University of Edinburgh
School of Informatics

Computer-controlled Agents for Complex 3D Environments in Real-time Computer Games

4th Year Project Report
Computer Science

Sean Hammond

May 24, 2004

Abstract: The design of complex agent characters for real-time computer games is an emerging science that promises great benefits for the realism and immersiveness of the game experience and is a potential application for the development of human level AI. This dissertation presents an investigation into the design of such agents using the Fly3D computer game engine to demonstrate the implementation of the solutions considered. Throughout the investigation the many and varied challenges posed by the agent design task are identified, and sets of solutions are proposed, considered and evaluated. A solution to the difficult problem of navigating a complex, arbitrary 3D environment is developed as well as a set of solutions to other challenges. These include controlling higher level agent behaviour, enabling agents to realistically engage in simulated combat with projectile weapons and an agent synthetic vision system. The investigation provides a clear identification of the techniques that could be used to develop flexible and robust computer game agents capable of existing in arbitrary, complex 3D environments in real-time and providing varied, convincing and engaging behaviour.

Acknowledgements

Thanks to my supervisor at The University of Edinburgh, Eric McKenzie, for guidance.

Thanks to Fabio Policarpo of Paralelo Computacao and the members of the Fly3D Internet forum for technical help with the Fly3D engine.

Thanks to Jay Bradley for pointing me toward sources of research information on game design, this assistance was invaluable to the project.

Thanks to CMP Game Group and Gama Network who provide Game Developer Magazine, The Game Developers Conference and in particular Gamasutra.com, an enlightening, invaluable and freely available source of game development articles which has been of great use throughout this project. Also thanks for mysteriously beginning to send me issues of Game Developer Magazine apparently free of charge.

Thanks to my fellow Informatics students at The University of Edinburgh who over the last couple of years have made regularly living in a computer lab until 4am a more than bearable experience.

Finally, thanks to the members of the Quake mod community who allowed me to use their carefully and skillfully crafted game environments for the purposes of this project.

Contents

0.1 Introduction to this report

This report presents the results of an investigation into the design of computer-controlled characters or 'agents' for 3D, real-time computer games.

The opening 'Background and synopsis' chapter gives an overview of the current state of AI in the field of computer games, a description of the Fly3D computer game engine and SDK used in the implementation portion of the project, a specification of the aims of the project and a summary of the main results.

The project was carried out in three major phases corresponding to the three seperate implementations that were produced to demonstrate the agent design solutions considered using the Fly3D engine. Each phase of implementation is described and evaluated in turn. Chapter 1 details and evaluates the first implementation, chapter 2 the second implementation, and chapter 3 details the third implementation and provides a final evaluation and identification of directions for further work. Closing conclusions are given in a final chapter.

The appendices provide results from the automatic area awareness data generation algorithm of the third implementation and implementation details relating to the the Fly3D engine and SDK and each of the three implementations in turn. A very large amount of code was implemented for the purposes of this project, and many of the classes written are very complex. The code itself is not described in detail as this would make the appendix unreasonably large. In general, each major class is described but fields and functions of classes and utility functions are not described. In particular, the second and third implementations include very large and complex classes that are not described in detail. Implementation has accounted for a significant portion of the time spent on this project, and there have been many difficult implementation problems relating to apparent bugs and other features of the Fly3D engine itself which have hampered progress throughout the project.

1. Background and synopsis

1.1 Project proposal and justification

The project proposal is to use the Fly3D SDK 2.0 games system as the base to develop a 3D computer game. Within this proposal, it was decided to carry out an investigative project into the area of Artificial Intelligence for real-time, interactive computer games. This choice was made following a number of influences:

- Of the many features commonly provided by public-domain computer game SDK's or game engines such as Fly3D, few provide coverage for the area of game AI. Graphics related areas such as rendering and animation are already well covered by the available SDK's, as is plugin-oriented game engine design. Sets of common solutions for use in these areas are well-known and well-grounded, leaving less room for innovation. In contrast, the area of game AI does not yet have a complete set of well-founded solutions, this grounding is currently being developed within the games industry and there is potential for much innovation.

- In terms of quality the area of graphics is currently saturated in computer games [14] and is becoming more so. High quality graphics are seen in practically every game release, and are beginning to be taken for granted by the game-playing public and game developers alike [26]. Although games are still sold today on the grounds of new, more advanced graphics features, graphics has declined notably as a major selling point for games. Due to the saturation of quality, it is difficult for a game to stand out today due to its graphical features, as has been the trend in the past.

- Key upcoming areas for development in computer games today are better physics and better AI. These areas have been neglected in the past while areas such as graphics and animation have developed quickly. However, as more impressive, realistic and interactive worlds for 3D, real-time games become available, the gap in sophistication between graphics and physics and AI becomes more pronounced. In todays computer game worlds, weak physics or AI can significantly hurt the users game experience.
  Due to the saturation in graphics quality, developers are beginning to turn to other features to make their game stand out and sell. More and more often, games are appearing with more advanced physics or AI as major selling features. For example, game franchises such as Ion Storms [36] 'Thief' [27] series published by Eidos [44] and the 'Hitman' [28] series from IO Interactive [37] also published by Eidos [44] are sold on the basis of unique

'stealth' styled game play that relies heavily on robust, intelligent, realistic AI. The 'Black and White' [29] series from Lionhead Studios [38] published by Electronic Arts [45] is sold on the basis of unique, interesting game play that is provided primarily through unusually advanced AI characters.

- Realistic physics for real-time, 3D game environments is quickly being realised and becoming more common place. For example, recent and upcoming titles like 'Half-Life 2' [30] developed by Valve Software [39] and published by Sierra Entertainment [46] and 'Max Payne 2' [31] developed by Remedy [40] and published by Rockstar Games [47] boast highly advanced in-game physics provided by the 'Havok' [33] game dynamics SDK which can be licensed by game developers for use in their games. Because of this trend I believe that realistic physics solutions will soon become firmly grounded within the computer games industry. For realistic AI, this is not yet the case.

- It has been widely predicted for some time that multi player gaming via the Internet, in which human players interact with other human players within a game environment, is set to be the next major evolution in the computer game industry. So far, no major online gaming revolution has emerged. The market for multi player online games remains a small corner of the computer game market, particularly in the area of home console games which represent the biggest section of the market. Though analysts still predict that online multiplayer games will be the next big market expansion [17], some major developers and publishers of computer games have downplayed the importance of gaming on the Internet [16]. Interesting and intelligent computer controlled characters are therefore an important element for the games industry today, and will remain so for the foreseeable future. As online gaming expands, computer controlled characters will be needed in this market also to fill the many roles not suitable for human players.

1.2 Overview: The current state of AI in 3D, real-time computer games

Over the last few years Artificial Intelligence has for the first time been recognised by major computer game developers as an important part of the design process of real-time interactive games [11].

At the Game Developers Conference (GDC) [12] AI roundtable discussions in 2000 a clear trend was beginning to be noticed towards AI becoming a more and more important feature of interactive games [11]. At the GDC in 2000, nearly 80 percent of developers present reported at least one team member working full

time on AI and nearly a third reported two or more team members, an increase from 60 percent and 10 percent respectively in 1999, and from 25 percent and less than 5 percent respectively in 1997 [11].

In addition, there has been a steady increase in recent years in the portion of CPU resources game developers are devoting to game AI [11] [26].

These trends are fueled by a number of factors:

- The increase in available CPU speed due to Moore's Law [6], which allows developers to devote more CPU cycles to a games AI.
- The development of advanced graphics processing chips, again following Moore's Law [6], which take more and more graphics processing cycles away from the CPU freeing the CPU up for use in other areas including AI.
- Computer game developers actively looking for AI features to differentiate their product from the competition.
- Increased use of modular AI design techniques for computer games so that AI behaviours can be easily tweaked and altered as game play changes, allowing AI to be incorporated much earlier in the development process.

The result is that an increasing number of games are being released with sophisticated AI features, often emphasising interesting AI features as a major selling point. Today, a games AI is more and more often considered to be as important a factor as a games graphics, and the games industry is realising that good AI is just as complex and difficult to produce as good graphics or animation.

Recent trends in game AI

Artificial life (A-Life) techniques provide an inherently simple and flexible means to introduce realistic, life-like behaviour. Solutions to low-level problems such as 'walk to an item' and 'pick up an item' are combined by a decision making hierarchy perhaps driven by simulated emotions or needs. The result is complex, apparently intelligent emergent behaviour that does not need to be explicitly coded.

For example, Maxis' [41] highly successful series 'The Sims' [32] published by Electronic Arts [45] uses an A-Life technique called 'Intelligent Terrain'. In 'The Sims', items in the terrain broadcast information to AI characters. For example, a fridge might broadcast a 'food' signal to nearby characters, or a television might broadcast an 'entertainment' signal. The characters then make decisions based on the signals they are receiving and their current simulated needs or emotions, and employ low-level functions to carry out these decisions resulting in life like behaviour.

'The Sims' AI behaviour includes an example of a 'need mechanism' in which an AI characters needs (such as food or entertainment, or perhaps ammunition in a typical combat game) are modeled by several variables based on the 'amount' of a particular need that the character already has at the current time (for example how much has the character recently eaten, how entertained are they feeling, how much ammunition are they currently carrying). Needs are weighed against each other in order to decide behaviour. [7] describes the approach taken to developing a need mechanism for a simulation of families of gorillas.

A-Life techniques are also used within other, more traditional game genres. For example group movement of AI characters using A-Life flocking algorithms such as 'Boids' [19] has become common in recent years.

Finite State Machines (FSM's) and hierarchical AI techniques remain popular in games development [13] [26]. Due to their simplicity they are considered more useful than more academic techniques such as neural networks and genetic algorithms. They are flexible, easily customisable and easy to test and can result in convincing behaviour with relatively little complexity. Increasingly the trend has been toward the use of Fuzzy State Machines (FuSMs) which employ fuzzy (as opposed to binary) logic [8] [9] to decide state changes. This allows for the evaluation of non-binary conditions and can produce varied and convincing behaviour. Fuzzy-logic has been applied in commercial games including Epic Games' [42] 'Unreal' [34] and Loki Games' [43] 'Civilisation: Call to Power' [35] published by Activision [48].

Advanced path-finding has been a major topic of development in recent years, possibly fueled by a spurt of games that were highly criticised for bad path-finding [13]. The computer games industry is just now beginning to get the problem in hand and look at more advanced techniques involving realistic movement in complex 3D environments, path-finding for vehicles with varying movement restrictions (for example the minimum turn radius at a given speed), path-finding with groups of characters and terrain-analysis path-finding. Innovative uses of common algorithms such as the A-star algorithm have recently appeared. For example, time-slice path-finding allows a path-finding algorithm to be run in 'slices' across multiple frames of simulation, so that sophisticated paths can be produced without affecting the games frame rate. Hierarchical path-finding allows high level AI to give high-level movement orders and leave the specific route up to lower level modules or characters. This might be used by a strategical AI to issue unit movement orders, leaving the exact route up to the individual units themselves but without ordering units to areas which they cannot reach, or could be used to guide a character across a very large level without at once computing and storing a large amount of path detail.

Traditionally there has been a mutual lack of interest between the fields of computer game AI development and academic AI research. There are a number of

reason for this:

- Though computer games have adopted some techniques from academic AI, for example decision trees and the A-star algorithm [14], academic research does not tend to cover areas of interest to computer game AI such as non-trivial pathfinding or agents with controllable personality parameters.
- Computer game AI typically requires simple solutions using minimal processor and memory resources, whereas academic AI solutions are typically very large programs with large requirements.
- Academic AI tends not to use computer games (other than board or card games) for research because modern games do not lend themselves well to techniques for precise, empirical evaluation.
- Finally, the goals of academic and computer game AI differ fundamentally. The aim of Academic AI is to create artificial intelligence, the aim of computer game AI is to create the illusion of intelligent behaviour.

The gap between academic and computer game AI however has begun to close in recent years. Professor John E. Laird of The University of Michigan cites three driving forces behind this convergence [14]:

- The increase in processor resources available to AI in computer games, as mentioned previously, which opens up possibilities for using academic AI techniques that previously would have been too resource intensive.
- A growing sociological effect due to students who grew up playing computer games taking advanced degrees in a AI and bringing a cross-pollination of ideas between the two fields.
- The game playing public who are increasingly beginning to demand more realistic AI features from computer games, and are increasingly dissatisfied with typical half-way techniques employed for AI in computer games.

As already mentioned, computer game titles are beginning to appear which boast advanced AI features. On the other side, computer games are beginning to be used as inexpensive, reliable and accessible research environments for academic AI. For example, Professor John E. Laird's research group at The University of Michigan uses numerous computer games as the environment for research into human-level AI [20]. Laird proposes that interactive computer games are the 'killer application' for human-level AI - the key application area in which the goal of human-level AI can be successfully pursued [23]. The academic world is fast beginning to realise that computer games are a fitting subject for education, academic study and research and that the task of developing AI for computer games is much more than 'just an engineering problem'.

1.3 Pathfinding

The pathfinding problem has been a major topic of the work undertaken for this project, and so it is given special treatment here. At the root of the challenge of making an intelligent-seeming AI character for a modern computer game is the problem of making the agent capable of navigating an arbitrary, complex 3D environment in a human-like manner, called **pathfinding**. In order to take part in the game, computer-controlled characters of all types require the ability to navigate their way from point to point throughout the environment in order for example to find objects and characters such as the player or other entities. Pathfinding decisions are the most common form of decision in computer game AI across all game genres and all types of game entity [55] and pathfinding forms a core component of most modern games [57].

The pathfinding problem is far from a trivial problem and so despite its importance has not always been solved well in commercial games in the past [55], and developers of recent games continue to report significant difficulties in developing pathfinding solutions. In [58] for example game developers from Raven Software discuss their struggle with the pathfinding for their title 'Star Trek: Voyager - Elite Force'. This is a complex and difficult problem for which not all questions have yet been adequately resolved [57]. Traditional techniques used with 2D or simple 3D games in the past fail when faced with the complex, 3D environments of todays games [54]. In computer game design today pathfinding, to quote [54] is 'one of the hardest problems to overcome efficiently and believably' and in particular, the generation of descriptive data on which to execute pathfinding routines such as graph searching algorithms is 'a herculean task'. As of yet, no simple solution has been proffered to pathfinding in true 3D. There is no specific best approach, the solution depends on the nature of the game, its characters and other factors [57]. A general approach is unlikely to be found for 3D games of all genres, but a general approach for a broad genre of games is not implausible and would be a significant goal.

1.4 Overview: The Fly3D 2.0 SDK

Fly3D is a plugin-oriented, OpenGL-based 3D games engine and development kit developed by Paralelo Computacao [3]. It offers rendering, input, sound and scene control methods for game programmers, as well as a variety of tools for game developers to add and modify game content [2]. The following is a brief overview of the Fly3D 2.0 SDK, for a detailed description of the features of Fly3D relevant to this project see appendix B.

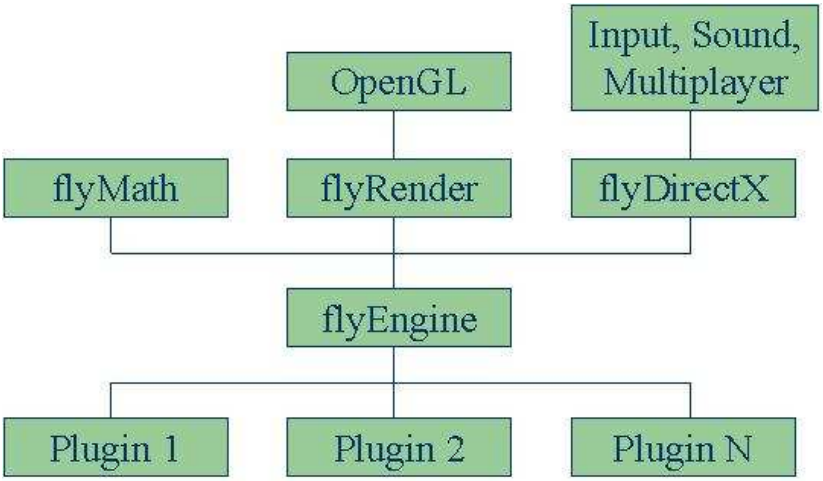


Figure 1.1: The diagram shows the architectural relationships between Fly3D's major DLL modules.

1.4.1 Software architecture

The software architecture of the Fly3D engine reflects a trade-off between two major design motivations: performance on the one hand, and ease-of-use and extendability on the other. For performance, highly related methods and classes are closely located and accessible in the code, avoiding unnecessary levels of indirection [1]. For extendability, plugin-orientation is provided making it simple to add new behaviour and features to the engine without recompiling the engine itself.

A new game or application for Fly3D can be developed as a plugin in the form of a DLL linked to the engine, making use of Fly3D's interface classes, methods and variables [1].

The following is a brief description of each of the major modules, and features of Fly3D, for more detailed information on each module refer to the Fly3D SDK 2.0 documentation [1].

flyMath The `flyMath` module exports several classes implementing all the mathematical operations needed by the simulation. The main classes provided represent common entities in the mathematics of 3D graphics and their associated operations: `flyVector`, `flyMatrix`, `flyQuaternion`, `flyPlane` and `flyVertex`.

flyDirectX The `flyDirectX` module acts as the interface between Fly3D and the Di-

rectX API. DirectX is used to provide sound, input and networking (for multi player gaming) features.

flyRender The `flyRender` module uses the OpenGL graphics API to perform all the rendering operations of the simulation.

flyEngine The `flyEngine` module integrates together the other three main modules above (`flyMath, flyDirectX` and `flyRender`) and provides the interface through which user-programmed plugins can access these back-end modules in order to make use of Fly3D's features [1]. The `flyEngine` module also handles initialising the simulation and plugins, per-frame update of the simulation and can coordinate several plugins at once. The `flyEngine` class encloses the most vital data and methods of Fly3D including data loading, lighting calculus and BSP recursion functions, the BSP tree itself, geometry data in the form of vertices and faces and all the simulation global parameters.

1.4.2 Front-ends

Fly3D provides five default front-end applications. These include tools for shader editing, on-line multi player servers and ActiveX control for running Fly3D applications within web browsers. For simulation and development, the main front ends are FlyFrontend and FlyEditor:

FlyFrontend.exe FlyFrontend is the primary front-end application of the Fly3D engine. It provides a rendering window in which simulation occurs as well as a menu which allows Fly3D scenes to be loaded for simulation and several options to be switched. Options include display resolution, texture filtering modes, input configuration and more.

FlyEditor.exe FlyEditor is the primary editing utility of the Fly3D engine. It combines a categorised tree-like view of all objects in the scene, an area for viewing and editing object, plugin and global parameters, and a rendering window in which simulation occurs. FlyEditor opens and saves Fly3D scenes, and can modify, add and delete entities and plugins in a Fly3D scene.

1.5 Project specification

The specific task approached by this project is to develop a 3D computer game in which the human player is opposed by a computer-controlled 'agent' or 'agents'. 'Agent' is a generic term in AI to describe software that perceives a world, thinks, then effects actions on the world [4]. In the field of computer game development,

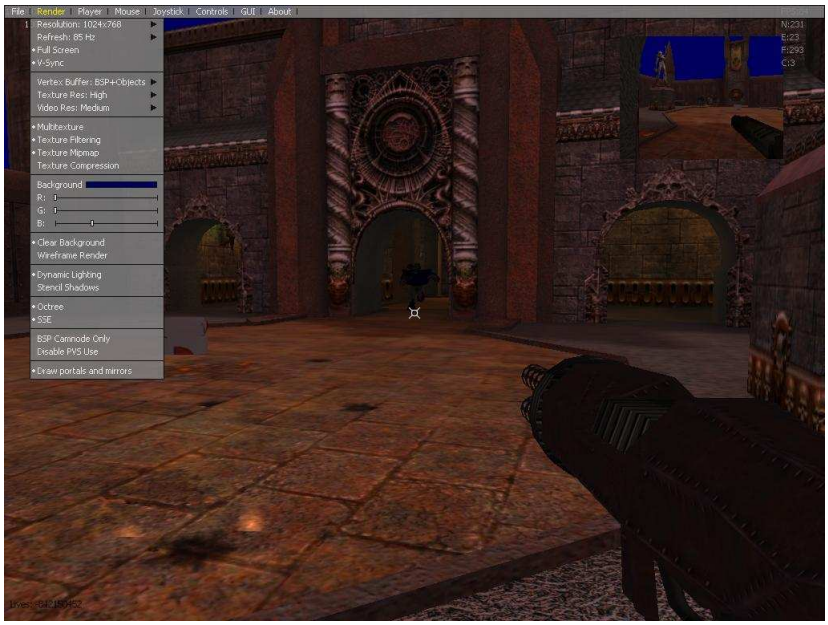


Figure 1.2: The FlyFrontend application

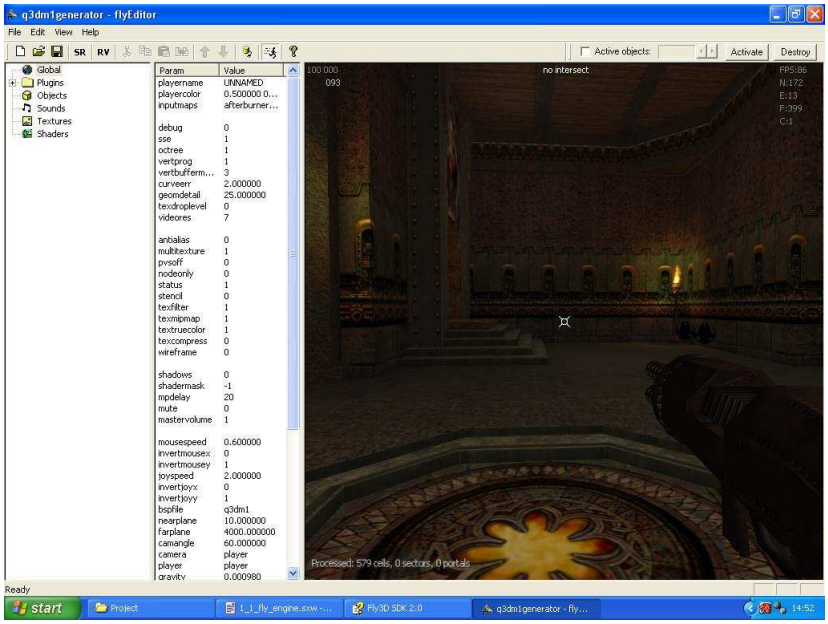


Figure 1.3: The FlyEditor application

agents are often referred to as 'bots' (a shortening of robot), or as 'units' or 'NPC's' (Non-Player Characters). The design of the game used is influenced heavily by the resources provided by the Fly3D engine. The features provided by Fly3D, as with most computer game SDK's available, are heavily biased toward the development of games of the 'First-Person Shooter' (FPS) genre. In these games the player views the world in a first-person perspective from the player character and is pitted in combat against his opponents using a variety of weapons, usually projectile weapons such as machine guns, laser guns and rocket launchers.

The game will be developed as a plugin for the Fly3D engine. The game will also make use of features provided by the Fly3D standard plugin 'walk' (see appendix B). The 'walk' plugin implements basic animation and physics functionality in order to provide a fully animated person character that can run and walk around an environment, and can carry and fire a projectile weapon. Variations on this character are used by the player and various agents in the game. Fly3D also provides a tool which can convert environments from the infamously violent FPS game 'Quake 3 Arena' for use in the Fly3D engine. The environments used in my game are environments designed for 'Quake 3 Arena' which have been converted to Fly3D. A wide range of environments is available in terms of size and complexity, and the Fly3D conversion tool provides items such as weapons, ammunition and healing bonuses scattered around the environment which can be collected for use by game characters.

The plan is to develop the game iteratively, starting with a relatively simple implementation using a simple environment and straight-forward agents which is used as a base to develop more complex solutions using complex environments. As such the work of the project will be split into a number of seperate implementations, each building on the work of the others.

The goal of the project is to investigate the design of agent characters for games of this type, consider solutions to the major problems posed such as agent navigation and agent behaviour and implement agent characters to demonstrate solutions to these problems.

1.6 Summary: Main results

The work was carried out in three distinct implementations.

The first implementation achieved a complete game in which the player must battle computer-controlled agents in order to achieve a task. The game used a relatively small and simple environment and contained relatively simple agents to operate in this environment. The work on this implementation identified the

major problems posed by the task of developing agents for this type of game: agent navigation, agent use of projectile weapons and controlling overall agent behaviour, and provided a set of basis solutions on which to build more complex agents for use in larger and more complex environments. The structure and workings of the Fly3D engine and the technique of developing a plugin for the engine were learned in order to develop this implementation.

