HONOURS PROJECT REPORT

Environmentally Aware Game Bots

WESLEY KING
wking@cs.uct.ac.za
Computer Science Department
University of Cape Town

SUPERVISED BY:

PATRICK MARAIS
patrick@uct.ac.za
Computer Science Department
University of Cape Town

SIMON PERKINS
sperkins@cs.uct.ac.za
Computer Science Department
University of Cape Town

	Category	Min	Max	Chosen
1	Software Engineering/System Analysis	0	15	0
2	Theoretical Analysis	0	25	0
3	Experiment Design and Execution	0	20	15
4	System Development and Implementation	0	15	15
5	Results, Findings and Conclusion	10	20	10
6	Aim Formulation and Background Work	10		10
7	Quality of Report Writing and Presentation	10		10
8	Adherence to Project Proposal and Quality of Deliverables	10		10
9	Overall General Project Evaluation	0	10	10
Total marks		**80**		**80**

November 6, 2009

Contents

1 Introduction **1**
1.1 Key Success factors . . . 2
1.2 Ethical Issues . . . 2

2 Background **3**
2.1 Spatial Awareness Framework . . . 3
2.2 Environment Representation . . . 3
2.3 Level of Detail Rendering . . . 4
2.4 Shadow Techniques . . . 5
2.5 Simulation control . . . 6
2.6 User Interaction . . . 7
2.7 Data Visualisation . . . 7

3 Design **9**
3.1 System Overview . . . 9
3.1.1 Base AI and Competition Simulation . . . 9
3.1.2 High Level AI And Environment Querying . . . 9
3.1.3 Rendering and Data Visualisation . . . 9
3.2 Rendering and Data Visualisation Overview . . . 10
3.3 Interaction With Other Project Components . . . 10
3.4 Terrain Representation . . . 11
3.5 GUI Components . . . 12
3.6 Graphics Rendering and GUI Components . . . 13
3.7 Shadows . . . 14
3.8 Level of Detail . . . 15
3.9 Miscellaneous Optimisations . . . 15
3.10 User Interaction . . . 16
3.11 Data Visualisation . . . 17

4 Implementation **18**
4.1 Developement . . . 18
4.2 Heightmap Rendering, Texturing, DIA Conversion and Basic User Interaction . . . 19
4.3 Shadow Mapping . . . 24

4.4 Simulation Control and GUI Components 27
4.5 Vertex Arrays and Chunking 29
4.6 Integrating Prototype Code Into Single Application With All Features 30
4.7 Integrating Code From This Part of The Project With The Other Parts 31
4.8 Visualisation, User Interaction and Additional Features . . 31

5 Testing and Evaluation 38
5.1 Introduction 38
5.2 Required User Testing 38
5.2.1 Rendering Efficiency Testing 38
5.2.2 Data Visualisation Testing 39
5.3 Expected outcomes 40
5.4 Results And Analysis 40

6 Conclusion 44

A User Test Presentation 47

B User Test Questionnaire 56

C User Test Results 58

List of Figures

2.1 2.5D Example: Mt. Saint Helens Rendered In Wireframe https://visualization.hpc.mil/wiki/2.5D_Visualization 3
2.2 Hierarchical Triangulation 4
2.3 Chunking renders the terrain at different levels of detail, depending how close a Chunk is to the camera 5
2.4 3D Graphics libraries do not provide shadow generation and require the developer to implement their own shadowing system 5
2.5 Demonstration of Shadow Mapping technique 6

3.1 Interface between Engine and Simulator 10
3.2 Example of a heightmap 11
3.3 Advanced shadow implementation for a game 14
3.4 The camera can perform pitch and yaw rotations 16

4.1 Examples of possible mesh triangulations 19
4.2 A mesh triangulation that can make use of $GL_TRIANGLE_STRIP$ to reduce the number of vertices 19
4.3 2D example of normal averaging at vertices 20
4.4 3D render without and with normal averaging 21
4.5 Calculation of triangle primitive normals 21
4.6 Testing of different Gimp paintbrush configurations for heightmap generation 22
4.7 Converting a dia map to a heightmap 23
4.8 A heightmap and heightmap-texture 23
4.9 Textured heightmap 24
4.10 Example of multi-texturing: Dirt 24
4.11 Example of multi-texturing: Rock 25
4.12 Shadow Mapping without the second pass, which renders the areas in shadow 26
4.13 A Shadow Map with its corresponding rendered terrain with shadows 26
4.14 Comparison between scene rendered with and without shadows 27

4.15 Example of jumping over vertices to render at a lower level of detail 29
4.16 Chunking in wireframe mode to show the change in detail at different distances 30
4.17 Bot health and status icon. The icon pivots such that it is always facing the camera 32
4.18 Status Icons for each state of the bots 32
4.19 Before and after pictures of centering the camera on a point without changing the orientation of the camera 33
4.20 Before and after pictures of centering the camera on a point without moving the camera 34
4.21 Controls for controlling the simulation 34
4.22 Data and menu options which are displayed when a bot is selected 35
4.23 Paths showing the routes which the bots plan to take to get to the flag 35
4.24 Trail showing the the route taken by the bot as well as its state at each point 36
4.25 The team menu. Only 2 of 4 bots are still alive and the team currently has the flag 36
4.26 Red areas are potentially dangerous as a bot has died there. A Green area is possible advantageous since a bot has been killed from that location 37

5.1 Graph showing the trade-off between Detail and Frames per Second 41
5.2 Paired t-test of the Scores achieved with data visualisations turned on and off 42
5.3 Paired t-test of the Confidence experienced with data visualisations turned on and off 43

C.1 Data recorded during the user test 58

Abstract

This report details the design, implementation, and testing of a system which provides efficient graphics rendering as well as rich data visualisations. The scene being rendered is a simulation of competing bots within a virtual environment and requires users to be able to easily identify features occurring within the simulation. Techniques to increase realism and rendering performance are discussed and a level of detail system, Chunking, is implemented. The methods used for displaying the data to the user are tested via user testing and are found to increase understanding of the simulation by 14%.

Chapter 1

Introduction

A Spatial Awareness Framework[14] system has (and still is) been developed by Simon Perkings. The framework allows for querying of environment specific information of a 2D virtual world. The utility of the framework in the area of game bot design is to be tested. Bots competing within a virtual environment can make use of the framework and use information about the surrounding environment to make more informed decisions on what actions should be taken to beat their opponent. To test the utility of the framework an application wass to be developed which allows a user to run a simulation with different rule files and then make decisions based on the behaviour of the bots. The project work has been split between three group members:

Gina Morris Developed the high level bot decision making, making use of a rule file loaded from disk

Mike Talbot Developed the low level bot actions such as flocking and pathing

Wesley King My section is responsible for the rendering of the simulation and providing the user with data about the simulation in a rich and interesting way

The bots within our simulation compete in a Capture The Flag type game. The game consists of two teams of bots competing against each other. Each team has a base and there is a flag somewhere on the map. The game is won when a bot manages to pick up the flag and return it to its base. The bots can shoot each other and if the flag carrier gets killed, the flag gets dropped. Alternatively If a team manages to kill all the bots on the opposing team, the game is also won. The rest of this report is focused on the rendering and data visualisation component of the overall project.

1.1 Key Success factors

The success factors for this section are designed to ensure that the project can either be deemed a success or failure. Development of the project is directed at fulfilling the following key success factors.

Key Success Factors:

1. Efficiently render the simulation - goal of at least 30 frames per second on low-end hardware
2. Provide data visualisations that aid the user in understanding the simulation for better decision making

1.2 Ethical Issues

The rendering component makes use of the open source library SDL and does not infringe its license (GNU LGPL).

Code for mathematic operations on 4×4 matrices is taken from an example in the OpenGL Super Bible [21] and is reference in the code.

For the user testing that was conducted, receipts of payment were kept for reimbursement. The tests however remained completely anonymous as the a receipt cannot be tracked to a specific answer sheet.

Chapter 2

Background

2.1 Spatial Awareness Framework

The Spatial Awareness Framework[14], developed by Simon Perkins, allows for the querying of environment information. The framework can give information about width, curvature and connectivity within a virtual environment. The Spatial Awareness Framework loads maps stored in a DIA format. DIA is an xml file which holds vector graphics. The environments which the framework uses are limited to 2D. The framework uses the polygons of the map file to generate a skeleton structure. The skeleton holds the environment information and can be queried. The Spatial Awareness Framework could be used to provide better rule systems for bots interacting within a virtual environment.