The second implementation investigated the extension of the solutions used in the first implementation to a significantly larger and more complex environment. The work on this implementation significantly improved the agent navigation system with the development of a hierarchical navigation system as well as other improvements and developed new agent navigation sub-systems to deal with the unexpected complexities of the new environment. The agents behaviour control was also improved and the agents use of projectile weapons was improved significantly with the development of a realistic and parametrisable system. Ultimately however the underlying basis of the agent navigation system was shown to be too weak for complex environments and the improvements to the system developed though useful could not compensate for its shortcomings.

The third implementation investigated the development of a new navigation system aimed at conquering the fundamental weaknesses of the previous system. The result was a success, a fully-automated system was developed capable of navigating arbitrary, complex 3D environments and showing significant promise for further development. The agent character was successfully transferred to the new navigation system and further enhanced with an additional synthetic vision system, improving the realism of the agent character. A game was completed in which the player battles an agent in one of many available environments, both the player and the agent must navigate the environment in order to find items such as weapons, ammunition and extra health and to hunt down and defeat their opponent. The agent judges its need for each type of item available and seeks out items accordingly. The agent is capable of using each of the different types of weapon available, and will attack if it sees the player.

The work undertaken represents a thorough investigation into the design of agents for real time, 3D computer games, the various challenges this poses and the available solutions. The work demonstrates a fully-automated, general, flexible and robust solution to the complicated problem of pathfinding in arbitrary, complex 3D environments and a set of solutions to other major problems and presents significant promise for further development.

2. Implementation one

The aim of the first implementation was to explore the use of the Fly3D engine and develop a simple game using computer-controlled agents as a basis for further work. This implementation identifies the basic problems that must be solved for computer controlled agents in this type of game and develops the initial framework for solutions to these problems. The game itself was designed with these aims in mind. It is simple enough to be implemented in a playable form in a reasonable time and requires only simple agent characters.

2.1 The game

- The player starts off in a designated safe area somewhere in the environment.
- A number of friendly agent characters are positioned in the environment. The players aim is to find as many of these characters as possible and lead them safely back to the safe area in order to score points.
- Throughout the playing of the game enemy agent characters appear from 'spawn points' in the environment and attack the player and friendly agents. The player must fight back these characters to succeed.

2.2 Agent navigation

Well known search algorithms from the fields of graph theory and conventional AI, which produce paths through discrete graphs, can be applied to the pathfinding

Figure 2.1: Screenshots of the environment used for the first implementation. From left to right, the environment consists of a courtyard area, a series of interconnected corridors and a large open room.

problem for agent navigation. The first difficulty is defining a suitable graph to act as the search space for these techniques. In 2D games such as strategy games, a grid on which to run searches can easily be overlayed on the environment, but a continuous, real-time, 3D world does not readily lend itself to easy discretization of this kind. In this first implementation the problem was solved by applying a global structure of navigation nodes and edges between nodes which can be accessed by all agents in the game.

Each navigation node consists of a position (a point in 3-space) and a list of pointers to its neighbouring navigation nodes (those which it is connected to by edges), and the navigation nodes are positioned carefully so that the nodes and edges form a graph describing the important features of the shape of the environment (see figure 2.2). The graph formed should be a minimal graph providing enough detail for the agents to satisfactorily navigate the environment. A new navigation node structure must be built by the game designer for each environment, and this is achieved using the FlyEditor application and the `myCamera` class (see appendix C.4). Each navigation node is an instance of the class `NavPoint` (see appendix C.9) and is added to the scene by the designer using the FlyEditor application. Nodes are placed following some simple rules:

- From any reachable[1] position on the map there must be a direct, unobstructed line of sight to at least one node.
- A nodes neighbours are all those nodes to which it has a direct, unobstructed line of sight. Following this rule the edges of the navigation graph can be defined automatically once the navigation nodes have been placed.
- Neighbouring nodes should not be placed too far apart. For example, a long straight corridor could be navigated using a node at either end. Instead, nodes should be placed periodically along the corridor. This is to do with the way the agent navigates to a moving target, explained below.

The node structure can be used by an agent to navigate from any reachable point in the environment to any other reachable point in the environment.

The agents basic movement functionality allows it to orient itself in the direction of a particular vector and to move forward following this vector. If there is a direct, unobstructed line of sight between two points in the environment then the agent can move from one point to the other by following the vector between the two points.

When it needs to navigate to a point to which it does not have a direct, unobstructed line of sight the agent makes use of the navigation node structure and a path-finding function:

[1] A reachable point is any point in the environment which a character should be able to navigate to from its starting point.

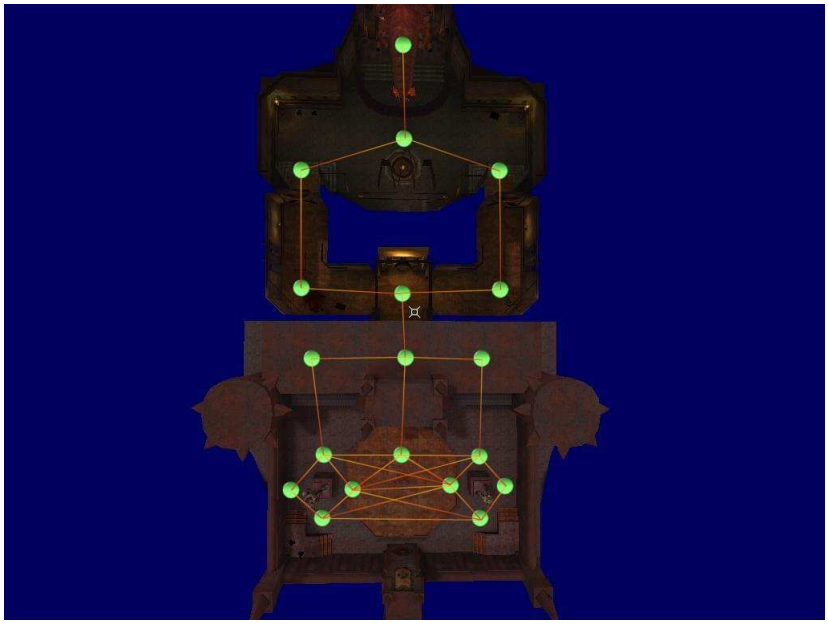


Figure 2.2: A plan view of the environment used for the first implementation, showing an example navigation structure constructed of 19 nodes and 31 edges.

- The agent requests a path to a particular location from the path finding function, specifying its own position and the target position.
- The path finding function finds the node nearest the agents position and the node nearest the target position, these are the start and target nodes.
- The path finding function then runs a search algorithm over the navigation node graph, finding a path of edges that leads from the start node to the target node. The path is returned to the agent as a list of pointers to nodes.
- The agent moves in a straight line from its position to the start node, then from each node in the path to the next node in the path, and finally from the target node to the target position. If the nodes have been placed in adherence with the guidelines above the agent will not be obstructed along its path.

There are several well-known algorithms for searching a graph that could be applied [55]. In this implementation, a form of heuristically-guided best-first search is used. This algorithm was chosen for its simplicity in comparison with alternatives such as iterative-deepening depth-first search and A-star search, and its speed of performance in comparison with breadth-first search or Dijkstra's algorithm. Nodes in the graph are dynamically weighted according to a heuristic measurement: the Euclidean (straight line) distance from the node to the target node. The search algorithm expands the nodes with the lowest cost according

```
1.best_first_path_find(vector start_position, vector target_position){
2.  start_node = nearest node to start_position
3.  target_node = nearest node to target_position
4.  push start_node onto path
5.  if start_node = target_node return path
5.  push start_node onto visited
6.  loop{
7.    let current_node be the last node on path
8.    iftarget_node is a neighbor of current_node:
9.      push target_node onto path and return path
11.   if all neighbours of current_node are on visited then backtrack:
12.     pop current_node from path
13      goto 6
14.   consider all neighbour nodes of current_node that are not currently on
      visited, and push the node with the lowest heuristic cost onto
      path and visited
15. }
16.}
```

Figure 2.3: Pseudo-code for the best-first search pathfinding algorithm. A list of nodes previously visited by the search `visited` is used to avoid looping, and a list of nodes `path` is returned representing a path from `start_position` to `target_position`.

to the heuristic first and so heads in a direct manner toward the target node. Pseudo-code for the pathfinding algorithm is given in figure 2.3.

Different agent behaviours can be built on top of this basic navigation functionality in order to produce characters such as the friendly agents and enemy agents in the game.

2.3 Following a moving target

If the agent is attempting to navigate to a moving target such as the player or another agent it must continually update its path to reflect the target's current position. If the path is not updated often enough, the agent will not effectively track down its target, but if the path is updated too often the agent can get caught in a loop continuously running back and forward between two navigation nodes.

The update time must be short enough to keep the path up to date, and the navigation nodes must be placed such that the agent can cover the distance

between any two neighbouring nodes within the update time.

2.4 Agent behaviour

Two simple agent behaviours are required for the game:

- Friendly agents must wait for the player to find them, then once found they must follow the player until they reach the safe area. If a friendly agent loses sight of the player it must be able to autonomously find its way back to the player.
- Enemy agents must autonomously find their way to the player whenever the player is not in sight, and must engage and attack the player when they see him.

The agents in this game can be described by a decision tree in which the taken action is decided at each step based on the agents current inputs by recursing the tree (see figure 2.4).

This rule-based approach has traditionally been the most common in game AI design for a number of reasons [10]:

- The approach is familiar to programmers because it is similar to common programming paradigms and can be coded in a straight forward imperative way for example using **if** statements.
- The designs produced are generally predictable and easy to test.
- Game developers traditionally lack in-depth knowledge of more advanced AI techniques.

2.5 Using projectile weapons

The enemy agent character carries a 'laser gun' type weapon which it uses to attack the player. It is a simple task for the agent to align itself with a target and then fire its weapon, but this behaviour will not produce satisfactory results for a computer game agent. It is not desirable for the agent to hit its target every time, since this does not produce a very exciting opponent and if the agents weapon is strong enough he may be impossible for a human player to defeat.

To address this problem the enemy agent character rotates its aim by a random angle (computed within constraints) before each shot it fires. The accuracy of the agent can be tuned by altering the constraint within which the random angle

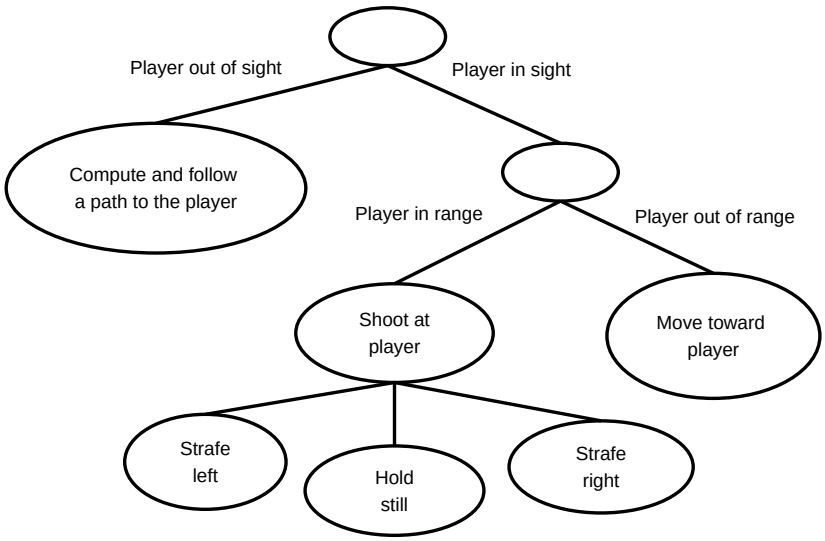


Figure 2.4: An example decision tree for the enemy agent character. In order to introduce some variety into the agent behaviour, the range for which the 'player in range' decision is made is altered by a random value (within constraints) each time the decision is made. The decision to strafe left or right or hold still is made by random choice and continuously updated while the player remains in sight and range, each update occurring after a timeout of random length (within constraints). The term 'strafe' is commonly used in reference to computer games and refers to the action of rapidly moving from side-to-side, perhaps in an unpredictable manner, in order to present a difficult target to an opponent using weapons such as projectile weapons while still being able to aim and fire at the opponent. Another common term 'circle-strafe' refers to the act of tracing a roughly circular path around an opponent via strafing, with the same aim in mind.

is produced. With a large range of values the agents aim is highly erratic and inaccurate, with a small range of values the agents aim is tight and accurate.

The effect can be thought of the agent shooting into a circle centred at the target of its aim. The shot will be placed at a random point within the circle. Inaccurate agents have large aiming circles, more accurate agents have small aiming circles.

2.6 Evaluation

The environment used is representative of the simpler sort of environment that might be seen in computer games, but is small and lacks the complexity and range of features you would expect to find in many game environments. This environment then acts as a good starting point to investigate the concepts of agent design, but solutions must later be extended to handle more complex environments.

2.6.1 Agent navigation

The use of a graph to describe the environment and a graph searching algorithm to produce paths for the agent to follow shows strength:

- The simple graph structure implicitly describes a lot of information about the environment - spacial, shape, paths and connectivity, and allows for the application of many well-founded graph searching algorithms.
- The search space produced is very small, 19 nodes and 31 edges for the simple environment used, meaning demands on processor time for searching and on memory space for storage are minimised.
- The graph is simple to construct following the small set of rules devised. For larger environments the search space will be larger and more difficult to construct, but should be of the same order.
- The paths produced, as lists of nodes, allow a path-following function that is a simple extension of the agents basic orientation and vector following movement functions to be used.

The system has solved the problem of agent navigation for the environment used very well, and shows promise for extension to a more complex environment.

The effect of the agent following a path produced by the system is less natural than might be desired. The agent moves in short straight lines and performs sudden turns (whenever it reaches a node). Of course the player cannot see the underlying nodes and edges that the agent is using, but none the less the agent

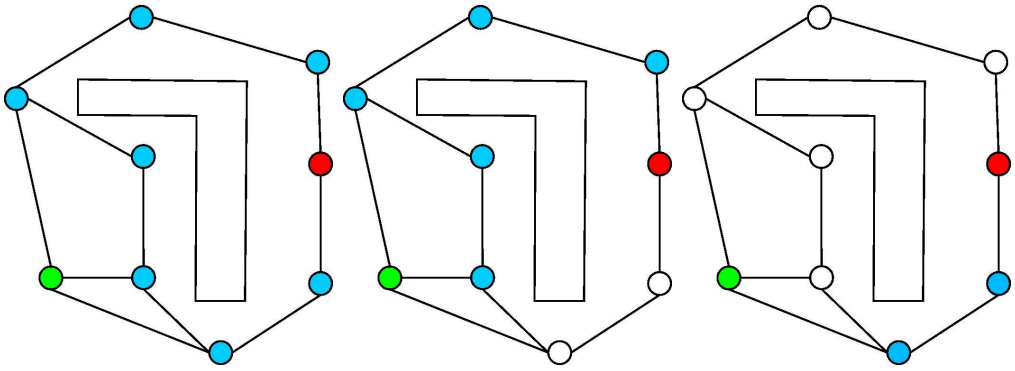


Figure 2.5: From left to right, example executions of breadth-first search, depth-first search and best-first search with the Euclidean distance heuristic. A path is requested around the polygonal obstacle from the green node to the red node, blue nodes are those considered by the algorithm in each case. Breadth-first search considers all nodes along the way, depth-first search considers few but produces an obtuse path, best-first search considers few but is guided by the heuristic along a direct path.

appears quite robotic. Human players move in more smooth, curved paths and an effect more like this would be desirable for the agent. There are a number of ways this could be attempted, some of which are discussed in section 4.5.

The path-finding function based on best-first search is effective but imperfect. The search itself does not spread out considering all directions through the graph as would a breadth-first, bi-directional breadth-first or Dijkstra's algorithm search and does not tend meander down many long, obtuse paths through the graph as an unguided depth-first search would do. The heuristic guides the search toward the goal so a result is found quickly and in a direct manner, without considering an inordinate number of nodes (see figure 2.5). This is important because many paths for many different agents often need to be computed per-frame in real time, so the algorithm must terminate quickly and this demand will grow stronger when larger and more complex environments are used.

The best-first search algorithm has three notable weaknesses:

1 The algorithm is a form of depth-first search and so is not guaranteed to find the optimal (shortest) path to the target. The algorithm is a *steepest-ascent hill climbing* [59] search (with the addition of a backtracking mechanism to assure that a path is always found if one exists).

2 The Euclidean distance heuristic used is not always accurate and may underestimate the cost to the target node from a particular node severely. For example a node may be very near the target node but on the other side of a wall, the heuristic will give a very low cost but in fact the path from the

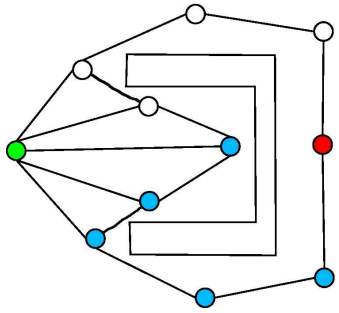


Figure 2.6: An example situation in which the best-first search algorithm returns an obtuse path due to the sometimes misleading nature of the Euclidean distance heuristic, the deth-first nature of the aglorithm and the failure of the algorithm to consider the length of the path taken so far to reach a node when deciding which node to consider next.

node to the target node may be very long due to the need to find a way around the wall (see figure 2.6).

3 The algorithm does not deal with nodes of different weights. For example in the way that Dijkstra's algorithm dynamically weights a node according to the length of the previous path from the start node to the node. Alternatively weighted nodes may be needed to represent different forms of terrain, for example it may be more costly to travel through a thick swamp than around it even though the path around the swamp is longer. The best-first algorithm is likely to return a path that wades directly through the swamp in order to reach the target.

These weaknesses combine to mean that the paths produced are not always optimal and can sometimes be quite obtuse and longer than necessary. This does not present a significant problem in this implementation because the environment used is quite small and simple. The agents sometimes follow slightly odd paths but will get to their goal in a more or less direct manner. The effect is also hidden by the confusion of using many agents and may even produce some extra variation making the agents appear more complex than they actually are. It is clear to see however that in a significantly larger and more complex environment the effect of non-optimal paths is likely to be more pronounced and may become problematic. The algorithms lack of support for weighted nodes represents a lack of potential flexibility. Though none is necessary in this implementation, some flexibility may be required of the search algorithm to deal with more complex situations such as those discussed in section 4.5.2.

2.6.2 Agent behaviour

As mentioned earlier, an AI agent is a piece of software that perceives a world, thinks, then effects actions on the world [4]. An agent can be thought of as a box which receives inputs from the virtual world and outputs effects on the virtual world. What goes on inside the box is the 'thinking' phase and the nature of this phase gives rise to a categorization of game agents presented in [53] as reflex agents, reflex agents with internal state and goal-based agents. [22] equivalently categorises game agents naming them reactive agents, context-specific agents and flexible agents:

- *Reflex or reactive agents* - these game agents react to their inputs in a rule-based manner, and can be implemented by a set of '**if** condition **then** action' rules and thought of as a condition-action lookup table or in a hierarchical manner as a decision tree. The agents have the advantage that they are usually simple in terms of computational requirements and design and can respond very quickly to changes in the environment that they perceive at their inputs. Their simplicity is also their weakness, these agents have no capacity to remember past events or inputs. For example if the agents opponent moves behind an obstacle so that the agent no longer sees the opponent then the agent is not able to remember that the opponent was visible just before it moved behind the obstacle and act accordingly.

- *Reflex agents with internal state or context-specific agents* - these are reflex or reactive agents with the additional capacity to remember past events. They can be implemented by a finite state machine or hierarchical finite state machine with persistent state which acts as a form of internal memory. Alternatively the agents internal memory could be programmed explicitly as a record of a set number of previous states of the agents inputs, or a combination of the two approaches could be used. A past record of the agents actions or outputs may also be stored either explicitly or within the agents state. If an agents opponent moves behind an obstacle the agent might find itself still in an 'attack' state, but now having lost sight of its target it might fall back on its record of previous inputs to guess the targets location and move to it.

  These agents none the less have the shortcoming that they employ no form of forward planning and do not consider the effect of an action before performing it, beyond any such consideration that is explicitly programmed or designed into the agent by its human designer. The use of a pathfinding function to plan routes may be considered a form of forward planning, but the details of the navigation system are ignored here and considered low level functionality as opposed to high level behaviour.

- *Goal-based or flexible agents* - these agents consider the consequences of actions rather than just reacting to the current state of their inputs. They have a choice of high level tactics with which to achieve their goals and a choice of low level behaviours with which to implement their tactics. For example they might use search algorithms to generate action sequences that act on the world-state and achieve a goal. The behaviour of these agents can be emergent - behaviour may result that the designer did not anticipate. The SOAR game agent [20], [22], [24] is a goal-based agent that has a hierarchical structure of goals, tactics and behaviours constructed as a hierarchical finite state machine and an inference engine which decides the agents state within the structure based on its knowledge, memory and sensory inputs. Other approaches such as the need-based mechanisms used in [7] and [32] are also examples of goal-based agent designs.

Another categorization of game agents specified in [4] consists of top-down and bottom-up designs:

- *Top-down* - with a top-down agent design the developer has complete knowledge of how she wants the agent to behave and creates a system to behave in this way. Agents built explicitly as decision trees or finite state machines are often top-down designs, and the SOAR game agent is an example of a top-down design.

- *Bottom-up* - with a bottom-up agent design high level behaviour emerges from the interaction of low level mechanisms. The designer might implement a set of simple reactive rules and apparently complex behaviour which was not explicitly programmed can emerge from the interaction of these rules. The 'boids' [19] flocking behaviour (see below) is an example of a bottom-up design.

Given these categorisations, the agents developed in this implementation are reactive, top-down agents, with the added feature of non-determinism due to the use of randomised decision making in the decision tree. Even with this 'fuzzy' element, the agents remain simple and more or less predictable from the point of view of the player. The agents provide a reasonable challenge when playing the game but rely on several factors to do so:

- The agents present in the game environment always outnumber the player.

- There is a constantly replenishing supply of agents throughout the game.

- The agents have unfair knowledge of the players position so that they can always path-find directly to the player and attack him. This in particular allows for apparently more than reactive behaviour from the agents. For example, if an agents opponent (the player) runs out of sight behind an obstacle the agent has no capacity to remember the player moving behind

the obstacle, but having lost sight of the player will default to the pathfind state and compute a path around the obstacle to the player. The agent will run around the obstacle in order to get to the player, as you might expect a context-specific agent to do. This sort of trick however will not fool a human player for long, and they will soon deduce that the agents are cheating, a common complaint against computer game AI.

This means that the agents don't provide a very interesting challenge to the player, and the use of these sort of gameplay inbalances instead of genuinely challenging AI opponents is fast becoming unacceptable to the gameplaying public.

For the purposes of implementing an agent that plays a human or human-like character in a modern game environment, reactive agents are too simplistic and can only form the basis on which to develop more complex agents. None the less, reactive agents do have their place in modern, complex computer games.

Reactive behaviours can be used in 'behavioural animation' techniques in order to simulate for example the flocking behaviour of birds, fish or herds of mammals. The 'boids' [19] flocking algorithm for example is a reactive, bottom-up approach that implements flocking behaviour. 'Boids' was used in the Disney animated film 'The Lion King' in 1994 to animate herds of animals [4].

The 'boids' simulation is based on global direction vectors and the position of AI entities with respect to other nearby AI entities. Each entity follows a simple set of three rules in priority order:

1 Avoid collisions with nearby entities in the group.

2 Match velocity with nearby entities in the group.

3 Stay close to nearby entities in the group.

The result is a startlingly realistic, emergent, group behaviour similar to the flocking behaviour of animals in the real world. More advanced systems of behavioural animation have been developed involving sensory models (e.g. models of vision) and physics [4]. In this context, reactive agent design can play a useful role in 3D environments for computer games, controlling groups of animals or other units and even crowds of people that need to move as a group without colliding with each other (the problem of large crowds is very difficult to solve using conventional agent navigation and obstacle avoidance techniques).

Though they are not suitable to provide the leading agents of a complex game, a variety of reactive agents playing background roles, such as flocks of birds or other animals for example, can provide convincing and entertaining background activity in a game environment. The technique of combining a number of different simple reactive behaviours in order to create the impression of greater complexity

has a long history in computer games, this for example is the basis for the 'ghost' characters that provide the famously addictive gameplay in some implementations of the arcade game 'Pacman'.

2.6.3 Using projectile weapons

Currently the agents use of projectile weapons is a passable but limited effect.

The randomisation technique used to simulate human-like imperfect aiming and varying degrees of aiming skill works well but looks slightly odd. Shots are seen flying away above the players head and into the ground well ahead of the player as often as they are seen skimming from side to side of the player, and these sort of shots seem like they should be more unlikely. A human does not aim randomly within a circle around its target. For a human the task of aiming a weapon involves judging the movement of the target and aiming so that the projectile will connect with the target. A players aim is usually held steady in the up-down axis but varies in the left-right axis as she attempts to judge her targets movement and adjust her aim accordingly. This is because typical targets such as the characters seen in this game that run and walk are more likely to be moving left or right in the field of vision than up or down. The agents aiming model does not take into account this 'leading aim' technique and hence the effect is not as convincing as it could be. This leads to the further limitation that the agent cannot hit a moving target unless it is moving directly toward or away from the agent or the agent achieves a lucky shot via the randomisation of the aiming position. The agent will always centre its aiming circle directly at the targets current position, a moving target such as the player will likely no longer be in this position by the time the projectile reaches it. This effect is made less noticeable by the confusion of having many agents shooting at the player from different directions at once, but will none the less soon be picked up on by human players.

The agents also use only one weapon, the laser gun which they are given when they are created, and never run out of ammunition for this weapon. The player on the other hand experiences a situation much more common in realistic computer game environments in which she has a limited amount of ammunition and must collect extra ammunition from the environment as the game progresses. The player can also find new weapons in the environment and use a variety of different weapons often carrying more than one at once and choosing which is most suitable.

3. Implementation two

The aim of the second implementation was to develop a new game, similar in design to the game of the first implementation, that furthers the work undertaken during the first implementation. This implementation explores in more detail the navigation and agent behaviour concepts encountered in the first implementation and also considers some more advanced concepts.

The game in this implementation is set in a much larger, more complex environment which was chosen to be representative of a large portion of the environments commonly encountered in 3D games. The environment includes a wide set of real-world type features as would be encountered in a realistic commercial game (see figures 3.1, 3.2, 3.3, 3.4 and 3.5).

3.1 The game

In this game the player is pitted against a single computer controlled agent. Items to be collected by the player and agent are scattered around the environment, including weapons to use and different types of ammunition for each of the weapons, and health items that regenerate health points lost in encounters. Either the player or the agent is defeated when his health parameter reaches zero, and both must roam the environment attempting to defeat the other while collecting items to help and heal themselves.

3.2 Agent navigation

The first task when developing the second implementation was to transfer the agent navigation system over to the new, more complex environment. This presented a plethora of difficulties that were not foreseen during the first implementation and required significant advancement of the navigation system itself.

3.2.1 Improving the graph structure

The navigation node system had to be extended to deal with three major problems:

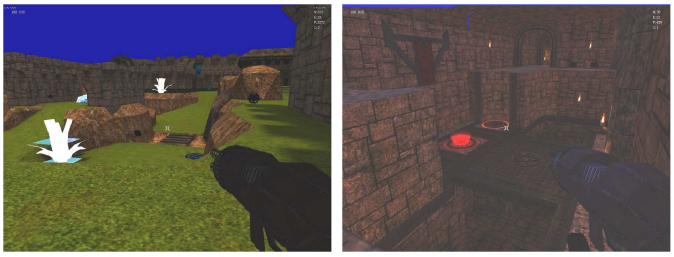


Figure 3.1: Multiple levels, raised platforms and drops.

Figure 3.2: Hills of various shapes and sizes.

Figure 3.3: Stairs of various shapes and sizes (left), lakes and bridges (right).

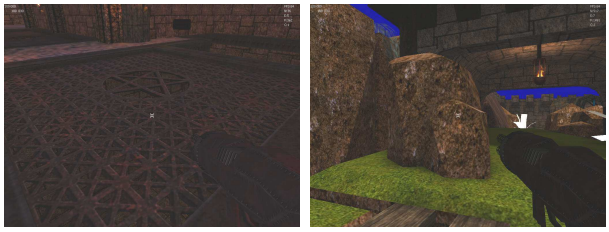


Figure 3.4: Grated flooring (left), arbitrary shaped obstacles (right).

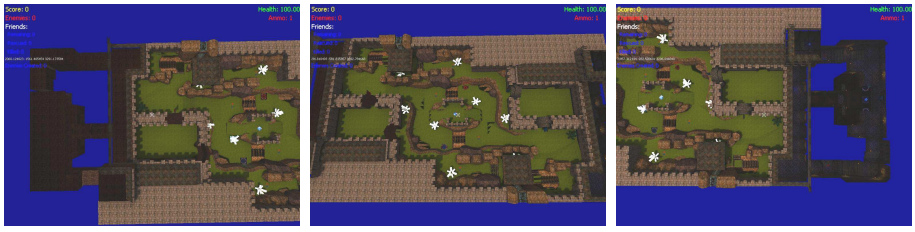


Figure 3.5: Large, complex layout.

- The 'line of sight' rule for defining edges between nodes developed in the first implementation: 'a nodes neighbours are all those nodes to which it has a direct, unobstructed line of sight' does not hold in the new environment.

  Multiple levels, raised platforms and drops (3.1) result in edges that should not exist and can cause an agent to run off ledges or into walls. For example, if two raised ledges are positioned at opposite ends of a lower area there can be a direct, unobstructed line of sight between a node on one ledge and a node on the other. If the agent attempts to move between these nodes by following the vector between them it will fall off the first ledge. There may also be a direct, unobstructed line of sight between a node in a lower area and a node in a raised area. Following the edge between these nodes may cause an agent to either run off a ledge or to run into a wall depending on which direction is being attempted, or it may be that there is a hill or staircase connecting the two areas in which case it is perfectly safe for the agent to traverse the edge.

  Large hills or staircases (3.2, 3.3) can cause erroneous discontinuities in the graph. It may be possible to move following a vector from one node to another by running up a hill or staircase, but because of the hill or staircase there may not be an unobstructed line of sight between the two nodes.

  With an environment the size of the one being used, which is representative of many environments found in modern computer games, the task of placing navigation nodes becomes prohibitively difficult, time consuming and error prone. If the designer were also required to manually define the edges of the graph the situation would become unmanageable.

- Searching the graph structure for path-finding is an inefficient and wasteful process. For games of this genre path-finding must be done in real-time and it may often be necessary for many different paths, for different agents, to be computed at once. The graph structure is large and complicated so the path produced can also be large and can take a long time to compute.

  This problem cannot be addressed fully by improving the search algorithm used for path-finding. It is necessary in complex environments for the paths followed to be optimal (see section 3.2.3), and even the fastest search algorithm that finds optimal paths will not perform in real-time when carrying out multiple searches over a very large graph.

In this implementation these problems were dealt with by developing a hierarchical graph structure. The environment is split into a number of distinct areas of roughly equal size, each of which is small enough to be searched easily in real time. Areas must be defined so that within any area the 'line of sight' rule for edge placement holds. For example a raised platform that overlooks a lower level must always be defined as a seperate area from the lower level, so that within

one area there are no drops or ledges.

Figure 3.6: An example screenshot showing the area subdivision of a portion of the environment.

The designer must now define the areas of an environment then a graph of nodes within each area. Defining areas and placing nodes within an area is not overly difficult, and is comparable to the task of designing the graph for the simpler environment in the first implementation. Edges within areas can be placed automatically following the 'line of sight' rule. The designer must then manually define additional edges that lead from one area to another wherever it is safe for an agent to traverse areas, and this is a relatively trivial task. Nodes connected by these special manually defined edges are called **endnodes**, and the areas and edges between endnodes form a higher-level graph which connects the underlying area subgraphs.

Overall the graph structure is still very large and so specialised debugging tools must be used to ease verification of the correctness of the structure and to refine the structure when necessary (see section 3.2.2).

For pathfinding, the search algorithm can now be run on individual areas in real time, avoiding the need to run large searches over the entire graph (see section

3.2.3).

Figure 3.7: An example screenshot showing a section of a navigation node graph. The white lines overlayed on the environment are the edges of the graph, nodes are not actually drawn but are positioned at the endpoints of edges.

3.2.2 Improving the graph creation process

To develop a navigation node graph structure for an environment, such as that developed in implementation one, a designer must place a set of nodes such that[1]:

- From any reachable point on the environment there is a direct, unobstructed line of sight to at least one node.
- The nodes form a minimal graph that adequately describes the shape and key features of the environment, and are not placed too far apart.

Within a large and complex environment the task of placing navigation nodes becomes significantly more difficult and prone to error. For this reason, some

[1]See (section 2.2).

specific tools were developed to aid in the task. The designer can lay down a preliminary graph and then use these tools to debug and refine the graph.

- The `myCamera` object (see appendix C.4) is adapted to aid in this task. The `myCamera` object is a controllable floating camera from which the scene can be viewed. With the plugin in debug mode the user can easily look around and move about within the scene via the `myCamera` object while the simulation runs. In this implementation, the `myCamera` object is configured to continuously output its x, y, and z position coordinates and the integer ID of nearby navigation nodes.
- When in debug mode the plugin graphically overlays the navigation graph onto the scene, drawing the edges between nodes (see figure 3.7). Combined with the `myCamera` object this allows the user to easily inspect the entire navigation graph node by node, giving an in-scene visible representation of each node and the edges coming out of it, or to observe an overview of the overall structure of the graph or the overall structure of a particular area or section of the graph.
- The graph itself is specified by an input file stored in a simple human readable from. The input graph is specified in an XML-style format and consists of a list of areas (identified by integer ID) and nodes within areas (also identified by integer ID). The manually created edges which interconnect areas via endnodes are also specified in the input file. The file format looks like this:

```
<Input file>
    Graph
</Input file>

<Graph>
    Area+
</Graph>

<Area>
    (node | endnode)+
</Area>+

<node>
    float float float
</node>+

<endnode>
    float float float int
</endnode>+
```

The input file defines a single `Graph`. A `Graph` consists of one or more `Area`s, each of which consists of one or more `node`s and/or one or more `endnode`s in any order. A `node` consists of three floating point values which represent its x, y and z position coordinates. An `endnode` is a `node` with an additional int value which specifies the integer ID of the `node` or `endnode` in a neighbouring area to which this `endnode` is connected. The integer IDs of `area`s, `node`s and `endnode`s need not be explicitly specified in the input file. The first `area` in the file is taken to be `area` 0, the second `area` 1 and so on, and similarly for `node`s and `endnode`s. IDs are defined uniquely across all `node`s and `endnode`s in the graph as one, such that a `node` may not have the same ID as another `node` or an `endnode` in any `Area`. To aid readability comments can be inserted between '/' characters in the file, for example it may be useful to add ID numbers or descriptions of areas as comments.

- The plugin additionally writes an output file for the graph. This file contains the data read from the input file supplemented with the extra data generated by the plugin in order to complete the graph: the edges between nodes within an area computed via the 'line of sight' rule for edge placement.

Using this set of tools the designer can dynamically add, subtract and move nodes and manual edges in order to debug and refine the graph structure for an environment, and can immediately observe the results and check for particular properties of parts of the graph.

The user must check that nodes within an area have been placed correctly so that there are no erroneous discontinuities in the graph. For example nodes leading up stairs and hills or round corners or obstacles must be placed carefully to maintain line of sight between neighbouring nodes. The definition of areas must also be checked, ensuring that each area is defined within a level so that there are no erroneous edges leading over ledges etc. An area is defined entirely by the nodes that belong to it. Finally it must be checked that the correct endnodes have been specified in order to create edges between areas wherever needed and avoid specifying any invalid edges that could lead an agent into trouble.

Problems such as these are now easily spotted and corrected, and combined with the simplifying nature of the hierarchical graph structure these development tools make the task of designing graphs for large, complex environments more feasible.

3.2.3 Improving the path-finding function

The path-finding function now acts on the hierarchical graph structure and returns two different kinds of path:

- The agent first requests a **high level path** which is a list of **endnodes** representing a path from area to area in the higher level graph, leading from the area containing the start node to the area containing the target node. The target node itself is also appended to the list.
- The agent then removes the first endnode from the high level path and requests a **low level path**, a list of nodes within the area graph of the agents current area leading from the agents current node to the node removed from the high level path.
- The agent repeats the low level path request each time it reaches the end of a low level path. When the high level path is found to be empty the agent has arrived at the target node.

The search algorithm itself has also been replaced. As discussed in section 2.6, the best-first search is fast and direct, but it fails to consider the accumulated cost of a path and so does not return the shortest path in all situations. This weakness is quite acceptable in fairly small environments such as that used in the first implementation, but with large and complex environments such as that used in this implementation it is magnified several fold and the results are not acceptable at all. An agent using the best first search will often follow highly obtuse paths to reach a target when to an observing human there is an obvious and much shorter optimal path. Not only does this have a negative effect on the agents performance in the game, to quote [56] 'success or failure is often a consequence of being in the right place at the right time', it makes the agent appear very unintelligent and so fail in one of its fundamental design aims.

It is apparent that a search algorithm which always returns an optimal path is required, but that a simple use of breadth-first search, bi-directional breadth-first search or Dijkstra's algorithm will not suffice - these algorithms do not run fast enough to perform multiple searches in real-time over large graphs and it is not desirable to depend on the hierarchical graph structure to reduce the search space far enough for these algorithms. The graph structure is already responsible for describing the navigability of the environment to the agent and so must be designed carefully, a major aim with this implementation has been to ease this design task and so we do not want to make it more difficult here.

The solution used was the **A-star** search algorithm. A-star is the best established algorithm for general searching of optimal paths in a graph, it combines the heuristic estimate of best-first search with a measure of the previous path length

as seen in Dijkstra's algorithm and so finds the target node directly and efficiently while still producing optimal paths. Pseudo-code for the A-star algorithm is given in figure 3.8.

A-star is guaranteed to return the shortest path to the target if the heuristic estimate used is **admissable** - the estimate of the cost from a node to the target node must never be greater than the actual cost. The Euclidean distance heuristic used in the first implementation is clearly admissable and so is re-used here[2]

3.2.4 Improving the low level movement mechanism

When developing this implementation it became apparent that improving agent navigation purely by improving the underlying navigation structure, while maintaining a highly simple path-following mechanism - following a vector from one node to the next, is not an ideal approach. Though the task of creating the navigation structure has been eased significantly, it remains a fact that a structure must be designed that covers a large and complex environment and that the designer must constantly compensate for the simplicity of the path-following function by carefully designing the structure. In a full size game, navigation structures would need to be built for a large number of complex environments and each one designed with great care.

Secondly, the agent sometimes moves independent of the graph structure, not attempting to follow a line from a node to a neighbouring node. This occurs for example when the agent is engaged in combat with its opponent, the player. The agent may move toward or away from its opponent, or may move side to side or run circles around its opponent in order to make itself a difficult target or to avoid incoming projectiles. The agent may also want to divert from following its usual path to pick up a nearby object such as a weapon or a healing item. In the environment used in the first implementation this did not present a serious problem, but in the new environment it can present a crippling weakness. The navigation node structure is designed to guide the agent around the new complexities observed in this environment - solid obstacles and traps presented by concave obstacles, ledges and drops and so on. Once the agent strays from this structure it becomes vulnerable to all of these obstacles all over again.

In order to reduce the dependency on careful graph design advancements were made to the path-following functionality. When moving from one navigation node to another, if the agents path is blocked the agent will pick a direction that is

[2]In the field of AI the A-star algorithm should technically be referred to as just A when the heuristic used is an underestimate. I have used A-star in both cases for simplicity and because in the field of game development no distinction is commonly made between the algorithms A and A-star.

```
1.a_star_pathfind(start_position, target_position){
2.  start_node = the node nearest start_position that has a direct, unobstructed
    line of sight to start_position
3.  target_node = the node nearest target_position, or the list of nodes
    nearest each position in the list of target positions
4.  push start_node onto open_queue
5.  while open_queue is not empty{
6.    pop the first node from the open_queue and call it current_node
7.    if current_node is a goal node the search is complete, construct and return
      the path
8.    for each neighbour node of current_node{
9.      neighbour.g = current_node.g + the distance from
        current_node to neighbour
10.     if neighbour already exists on open_queue with equal or lower cost skip
11.     if neighbour already exists on closed_list with equal or lower cost skip
12.     remove any occurrences of neighbour from open_queue and closed_list
13.     set the parent of neighbour to current_node
14.     neighbour.h = the Euclidean distance from neighbour
        to target_node, or the nearest node in the list of target nodes
15.     push neighbour onto open_queue
16.   }
17.   push current_node onto closed_list
18. }
19.}
```

Figure 3.8: Pseudo-code for the A-star algorithm used. Along with `start_position` the algorithm takes a parameter `target_position` which can be either a single position or a list of positions, the algorithm can search for one or multiple targets. The algorithm maintains two data structures: `open_queue` is a priority queue containing all nodes currently open for consideration by the algorithm, priority of nodes is determined by the **f** (see below) value of each node in lowest-first order, and `closed_list` is a list of all nodes that have been considered so far by the algorithm. The `open_queue` and `closed_list` must be traversed in an inner-loop denoted by lines 10, 11 and 12. For each node four important fields are maintained: the length **g** of the path from `start_node` to the node, a pointer to the parent of the node (the node before it in the path) used to construct the path when a target node is found (line 7), the heuristic estimate **h** of the cost from the node to the target node and an estimate of the total length **f** of the path to the target node going through this node computed as $\mathbf{f} = \mathbf{g} + \mathbf{h}$. In implementation the function actually accepts a pointer to the search graph itself as an additional parameter, allowing searches to be run on the high level graph or any of the low level subgraphs separately.

Figure 3.9: Left - a typical example of the first obstacle avoidance technique in action, the agent encounters an obstacle on the path to a node, moves off in a random direction and continues unobstructed. Right - a typical example of the second obstacle avoidance technique, the agents original curving path, part of a strafing maneuver, is represented in red, in blue is the adjusted path to handle the obstructing wall.

clear and move in that direction a little way, then once again move in a straight line toward its target point. This allows the agent to find its way round minor obstacles in a robust manner. Any major, large obstacles would not be passed by this technique and these must be dealt with by graph design.

The robustness of this technique is provided by the manner in which the new direction is picked - by random choice. The agent moves in a random direction away from the object and if this is not successful will try again in another direction, and in this way will simply and reliably find its way round minor obstacles.

To deal with the second difficulty, when the agent is executing maneuvers independent of the navigation node structure, a slightly different algorithm was applied. When moving in a direction, if the direction is blocked by an obstacle the agent will always move in the closest approximation to its desired direction that is not blocked, this direction being updated continuously as long as the desired direction is blocked. This allows the agent to robustly trace round obstacles while maintaining similar overall behaviour in situations such as combat with the player.

The greatest difficulty in applying both of these techniques is in determining when the agents path is blocked. It is not enough to simply wait for the agent to collide with an object. Apart from looking highly unintelligent this method will not help the agent if it walks into a hole or off a ledge, or into a 'trap' presented by an awkwardly shaped object such as a concave object.

A technique known as **environment sampling** was applied for obstacle detection. The basic idea is that the agent continuously casts a ray to a set distance ahead of itself in the direction of movement. If the ray is found not to collide with

any obstacle then the path is clear, if the raycasting function returns a collision the obstacle avoidance technique is triggered.

In practice, environment sampling is much more complicated and a number of special cases must be accounted for:

- An obstacle may easily be missed by the ray. For example perhaps the obstacle only obstructs one of the agents shoulders or is only knee-high to the agent. This is enough to block the agents path, but a ray cast forward from the centre of the agent may not find the obstacle. This situation can occur in many forms and must be accounted for by casting multiple rays from the agent. The difficulty is in deciding how many rays to cast, more rays will provide more robust detection but since they must be cast each frame will require more processing time. In this implementation, the agent uses six rays, one from each knee, one from either side of the waist and one from either side of the head.

- When approaching steep staircases or hills the rays may erroneously return collisions (with a part of the hill or staircase ahead) when in fact the agent can continue forward and climb the hill or staircase. This problem is overcome by reducing the distance ahead of the agent to which rays are cast. A shorter distance represents a higher steepness tolerance for hills and staircases, any object that exceeds the tolerance will be treated as an obstacle. Reducing the raycasting distance of course also reduces the distance at which genuine obstacles such as walls are detected and may cause some obstacles to be mistaken as passable hills or staircases, so a balanced setting must be found by trial and error.

- Holes, gaps and ledges may also present obstacles but are not objects with which a ray can collide. The agent must cast additional rays ahead of its feet in order to detect the ground, and initiate obstacle avoidance if no ground is detected. This can be difficult, since if an agent is moving down a hill or staircase it may not detect the ground but should continue anyway. To achieve this, rays must be cast a certain distance below the agents feet which represents a steepness tolerance when moving down hills and staircases, the agent will avoid any drop steeper than its tolerance. Another difficulty is presented by grated floorings which the agent can safely walk on, but which may at times not be detected by rays which pass through the grating and so erroneously initiate obstacle avoidance. To overcome this problem three different rays are cast at different distances ahead of the agent such that it is unlikely that all three rays will be erroneous. The distances are spread so that if any one (or more) of the rays returns a collision it is guaranteed that the agent can safely move forward, the maximum possible size of the gap encountered by the other rays is known to be too small for the agent to fall into.

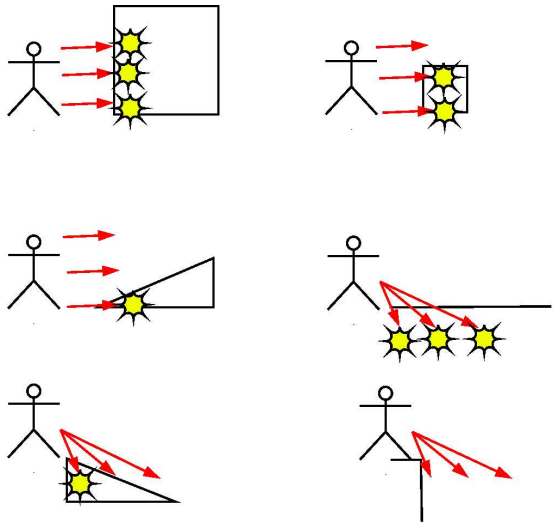


Figure 3.10: Examples of raycasting results with obstacles, passable hills and ledges.

3.3 Agent behaviour: finite state machines

The agents high level behaviour was designed and coded explicitly in a finite state machine (FSM) form (see figures 3.11 and 3.12). This gives a clear relation between code and design and is easily extensible. Figure 3.11 shows the FSM that controls the agents high-level behaviour. The `init` state of this FSM is a special case in which the agent must make a decision whether to attack its opponent or to search for an item of a particular type. Techniques for evaluating decisions such as this and the similar decision the agent must make about which of its current weapons to employ at a given time are discussed in section 4.5.3 and one approach to the technique was implemented in the agent for the third implementation.

Figure 3.12 shows the FSM which controls the agents skirmish behaviour - the movement and tactics it employs when it encounters the player. The aim was to reproduce some common behaviour exhibited by human players of similar games in a simple yet flexible and extensible manner. The agent has at its disposal a set of techniques designed to make it a difficult opponent which it applies selectively based on its `attack skill` parameter.

3.4 Using projectile weapons

The agents use of projectile weapons has been made more realistic for this implementation. Similar to the skirmish behaviour above the aim is to reproduce

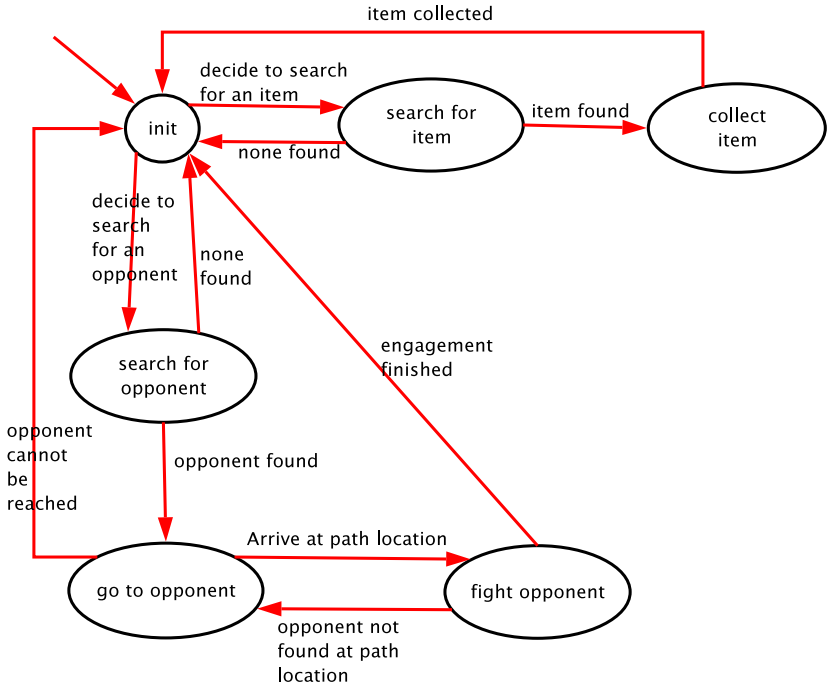


Figure 3.11: An example finite state machine for the enemy agent character.