2.2 Environment Representation

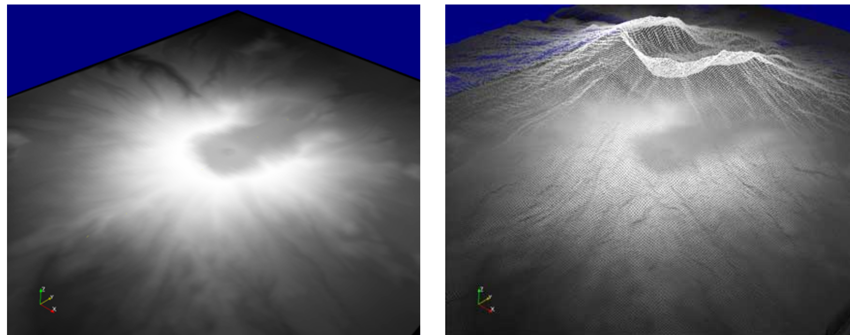


Figure 2.1: 2.5D Example: Mt. Saint Helens Rendered In Wireframe https://visualization.hpc.mil/wiki/2.5D_Visualization

Virtual environments are represented in one of three ways: 2D, 2.5D and 3D. In a 2D environment the world is represented as a grid. Each cell of the grid holds information about the environment at that point. 2D environments cannot represent real world terrains as they are limited to only 2 Dimensions. A 2.5D environment also makes use of a grid, but each cell has the addition of a height value. A 2.5D environment extends into all 3 dimensions but is limited to only having one height value per cell of the grid (Figure 2.1). Although a 2.5D environment can represent many real life terrains, it cannot display features such as caves. 3D environments allow for complete flexibility of what types of terrains can be represented, however a 3D environment is difficult to define and work with. The grid type nature of 2.5D environments makes them popular for use in games.

2.3 Level of Detail Rendering

Rendering large numbers of graphics primitives naively is computationally expensive. There is an overhead due to the large number of API calls as well as a limitation on the bandwidth between the CPU and GPU. Two faster solutions are proposed by Louis Bavoil [9]. The first solution is a vertex array, which is an array of vertices which can be sent to the API in one function call. The second solution is a display list, which allows for precomputation on static graphics objects.

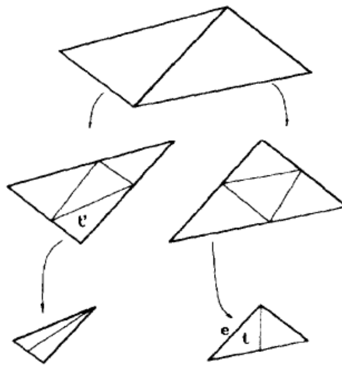


Figure 2.2: Hierarchical Triangulation

L. De Floriani and E. Puppo describe a level of detail method in [9] which uses a Hierarchical Triangulation (shown in Figure 2.2) to store triangle meshes. Level of detail systems allow for the detail of a scene to be dynamic, resulting in less graphics primitives where they are less needed. The hierarchy is designed such that greater detail can be achieved by traversing deeper down the tree and using more

triangle primitives. The hierarchy is constructed to reduce the error between the approximation and the original triangulation. This allows for the detail of the terrain to be dynamic as more detail in a region is required. Hierarchical Triangulation is complex and by using this technique you can't use display lists as the mesh is no longer static.

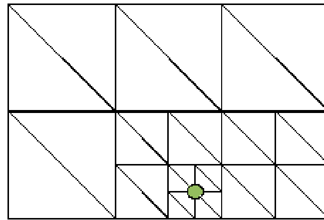


Figure 2.3: Chunking renders the terrain at different levels of detail, depending how close a Chunk is to the camera

A simpler level of detail system is Chunking [15][12]. Chunking divides the environment up into cells. Each cell holds its allocated area at different levels of detail. To render the scene all the cells are rendered, and the distance to each cell determines the level of detail that the cell will be rendered at. Closer cells are rendered at greater detail. This results in a greater number of triangle primitives closer to the camera resulting in greater detail where it is needed (Figure 2.3).

2.4 Shadow Techniques

Figure 2.4: 3D Graphics libraries do not provide shadow generation and require the developer to implement their own shadowing system

Realism can be added to a scene by adding shadows. A technique sometimes used in video games is Fake Shadows [20], where only shadows projected onto the floor are taken into consideration. This technique increases efficiency because you only need to project onto a

plane. We could use fake shadows in our simulation because the Spatial Awareness Framework assumes a 2-Dimensional environment. F. Crow [8] suggests that the best way to do lighting is by using Shadow Volumes. Shadow Volumes store shadow data as invisible 3D objects in the object space. To determine if a point is in a shadow or not, a shadow count is calculated from the camera source. When a shadow volume is entered the count is incremented and when it is exited it is decremented. If the final shadow count is 0, then the point is not in any shadow.