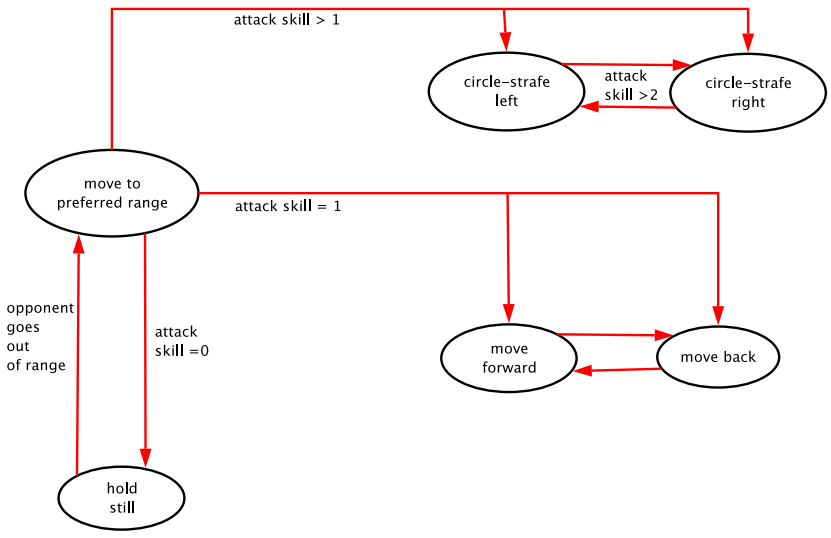


Figure 3.12: An example finite state machine for the agent skirmish behaviour.

to an acceptably convincing level the behaviour of a human player in the same situation with as simple and flexible a system as possible.

In the previous implementation the agent characters carried a 'laser gun' which they used throughout the game. In this implementation the agent collects weapons from the environment as it plays, and so can use a variety of different weapons.

For each type of weapon found in the game the agent has a preferred range which it attempts to maintain between itself and the player when using this weapon in order to maximise the effect of its weapon. For example, if carrying a shotgun the agent will attempt to close to a short range in order to inflict the most damage. When using the rocket launcher the agent will attempt to avoid close range in order avoid receiving collateral damage.

In order to improve the realism of the experience the agents aim accuracy parameter is altered depending on which weapon it chooses. The agent is less accurate with long range weapons which are difficult to aim such as the laser gun and more accurate with simple weapons such as the shotgun.

If the agent possesses more than one weapon at once then in must choose which one to employ. There are many possible ways to evaluate this choice, discussed in section 4.5.3. In this implementation a priority system is used in which the agent has a preferred ordering of weapons and always chooses the highest order weapon which it has available.

As well as the aim accuracy factor used previously the agents use of weapons is now parametrised by an additional 'aim skill' factor:

- If the aim skill is set greater than 0 the agent employs 'linear leading': based on the speed of the projectile fired by the agents current weapon, the velocity vector of the agents target and the distance from the agent to the target the agent estimates where to fire the projectile so that it connects with the target.
- If the aim skill is set greater than 0 the agent will refrain from firing the 'rocket launcher' if he is standing too close to a wall or obstacle, to avoid receiving collateral damage from the rockets explosion.
- If the aim skill is set greater than 1 the agent will adjust its aim so that it fires at the targets feet when using the 'rocket launcher'. Fired in this way, the rocket is likely to inflict damage even if it does not hit the target because it will explode on the ground nearby the target instead of flying past the target.

The aim accuracy is used similarly to the first implementation. After the agents aim has been adjusted to account for linear leading etc, the projectile is fired into a sphere centred at the agents aim position. The size of the sphere is determined

by the aim accuracy factor and the projectile is fired at a random point within the sphere. The sphere is longer horizontally than vertically, a sphere like this is used in place of the circle used in the first implementation because it produces a more realistic effect.

The agent then is configurable in terms of gameplaying difficulty via three parameters - aiming accuracy, attack skill and aim skill.

3.5 Evaluation

3.5.1 Hierarchical pathfinding

The idea of a hierarchical navigation structure and pathfinding technique has proved to be both useful and flexible.

- The hierarchical structure serves to reduce and simplify the workload of the designer. After splitting the environment into sub-areas, the designer can define smaller navigation graphs for each sub-area so that the task is clearly divided into a set of much simpler tasks, and furthermore the 'line of sight' rule for edge placement holds within subgraphs so that edges can be defined automatically - the designer only need place the nodes and join up the subgraphs after they are completed.
- By splitting the navigation data into a higher level graph and a set of lower level subgraphs the hierarchical structure successfully reduces the search space of each individual search operation and is a key component in allowing the pathfinding function to run in real time while still producing the optimal paths required for the environment.
- The hierarchical pathfinding scheme lends itself well to some possible extensions. For example, agents could be developed that respond to higher level tactics such as 'defend the base', 'hold the central area', 'hold the area around important item x' or 'attack the enemy base'. Tactics like this can be expressed in terms of the underlying navigation structure by referring to areas or subgraphs of the navigation graph. Agents may be created with particular tactics or may include a tactical module which evaluates which tactic to apply, or alternatively tactics could be passed down to agents as orders from human players or from seperate 'commander' agents.
- The major weakness of the system is that the subgraphs need to be manually defined in a particular way, so that the 'line of sight' rule for edge placement holds within each subgraph, putting extra responsibility on the designer It is advantageous for the creation of navigation data to be automated as

far as possible, because any manual input will always open up the system to error and require careful debugging, and this difficulty becomes more pronounced and time-consuming as the environments used become larger and more complex.

- It is because of the need to maintain the 'line of sight' rule within subgraphs that the hierarchical structure for an environment must be defined manually. If this necessity were removed then a hierarchy could be automatically defined in order to gain the advantage of reduced search spaces without requiring any work from the designer, as discussed in section 4.5.

3.5.2 Navigation points for agent navigation

In this second implementation the navigation point graph system was shown to be a less than ideal approach to agent navigation. The navigation point graph suffers from two fundamental weaknesses:

- Direct human input is relied on to divide the environment into areas and to place navigation nodes forming the subgraph for each area. The agent relies heavily on the graph structure to describe how it may safely move, and has no concept of its own of the shapes and connectivity of the environment. The agent is effectively blind to the environment and sees only the navigation graph, meaning that the design of the graph must be near perfect, any mistake is crippling to the agent. Very careful placing of navigation nodes and thorough debugging is necessary, and this task quickly becomes more difficult and time consuming as the size and complexity of the environment increases, as is shown by the significant increase in difficulty of designing the navigation graph for the second implementation compared with the first implementation. There is no obvious technique to automate the placing of navigation nodes of this kind.

- Although the navigation node graph encodes a lot of descriptive information about the environment (see section 2.6), its descriptiveness has shown to be weaker than necessary. The graph does not properly describe the nature of the environment, specifically shapes and volumes are not described. An agent using the navigation graph may be able to follow a path down a corridor and into a room, but the agent has no concept of the shape or size of the corridor or room.

  This shortcoming represents a fundamental lack of flexibility and robustness in the use of navigation points as the basic underlying system of agent navigation. This is of key importance when the agent wants to move in ways which do not follow the edges between nodes of the graph structure, for example to avoid an obstacle, to pick up an item, to go to a target such as

the player or to dodge left and right etc when in combat. Maneuvers like this effectively decouple the agent from the safety of the navigation graph and it must rely on secondary information provided by the environment sampling techniques developed in order to remain safe. Without the graph structure to guide the agent complexities such as arbitrarily shaped obstacles, gaps, ledges and drops become very difficult problems to overcome.

The lack of robustness of the technique presents other problems also. For example, when an agent requests a path giving an (x,y,z) start position and (x,y,z) target position the corresponding start node and target node in the graph must be determined. In the first implementation this was achieved successfully by computing the nearest node to each of the positions specified. In the more complex environment of this implementation however this technique will sometimes fail. It is possible that the nearest node to an agent may be on the far side of a wall, and the correct start node is in fact not closest to the agent. This particular situation can be handled by instead computing the nearest node to which the agent has a direct, unobstructed line of sight. Other situations cannot be dealt with this way, for example the nearest node to the agent may be out of reach on a ledge above or below the agent but still have a direct line of sight to the agent. In this case, the nearest node will be defined as in a seperate subgraph to the correct start node, but the subdivision system cannot solve this situation because there is no way to compute which area or subgraph the agent itself currently belongs to. The areas into which the environment is divided are not defined as shapes or volumes which the agents position could be checked against but as subgraphs, subsets of nodes from the navigation graph. It is not possible either to keep a constant record of the subgraph the agent currently belongs to, updating it whenever the agent following a path traverses an edge between subgraphs, because as mentioned the agent is not always coupled to the graph structure. At this point the inherent simplicity of the navigation node graph structure which was one of its major strengths in the first implementation has been lost. Rather than further extending the already over-complicated navigation node system, a technique which naturally subdivides the environment in terms of shapes and volumes is desirable.

The conclusion reached is that the system should not be extended to deal with multiple environments of equal or greater complexity than the one used, as would be the case in a real world computer game, because the workload required to design the graphs would be too large and the system is not reliable enough.

Navigation points however do have potential uses for 3D games as a supplement to another underlying structure for basic navigation. Navigation points could be placed by a designer to mark key points or objects in the environment that an

agent might want to move to, for example to mark good hiding places or positions from which to lay ambushes for the player. There is some scope also for navigation points to be dynamically created as the simulation runs, for example the player may incrementally drop navigation points as he navigates the environment. Whenever the player is in sight of a particular agent navigation points are dropped from the player that are visible to that particular agent, allowing the agent to follow the player for as far as it sees the player go, if the agent loses sight of the player then it does not know where the player moved to.

3.5.3 A-star search algorithm

The use of the A-star search algorithm for the pathfinding function has proved successful. A-star is a fast and flexible algorithm that returns optimal paths and is the current de-facto standard search algorithm for pathfinding problems in computer games. Combined with the hierarchical graph structure A-star is successfully used over both the higher-level and lower-level graphs to produce ideal paths in real time. The algorithms flexibility allows it to be extended to deal with requests involving multiple targets such as a request for a path to any item of a particular type as opposed to the previous implementation which involved only requests to particular locations. The algorithm also promises extensions and efficiency improvements to deal with more complex situations, discussed in section 4.5.

3.5.4 Obstacle avoidance

The environment sampling techniques developed did not prove to be either robust or flexible when used as part of the basic agent navigation system. Their limitation lies in the limited descriptiveness of environment sampling techniques using raycasting. A raycasting operation either returns no collision or returns a collision and can specify the distance to the collision and the type of object intersected. All parts of the static environment - walls, hills, staircases and any static objects cannot be distinguished by the object type returned by the raycasting operation, though it can distinguish a static part of the environment from a dynamic object such as another game character. The information received from a raycasting operation cannot specify the shape or size of an obstacle or be used to plan ahead of time a route around the obstacle. Further, it is difficult to develop techniques based on raycasting that deal robustly with all kinds of obstacles, as seen by the need to develop a system involving multiple rays and a seperate system to deal with gaps and ledges. Distinguishing between passable objects such as staircases and hills and genuine obstacles is also difficult to achieve robustly using the limited information provided by raycasting.

Along with the use of raycasting for obstacle detection, a pair of obstacle avoidance techniques were also used. These techniques attempt to move around obstacles while relying on only the information provided by the raycasting operations. They are therefore weaker, less robust and less realistic than any technique that looks ahead and plans a route around an obstacle for example using a search space and search algorithm.

The first obstacle avoidance technique used was intended to deal with situations in which an obstacle is detected between the agent and its next node when the agent is following a path. This was developed so that the placement of navigation nodes (and therefore the paths computed for agents) does not need to be perfected, minor obstacles can be dealt with by this avoidance technique. The technique has the advantage of simplicity and can handle most minor obstacles, but does not produce very intelligent looking behaviour and cannot handle certain obstacles. Large obstacles are not likely to be handled by the technique, or will be handled very badly. This is acceptable because large obstacles should be handled by the layout of the navigation nodes themselves (i.e.: they are handled by the search space and search algorithm). Concave shaped obstacles are not handled either, and examples of concave obstacles in the environment used can cause the agent to become trapped.

A better obstacle avoidance technique would be to trace around the obstacle: using its obstacle detection ability, the agent traces around the perimeter of the obstacle until it finds its desired path is once again clear, then continues on its path. This technique was not developed in this implementation because the obstacle avoidance technique was not intended to pass large obstacles and the obstacle detection technique was too troublesome. This obstacle tracing algorithm however is potentially useful within other navigation systems, as discussed in section 4.5.

The second obstacle avoidance technique was used when the agent is moving independent of the navigation node structure, for example when tracking an opponent or circle-strafing an opponent, or sidestepping to avoid taking damage from incoming projectiles. The technique avoids obstacles successfully and is convincingly realistic, and could be extended for use within other navigation systems, as discussed in section 4.5.

Both of these techniques are made difficult to implement and less robust by the difficulty in reliably detecting obstacles with environment sampling. Situations in which an agent avoids a safe hill or staircase or moves over a ledge which it considers to be a safe hill could not be fully removed.

4. Implementation three

The third implementation was designed to address the inherent weaknesses in the use of navigation node graph structures as seen in the previous implementation, achieving two major aims:

- The agent should be able to navigate 3D environments in general without any human-created guidance. An algorithm to generate navigation data for arbitrary environments is used.
- The navigation system should overcome the inherent descriptive limitations of the navigation node graph, encoding fully three-dimensional knowledge of shapes and volumes as well as interconnectivity.

This must also be achieved while maintaining a real-time pathfinding process such that many agents could use the system at once without reducing simulation performance.

The game for this implementation is the same as that used in the previous implementation - the player is pitted against a single agent in a competition that involves both characters navigating the environment to collect health bonuses, ammunition and weapons which they must use to defeat each other.

4.1 Sectors and portals for agent navigation

4.1.1 Overview

The navigation sub-system is now based on the concept of **sectors** and **portals**, which provide a form of **area awareness** data, describing the shapes and sizes of volumes making up the environment as well as connectivity for path-finding:

- A sector is a cuboid [60] defined according to the following rule: From any point within a sector it is possible to move unobstructed, following a straight line, to any other point within the sector. This effectively means that an agent is free to move however it pleases, forward, backward, side to side, round in circles, and is guaranteed not to be obstructed by the scenery as long as it remains within a sector.
- A portal is also a cuboid that adheres to the above rule, but is defined more strictly. Portals denote a volume which interconnects two sectors, such that an agent can move in a straight line from any point in sector A to any point

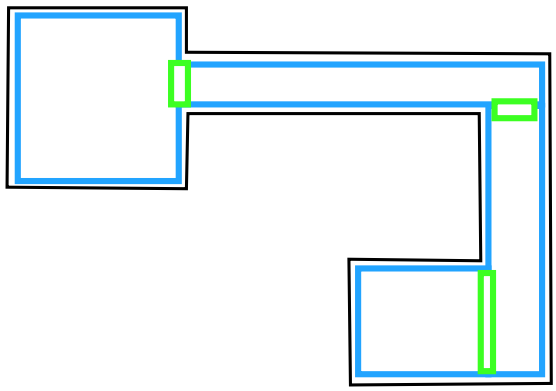


Figure 4.1: A plan view of an example set of sectors (outlined in blue) and portals (outlined in green) representing a series of rooms and corridors.

in portal AB, and from any point in portal AB to any point in sector B, without being obstructed.

Intuitively, sectors can be thought of as defining the volumes of cuboid rooms and corridors, and portals defining the doorways that interconnect them.

In practice the sectors and portals need not necessarily correspond directly to intuitive notions such as rooms and corridors. The system can describe through combinations of multiple sectors and portals general areas including arbitrary (non-cuboid) shaped rooms, corridors and doorways interconnected in any way, and even outdoor environments that are not divided into rooms and corridors at all.

The set of sectors and portals describing an environment also define a search space, on which a graph searching algorithm such as A-star used previously can be run in order to provide path-finding functionality. Sectors form the nodes of a graph, and portals form the edges interconnecting the nodes. To compute a path to an arbitrary location, an agent must determine which sector its current position lies in and which sector the target position lies in then apply a search algorithm to compute a path, defined as an ordered list of portals, from the start sector to the target sector. Since the sectors are cuboids, an agents (x,y,z) position parameter can easily be compared against each sector to determine which sector the agent is in.

The agent can then navigate from one point to another following a simple algorithm - remove the first portal from the path list, turn to face the portal, move forward. On arriving at the portal, remove the next portal from the task list, turn to face the portal, move forward. The sequence is repeated until the final portal is arrived at, at which point the agent can move directly to its target location.

4.1.2 Auto-generation of sector and portal data

A process was written capable of automatically generating sectors and portals to describe an arbitrary environment. This process was derived from [54] which gives a general outline of the technique. The process is large and complex, and for large environments can require a lot of memory and take a long time to complete. This is acceptable however, because it is a once only pre-processing step. Once the data has been generated for an environment it is compacted and written to file and can be easily read in from file and stored in memory in future runs of the simulation. An XML-style file format similar to that in the previous implementation is used. It is necessary however for the data structures representing cells (see the next section), sectors and portals to be as small as possible because a large number will be needed in memory at once. When developing a game, the data would be pre-computed for each environment in the game and distributed in a file along with the game itself. The process is split into four major stages:

4.1.2.1 Stage one

The first stage of the algorithm is based on the notion of a **flood-fill** [61] (also called a **seed fill**): The flood fill is a recursive algorithm which determines connectivity between elements in an array often used by the familiar 'bucket fill' tool in bitmap image editing applications.

The algorithm takes three parameters: **start element**, **source symbol** and **destination symbol** and changes every element in the array that is connected to the start element by elements containing the source symbol to the destination symbol.

In the bucket fill example for a bitmap editing application the start element is a pixel clicked on by the user, the source symbol is the colour at this pixel and the destination symbol is another colour. The area surrounding the start element is 'filled in' with the destination colour for as far as the source colour extends.

The algorithm proceeds as follows:

```
1.Fill the start element with the destination colour and push it on the task list.
2.While the task list is not empty{
3.  Remove the first element from the task list and call it the current element.
4.  For each neighbouring element to the north, east, south and west of the
    current element in turn{
5.    Ifthe element contains the source colour{
6.      Fill the element with the destination colour.
7.      Ifthe element is not already in either the task list or processed list:
8.        Add the element to the task list.
```

```
9.    }
10. }
11. Add the current element to the processed list.
12.}
```

To generate navigation data the array is replaced with a continuous 3D environment. An element is defined by the notion of a **cell**: a cell is a cuboid defined by the **bounding box** of the agent characters polygon mesh. The start element is the cell placed at the starting position of the agent. The source symbol test in line 5 is replaced by a notion of reachability derived by emulating the in-game movement of the agent character described below. The cell is the destination symbol. The algorithm proceeds as follows:

```
1.Add the start cell to the task list.
2.While the task list is not empty{
3.  Remove the first cell from the task list and call it the current cell.
4.  For each cardinal direction north, east, south and west in turn{
5.    Simulate the agent moving one cells width in the direction.
6.    Ifthe move is completed without the agent colliding with anything{
7.      Create a new cell at the agents new position.
8.      Ifthe new cell does not already exist in either the task list or the
        processed list:
9.        Record the one-way link from the current cell to the new cell and add
          the new cell to the task list.
10.     else
11.       Record the one-way link from the current cell to the already existent
          cell and discard the new cell.
12.   }
13. }
14. Add the current cell to the processed list.
15.}
```

For the collision test in line 6 the agent movement and bounding box collision physics used when running the actual simulation must be emulated. This is the reason for the use of the agents bounding box as the cell object. A 'dummy' agent is created and manipulated in order to perform the necessary north, east, south and west steps to determine where cells should be placed. The test is then passed by any move which the agent could make unobstructed in the actual simulation: walking across flat ground, walking up or down a hill or staircase etc, and is failed by any obstructed move: walking into a wall or obstacle. A number of special cases must also be defined, depending on the simulation and the agent character. For example, any move that causes the agent to walk off a ledge could be considered invalid, or alternatively it could be considered valid so long as the agent falls onto safe ground and does not fall far enough to receive significant

harm. This second possibility will result in an agent willing to jump off a ledge in order to reach a location as long as it judges the fall not to be harmful. A move that takes the agent into a pool of water may be considered valid, or if the agent is afraid of water or cannot swim it may be considered invalid.

When the flood fill completes the result is a set of cells and connections between them that fully describe the reachability relations over the environment for this particular agent (see figures 4.2 and 4.3). This set of cells defines a search space with the cells as nodes and the connections recorded between cells as edges. The search space can be thought of as a grid - the same structure used for pathfinding in simpler 2D environments, but because the simulations movement and collision physics are used the grid will go up and down hills or staircases, around corners and obstacles, over bridges or deal with any other feature that the in-game character can traverse, and equally the grid will not traverse any areas that the agent cannot traverse and there will be no connectivity between parts of the grid that cannot be walked between such as a lower area and a higher, raised platform, or two areas separated by a large gap. A search algorithm could be run over the grid of cells in order to compute a path in the form of a list of cells which the agent could follow by moving in north, east, south and west steps between neighbouring cells.

For a different type of agent the algorithm will need to be re-run using the different agents bounding box for the cell and the different agents movement and collision physics. A small agent perhaps representing a rodent would have a different search space to an agent representing say a large dragon, so that small agents would follow paths leading through small gaps for example while larger agents would only attempt to follow paths through which their bounding box could fit.

The cells however cannot realistically be used directly as the search space for a real-time game. Even for a small environment, the number of cells produced will be large, and a large game environment may hold thousands or millions of cells. The sheer volume of cells produced can be seen in figures 4.2 and 4.3. The data is far too large to hold in memory as the game runs, and far too large to run search algorithms over in real time.

4.1.2.2 Stage two

The cell data is a very inefficient representation of the environment. A large cuboid volume may be represented by a very wide and long array of connected cells. This same space could be represented by a single sector that is the size of many cells. The task in this stage is to find connected groups of cells and combine them into sectors.

The algorithm proceeds as follows:

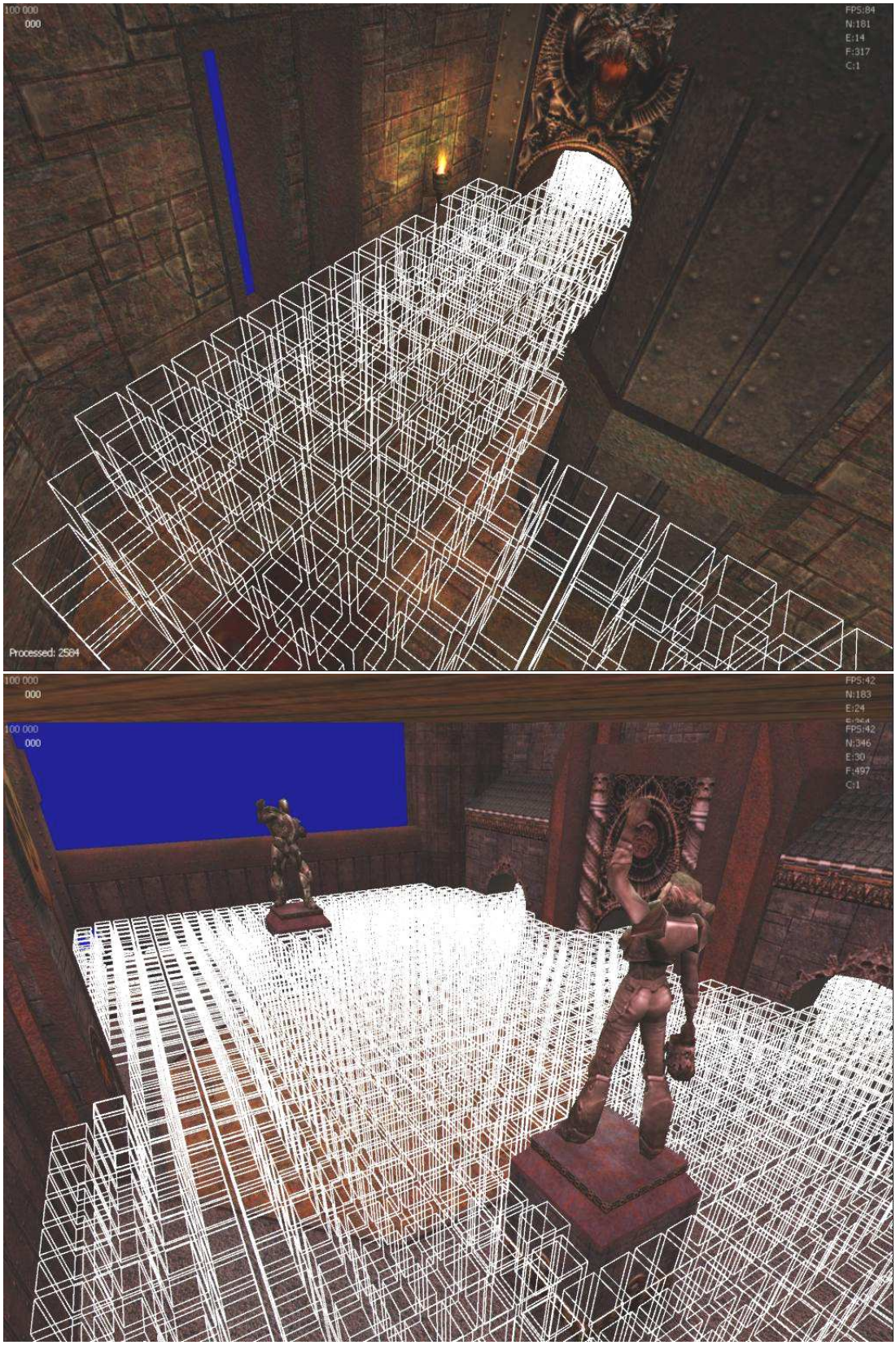


Figure 4.2: Example screenshots showing stage one area awareness data for the environment used in the first implementation.