P. Atherton, K. Weiler, and D. Greenberg [4] present a way in which shadows can be generated using transformations and hidden surface algorithms. In this method the scene is transformed to the view of the light source. Using hidden surface algorithms, polygons which are completely shadowed are removed. The remaining polygons are added to the original scene and used for surface detail calculations. This method is difficult to implement due to the hidden surface removal, but provides great advantages where knowing shadowed and illuminated areas is useful in a polygon form [20] [4]. Our project has no use for knowing shadowed areas in a polygon form and both calculating shadow volumes and doing hidden surface removal are complex. A simple and practical solution to shadowing is Shadow Mapping [20][17][5]. Shadow Mapping involves rendering the scene from the light sources' point of view. Data can then be extracted from the depth buffer (z-buffer) and stored in a texture so that when the scene is rendered from the camera's point of view, points which are visible from the light source can be illuminated.

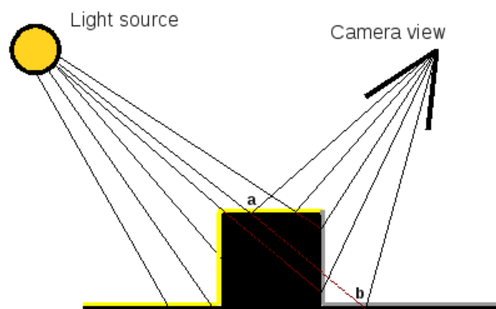


Figure 2.5: Demonstration of Shadow Mapping technique

2.5 Simulation control

Various design models exist for the implementation of a simulation loop[19]. The simulation loop is responsible for updating the simu-

lation to the next step in time, update(), and re-rendering the scene, render(). Two models are presented by L. Valente, A. Conci and B. Feijo [16], Coupled and Uncoupled. The Uncoupled Model differs from Coupled by allowing for asynchronous execution of the update() and render() methods. The advantage of the Uncoupled Model is that maximum frames per second can be reached, whilst keeping the simulation running at a constant speed. This makes the simulation deterministic. Determinism means that if the simulation is started with the same initial starting state, it will follow the same behaviour [10]. To record or playback a simulation, only the initial conditions need to be recoded oppose to storing the data at each time step.

2.6 User Interaction

An easy way for the user to interact with the environment and the agents is to follow the Real Time Strategy Game Paradigm described by H. Jones, and M. Snyder[11]. This paradigm has the user as a superior over autonomous agents. The view is usual 3-Dimensional and places the user above the agents with a relatively large area of the environment in view. Agents are represented by avatars and the user can click to select single agents or click and drag to select multiple agents, in order to interact with them. When an agent or agents are selected, the user is provided with options for those agent(s) on the Heads Up Display. The user can pan the environment by moving the mouse cursor to the edge of the screen, in the direction they wish to pan. Users are much better at navigating a 3-Dimensional interface when given a global overview of the space [18]. This idea supports the need of a mini-map which is often found in RTS games.

Jakob Nielsen's Ten Usability Heuristics [13] suggests that a system should include visibility of the system status and provide the user control and freedom. These heuristics are highly satisfied by the Real Time Strategy Game paradigm [11].

2.7 Data Visualisation

Being able to transform raw data into a graphical format allows users to better understand the data and often interpret unexpected results from the data [7]. Real Time Strategy Games have very little information which they need to show for the game agents. This information is usually the agents current health and a symbol to represent the agent's state, on guard, attacking etc. This data is small enough to be shown above each agent's head. For our data visualisations we will mostly want to show data that has some kind of space value. A user might

want to see what areas of the map a specific agent has covered, or what areas of the map most of the kills have taken place etc.

The Visual Information Seeking Mantra is ”Overview first, zoom and filter, then details-on-demand” (B. Shneiderman). By borrowing the RTS paradigm, a lot of the mantra is achieved for the general interface. An overview is presented to the user in the form of a minimap where quick navigation can take place. The user if often able to control the zoom of the camera and can choose to select only a certain type of unit by double clicking one of that type. Details about a specific unit can be accessed by selecting the unit. This mantra was extended in [6] to include the steps Relate, History and Extract. The user needs to be able to compare multiple pieces of data to find relations. An example of this would be simultaneously visualising the areas which each team has covered. History allows for data to be recorded and replayed at a later stage and Extract allows the user to save data produced by the system.

Chapter 3

Design

3.1 System Overview

The project has been split up into three components. Each component is designed to stand on its own, allowing for its success to be evaluated without relying on the other parts. This report is based and focused on the Rendering and Data Visualisation component. These project components are described below:

3.1.1 Base AI and Competition Simulation

This part of the project is responsible for running the competition between the two teams of Bots. It also provides the bots with low level AI functionality. This includes path-finding and flocking.