Figure 4.3: Above - partially generated stage one area awareness data for a very complex environment. Below - stage one area awareness data for the environment used in the second implementation.

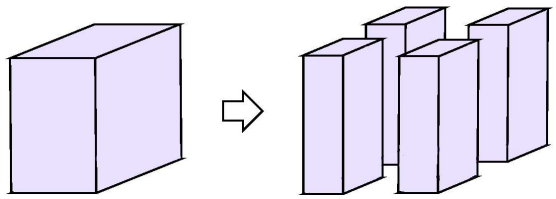


Figure 4.4: Because during the flood fill the cells remain always at the same orientation relative to the world and move only in north, south, east and west steps there are some volumes, perhaps a doorway for example, that an agent character could fit through by rotating itself but that the flood fill would not move through. To account for this the cells used are actually constructed by splitting the agents bounding box into four, as depicted. A cell has the same height as the agents bounding box but has half the width and depth.

Pick a cell from the processed list (which contains all cells created by the flood fill of the environment) and add it to a new sector. A sector at this stage is represented by a two-dimensional array of cells. The cell is not removed from the processed list but is marked as taken by a sector.

Expand the sector - the sector is grown as far as possible in all directions by adding to it neighbouring cells of its current cells. Every cell added to the sector is marked as taken by a sector. A sector of maximal size must be produced while adhering to the defining rule of a sector - that an agent can move unobstructed, in a straight line, from any point in the sector to any other point in the sector. This means that the sector must be of cuboid shape and that every new cell added to the sector must have a **two-way** link to a neighbouring cell that is already in the sector. Clearly, this neighbouring cell must be on one of the current edges of the sector in order to have a link to a cell outside the sector.

When completed the sector is added to the list of sectors and a new, non-marked cell is picked from the processed list to start a new sector.

The algorithm terminates once every cell in the processed list belongs to a sector.

Although a cell already belonging to a sector will never be used to start a new sector, it may be added to a new sector as the sector expands - the sectors produced in this stage may overlap (see figure 4.6). The next stage of the algorithm is concerned with resolving these overlaps in order to represent the environment with as few sectors as possible. The stage two algorithm can be altered so that marked cells are not added to an expanding sector, in which case stage three is not necessary, but the set of sectors produced is not likely to be very efficient.

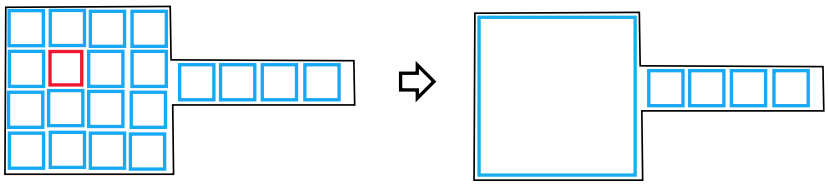


Figure 4.5: An example showing the creation of a sector. On the left, the red cell is picked to start a new sector, the sector is expanded as far as possible in all directions while maintaining a cuboid shape resulting in the completed sector show on the right.

4.1.2.3 Stage three

In this stage overlaps between sectors are located and the smaller of the sectors is deleted. A new sector is then created starting from one of the newly freed cells due to the deletion of a sector. This new sector expands as far as possible as in stage two but with the additional constraint that no cells already belonging to a sector may be added to the new sector. In some cases more than one new sector may need to be created in order to re-use all the newly freed cells.

Once all overlaps have been resolved and the final set of sectors has been defined most of the cells created by stage one can be discarded. A sector is first re-defined as the bounding volume computed from all points of all the cells it contains, then all its **internal** cells are deleted. The sectors **external** cells, those located on one of the four edges of the sector, must be maintained for now. The nodes of the navigation graph have now been identified.

4.1.2.4 Stage four

The final stage of the process is concerned with identifying the edges of the navigation graph - the portals which interconnect sectors. The remaining cells, the edge cells of the sectors, are the key to this algorithm.

The algorithm proceeds by identifying contiguous groups of cells on a particular edge of a particular sector which all have a link to cells belonging to the same other sector. The links can be one way or two way, and will define a one-way or two-way portal accordingly. When a group of cells like this is identified the cells along with their neighbours in the other sector are added to a new portal belonging to the source sector and leading to the destination sector. If the links are two-way a corresponding portal is defined in the destination sector leading to the source sector. A portal then is defined as an n-by-2 array of cells.

Once all portals have been identified each can be re-defined as the bounding volume computed from all the points of all the cells it contains, the remaining

Figure 4.6: Above - stage two area awareness data for the environment used in implementation one. Below - stage three area awareness data for another environment.

cells can then be discarded.

The sectors and portals created now define a graph that fully describes the environment for navigation purposes and is of an order of size suitable for real-time path finding (see figure 4.7, and appendix A which contains images of completed area awareness data for several environments along with the sector and portal counts giving the size of the search space).

A final detail is that any sector which has no portals leading out of it must be eliminated. It is possible for sectors arrived at by a one-way portal which have no exit portal to appear, the simplest example is a hole that can be dropped into but not escaped from. Of course, an agent will not usually compute a path that goes through this sector, but it is possible that the agent will move into such a sector to retrieve an item or arrive at a target such as a human player that has ended up in the sector. These sectors are removed to prevent this.

4.2 Path finding

Pathfinding in this implementation uses the same A-star algorithm with the Euclidean distance heuristic as in the second implementation. The size of the search space produced by the sectors and portals is larger but of the same order as the search space in the second implementation. In place of the hierarchical search space in the second implementation, this implementation uses **time-slice pathfinding** in order to run searches over the entire search space without affecting the frame rate of the simulation.

The state of an A-star search is represented entirely by the open list, the closed list and the target sector or list of target sectors. This state can be saved and the search resumed at a later time. In this way, a search is spread across multiple frames of simulation controlled by a parameter which determines the maximum number of nodes of the graph that can be expanded in each frame. The frame rate will not be affected by the pathfinding algorithm, but an agent may have to wait an undetermined number of frames for the result of a path request. In practice, this does not present a problem. Although the search algorithm could not service multiple searches over a large graph in real-time, it is none the less quite efficient and will return searches within a second or so at the very most when using time-slices. The worst effect that is seen is an agent pausing for a very brief moment before continuing onto a new path, and since the agents controlling functions, animation and physics are still running this does not look unusual or cause the agent to disregard any events that occur while waiting for a search to complete. The effect could actually be considered quite positive in a game environment, since it effectively corresponds to the agent stopping to think

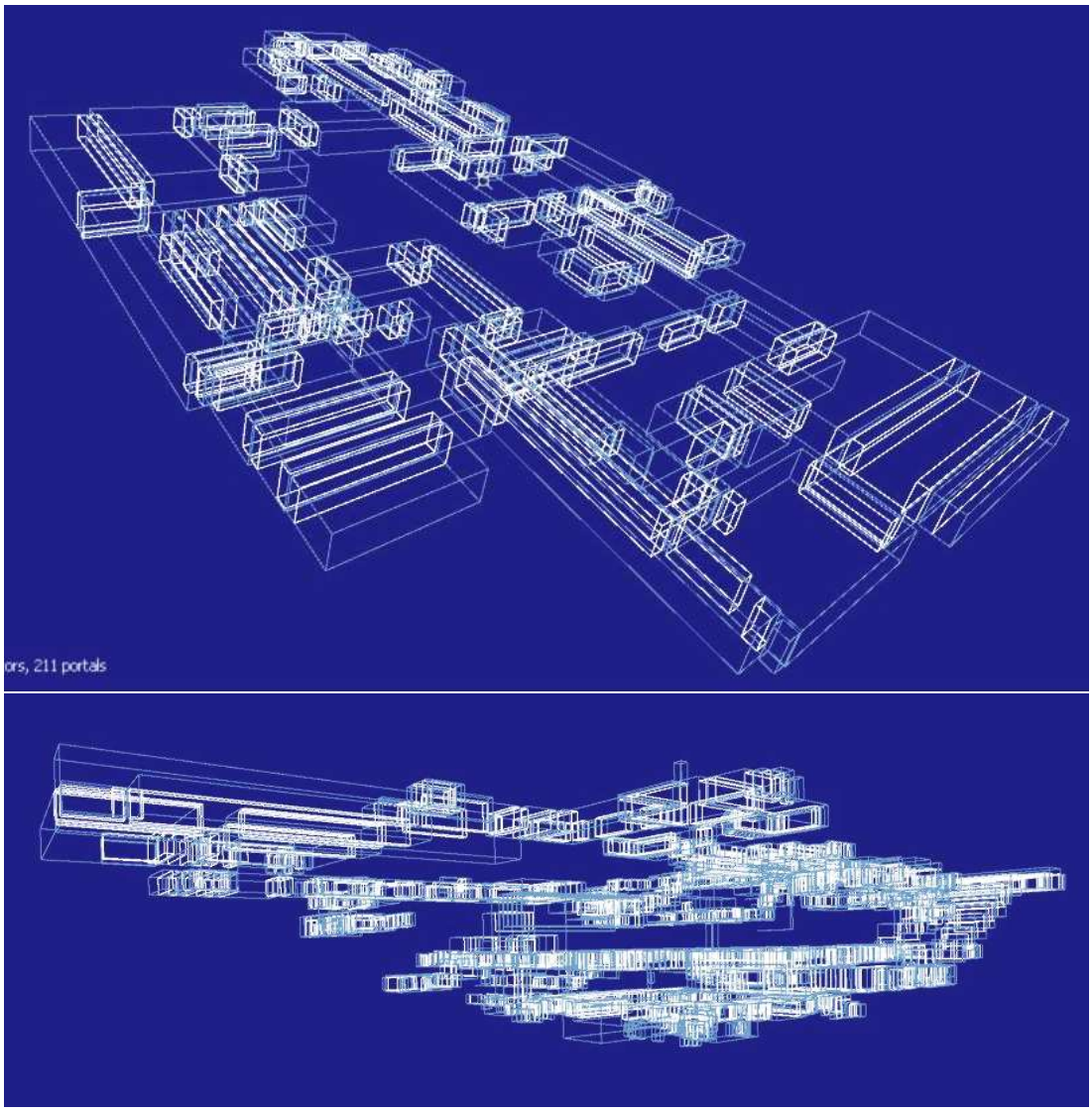


Figure 4.7: Screenshots showing final area awareness data dealing with two complex environments and showing the 3D nature of the data. The actual BSP environment is not drawn in order to show the sectors and portals clearly. Sectors are outlined in blue, portals in white.

about where it is going, a behaviour often seen in human players. Pauses such as this are commonly programmed explicitly into computer game characters in order to make them appear more human and not unrealistically fast at particular tasks.

4.3 Agent design

The agent in this implementation is controlled by a simple needs-based mechanism:

- If the agent is low on health then it searches for healing items, this rule has highest priority so that if this rule applies it will always be followed regardless of which other rules apply.
- If the agent is low on ammunition then it searches for ammunition with second highest priority, this rule can only be overridden by the health rule above.
- If the agent sees the player then it will attack the player with third highest priority.
- If no other rule applies the agent will roam around collecting items of any kind, this is the lowest priority rule.

This system acts as a straight-forward demonstration of the needs-based evaluation concept which could be extended to provide a much more complex behaviour.

4.4 Agent sensory system

For this implementation the realism of the agent was extended with a simulated vision system. The agents attention is drawn to items that enter its field of vision, and in particular the agent will attack if it sees the player. The agent will not see anything outside of its field of vision, so if the player can navigate herself to a position behind the agent without being spotted she can gain an advantage.

The agents visual system is simulated by three simple tests which an object must pass in turn in order to be defined as visible:

Range A circle is defined centred at the position of the agent with the agents range of vision parameter as its radius. Any visible object must be within this circle. This is tested by checking that the Euclidean distance between the agent and the object is less than or equal to the range of vision parameter.

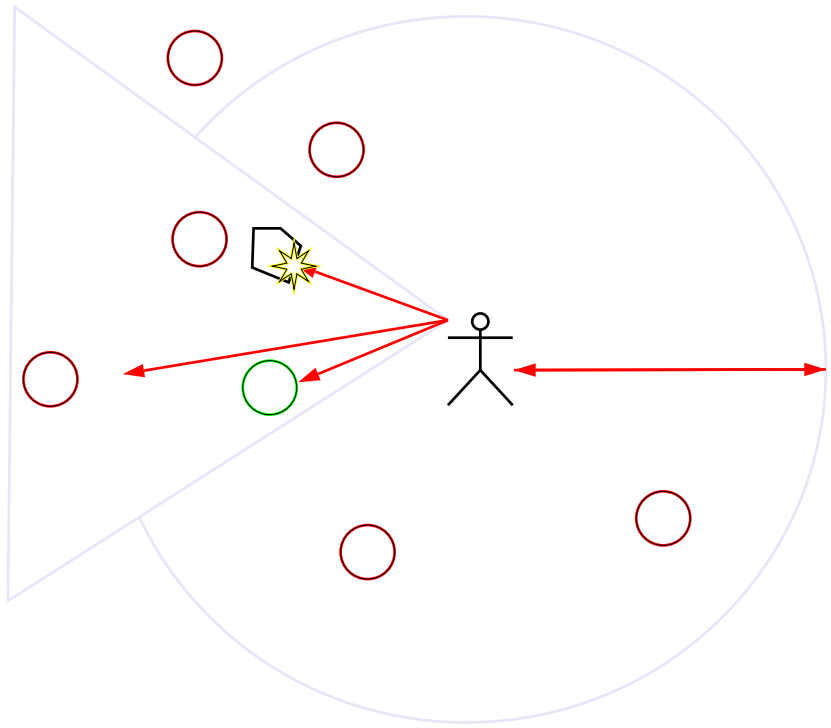


Figure 4.8: The agents synthetic vision system. The green circle represents an object that is in sight, the red circles represent objects that are out of sight either because they are out of range, out of the view frustum or blocked from view by an obstacle.

Field of view A frustum is defined emanating from the agents eyes which represents the agents field of view. Any object that is visible must be within this frustum. This is tested for by clipping the objects bounding box against the frustum.

Line of sight Finally, for an object to be visible there must be a direct, unobstructed line of sight between the agent and the object. This is tested by casting a ray from the agent to the object and determining whether the ray collides with any other object before reaching the target object.

The ordering of the tests is defined intentionally simplest first in order to reduce the workload of determining which objects of interest are visible in each frame of simulation.

4.5 Evaluation and Directions for Further Work

4.5.1 Agent navigation

The sector and portal system for agent navigation has proved a robust and flexible system which avoids all of the difficulties encountered when using the previous navigation node system with complex environments. The start and target sectors of a path request can easily be computed in all cases and the search space can be relied upon to avoid all obstacles without the need for careful work by a human designer.

The systems superiority to the navigation node structure is due to two features:

- Because a sector or portal is a cuboid volume within the environment, as opposed to a navigation point which is simply a point on the environment and and edge which is simply a connection recorded between two points, the sector and portal system provides more information about the environment.
- Sectors and portals can be generated automatically for any environment.

An agent can navigate the environment in a robust and flexible way using sectors and portals much more easily than an agent using navigation points, and the sectors and portals can further be used to tell the agent which moves are safe to make when not following a path. If the agent for example is strafing left and right or dodging an incoming projectile, it is known that moving in a given direction is valid provided the move does not take the agent through a sector wall. If the move goes between sectors it must do so via a portal. The second obstacle avoidance technique used in the previous implementation when the agent was decoupled from the navigation graph (see section 3.2.4) can be used here, with the environment sampling technique for obstacle detection replaced by a simple check of whether the agents current velocity will likely take it through a sector wall within the next frame.

There was some difficulty in emulating the movement physics of the agent for the first stage of the generation process. The movement functions used by the agent character in this implementation are extended from those of the Fly3D standard plugin 'walk', which was not designed for the type of use required by the generation process. It is necessary for the first stage of the algorithm to move the agent in a given direction instantaneously (in a single frame of simulation) across a distance so that the agents bounding box at its new position is close to but not intersecting the bounding box at the original position, and return a value indicating whether the move was valid or a collision occurred. The emulation technique used has a tendency to return false collisions when steep hills or stairs are encountered, so that the flood fill will not traverse the stairs or hill. A pair

of parameters described in appendix E can be tweaked to control the 'steepness tolerance' of the algorithm, so that if a particular hill for example is not traversed the tolerance can be increased and the algorithm re-run until the hill is traversed. Unfortunately particularly large, steep hills or staircases can demand too high a tolerance so that by the time the parameters have been increased so that the flood fill will traverse the hill or staircase they are so high that non-traversable obstacles such as boulders or raised ledges are leapt onto in a single bound by the flood fill, producing invalid moves. This implementation difficulty is a result of attempting to adapt the movement functions provided by Fly3D to a purpose for which they were not intended. The solution to the problem is to rewrite the movement functions with the generation process in mind, a detail which there was not time to attend to. A similar technical difficulty occurs when the flood fill moves over a ledge and a fall must be simulated, in this case the algorithm is caused to temporarily run very slowly but will attain correct results.

The navigation system has much potential for extension by further work, some of the possibilities are described by the following subsections.

4.5.1.1 Hierarchical navigation

A hierarchical navigation structure could be added in order to reduce the search space for individual search operations, which would reduce the time required for each path request and help to extend the system to deal with larger environments containing many agent characters while still maintaining fast pathfinding performance. A process can split the graph into subgraphs such that each subgraph has a roughly equal number of edges and/or nodes, and the ideal number of subgraphs can be computed which balances the size of the higher-level graph with the average size of lower-level graphs. The subgraphs do not need to correspond to any subjective subdivision of the environment which a human designer might specify, such as rooms and hallways, indoors and outdoors and so on. If a hierarchical structure was used in this way as opposed to using a hierarchical structure defined by a human designer then there is perhaps less potential for the use of the structure in a system for agent tactics, since the subgraphs will no longer correspond to areas such as bases, but there is still some potential. For example a higher level tactical function or agent may wish to give movement orders to lower level agents. The tactical function could carry out a search on the higher-level navigation structure in order to see which areas an agent can reach and issue orders, but leave the particulars of lower level pathfinding up to the individual agents themselves.

4.5.1.2 Path queuing and time-slice pathfinding

Path queuing - In a game environment involving a large number of computer controlled agents the added strain due to multiple simultaneous or overlapping path requests can be removed by path queuing. With this technique, a global queue of path requests specifying start and target position is maintained. When it needs a new path an agent places a request on the queue. The pathfinding function services one request per frame from the queue as long as the queue is non-empty, and each agent must wait to receive the result of its request. The disadvantage of this technique then is that an agent may have to wait an unspecified number of frames to receive a path. The agent itself can be written so that its controlling modules, physics and animation and so on are still active while it waits for a path, having the effect of the agent appearing to stop and think before following a path but to continue interacting with the environment. If an event occurs that demands it the agent may abandon its path while still awaiting the result and pursue another course of action, removing the unserviced request from the queue. If a new path is required the agent could be allowed to swap its request for another rather than having to go to the back of the queue again, which would avoid potentially problematic extended waits occurring if the agents situation changes. A further disadvantage is that the number of frames that an agent would have to wait to receive the results of a request would be determined by the number of path requests from other agents already in the queue, which from the point of view of each individual agent is quite an arbitrary factor. A request for a large path for example is no more likely to suffer an extended wait than a request for a very short path. It might look strange to see an agent apparently stopping to think for some time before following a very simple route. There is no obvious correct solution to this potential problem, since it is not possible to tell before hand how complex a path between two points will be. The distance between the two points, however, is a reasonable heuristic and so path requests could be serviced shortest-first. Extra mechanisms would then be necessitated to ensure that long path requests are not starved by many incoming short requests. Ordered queuing I would predict is most likely an unnecessary over-complication, since the negative effect of varied queuing times is not likely to be great enough to justify a complex queuing mechanism. Time-slice pathfinding below provides a more realistic solution to the problem of multiple simultaneous path requests.

Time-slice pathfinding - The basic time-slice pathfinding technique used in this implementation succeeds at allowing the pathfinding function to run in a real-time simulation. For a more complex simulation with larger environments and many agents the time-slice technique can be extended to dynamically handle multiple simultaneous or overlapping path requests without increasing the per-frame processing requirement for pathfinding. The simplest way to control the time allowed for pathfinding each frame is to allow a set number of nodes to be

considered by the search algorithm each frame before the state must be saved for continuation in the next frame, as implemented here. A more accurate solution might be to measure the time taken and continue considering nodes until a time limit is reached. In an environment containing many agents there may be several outstanding search requests at once. In this situation the available search time for a frame could be divided equally among the currently outstanding searches, so that many searches can be processed simultaneously with a fixed per-frame processor usage. If there are many simultaneous searches then each will take longer to complete, but if there are few or only one outstanding search request at a given time the full processing time available will still be utilised so the search will complete sooner. As a general rule, requests for long and complex paths will still take longer to complete than short requests, but the completion time of a request is also effected by the number of currently outstanding requests.

A minimum amount of processor time per request should be guaranteed in order to prevent the system becoming overloaded. To achieve this a maximum limit must be applied to the number of requests that can be processed simultaneously. Once this limit is reached any incoming requests must be queued and a request can be dequeued for processing whenever a request is completed.

4.5.1.3 Handling additional environment features

[54] describes a technique to extend the functionality of the sectors and portals system to deal with some additional environment features. Special mechanisms can be added to deal with features such as closed doors which can be opened, elevators or ladders. The normal flood fill would not traverse these kind of features, because they are not traversed by simply walking around, they involve special activities such as climbing, pressing a button (for example) to open a door or operating an elevator. These features could be handled as 'teleports' within the navigation graph. Before running the generation process a designer specifies special 3D volumes called 'entry zones' and 'exit zones' for each teleport mechanism in the environment. During the flood fill stage of the generation process when a cell is removed from the task list it is checked against all entry zones and if it is contained entirely within the entry zone of a teleport mechanism then a new cell is created in the corresponding exit zone and added to the task list. This ensures the the flood fill continues on the other side of the teleport mechanism (the other side may be unreachable except via the teleport mechanism itself), note that a link between the cell in the entry zone and the cell in the exit zone is not recorded.

After the generation process has completed the entry and exit zones of teleport mechanisms must be linked to the sector data. Intersections must be found between the entry and exit zones and a pair of sectors, and if intersections are found

then a portal can be added between these two sectors defined by the bounding boxes of the intersections of the sectors with the entry and exit zones. Specific information about the teleport mechanism is stored with the portal, for example to tell an agent it must open the door, climb the ladder or operate the elevator.

This portal mechanism shows promise for extension to all sorts of devices, not just simple doors, ladders and elevators. Information about a particular key required to open a door could be stored with the portal, so that an agent not holding the correct key will not attempt to operate the door for example. Other mechanisms might include futuristic teleportation devices, or large leaps or other maneuvers required for example to traverse a chasm, information about the direction, timing and speed of the maneuver would be stored with the portal. Teleport maneuvers could be created dynamically as a game is played, allowing an agent for example to replicate a heroic leap that it observes a human player executing in order to follow the player. Information about the speed, timing and direction of the players jump would be recorded by the agent and added to a dynamic portal between the sector from which the player leapt to the sector in which the player landed.

4.5.1.4 More natural paths

The agent follows paths by moving in straight lines between portals and executing sudden turns each time it arrives at a portal. Similar to the technique used with navigation points this path following technique produces somewhat inhuman, robot-like movement. Human players do not always move in straight lines and do not execute instantaneous turns, instead they will tend to execute turns over time while still moving forward so following smooth, curved paths. Techniques are available to smooth the path of the agent characters and produce more realistic movement.

[54] discusses a technique using bezier curves which is linked to the sector and portal navigation system. The idea takes advantage of the convex hull property of a bezier curve - any point on the curve is know to lie within the convex hull of the four control points of the curve. Once a path has been computed a post-process step is carried out which involves building a bezier curve from the points of the path and ensuring that every control point of the bezier curve lies within the set of sectors traversed by the path. An agent can then follow a smooth bezier curve while still remaining within the navigation sectors.

4.5.1.5 Pathfinding for entities with more complex movement restrictions

Person characters such as the agents implemented in this project have movement restrictions in the sense that they are rooted to the ground, they can move up stair or hills but cannot move up large ledges and cannot maneuver freely in complete 3D as a flying object. On the other hand, characters such as these can start or stop more or less on the spot, can change direction instantaneously without appearing too unrealistic and if greater realism is needed can still perform arbitrarily tight turns by stopping and turning on the spot. In a complex environment it may be required to provide pathfinding and following techniques for entities that have more complex movement restrictions, a car for example has a limited turn radius and takes time to accelerate or decelerate, and these restrictions change depending on the current speed of the car, what kind of car it is, what kind of surface it is moving on and perhaps many other factors depending on how realistic a simulation is required. Pathfinding for vehicles such as this is a significantly more difficult problem than pathfinding for person characters, as a path must be computed that can be successfully followed with the movement restrictions of the particular vehicle that has requested the path. For example paths must not demand turns which are too tight for the vehicle to make, and vehicle orientation must be taken into account - a vehicle entering a sector at a given orientation may be able to make a turn in time to pass through a particular portal, but if entering the sector at another orientation the same portal might not be reachable. [54] suggests an initial direction for a solution to this problem that involves determining the reachable and non-reachable portals of a sector during the pathfinding search and forming spline curves from a path that obey a vehicles movement restrictions in order to tell the AI how to orient the vehicle so that it will not find itself needing to make an impossible turn. [57] discusses several other approaches to computing realistic curved paths which obey complex movement restrictions via enhancements to the A-star algorithm.

4.5.1.6 Dynamic obstacles

The obstacle avoidance techniques applied in the second implementation have potential use within the sector and portal navigation system, to avoid dynamic obstacles which are not part of the scenery and are not described by the sectors and portals themselves. These obstacles would most often be other agents or game characters in the scene. The environment sampling system would likely extend well to detecting these obstacles, and would not encounter the previous difficulties with detecting gaps, hills and staircases and so on. On obstacle detection an agent could instigate the obstacle tracing mechanism described in the second implementation before continuing on its path. There are complications however.

Tracing would have to be executed while remaining within the sectors, and this may in the worst case mean there is not room to trace around the obstacle. Some communication between dynamic entities would be necessary, such that one entity (the smaller of the two or randomly chosen if they are the same size) remains still and becomes the obstacle, the other entity executes the tracing. When there is not room for tracing the entities must agree to back up and avoid each other.

An alternative solution is to decompose a sector into equi-sized cells when a collision is detected and use the pathfinding algorithm to plan a route through these cells around the obstacle. This lends itself well to determining when the entities need to back up as the search function will fail if there is no route around the obstacle.

4.5.2 A-star algorithm

The A-star algorithm was successfully converted for use with the new navigation system and extended to handle time-slice pathfinding in place of the hierarchical pathfinding in the second implementation, in order to allow pathfinding to run in a real-time simulation. The algorithm again solves the searching problem very well and holds potential for further improvement in terms of efficiency and flexibility.

The efficiency of the A-star algorithm itself can be greatly improved over the straight-forward implementation used here. [57] discusses a technique which removes all memory allocation and list insertions from the operation of the A-star algorithm by replacing the Closed list and Open priority queue with a fixed-size two-dimensional array. Elements of the array represent nodes in the search space, and each stores:

- The cost from the start node to this node.
- The current best total cost through this node to the goal node, computed as the sum of the current best cost from the start node to this node and a heuristic estimate of the distance from this node to the target node.
- The [x][y] location in the array of the parent node of this node - the node before this node in the current path from the start node to this node.
- A boolean value denoting whether this node is currently in the open list or not.
- The [x][y] locations in the array of the previous and next nodes in the open list.

The size of the array is determined by the size of the search space, which if hierarchical pathfinding is used can be kept within limits even if the environment

is very large. The memory for the array is allocated at the start of the simulation and reserved throughout the simulation. With a 60x60 array for example, given that each array element requires approximately 16 bytes, a total of merely 57 kilobytes will have to be reserved for the array. Using this array structure for A-star search requires time in the order of n (where n is the number of nodes in the search space) to find the lowest-cost node at the top of the A-star loop, as opposed to order of log n if a priority queue is used as in the traditional implementation. The array structure however requires only constant time to insert and delete elements, which happens often, and the inner-loop of the A-star algorithm in which it must be checked whether neighbouring nodes already exist in the Open queue or Closed list is completely eliminated. In the traditional implementation this inner loop takes time in the order of n. Overall, [57] claims that this method is up to 40 times faster than the standard A-star implementation.

The efficiency of A-star search increases with the accuracy of the heuristic estimate. The Euclidean distance heuristic used in this implementation is likely to underestimate the cost and so the search is not as efficient as it could be if a more accurate heuristic were employed. The Euclidean distance heuristic however has the advantage that it is very simple to compute dynamically and does not require any additional storage space, making it an attractive choice.

The A-star algorithm is highly flexible and many variations on its implementation are possible:

- The algorithm can handle **weighted terrain**. In the current implementation the cost of moving to a node is represented by the Euclidean distance between the node and the current node. In a more complex simulation, it may be desirable for some forms of terrain, such as a swamp, to be more costly to travel over than others such as a road. A simple extension to A-star can alter the cost of traversing a sector depending on the type of terrain it contains, so that agents will tend to use the easier terrain unless it is genuinely faster to shortcut through tough terrain. This has many applications in computer game environments, for simple terrain if varying traveling difficulty such as roads and swamps, flat ground and hills, to more advanced notions. For example a person character could be programmed to consider pavements cheaper to traverse than roads, meaning people will tend to stay on the pavements and only cross roads if necessary. For cars roads would be cheaper than pavements to traverse. The relative costs of pavement sectors and road sectors would be altered until a good balance is found, and there is even scope to dynamically alter the balance of costs depending on the game state and the state of the AI character. A pedestrian fleeing from some form of danger for example or a car involved in a wild, high-speed chase may pay less attention to sticking to the correct terrain.
- There are many possible ways to adapt dynamic terrain weighting to dif-

ferent games. For example in a combat game a unit might tend to avoid paths that take it near groups of opposing units or near an enemy base. If a lot of agents are destroyed in a particular sector, perhaps it represents a good bottleneck area for enemy ambushes, this sector could become more costly to traverse so that agents will tend to avoid it. It may be desirable for agents to tend toward sectors which the player often passes through or is often seen passing through by agents, or it may be desirable for agents to tend to avoid these sectors. Sectors containing valuable items to collect may be considered cheaper to traverse, so that agents will tend toward paths which allow them to grab items along the way.

- The heuristic estimate used for A-star can be modified. If the heuristic estimate always underestimates the remaining cost, as in my implementations, the A-star algorithm is guaranteed to find an optimal path but the greater the underestimates that are made the more nodes will be expanded by A-star during its search. If the heuristic estimate is always completely accurate then A-star will expand the fewest nodes possible while still finding an optimal path. If the heuristic is an overestimate then an optimal path is not guaranteed, but the algorithm may complete more quickly. This trade-off between speed and accuracy can be exploited in situations in which an optimal path is not desired, or an optimal path is not needed but a fast computation time is needed. An initial minimum cost heuristic such as the Euclidean distance or Euclidean distance with varying, perhaps dynamic terrain costs can be taken and scaled by a dynamically controlled factor to alter the behaviour of the A-star algorithm. Depending on the particular game involved there is some scope to dynamically alter the accuracy of the heuristic based on processor speed, the current level of demand for pathfinding solutions, the difficulty level of the game or more complex factors. For example a soldier strolling around a friendly village may not require optimal paths, but a soldier creeping near an enemy base may require the best available paths to best avoid dynamically weighted areas where enemy soldiers are located.

- If a higher level navigation graph is also available, then the Euclidean distance heuristic can be improved upon by instead computing the shortest path between two points on the higher level graph and using this to estimate the length of the more detailed lower level path. This may avoid situations in which the Euclidean distance gives a particularly severe underestimate. For example a target node may be close to the current node but on the far side of a river or wall, the Euclidean distance will give a short cost but the actual cost to move around the obstacle can be high. If the higher level graph expresses the need to route around the obstacle then this underestimate is reduced.

4.5.3 Agent behaviour

The need-based priority system used to control the agent successfully produces a reasonably convincing behaviour, is robust and flexible and shows great promise for further work. The agent implemented uses the simplest form of needs-based evaluation, the agents needs are ordered by priority and each need has a trigger value, when the need exceeds this value the agent pursues this need. If more than one need is triggered at once the agent pursues the highest priority need. This system can be tweaked by altering the trigger values to produce for example highly cautious agents that respond very quickly to the need to search for healing items, or aggressive agents that largely ignore the need to manage their health and attack the player often. This hints at one of the outstanding features of a more complex needs-based system - personality parameters.

The concept of 'personality parameters' is a data-driven approach to program design which allows multiple agent behaviours or personalities to be produced using the same program code, by feeding different input parameters into the code. Agent behaviours can be tweaked or new agent behaviours created simply by altering personality data files. Personality parameters fit well with agent behaviour design concepts such as needs-based mechanisms [7] and fuzzy evaluation [65] [8] [9]. Sets of carefully chosen personality parameters can combine in interesting ways producing behaviour that was not originally visualised, this is a bottom-up design approach, and a flexible system can allow for a great variety of imaginative behaviours. [64] discusses a flexible and extensible approach to adding a complete set of personality parameters to a game agent design. [7] discusses the design of an advanced need-based behaviour mechanism incorporating various behaviour parameters. The system is an extension of the basic need-based mechanism implemented here which allows for more complex interaction of needs and emergent behaviour.

In this implementation agent needs are represented by a single trigger value. The agent is either currently in need of a particular resource or other need (the need to attack an opponent, the need to avoid damage by running for cover..) or is not currently in need of it. This is a Boolean evaluation of needs. Fuzzy logic is a superset of Boolean logic which extends the system to handle the concept of partial truth or falsity, representing values in a range between completely false and completely true [8] [9]. Fuzzy logic was introduced by Dr. Lotfi Zadeh of UC/Berkeley in the 1960's as a means to model the uncertainty of natural language [65]. In typical implementations of fuzzy logic a value in the range [0,1] is attributed to a logical element using a function called a function called a 'fuzzy relation'. Fuzzy logic can be used in the design of game agents to express things like the agents current need for a particular item or behaviour. In the game implemented here, fuzzy relations could be applied to the agents decision

to collect healing items, collect weapons or ammunition, or attack the player and to the agents decision on which weapon to use at a given time when it is carrying multiple weapons. At particular times, perhaps at regular intervals in the simplest case, the agent would decide which behaviour to apply or weapon to choose by evaluating the fuzzy relations and selecting the item with the highest value. The fuzzy relations would be based on variables such as how much of an item the agent already holds, personality parameters expressing the agents preference for an item or behaviour and the current state of the game world (for example, whether there is immediate danger from a nearby opponent which might increase the value of attack or hide behaviours). The computer game 'Quake III Arena' employs this form of decision-making for its agent characters [65]. Fuzzy relations represent a relatively simple decision making technique that can provide variety and convincing behaviour with low memory and processing requirements, and can be extended to produce complex behaviours by increasing the number and complexity of the fuzzy relations and the number or variables considered by the fuzzy relations.

[65] suggests that the task of balancing a complex set of fuzzy relations and preference values (or equivalently a set of personality parameters) in order to produce a suitably pleasing behaviour can be difficult and time consuming. For the computer game 'Quake II Arena' [65] this difficulty was handled by applying a process of genetic selection to a set of pre-designed agent behaviours. A number of agents are rated in terms of behaviour quality, the method of rating agents depends on the specific game being implemented and the sort of behaviours desired. The agents are then split into sets of three: two parents and a child based on the ratings assigned and an element of random chance with higher rated agents more likely to be selected as parents. The behaviours (fuzzy logic, preferences or personality parameters) of parent agents are 'bred' together using a genetic-algorithm technique such as averaging between each pair of values with an added random element, and the resulting behaviour replaces the child behaviour. This process is repeated a number of times until a set of satisfactory behaviours is produced.

A variety of different approaches to agent behaviour are also possible, including 'real' AI techniques such as neural networks or expert systems. [66] discusses the design of the SOAR game agent [24], [22], [20] which uses a complex tactical, hierarchical goal-based structure to implement agent behaviour. [66] discusses the addition of an 'anticipation' capability to the SOAR agent which gives the agent the ability to predict the behaviour of human players or other agents in order for example to set ambushes, a feature which makes for a significantly more engaging game experience. [67] discusses an approach taken for the computer game title 'Spec Ops II' based on a biological model of stimulus-response directives and servo feedback loops.

4.5.4 Synthetic vision

Agent 'senses' in computer game AI are a system for gathering information about items of interest in the simulated environment [62]. The relation to human or animal sense in the real world is simply a design metaphor. A key difference is that AI senses for computer games are typically active processes in which the agent scans the environment and decides for example what it sees, unlike in reality where stimuli arrive at senses of their own accord. The senses in computer game AI must only be as sophisticated as necessary to produce realistic and robust performance, and must be as efficient as possible. Further, for AI 'senses' to be worthwhile their effect must be clearly visible and understandable to the human player. For this reason, AI characters in games do not often simulate realistic senses of touch, taste or smell but often have simulated vision, hearing and pain. Sense can be simulated by inputs to agent behaviour state machines or other mechanisms. Simple pain responses can be simulated by programming an agent to respond accordingly if it receives damage and its health parameter is reduced, a simple task, or perhaps to respond to contact with other objects in the simulation whether they affect damage or not. Simple hearing can be simulated by feeding any sounds played within a set radius of an agent to its sensory inputs and programming the agent to respond appropriately according to the sound played, the object it emanated from and the position of the object. An obvious extension is to factor in the volume of the sound, so agents hear loud sounds further away than quiet ones. A further extension may be to combine different senses, so that an agent might respond differently to a sound emanating from an object which it can see and a sound coming from somewhere out of sight.

In this implementation a simple vision system was developed. Though simple, the system is robust and provides realistic performance that is highly visible to the player. Also, due to its simplicity the system is highly efficient and should be extensible to scenarios involving many agent characters without reducing the frame rate of the simulation. The system also provides scope for extension to a more complex, realistic solution.

The famously successful computer game 'Half Life' [30] used an agent vision system that is largely equivalent to the one implemented here [62]. 'Half Life' models vision using a 'view distance', a 'view cone' and a line of sight check which correspond to the range circle, field of view frustum and raycasting check implemented here. 'Half Life' also includes a system of agent attention - if an agent does not want to look for objects of a particular type it will not see them and they will not be processed by the vision function, and a system of agent hearing which involves an agent 'hearing sensitivity' and a check to see whether a particular sound carries to the agent. Certain characters in 'Half Life' also respond to 'pseudo-sounds' emitted by particular objects but not played to the

player, so that they appear to be able to 'smell out' particular items.

An example of a more advanced agent sensory system is the 'Thief' [27] series of computer games [62]. In these games the player takes on the role of a thief and encounters situations in which he must sneak past guards in the dark without alerting them in order to complete his task. The gameplay of 'Thief' focuses heavily on the agents sensory systems. The AI senses of 'Thief' are an extension of the same core concepts used in 'Half Life' and in this implementation. In 'Thief', senses are described in terms of levels of awareness, rather than simply 'seen' or 'not seen', 'heard' or 'not heard'. Awareness values are stored as links between an agent and another entity or a position in the environment, along with information such as the time and location of the sensing event, whether the agent had a line of sight to it, and so on. These 'sense links' act as the primary 'memory' of the agents. The awareness values combine to form an alertness value for an agent, which is fed back into the agents behaviour system. A guard that hears a nearby suspicious sound suddenly becomes more alert and may be prompted for example to start looking around in the direction of the sound. An agents reactions to its senses are made highly visible and verbose, in order to highlight the performance of the sensory system and improve the players experience of the game. For example a guard hearing a suspicious sound from somewhere out of sight might loudly say 'What was that?', a guard hearing a sound and sighting the player that caused it will blurt out 'Thief!'. If no more relevant senses are received (i.e.: no more senses in the same sense link - between the same agent and other entity or agent and position), the agents alertness slowly decreases over time and it will return to its original state.

Rather than using a single view frustum or field of view, agents in 'Thief' have an ordered set of view frustums which are rotated according to the direction the characters head is facing. Different frustums correspond to short and long range vision, central vision and peripheral vision for example, and have varying levels of sensitivity to general acuity and movement for example. A discrete visibility factor is applied to an object sighted by an agent according to factors such as the general level of visibility of the object, how brightly lit or shadowed it currently is and which of the agents frustums it was sighted in. 'Thief' also employs a pseudo-sense similar to the pseudo sounds of 'Half Life' in the form of a small frustum which looks out of the back of the agents head and is sensitive to movement, causing the agent to respond to movement occurring very close behind it even if no sound is made.

Using this sensory system as input to finite state machines to control agent behaviour, and without significantly increasing the complexity of design of the agent state machines, 'Thief' provides a highly convincing and compelling game experience that is centred around the sensory systems of the games agent characters.

Alternative and perhaps more complex sensory systems are possible for computer

game agents. [63] for example discusses a sensory system based on the real-time rendering of the agents view of the world, also implemented using Fly3D.

5. Conclusions

The goal of this project has been to investigate the design of computer-controlled agents for modern, real-time, 3D computer games, identify the major challenges posed by agent design and the available solutions, and to implement a computer game agent in order to demonstrate the application of some of the solutions considered. An action based 'first-person shooter' game style similar to well known commercial titles such as 'Quake III Arena' and 'Unreal Tournament' was chosen following the influence of the available resources.

An iterative development plan was taken and as a result the implementation portion of the project was in fact carried out as three seperate implementations, with each successive implementation building on the work of its predecessor(s).

The work of the first implementation involved studying and understanding the Fly3D game engine and SDK, and learning the techniques needed to develop a game as a plugin to the Fly3D engine. The first implementation succeeded in developing a game set in a simple but illustrative 3D environment, identified the set of major challenges to be addressed when designing an agent for a game of this type and demonstrated an initial set of basis solutions to the challenges. The second implementation aimed to extend the design used in the first implementation to a much more complex environment chosen to be representative of the more complex environments typically found in modern commercial computer games, as well as extending the functionality of the agent itself to produce a more realistic game character.

The second implementation identified many difficult complexities presented by the environment and some useful extensions to the agent design solutions, but was troubled by persistent difficulties due to fundamental shortcomings of the underlying navigation structure which was inherited from the first implementation. As such, the second implementation was never developed into a complete game.

A third implementation was completed which represented a fundamental change in the design of the agent navigation system aimed at overcoming the limitations of the previous system and achieving the desirable aims identified by the second implementation. The third implementation succeeded in developing a complete game demonstrating an agent design solution capable of navigating arbitrary complex 3D environments without requiring guidance in the form of human input.

Due to its complex and difficult nature the navigation or 'pathfinding' problem for agents in arbitrary, complex, 3D environments has been the major focus of the work undertaken. The design of agents for computer games however is

a multifarious problem of which navigation is only one part. Throughout the project, solutions to problems including methods of controlling high level agent behaviour, enabling agents to realistically use the projectile weapons seen in the game and engage in combat maneuvers and an agent visual sensory system were developed. The implementations of these problems represent basic solutions and the investigation has identified the basis of extensions to these solutions to produce complex and varied results.

The knowledge gained during this investigation gives a clear indication of the techniques that could be used to develop flexible and robust computer game agents capable of existing in an arbitrary, complex 3D environment in real-time and exhibiting varied, convincing and engaging behaviour.

Using the automatically generated sectors and portals navigation data to describe arbitrary environments, combined with a special 'teleport' mechanism and specialised routines to handle devices such as closed doors, ladders, elevators and many others and an optimised implementation of the A-star algorithm extended to handle concepts such as dynamically varying terrain costs and a dynamically scaled heuristic function an agent is equipped to take on the array of complexities presented by realistic environments. Given a simple command such as 'go to position (x,y,z)' from a human player or higher level controlling module an agent can autonomously execute a series of maneuvers, navigating rooms, corridors and other environments, operating doors and elevators and avoiding obstacles to arrive at its destination. With a combined, dynamically controlled system of time-slice pathfinding and path queuing and a suitable obstacle avoidance routine based on collision prediction and obstacle tracing or routing methods, a large number of agents can co-exist simultaneously in the same environment. Add to this a routine integrated into the sectors and portals that generates bezier-curves from the paths returned by the pathfinding function and agents can move in a smooth, sweeping, natural manner. With the navigation problem solved, these characters, set against a backdrop of reactive, rule based agents representing birds, cats and dogs or fish for example using behavioral animation and flocking techniques, would be controlled by combinations of mechanisms such as finite state machine, need-based and fuzzy-evaluation systems. Carefully designed with respect to the needs of the individual game and fed by inputs from artificial sensory systems allowing human-like interactions with the environment these systems would produce widely varied, flexible and robust higher level behaviour.

The design of complex agent characters for computer games is an emerging science that will in the years to come extend the realism and immersiveness of real-time 3D games by an order of magnitude. I envisage an environment of arbitrary size, variation and complexity inhabited by agent characters that deftly handle the manifold challenges posed by this setting with the appearance of something approaching human-level skill and finesse.

Appendix A. Additional examples of completed sector and portal data

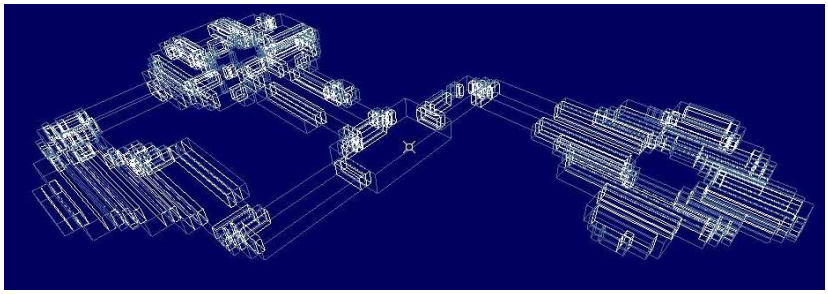


Figure A.1: 2093 cells reduced to 121 sectors and 297 portals.

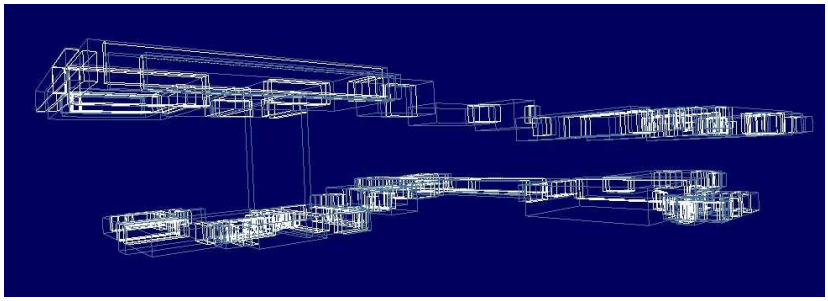


Figure A.2: 2905 cells reduced to 98 sectors and 232 portals.