3.1.2 High Level AI And Environment Querying

This part of the project provides a logical rule system for the agents. It allows for different rules to be loaded into the bots for testing and comparison. The high level rules govern how the low level behaviour is carried out. This part is also be responsible for interfacing with the Spatial Awareness Framework to allow for rules to be made which include environment data.

3.1.3 Rendering and Data Visualisation

This part is responsible for the rendering of the environment and the bots. It also needs to provide the user with functionality to access information about the simulation.

3.2 Rendering and Data Visualisation Overview

There are four main modules making up this component of the project:

- Rendering of the terrain
- Rendering of GUI components
- User interaction
- Data visualization

This component has been broken down into these modules so that prototypes of each can be developed and tested independently.

3.3 Interaction With Other Project Components

The rendering component of the project needs to interact with the simulator. The game/simulation loop will be responsible for getting user input, calling the simulator to update the simulation state, and finally rendering the graphics. The loop controls these functions, such that the simulation updates at a steady rate, and the rendering can be performed at the maximum number of frames per second. It calls the simulator's update function, and queries the simulator so that it can render the bots and flag as well as any additional information stored by the bots.

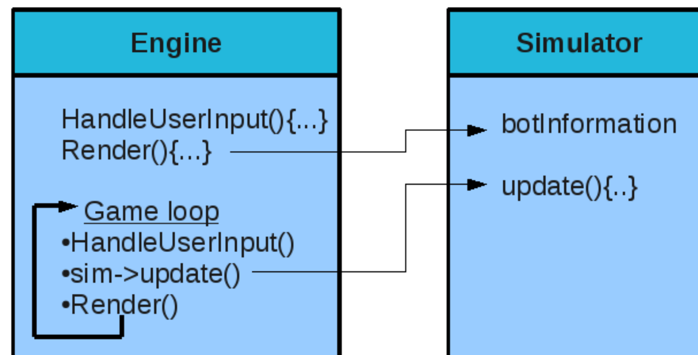


Figure 3.1: Interface between Engine and Simulator

3.4 Terrain Representation

There are three ways in which the terrain can be represented. The simplest is a 2-Dimensional representation which would consist of a 2D grid. Each cell of the grid would hold information such as whether it is walkable. A 2D environment can be rendered extremely efficiently, but it is restricted in that it is not visually appealing and does not match a real world environment. The lack of authenticity to real environments could hinder the analysis of the bots behaviour. A 2.5D representation also consists of a 2D grid, with the addition of a height value for each cell. A 2.5D environment provides greater realism and can represent many real world environments. A 2.5D environment however cannot represent caves or overhangs because each cell only has one height value. The grid nature of the 2.5D representation is beneficial to the simulation as it provides a 2D co-ordinate system for the positioning of objects within the environment. A 3D representation provides complete flexibility with regards to what environments can be rendered, however this method is complex in the way in which it is stored and interacted with. Calculating collisions with terrain in 3D is too computational expensive.

The terrain used in the Spatial Awareness Framework is all 2-Dimensional and the framework only provides querying of 2-Dimensional environment data. A logical representation of the terrain for rendering is therefore either 2D or 2.5D. We will use a 2.5D representation for the rendering, but the simulation will use the terrain as if it were in 2D.

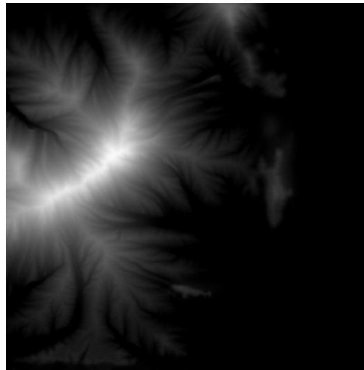


Figure 3.2: Example of a heightmap

The Spatial Awareness Framework uses the terrain stored in a dia file format. The format specifies the terrain as a collection of polygons, each stored as a collection of points. The dia format is therefore a vector format. Each polygon in the file defines an area which is non-walkable by the bots. To convert the terrain to a 2.5D heightmap 3.2,

it needs to first be rasterized to a grid. Rasterization is the process of converting vector graphics into a discretely defined format, generally performed by sampling points. This can be done by parsing the dia file and drawing the polygons into an image. The image then needs to be manipulated to provide realistic elevation features. The height for each cell can be calculated using an algorithm for terrain generation or can be done by hand using an image editing program. There are many terrain generation techniques which use things such as Perlin noise and fractals. Such an algorithm would however need to take into account the walkable and non-walkable parts of the terrain. This would add a lot of complexity to the project and will rather be left as an extension. The rasterized dia file will rather be edited manually to create realistic height maps.

3.5 GUI Components

The interface for the application will require GUI components. These components are required for controlling the simulation as well as for moving around the virtual environment. The GUI components enable the user to activate various data visualizations to find out more information about the simulation. The most basic GUI component will be a button object. The object will be given callback functions which will be called on specific events such as mouse-over, mouse-click and mouse-release. The button will also require textures for each of its states: released, mouse-over and pressed. The basic button object can be extended to created checkboxs. A check box would have additional states, and will be linked to a boolean variable for its current checked/unchecked state. More advanced components may be required such as scroll bars. These will be a bit more complex than a basic button. All GUI components for a current window will be stored in a menu object. The current menu object can therefore be changed to change what controls are available to the user.

To determine whether an object has been interacted with, a linear search over the objects on the current menu will take place. If the mouse is over an object, its events will be called depending on what operations the user is performing. This process can be optimized using a quadtree for quick elimination of objects which are not being interacted with. A quad tree recursively divides the screen area into 4 quads forming a tree structure. All the GUI components are inserted into the quad tree when they are created. The location of a component is determined by which quads they are inside or overlap. To test what component is being interacted with, the quadtree is traversed only following the nodes which the mouse is lying in. The benefits of a quad tree will only be gained for a large number of GUI objects, due to the overhead of using a quadtree.

3.6 Graphics Rendering and GUI Components

Developing our own renderer gives us more freedom for customizations and optimizations which can be implemented. For the rendering to be a success it needs to run efficiently on basic hardware. The overhead of a sophisticated engine could lead to slow rendering, poor customization for data visualizations, and difficult integration with the Spatial Awareness Framework. OpenGL[1] was chosen for the rendering of the graphics in our application, using SDL[2] as a container. OpenGL is cross platform and allows for easy compilation for different architectures if necessary. SDL was chosen over Freeglut[3], as it is more advanced and provides external libraries for tasks such as image loading.

Naive simulation loops make the simulation and rendering dependent on the speed of hardware it is being executed on. With basic time management, it is possible to achieve a loop that keeps the simulation at a constant rate, however this fixes the render rate to the same frequency. We will use a more advanced simulation loop that is designed to get the maximum rendering frames per second, whilst keeping the update rate of the simulation constant. Furthermore we will allow for a cap on the maximum fps of the renderer. Rendering above a certain fps will not add any benefit to the user.

To use vertex arrays we are required to build an array of the vertices of the heightmap as well as building an index array for the ordering in which they form the triangulation. Using indices also means there is less redundancy where vertices are used by multiple triangles. This reduces the number of vertices sent to the graphics card and increases performance.

The normals of the triangles are needed for performing lighting calculations by OpenGL. The normal at each point of the triangle mesh will be taken as the average of the normals of its surrounding triangles. This helps give the terrain a natural look as the edges between triangles are not obvious due to different lighting conditions.

Multitexturing is used to allow for the merging of multiple textures over the terrain to increase realism. The first layer provides a base colour for each point. A one dimensional texture allows for a mapping of a height onto a colour. This technique has been extended and also takes into account the slope of the terrain at the given point. The height and slope values are used to lookup a colour in a two dimensional texture. Slight randomness to the position in the texture adds more randomness to the terrain and increases authenticity. A high-detail texture is overlayed over the basic colouring providing a sharper more realistic and rugged terrain.

3.7 Shadows

Shadows on the terrain add realism and also allow for shadow information to be queried by the bots as a possible extension. While OpenGL does provide lighting functionality it does not provide shadows. With lighting objects in the scene will not be darkened when they are obscured from a light source.

Shadows can be implemented using either Shadow Mapping, raycasting, or generated by hand. Raycasting is computationally expensive, but is a possible non-realtime solution for static objects in the scene. For example the terrain. Raycasting projects a line from the light source to each vertex. If the line reaches the vertex before hitting any other geometry, the vertex is regarded as lit as there is nothing obscuring it from the light source. The complexity of calculating the geometry intersection is undesirable, especially since the technique will not function in real-time.

An easier technique is to generate the shadow file by hand and simple texture it onto the terrain using multitexturing. Generating the shadows by hand can be slow and inaccurate, and also only provides shadows for static objects in the scene.

Figure 3.3: Advanced shadow implementation for a game

We will implement a better technique known as Shadow Mapping, which is efficient enough for realtime lighting on dynamic objects. Shadow Mapping works by performing an additional rendering pass of the scene from the point of view of the light source. This limits Shadow Mapping to directed lighting. After the render of the scene, the depth buffer is stored into a texture. During the render of the scene from the camera's point of view, the texture is used to determine whether a vertex is lit or shadowed. If the distance from a vertex to the light source is greater than the distance stored in the texture for that vertex, then the light must have been obscured before reaching the fragment and is therefore in shadow. This technique is similar to ray casting, but takes advantage of being executed on the graphics card in parallel.