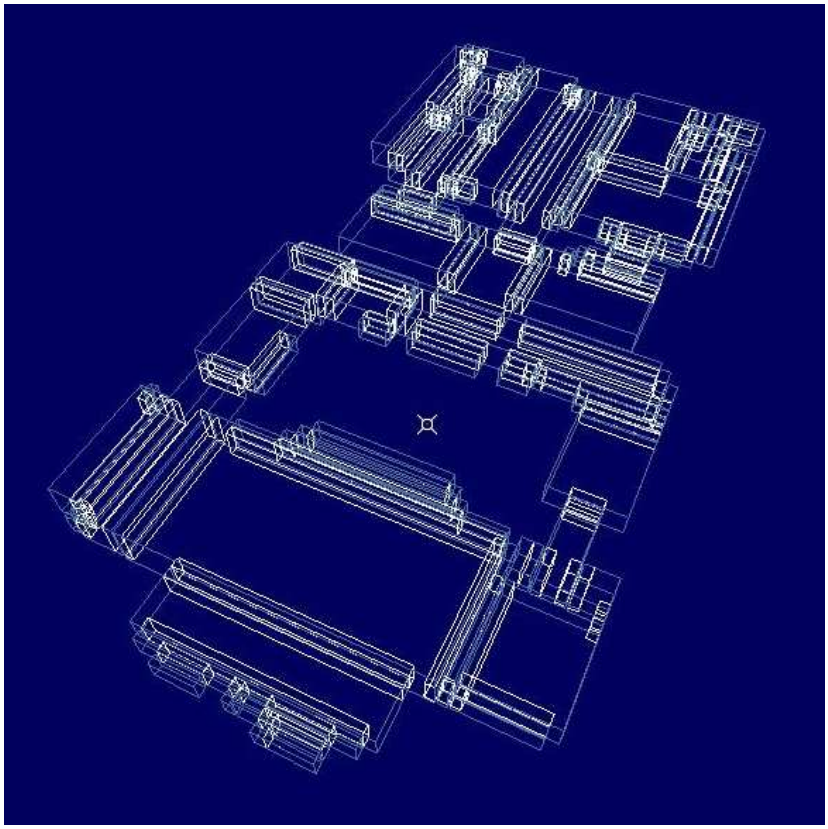


Figure A.3: 2481 cells reduced to 69 sectors and 184 portals.

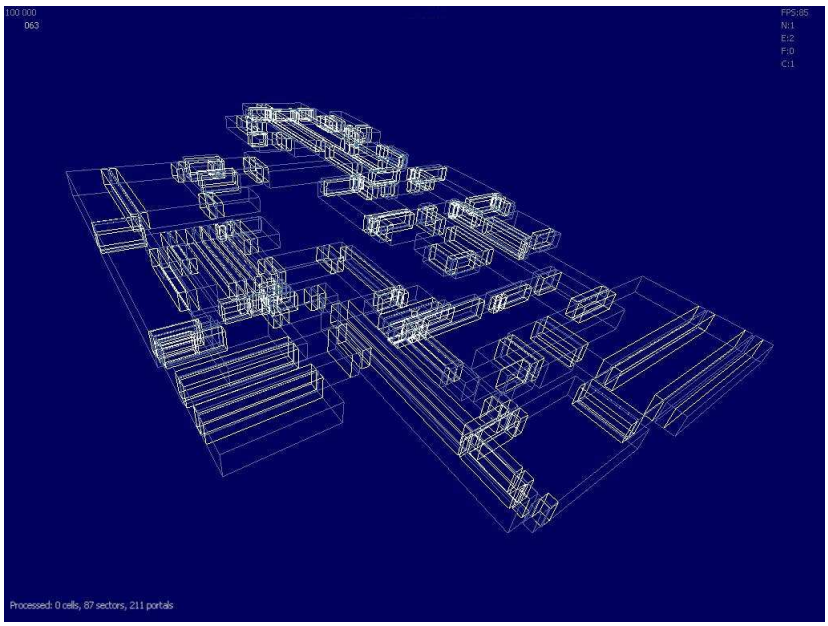


Figure A.4: 3295 cells reduced to 87 sectors and 211 portals.

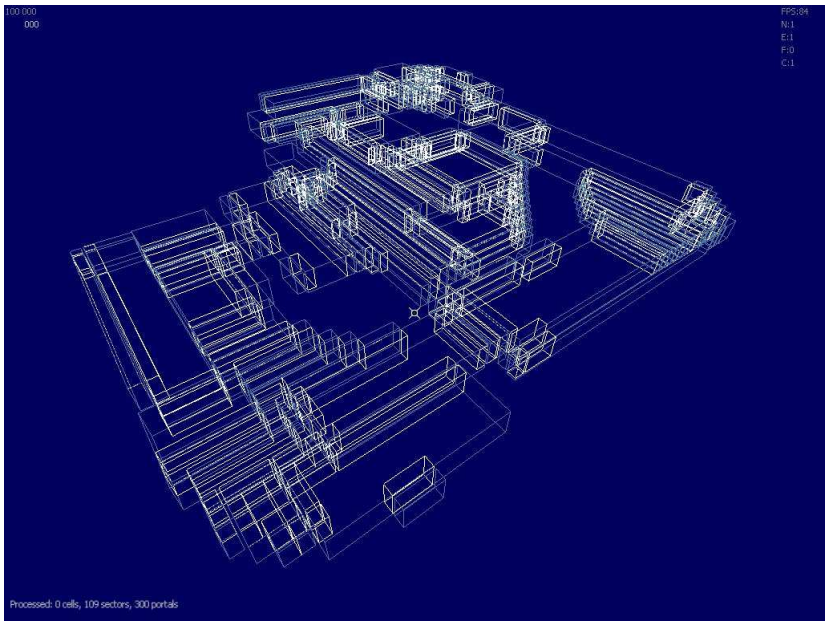


Figure A.5: 2341 cells reduced to 109 sectors and 300 portals.

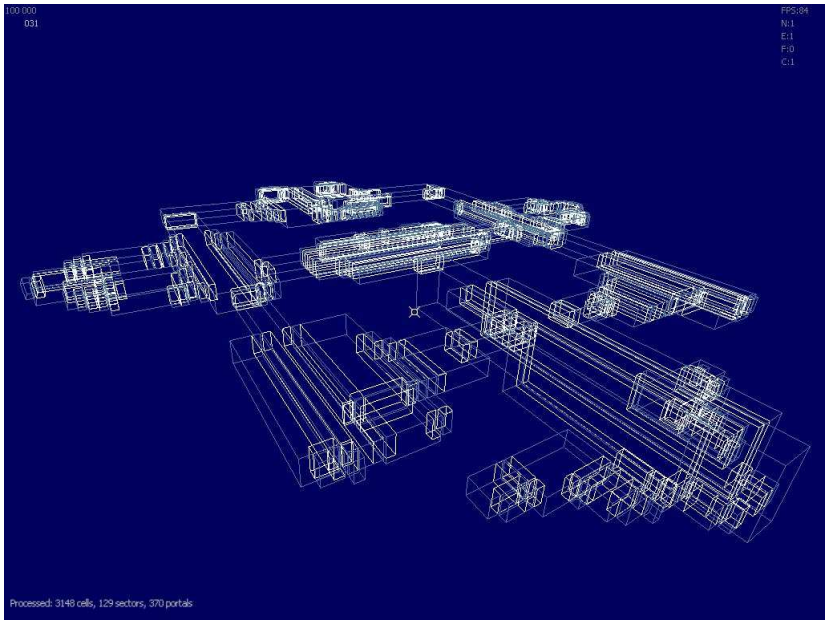


Figure A.6: 3148 cells reduced to 129 sectors and 370 portals.

Appendix B. Fly3D

The `flyEngine` module exports the important `flyBspObject` class which represents an object in the BSP scene. Every plugin-implemented object that wishes to exist in the BSP scene must derive from the `flyBspObject` class. The `flyBspObject` class provides several virtual functions which must be re-implemented by a plugins' classes in order to produce the desired behaviour for a BSP object.

B.0.5 flyBspObject virtual functions

- Default Constructor
  The default constructor must assign initial values to the class' member variables to assure that a recently instantiated object does not contain invalid data that could cause a crash in the system.
- Copy Constructor
  The copy-constructor must copy all the member variable values from the given source object, allocate additional memory and pointers for the new copies of data and call the copy-constructor of all the parent classes.
- `void init()`
  This function is called whenever the object is *activated* (see B.0.6) in the simulation, and should not be confused with the constructor. `init()` should implement any actions to be taken upon inserting an object into the simulation, such as initialisation and validation of the objects attributes.
- `int step(int dt)`
  This function is called by the engine once per frame for every active object, and allows each object to update its state. For example objects may wish to update position or velocity attributes, or check for input etc.
- `void draw()`
  The draw function is called by the engine whenever it requires an object to draw itself, ie.: whenever a camera is facing the object. The draw function may not be called for a particular object and frame if no camera is facing the object, or may be called more than once if multiple cameras are facing an object.
- `void draw_shadow()`
  Similar to the draw function, this function is called by the engine whenever it requires the object to draw its shadow.

- `flyMesh *get_mesh()`
  If required an object must implement this function to return a pointer to its polygon mesh instance.

- `flyBspObject *clone()`
  This function is called whenever the engine needs to make a copy of the object and should call the objects copy-constructor and return the newly allocated pointer.

- `int message(const flyVector& p,float rad,int msg,int param,void *data)`
  An object must implement this function if it needs to receive messages about events that occur in the simulation, or to send messages. This messaging system is used for communication between objects. Different message types are used for different communications, and an object responds only to the message types that it needs to respond to. For example, a dynamic light may send out illumination (`FLY_OBJMESSAGE_ILLUM`) messages to nearby objects, and any object that needs to be dynamically lit must respond to messages of this type.

- `int get_custom_param_desc(int i,flyParamDesc *pd)`
  This function is used by the flyEditor application to edit object parameters. An implementation must fill in the `flyParamDesc` parameter object which represents a parameter description including parameter name, parameter type and a pointer to the actual parameter data. The `i` parameter value denotes which of the objects parameters is being requested from the function. The return value should always be the total number of editable parameters exported by the object.

- `flyMesh *ray_intersect(const flyVector& ro,const flyVector& rd,flyVector& ip,float& dist,int &facenum)`
  Must implement ray intersection with the object, for example intersecting with the objects actual polygon mesh or just with the objects bounding box. The function must compute the intersected point and face and return the intersected mesh.

- `int ray_intersect_test(const flyVector& ro,const flyVector& rd,float dist)`
  This function is similar to `ray_intersect` but must only test for intersection returning true or false, not computing the intersection point or face, and so should be faster than `ray_intersect`.

B.0.6 Stock and Active objects

During a simulation, many objects of different types can be inserted into and removed from the scene, and multiple independent copies of the same object may

exist in the scene at once. Each copy of an object is inserted into the scene with the original parameter values for its class, but is controlled independently of other objects and maintains its own parameter values while in the scene. This is achieved by the use of stock and active objects. For each class a single stock object holds the initial parameter values for that class of object, and whenever a new object is added to the scene the flyEngine uses the clone function of the stock object and inserts the new copy into the scene as an active object.

B.0.7 Files and file formats

Fly3D utilises a number of custom file formats enclosing data which is processed by the flyEngine for scene loading and simulation. File types include map files for scene geometry information, BSP files containing BSP trees for scenes, lightmaps and shaders for lighting information, static and animated mesh information for polygon mesh objects and more.

A key file type is the `.fly` file, which is the scene information file. It includes all global and plugin-specific parameter values for a scene, as well as parameters for every entity (stock and active objects) present in the scene, and the locations of data files such as the BSP file needed for a scene. The `.fly` file is the main file type opened and saved by the development and simulation front-ends.

B.0.8 Fly3D standard plugins - walk

Fly3D comes equipped with a set of standard plugins, prewritten by the creators of Fly3D, to demonstrate the capabilities of the engine and to act as a platform for Fly3D users to build upon when creating their own plugins. When creating plugins, users of Fly3D can incorporate standard plugins in a scene along with their own plugins, or can create their own plugins as extensions to the standard plugins.

The leading standard plugin for Fly3D is a 'ship' game which involves maneuvering a flying spacecraft through a maze-like level in a hunt to destroy opposing spacecraft. Another standard plugin of interest to this project is the walk plugin. The walk plugin provides a person character with a 3D mesh, animation, physics and movement functionality. The character responds to keyboard inputs and includes the ability to use a projectile weapon and to be hit by weapons, take damage and be destroyed. Additionally, the walk plugin includes several projectile weapons for use by the character which can be placed within a scene and picked up by the character. Health bonuses and extra ammunition are also available to be picked up. The player in my implementations controls an object instance of a class which is an extension of the walk class, and the agents I have

implemented are also extensions of the walk class with alterations to make the class suitable for computer-controlled characters.

Appendix C. Implementation one

C.1 Files and Classes

Files:

```
myPlugin.h myPlugin.cpp Agent.cpp EnemyAgent.cpp myCamera.cpp SpawnPoint.cpp
```

Classes written:

```
myPlugin myCamera spawnPoint NavPoint EnemyAgent FriendlyAgent Agent
Player
```

Classes used from Walk:

```
gib diemesh powerup teleport jumppad person
```

C.2 myplugin.h

The header file must conform to all the requirements of a normal C++ header file and declare all classes, class members, class functions and plugin-global variables and functions defined by the plugin.

In addition, the header file must enumerate a number of constant types which are used as the 'type' field for each of the plugins exported classes. Each class in a Fly3D scene must have a unique integer 'type' field to allow Fly3D to distinguish objects of this class from objects of other classes, and to group objects of the same class so that accessing them in sequence is optimised [1]. The programmer must take care that the type field of each of his classes does not equal the type field of any of his other classes, or the type field of any other class exported by any other plugin in the scene. It is good practice then to enumerate the type constants used by the plugin at the top of the plugins header file so that clashes in

a particular scene can be identified and removed. Hence myplugin.h enumerates a constant value for each of myplugin's exported classes:

```
enum{ TYPE_MYCAMERA=19781982, TYPE_SPAWNPOINT, TYPE_NAVPOINT, TYPE_AGENT,
TYPE_ENEMYAGENT, TYPE_FRIENDLYAGENT, TYPE_PLAYER };
```

The header file must also define a class description meta-class for each of the plugins exported classes. The class description is a class that inherits from the Fly3D class `flyClassDesc` and re-implements three virtual functions that allow external modules to see what members the class exports [1]:

- The `create` method must return a new instance of the described class.
- The `get_name` method must return a string containing the friendly name of the class.
- The `get_type` method must return the type that defines that class.

For example, the description class for the `myCamera` class looks like this:

```
class myCamera_desc :  public flyClassDesc{
      flyBspObject *create() {return new myCamera;};
      const char *get_name() { return "myCamera"; };
      int get_type() { return TYPE_MYCAMERA; };
      };
```

C.3 myplugin.cpp

The `.cpp` file for the plugin contains the definitions of the plugins global functions and members.

Plugin global members

Values for recording the players score, game state etc:
`int score, enemies, friends, friendsDied, friendsRescued, enemiesKilled, enemiesCreated, game_time, game_paused, restart_time`

The flyArray containing the navigation points used by the agents:
`flyArray<NavPoint*> navpoints`

A flyString that the plugin writes to the screen every frame:
`flyString globalMessage`
The string can be filled in or appended to by any objects in the plugin, and is used to aid in development and debugging.

Functions

Any Fly3D plugin must also implement some specific functions in its `<plugin_name>.cpp` file:

- `int num_classes( )`
  Returns the number of classes exported to the Fly3D engine by this plugin.
- `flyClassDesc *get_class_desc(int i)`
  The parameter value `i` acts as an index into the list of classes exported by the plugin. The function switches on `i` and returns a reference to an instantiation of a class description for the corresponding class.
- `int fly_message(int msg, int param, void *data)`
  This function is used by the Fly3D engine to notify plugins of various types of event in the simulation. The function switches on the `msg` parameter and executes the plugins response to various types of message defined by Fly3D global constants:
  - `FLY_MESSAGE_INITSCENE`
    Initialise the plugin. As part of the initialisation of a Fly3D scene, the Fly3D engine will call `fly_message` with this parameter for every plugin in the scene. This gives each plugin an opportunity to carry out any plugin-global initialisation tasks it requires.

    For `myplugin`, all stock objects in the scene must be searched looking for objects of type `NavPoint`. A pointer to each `NavPoint` object in the scene is added to `flyArray navpoints`.
  - `FLY_MESSAGE_UPDATESCENE`
    Per-frame update of the plugin. The Fly3D engine calls `fly_message` with this parameter once per frame for each plugin in the scene. This gives the plugin the opportunity to carry out any plugin-global updates it requires each frame.

    `myplugin` plugin does not need to make use of this case.
  - `FLY_MESSAGE_DRAWSCENE`
    The Fly3D engine calls `fly_message` with this parameter once per frame for each plugin in the scene. One of the plugins in a scene must receive and respond to this message and call the appropriate Fly3D functions to draw a view of the simulation from some point, or nothing will be drawn. `myplugin` does not need to respond to this message as the `walk` plugin which works alongside will respond and draw the scene from the point of view of the player.
  - `FLY_MESSAGE_DRAWTEXT`

The Fly3D engine calls `fly_message` with this parameter once per frame for each plugin in the scene. This allows a plugin to draw any 2D graphics such as text or lines it wishes to output for each frame.

`myplugin` uses this case to output colourful textual information such as the players score, level of health and ammunition, game state and so on. If the game is over, this case is used to display a summary of the players performance

– `FLY_MESSAGE_CLOSESCENE`
The Fly3D engine calls `fly_message` with this parameter for each plugin as part of the scene closing procedure. This allows a plugin to carry out any plugin-global closing down tasks it requires.

`myplugin` does not make use of this case.

- `int get_global_param_desc(int i, flyParamdesc *pd)`
This function allows a plugin to specify any plugin-global parameters it wishes to export to the Fly3D engine. When the Fly3D engine requires information about the parameters exported by a plugin it uses this function.

  A plugin must switch on the parameter value `i` which specifies which of the plugins exported parameters is being requested, and fill in the `flyParamdesc` parameter object with details of the relevant parameter. The return value of the function should be the total number of parameters exported by the plugin.

  (EXPLAIN flyParamdesc class?)

  `myplugin` exports just one parameter:

  `flyArray<NavPoint> navpoints`, the array holding all the `NavPoint` objects in the scene.

C.4 The myCamera class

The `myCamera` class is declared in `myplugin.h` and implemented in `myCamera.cpp`.

The `myCamera` object is an extension of an object developed as part of one of the Fly3D tutorials [1]. It implements a floating BSP object which has no mesh, but can be set as the global `camera` parameter for a scene, meaning that the scene will be viewed from the point of view of the `myCamera` object. Using the mouse, a user can change the `myCamera` objects orientation so as to look around the scene, and using the keyboard a user can move the `myCamera` object forwards, backwards, left or right so as to float around within the scene.

So the `myCamera` object is an invisible, controllable floating camera from which the scene can be viewed. The object can be set to collide with scenery such as walls, floors and ceiling, or if convenient can be set to pass through scenery without colliding.

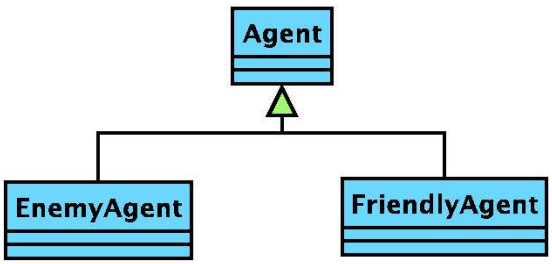
The purpose of the `myCamera` object is as a development and debugging tool. It is used to observe the behaviour of a plugin, and can be configured to output to the screen various types of information. `myCamera` appears throughout the different plugins I have implemented for this project, outputting different forms of data in each.

In `myplugin` the `myCamera` object is configured to output its current x, y and z position coordinates. This helps with placing and debugging the `NavPoint`, `SpawnPoint` and `BirthPad` objects in the scene.

C.5 The Agent class

The `Agent` class is declared in `myPlugin.h` and implemented in `Agent.cpp`.

The `Agent` class implements the behaviour and abilities common to all types of computer-controlled agent in the plugin. Other classes inherit from `Agent` and implement their own specific behaviours and abilities:



The `Agent` class inherits much of its basic functionality from the `person` class of the Fly3D standard plugin `walk` [1], but overrides most of the `flyBspObject` virtual functions inherited from `person` in order to provide its own behaviour.

The `Agent` class can be instantiated to produce an animated, computer-controlled agent in the scene complete with polygon mesh, laser gun, correct collision detection and physics behaviour, animation and the functionality to take damage from weapons and die when enough damage is taken. However this agent will only stand still. Although the agent class includes functionality to allow the agent to walk around, orient itself toward a target, compute a path to a target using the navigation structure and to follow a path, it does not contain the functionality to tell the agent when to perform these actions. It is the job of inheriting classes to

implement controlling routines that initiate the right functions provided by the `Agent` class at the right times.

C.6 The EnemyAgent class

The `EnemyAgent` class inherits from the `Agent` class and adds functionality to provide the games opposing agent characters.

C.7 The FriendlyAgent class

The `FriendlyAgent` class inherits from the `Agent` class and adds functionality to provide the games friendly agent characters.

C.8 The SpawnPoint class

The `SpawnPoint` class implements an invisible 'spawn point', a point in 3-space in the scene from which new `EnemyAgent` object instances appear. Agents appearing in this way are said to be spawned. The spawning behaviour of the spawn point can be controlled through various parameters.

C.9 The NavPoint class

`NavPoint` is a very simple class, needing only three parameters:

- Position: the `NavPoint`'s position in 3-space in the scene. This is provided by the `pos` parameter inherited from the `flyBspObject` class.
- Unique ID: an integer identifier to differentiate this `NavPoint` from other `NavPoints`, provided by the `int id` member variable.
- A list of neighboring `NavPoints` of this `NavPoint`, provided by the member variable `neighbors`, a `flyArray` of integer values referring to the unique ID's of `NavPoint` objects.

C.10 The Player class

The `Player` class is a simple extension of the Fly3D standard plugin `person` class. It customises the `person` class for the purposes of this plugin, setting some default parameter values and providing some extra accessor functions used by the plugin.

Appendix D. Implementation two

Files:

`Agent.cpp myCamera.cpp myPlugin.cpp myPlugin.h NavPoint.cpp Opponent.cpp SpawnPoint.cpp`

Classes implemented:

`Agent myCamera myPlugin Opponent`

Classes re-used from implementation one:

`NavPoint SpawnPoint`

Classes used from walk:

`gib diemesh powerup teleport jumppad person`

D.1 myPlugin.h

myplugin.h defines each global field and class used by the plugin, as well as the type constants and description classes required by Fly3D (see C.2).

Plugin-global fields:

`extern int score`
Records the players current score.

`extern flyArray<NavPoint*> navpoints`
The array of navigation points forming the navigation graph for the environment.

`struct area{std::list<int> joins;}`
The struct representing an area. The environment is split into areas which define

a high level navigation graph. Areas are represented by their links to other areas. Each `NavPoint` object contains a field declaring which area it belongs to (areas are identified by their index in the flyArray areas).

`extern flyArray<area> areas`
The list of areas which make up the higher level navigation graph.

`extern int enemies, extern int friends, extern int friendsDied, extern int friendsRescued, extern int enemiesKilled, extern int enemiesCreated, extern int waitpic, trackpic, movepic, followpic, followpathpic, extern int maxenemies`
Fields recording the current game state.

`extern flyString globalMessage`
For debugging output. See previous appendix.

`extern flyVector globalStart, extern flyVector globalEnd`
Additional fields for debugging output, allowinga class to request that a vector is drawn from `globalStart` to `globalEnd`.

`extern int game_paused`
Pauses the simulation.

`extern std::list<powerup*> lifepickup, extern std::list<powerup*> machineammo, extern std::list<powerup*> shotammo, extern std::list<powerup*> rocketammo, extern std::list<powerup*> railammo, extern std::list<powerup*> shotgun, extern std::list<powerup*> rocketgun, extern std::list<powerup*> railgun, extern std::list<Opponent*> opponents`
Lists containing entries for each relevant object in the scene, maintained for efficiency (so that the entire set of Fly3D entities in the scene need not be searched during simulation).

D.2 myPlugin.cpp

`myplugin.cpp` contains the definitions of the global fields declared in `myplugin.h`, and implements the functions required by a `<plugin_name>.cpp` file by the Fly3D engine (see C.3). In particular, the `fly_message` function responds to the `FLY_MESSAGE_INITSCENE` message by building lists of the entities on the scene relevant to the plugin, reading in the nodes of navigation graph from file and computing the edges of the graph and outputting the navigation graph debugging file. The `FLY_MESSAGE_DRAWTEXT` message is responded to by outputting information about the current game state to the player and the `FLY_MESSAGE_DRAWSCENE` message is responded to by drawing debugging information. A debug mode when activated allows the navigation

graph as well as additional information requested by any of the plugins objects to be drawn.

D.3 The Agent class

This class implements the agent character. The class is very large and complex and implements functionality including:

- The state machines controlling the agents higher level behaviours.
- Environment sampling and obstacle avoidance techniques.
- Hierarchical A* pathfinding.
- Agent movement and path following functions.
- The agents ability to track and chase a moving target.