3.8 Level of Detail

Level Of Detail systems reduce the amount of polygons or quality of texturing as the detail of the scene becomes less beneficial on the quality of the final rendered image. A classic example of this is for objects which are far away. Using less vertices will have little or no effect on the final rendered image, however as the object gets closer to the camera more vertices should be used. The performance gained by using fewer vertexes is achieved by minimizing the bandwidth used between the CPU and GPU. A level of detail system can be easily implemented if the nature of the geometry can be resembled as a fractal. The fractal can be iterated deeper as more detail is needed. This could even be done to an infinite depth. Our terrain however is generated by hand and cannot exploit any fractal nature. Generating the terrain using an algorithm has already been mentioned as an extension, and a further extension would be making this algorithm produce a fractal landscape.

A level of detail scheme called Chunking can be used. Chunking involves splitting the terrain up into a grid, with each cell, or Chunk, being allocated an area of the terrain. The Chunk holds that section of the terrain at different levels of detail. When rendering the terrain, each Chunk will get rendered but it's detail will be dependent on its distance from the camera. An algorithm needs to be implemented to down sample the terrain held by each chunk. This can be done by using a blurring function such as a Gaussian Blur. We will use a more privative method of skipping over vertices as there is a performance gain which can be utilized. Only the indices of the vertex array need to be different for different levels of detail. So for each level of detail the same vertex array will be used, but with different indices.

A problem with chunking is the need for seams. If two adjacent chunks are rendered at different qualities, the vertices at their common edge will not align perfectly. The most common solution is the addition of seams, which consist of additional geometry to patch up the gaps.

3.9 Miscellaneous Optimisations

Reducing the number of vertices reduces the bandwidth required between the CPU and GPU and increases performance. There is an advantage in removing vertices which will not affect the final rendered image and not send them to the GPU at all. To do this you need to know which areas of the terrain will not appear in the screen space after the transformations on the vertices. The splitting of the terrain into Chunks makes this process a bit easier. The corners of the screen can be projected onto the terrain producing a quadrilateral. Any chunks which lie outside of the quad will not need to be rendered. This technique will have an overhead and may not be beneficial for some camera

angles. The most commonly used view for our application will be looking down. This view will exploit this technique to its full potential and if the camera angle is changed the technique can be automatically turned off.

A problem with this technique is that it assumes 2-dimensional data is being rendered. It is possible that the base of a hill is not within the quadrilateral, but the top of the hill is within the view frustum. The hill will therefore not be rendered when it is suppose to be. This may not be a problem since the actual simulation is taking place in 2D. The hill can be seen as being removed for obscuring the important areas of the terrain.

3.10 User Interaction

The user needs to be able to easily move around the environment to analyse the behaviour of the bots. Movement can be either done using the traditional real time strategy paradigm where the user can pan around the terrain using the keyboard or moving the mouse to the edge of the screen where they wish to move to. Movement can also be done in flying type style where moving the mouse changes the camera direction. Pushing keys then moves the camera forward or backwards.

The camera position and orientation is defined by 3 values for position (x,y,z) and two for orientation (theta, phi). The theta angle is the rotation of the camera around the z-axis, or yaw. Phi is the angle between the environment plane and the camera direction, or pitch. The system is simplistic, but allows for all possible camera positions and orientations that will be required. A more advanced technique would be to use quaternions. This would allow for an additional camera movement, a roll. The added complexity of quaternions is unnecessary for the camera orientations required for this project.

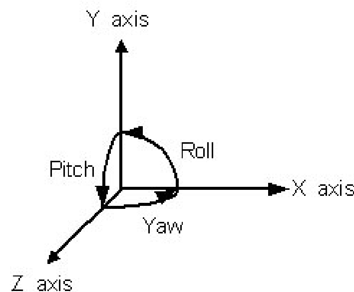


Figure 3.4: The camera can perform pitch and yaw rotations

The user needs to be able to select bots or at least be able to select a

single bot. The position of the mouse in world space can be calculated by unprojecting the position with the model-view matrix. The closest bot is then selected. A quad tree can increase the performance of selecting a single bot. For selecting multiple bots, both the start position of the click and the current position of the click need to be unprojected. These two points in world space along with the camera orientation will define a quadrilateral. All bots within the quadrilateral are selected.