D.4 The myCamera class

The `myCamera` class is carried over from the previous implementation and is extended to provide additional debugging information. The camera outputs its current position and the ID of the current nearest navigation node.

D.5 The Opponent class

The complex `Opponent` class inherits from the agent class and implements the behaviour specific to the opposing agent character, including state machine behaviour control, parameterised use of projectile weapons and combat maneuvers and the agents ability to seek out and collect useful items from the environment.

Appendix E. Implementation three

The third implementation consists of two independent plugins for Fly3D:

- `myplugin3` is a program which auto-generates sector and portal data for arbitrary 3D environments. This program was written as a plugin for Fly3D because this allows it to easily visualise the results produced, making the program easy to demonstrate and to debug. The plugin is loaded into a Fly3D scene and immediately begins work creating the sector and portal data. Using the `myCamera` object, a user can watch the graphical output of the program that appears in the Fly3D scene and the textual output in the Fly3D console. When its work is complete, `myplugin3` saves the sector and portal data to file. The plugin responds to various parameters. For debugging there are parameters which control the textual output of the plugin through the Fly3D console from completely silent to fully verbose settings. In verbose mode the plugin outputs detailed information about its progress. Additional parameters control the delay before the plugin starts and delays between individual steps in the plugin, allowing the data to be generated more slowly to observe specific behaviour, and parameters controlling the behaviour of the flood fill algorithm as described in the main body of the report (section 4.5).
- `myplugin3game` is a game plugin which reads the sector and portal data for an environment from the corresponding file created by `myplugin3`. The game provides a computer-controlled agent character that battles with the player and uses the sector and portal data to navigate the environment. A special debug mode allows a viewport showing the game from the agents point of view to be overlayed on the screen, as well as the sectors and portals themselves, the agents current path and the agents view frustum.

E.1 The generator

E.1.1 Files and Classes

Files:

`dummy.cpp myCamera.cpp myplugin3.cpp myplugin3.h`

Classes implemented:

`Box drawtrick Dummy myCamera`

E.1.2 myplugin3.h

Myplugin3.h defines each global field and class used by the plugin, as well as the type constants and description classes required by Fly3D (see C.2).

Plugin-global fields:

- `bool verbose`
  If set to `true`, the plugin outputs full detail information about its progress to the console, useful for debugging specific problems. If `false` the plugin only outputs minimal information.
- `Box box`
  Plugin-global instance of the Box class.
- `std::list<Cell*> task_list`
  The task list is used during stage 1 of the generation process. The initial cell is placed on the task lits, and at each iteration of the algorithm a cell is removed from the list and processed. As new reachable cells are discovered during the processing of a cell they are placed on the task list for processing by a later iteration. Once the task list is empty stage one is complete.
- `std::list<Cell*> cells`
  The list of processed cells. Once a cell has been removed from the task list and processed it is added to this list.
- `std::list<Sector*> sectors`
  The list of completed sectors.
- `int portalcount`
  Records the number of portals processed.
- `int stage`
  Records the current stage of generation, the algorithm has 4 distinct stages.

In addition, myplugin3.h defines some important structs used by the generation process:

- `struct Cell` This struct represents a cell used by stages one and two of the generation process, consisting of:

  - `flyVector pos`
    The position of the cell in the scene.
  - `bool bnorth, bsouth, beast, bwest`
    Boolean values denoting whether the cell currently has a north, south, east and west neighbor cell respectively.
  - `Cell * north, south, east, west`
    Pointers to this cells north, south, east and west neighboring cells respectively.
  - `bool operator==(const Cell x) const`
    The equality operator for Cell structs. Two Cells are defined to be equal if there position vectors are equal.
  - `bool operator!=(const Cell x) const`
    The negation of the equality operator for Cell structs.
  - `bool processed`
    Initialised to `false`, this value is set `true` once the cell has been processed by stage one of the generation process.
  - `int x,y`
    Coordinates representing the cells position within its sector, used by stages two and three of the generation process.

- `struct SectorCell` Once they have been collected into sectors, `Cell` structs are translated into the equivalent `SectorCell` structs. A `SectorCell` consists of:
  - `Cell *cell`
    A pointer to the `Cell` struct corresponding to this `SectorCell`.
  - `int x,y`
    Coordinates of the `SectorCell` relative to other cells in its sector.

- `struct Sector` This struct represents a sector, consisting of:
  - `std::list<SectorCell*> cells`
    A list of pointers to the cells contained in this sector.
  - `std::list<Portal*> portals`
    A list of pointers to the portals leading out of this sector.
  - `std::list<SectorCell*> northmost`
    A list of pointers to the cells along the northmost edge of this sector.
  - `std::list<SectorCell*> eastmost`
    A list of pointers to the cells along the eastmost edge of this sector.

  - `std::list<SectorCell*> southmost`
    A list of pointers to the cells along the southmost edge of this sector.
  - `std::list<SectorCell*> westmost`
    A list of pointers to the cells along the westmost edge of this sector.
  - `int id`
    A unique identifier that differentiates this sector from other sectors.
  - `flyVector min`
    The minimum corner of the sector.
  - `flyVector max`
    The maximum corner of the sector.
  - `bool complete`
    Initialised to `false`, `complete` is set `true` once the sector has been expanded as far as possible by stage three of the generation process.
  - `bool operator==(const Sector x) const`
    The equality operator for `Sector` structs, two sectors are defined to be equal if their `id` fields are equal.

- `struct Portal` This struct represents a portal, consisting of:
  - `std::list<Cell*> cells`
    A list of pointers to the cells contained within this portal.
  - `Sector *dest`
    A pointer to the destination sector of this portal.
  - `int destination`
    The unique identifier of the destination sector of this portal.
  - `bool done`
    Initialised to `false`, `done` is set `true` once the portal has been completed by stage four of the generation process.
  - `flyVector min,max`
    Vectors representing the minimum and maximum corners of the portal.

E.1.3 myplugin3.cpp

`myplugin3.cpp` contains the definitions of the global fields declared in `myplugin3.h`, and implements the functions required by a `<plugin_name>.cpp` file by the Fly3D engine (see C.3). In particular, the `fly_message` function responds to the `FLY_MESSAGE_DRAWTEXT` message by outputting progress information to the screen.

E.1.4 The dummy class

The `dummy` class implements an extension of `flyBspObject` which carries out the auto-generation process itself and handles the emulation of the agent physics and movement.

A dummy character, an entity with the polygon mesh, movement and physics functions of the `person` class used by the computer controlled agents but with no actual behaviour of its own is manipulated in order to determine reachabilities between cells.

This is a very large and complicated class which implements functionality including:

- Finding intersections between cells, sectors and portals of which there may be thousands in the scene at once.
- Drawing the cells, sectors and portals as the data generation progresses, to provide a graphical display of the process.
- Reading sectors and portals from file if the relevant file exists and writing the data to file once ith as been generated.
- Outputting debug information.
- Autonomously run through all 4 stages of the data generation process.
- Emulate the movement and physics of the agent character.

E.1.5 The Box class

`Box` is a utility class not exported to the Fly3D engine. It implements a bounding box style object designed for the purposes of this plugin that is more suitable than Fly3D's own `flyBoundBox` class. The `Box` class is used at all stages of the generation process to draw and compute collisions between cell, sector and portal objects.

Only a single, global instance of the `Box` class is used. It is initialised to a particular size and stored centred at the origin. In order to draw or compute collisions between different objects, the box is translated into the required position. Hence only a single `Box` is needed for all cells, sectors and portals. The box is initialised to the shape and size of a cell, and provides special functions for drawing sectors or portals which can be different sizes.

E.1.6 The drawtrick class

The `drawtrick` class implements an object with a large bounding box but no polygon mesh, collision or physics functionality. The class is designed to trick the Fly3D engine into drawing the cells, sectors and portals which are represented only by structs, not `flyBspObject` instances, and are not exported to the Fly3D engine.

Through its `step` function the `drawtrick` object ensures that its position is always centred at the position of the current scene camera. Combined with the objects large bounding box, this means that the Fly3D engine will request the `drawtrick` object to draw itself in every frame, by calling the objects `draw` function.

With its `draw` function the object does not draw itself but takes the opportunity to draw the current cells, sectors and portals using the `Box` class and according to the current stage of the generation process and the state of each cell, sector and portal. This gives the generation process its graphical output which makes the progress and procedure of the process clear, and is useful for demonstrating and debugging. This is achieved without any of the cells, sectors or portals being instances of `flyBspObject` or being exported to the Fly3D engine. Because a very large number of these objects may need to be produced by the process, depending on the size and complexity of the environment being used, it is desirable for speed and memory concerns to have them stored as structs of the minimum achievable size and complexity.

Before the introduction of this mechanism the process was prohibitively slow even for quite small environments, and could potentially have exhausted the available memory resources with larger environments.

The other `flyBspObject` inherited virtual functions for this class are not needed, and are empty.

E.1.7 The myCamera class

For this plugin, `myCamera` is configured not to output any information and is used only as an easily controllable camera from which to view the progress of the generation process.

E.2 The game

E.2.1 Files and Classes

Files:

`Agent.cpp myplugin3game.cpp myplugin3game.h`

Classes implemented:

`Agent`

E.2.2 myplugin3game.h

Defines the structs `Portal` and `Sector` (see E.1.2) and the description class (see C.2) for class `Agent`, and declares class `Agent` and a boolean value `debug` which switches the games debug mode on and off. The debug mode draws a lot of extra data into the scene to demonstrate how the agent is working and to assist in identifying problems. For example, the sector and portal data used by the agent to navigate is drawn, a 3D line representing the path the agent is currently following is drawn, and the agents view frustrum is drawn.

E.2.3 myplugin3game.cpp

`myplugin3game.cpp` contains the definitions of the global fields declared in `myplugin3game.h`, and implements the functions required by a `<plugin_name>.cpp` file by the Fly3D engine (see C.3). In particular, the `fly_message` function responds to the `FLY_MESSAGE_INITSCENE` message by reading in the sector and portal data from file.

E.2.4 Class Agent

This class implements the computer controlled Agent that plays against the player. The class is very large and complex, and implements functionality including:

- Pathfinding over the sector and portal data, using the time-slice A* technique.
- Agent movement and path following ability.

- Higher level agent behaviour control.
- The agents combat and weapons using behaviour.
- The agents synthetic vision system which provides inputs to the behaviour functions.
- The agents ability to look for and collect useful items around the environment.

Bibliography

[1] The Fly3D SDK 2.0 Documentation

[2] The Fly3D website: `http://www.fly3d.com.br/`

[3] The Paralelo Computacao (developers of Fly3D) website: `http://www.paralelo.com.br/en/index.php`

[4] 3D Games, Volume One: Real-time Rendering and Software Technology
Authors: Alan Watt and Fabio Policarpo
Addison-Wesley, 2001

The book accompanying the Fly3D SDK 1.0. Gives a detailed overview of the major areas of computer game development and the techniques involved.

[5] 3D Games, Volume Two: Animation and Advanced Real-time Rendering
Authors: Alan Watt and Fabio Policarpo
Addison-Wesley, 2003

The book accompanying the Fly3D SDK 2.0, giving an advanced overview of some more specific areas of game development.

[6] Paper: Cramming more components onto integrated circuits
Author: Gordon E. Moore
Published in 'Electronics', Volume 38, Number 8, April 19, 1965.

Gordon Moore's original paper, published in 1965, in which he first observed an exponential growth in the number of transistors per integrated circuit and predicted that this trend would continue. 'Moore's law' states that the number of transistors that can be put onto a single integrated circuit doubles every two years. This law has been held true since the publication of the paper and is expected to do so for at least a decade to come.

[7] Article: Designing Need-Based AI for Virtual Gorillas
Author: Ernest Adams
Published by Gama Network, December 22, 2000
(http://www.gamasutra.com)

An article discussing the design of need-mechanisms for AI characters, using the approach taken to the design of a 'Virtual Gorilla' simulation as an example.

[8] Series: Fuzzy Logic - An Introduction
Author: Steven D. Kaehler

Published in 'Encoder', the newsletter of the Seattle Robotics Society, March 1998
(http://www.seattlerobotics.org)

A six article series covering fuzzy logic and its application and explaining an example implementation in detail.

[9] Article: What is fuzzy logic?
Author not known
Taken from FAQ: Fuzzy Logic and Fuzzy Expert Systems, http://www-2.cs.cmu.edu, dated 15 April 1993

The article provides a short introduction to fuzzy logic.

A series of six articles

[10] Article: Game AI: The State of the Industry
1999 Edition
By Steven Woodcock (http://www.gameai.com)
Published by Gama Network, August 1999 (http://www.gamasutra.com)

[11] Article: Game AI: The State of the Industry
2000 Edition, part one
By Steven Woodcock (http://www.gameai.com)
Published by Gama Network, November 1, 2000 (http://www.gamasutra.com)

The article discusses the current state and recent developments of AI in the computer game industry, and the Game Developers Conference (GDC) AI roundtable discussions in 2001.

[12] Website: Game Developers Conference
http://www.gdconf.com/

The conference provides an independent forum for expert developers from around the world to share ideas, build skills, and learn about the latest tools and technologies.

[13] Article: Game AI: The State of the Industry
2000 Edition, part two
By David C. Pottinger
Published by Gama Network, November 8, 2000 (http://www.gamasutra.com)

The article discusses upcoming popular techniques for computer game AI, and the gap between game AI developers and academic AI researchers.

[14] Article: Bridging the Gap Between Developers & Researchers
By Prof. John E. Laird
Published in Game AI: The State of the Industry by Gama Network, November 8, 2000 (http://www.gamasutra.com)

An article discussing the various forces which are closing the gap between computer game AI developers and academic AI researchers.

[15] Article: Game Developers Conference 2001: An AI Perspective
Author: Eric Dybsand
Published by Gama Network, April 23, 2001 (http://www.gamasutra.com)

Discusses the AI related sessions from the Game Developers Conference (GDC) 2001.

[16] Article: Nintendo frowns on online
Author: Chris Morris
Published Marh 6, 2002 http://money.cnn.com/2002/03/06/technology/column_gaming/

An article discussing comments from Shigeru Miyamato of Nintendo Corporation on the area of multi player Internet computer gaming.

[17] Article: The Market for Online Games
Authors: Jessica Mulligan, Bridgette Patrovsky
Published by Peachpit, April 4 2003 (http://www.peachpit.com/articles/)

Detailed article discussing the market for multi player Internet computer games.

[18] Article: AI Middleware: Getting into Character
Author: Eric Dybsand
Published by Gama Network July 18, 2003 - July 25, 2003
A series of five articles covering currently available 'AI middleware' or SDK's for game AI.

[19] Website: 'Boids' (A-Life flocking alogrithm) by Craig Reynolds.
http://www.red3d.com/cwr/boids/

Craig Reynolds is the original author of the famous 'Boids' flocking algorithm. The website contains examples, sample code and information links covering all aspects of boids and other group movement algorithms.

[20] Paper: Using Computer Games to Develop Advanced AI
Author: John E. Laird

Discusses the use of computer games as a testbed for sophisticated AI techniques, and the development of the SOAR Quakebot, which uses the SOAR artificial intelligence architecture and the computer game Quake 2.

[21] Paper: The Role of AI in Computer Game Genres
Authors: John E. Laird and Michael van Lent

Discusses the different roles in which AI appears in different computer game genres.

[22] Paper: Developing an Artificial Intelligence Engine
Authors: Michael van Lent and John E. Laird
Artificial Intelligence Lab, University of Michigan

Discusses the application of state of the art techniques in AI research to computer games. In particular the SOAR/Games project which interfaces the SOAR artificial intelligence architecture with various computer games.

[23] Paper: Human-Level AI's Killer Application: Interactive Computer Games
Authors: John E. Laird and Michael van Lent
University of Michigan
Copyright 2000, American Association for Artificial Intelligence (www.aaai.org)

Proposes that AI for interactive computer games is an emerging application area in which the goal of human-level AI can be successfully pursued. The paper discusses research on AI and games, the role of AI in different game genres, potential roles for human-level AI and research issues and AI techniques.

[24] Paper: Intelligent Agents in Computer Games
Authors: Michael van Lent, John Laird, Josh Buckman, Joe Hartford, Steve Houchard, Kurt Steinkraus, Russ Tedrake
Artificial Intelligence Lab
University of Michigan

Discusses the application of research AI techniques to computer games.

[25] Paper: Design Goals for Autonomous Synthetic Characters
Author: John Laird
Artificial Intelligence Lab
University of Michigan

Discusses the evaluation of AI development techniques and characters in computer games.

[26] Paper: Computer Games With Intelligence
Authors: Daniel Johnson, School of Computer Science and Electrical Engineering
Janet Wiles, School of Psychology
The University of Queensland

Discusses the difficulties of developing intelligent characters for computer games, proven techniques and the value of colloboration between the game AI and academic AI communities.

[27] Website: 'Thief: Deadly Shadows', the latest release in the 'Thief' computer game franchise.
http://www.thief3.com

[28] Website: 'Hitman Contracts', the latest release in the 'Hitman' computer game franchise.
http://www.hitmancontracts.com

[29] Website: 'Black and White 2', the latest release in the 'Black and White' computer game franchise.
http://www2.bwgame.com/

[30] Website: 'Half Life 2', the latest release in the 'Half Life' computer game franchise.
http://www.sierra.com/product.do?gamePlatformId=470

[31] Website: 'Max Payne 2', the latest release in the 'Max Payne' computer game franchise.
http://www.rockstargames.com/maxpayne2/

[32] Website: 'The Sims' computer game series.
http://thesims.ea.com/

[33] Website: The 'Havok' dynamic game play SDK.
http://www.havok.com/

[34] Website: 'Unreal' computer game.
http://unreal.com/

[35] Website: 'Civilization: Call to Power' computer game.
http://lokigames.com/products/civctp/

[36] Website: Ion Storm, computer game developer.
http://www.ionstorm.com

[37] Website: IO Interactive, computer game developer.
http://www.ioi.dk

[38] Website: Lionhead, computer game developer.
http://www.lionhead.com

[39] Website: Valve Software, computer game developer.
http://www.valvesoftware.com

[40] Website: Remedy, computer game developer.
http://www.remedy.fi/

[41] Website: Maxis, computer game developer.
http://www.maxis.com/

[42] Website: Epic Games, computer game developer.
http://www.epicgames.com/

[43] Website: Loki Games, computer game developer.
http://www.lokigames.com/

[44] Website: Eidos, computer game publisher.
http://www.eidos.com

[45] Website: Electronic Arts, computer game publisher.
http://www.eagames.com/

[46] Website: Sierra Entertainment, computer game publisher.
http://www.sierra.com

[47] Website: Rockstar Games, computer game publisher.
http://www.rockstargames.com/

[48] Website: Activision, computer game publisher.
http://www.activision.com/

[49] Paper: Squad Tactics: Team AI and Emergent Maneuvers
Author: William van der Sterren
Published in AI Game Programming Wisdom
Charles River Media, Inc., 2002.

[50] Paper: Squad Tactics: Planned Maneuvers
Author: William van der Sterren
Published in AI Game Programming Wisdom
Charles River Media, Inc., 2002.

[51] Paper: Recognising Strategic Dispositions: Engaging the Enemy
Author: Steven Woodcock
Published in AI Game Programming Wisdom
Charles River Media, Inc., 2002.

[52] Paper: Towards a Game Agent
Authors: Christopher Niederberger and Markus H. Gross
2002

Gives a survey on state-of-the-art techniques and academic research in the field of artificial life and emergent behaviour and compares and classifies modern techniques for artificial intelligence in game agents.

[53] Artificial Intelligence: A Modern Approach.
Authors: S. Russel and P. Norvig
Published by Prentice Hall, 1995.

[54] Article: Polygon Soup for the Programmer's Soul: 3D Pathfinding
Author: Patrick Smith
Presented at the Game Developers Conference 2002 and published by Gama Network April 5, 2002
http://www.gamasutra.com/

Discusses how to attain good data on which to run search algorithms for pathfinding, covering the technique for auto-generating sector and portal data and goes on to discuss extending the sector and portal system to more complex problems.

[55] Article: Smart Moves: Intelligent Path-Finding
Author: W. Bryan Stout
Published in Game Developer Magazine, October 1996 and by Gamasutra, February 12, 1999 (http://www.gamasutra.com)

Discusses various well established alogrithms for obstacle avoidance and the pathfinding problem, considering efficiency and the quality of the results produced.

[56] http://ai-depot.com/BotNavigation/Design-Introduction.html

[57] Article: Toward More Realistic Pathfinding
Author: Marco Pinter
Published by Gama Network March 14, 2001
(http://www.gamasutra.com)

The article discusses in depth several techniques for achieving more realistic-looking results from basic, A-star based pathfinding, using techniques from Acitivision's computer game title 'Big Game Hunter 5' which displays 'startlingly realistic and visually interesting movement' for various different animals and considering efficiency and the quality of the results produced.

[58] Article: Postmortem: Raven Software's Star Trek: Voyager - Elite Force
Authors: Brian Pelletier, Michael Gummelt and James Monroe
Published by Gama Network February 7, 2001
(http://www.gamasutra.com)

An article detailing Raven Software's experience developing the title 'Star Tek: Voyager - Elite Force', including difficulties experienced in implementing pathfinding for AI characters.

[59] Definitions of the terms *hill climbing*, *steepest-ascent hill climbing* and *best-first search* when applied to graph searching algorithms in computing, from the website 'The Free Dicitonary': `http://computing-dictionary.thefreedictionary.com/hill%20climbing`

[60] A web page from the website 'MathWorld' giving a definition of the term 'cuboid': `http://mathworld.wolfram.com/Cuboid.html`

The website 'Wikipedia' provides an alternative, equivalent definition: `http://en.wikipedia.org/wiki/Cuboid`

[61] Encyclopedia definition of the 'recursive flood fill', from the website 'The Free Dictionary':
`http://encyclopedia.thefreedictionary.com/Flood%20fill`

[62] Article: Building an AI Sensory System: Examining The Design of Thief: The Dark Project
Author: Tom Leonard
Presented at the Game Developers Conference 2003 and published by Gama Network, March 7, 2003
(http://www.gamasutra.com)

An article discussing the design of sensory systems for computer game agents including simulated vision, sound and smell and using the computer games 'Half Life' and 'Thief' as examples.

[63] Thesis: An Investigation Into the Use of Synthetic Vision for NPC's/Agents in Computer Games
Author: Sebastien Enrique, directed by Alan Watt, The University of Sheffield and co-directed by Marta Mejail, Universidad de Beunos Aires
Universidad de Beunos Aires, Argentina, September 2002

A thesis that examines the role of synthetic vision in computer games and describes the implementation of a synthetic vision module for an agent character in a first-person shooter game based on two viewports rendered in real-time.

[64] Article: Personality Parameters: Flexibly and Extensibly Providing a Variety of AI Opponents' Behaviours
Author: 'Tapper'
Published by Gama Network, December 3, 2003
http://www.gamasutra.com/

An article describing techniques for providing agents with a wide variety of differing behaviours and a sense or personality. The solution presented is claimed to be flexible, robust and easily extensible, and is based on expe-

rience designing solutions for well known computer games titles including 'Men in Black: Crashdown', 'Monopoly Party' and 'Worms 3D'.

[65] Thesis: The Quake III Arena Bot
Author: J.M.P. van Waveren
The University of Technology Delft Faculty ITS, June 28th, 2001

A thesis presenting the design of the computer-controlled agents for the groundbreaking computer game title 'Quake II Arena'. J.M.P. van Waveren is the original designer of the Quake III Arena game agent as well as earlier groundbreaking agents, and the Quake III Arena agent was the first commercially developed artificial player to use fully automated pathfinding through arbitrary, complex 3D worlds without the need for human intervention to provide navigation information to the agent. The agent uses a volume based representation of the environment which has similarities to the representation used in implementation three in this project. A wide range of techniques and common-sense solutions are described which allow the agents to exhibit human-like behaviour within the game environment.

[66] Article: It Knows What You're Going To Do: Adding Anticipation to a Quakebot
Author: John E. Laird
University of Michigan
No date available

An article discussing the design of the SOAR Quakebot and the addition of an anticipation module allowing the agent to predict the behaviour of human players or other agents and to make use of these predictions to its advantage.

[67] Article: A Modular Framework for Artificial Intelligence Based on Stimulus Response Directives
Author: Charles Guy
Published by Gama Network, November 10, 1999
http://www.gamasutra.com

An article discussing a 'stimulus-response directives' approach to AI design for game agent behaviours, which was used in the computer game title 'Spec Ops II'.