3.11 Data Visualisation

Information about the bots needs to be displayed to the user. This is done by overlaying the terrain with information which has a space value. An example would be to mark areas in red where bots have been killed. To do this a texture of the information needs to be overlaid onto the terrain. Modifying the texture will require texture pixel access. OpenGL does not support this, but SDL can be used to modify the texture which will then be used by OpenGL. Pathing data relating to the bots can be shown as a series of connected lines on the terrain. Two types of paths need to be shown:

1. The path which a bot is planning on taking
2. The path which a bot has taken

Displaying the path that a bot wishes to take, provides extra information to the user and assist in the process of determining which rules produce better strategies. A bots path may indicate that it is avoiding a certain area of the map. Displaying the path which a bot took assists the user in determining what areas of the environment the bot was in the most. It also shows where bots changed their tactic. By viewing the path taken, the user can get an understanding of what the bot was doing over a period of time. When a bot is attacking another bot, a red line between the bots is drawn. This is a simple yet commonly used technique in games. Without this visualization it would be difficult to identify who a bot is attacking when they are in an attacking state.

Each bots status and health is indicated above their heads. The health is indicated by a green and red ring. Green represents current health and red represents lost health. The health indicator assists the user in seeing how a bots behavior changes depending on the bots current health. The bots state is indicated as an icon directly above its head and is rotated so that it is always visible to the user.

When a bot is selected addition information about the bot is displayed. A percentage bar for each state indicates how frequently the bot was in each state. Addition buttons are displayed to allow for the turning on and off of path rendering for the selected bot.

Chapter 4

Implementation

4.1 Developement

This part of the project was implemented using a prototype development model. The work which was meant to be completed was split up into sections. Each section has a main focus. The first few steps of development involved developing these sections of work into mini stand alone prototypes. The motivation for this is that the techniques required by each section can be tested and experimented on independently. This is advantageous since developing a single prototype would make it difficult to determine where errors are being generated.

The 4 prototypes are:

1. Heightmap Rendering, Texturing, Dia Conversion and Basic User Interaction
2. Shadow Mapping
3. Simulation Control and GUI Components
4. Vertex Arrays and Chunking

Once the code for each section is developed, It can supposedly easily be integrated into one application which has all of the required features. This can be done since the techniques will now be familiar and a "best way forward" established. The code from the prototypes serves as a reference and most code can be copied straight. The code from this part of the project then needs to be merged with the code from the other parts being developed separately.

The final 3 steps of developement are:

5. Integrating prototype code into single application with all features
6. Integrating code from this part of the project with the other parts
7. Visualization, User Interaction and Additional Features

4.2 Heightmap Rendering, Texturing, DIA Conversion and Basic User Interaction

Triangles are a common primitive used for rendering by graphics APIs. To render the heightmap the area needs to be divided into triangles to form a triangle mesh. It would make sense that each vertex of the triangle mesh represents a height found in the heightmap. The triangle mesh can be generated in multiple ways.

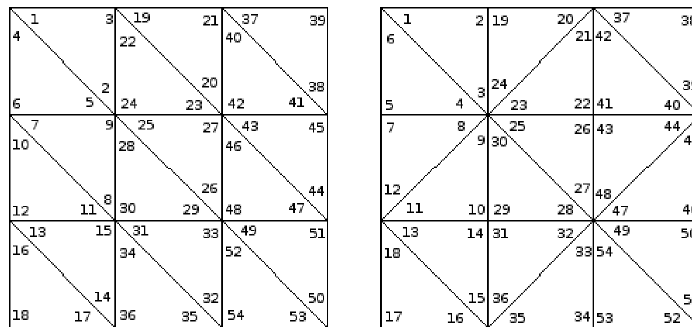


Figure 4.1: Examples of possible mesh triangulations

Instead of drawing each triangle individually (using $GL_TRIANGLES$) OpenGL provides a $GL_TRIANGLE_STRIP$ API call. This enables the drawing of multiple triangles where the last 3 given vertices form a triangle. This reduces the number of API calls and makes using certain triangulations beneficial.

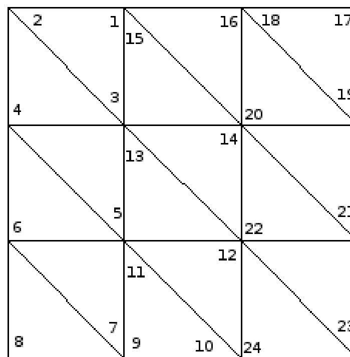


Figure 4.2: A mesh triangulation that can make use of $GL_TRIANGLE_STRIP$ to reduce the number of vertices

Each vertex is to be given a height value to represent the desired terrain. From the above triangulation it can be seen that vertices have common height values (2 and 10 for example). The height values are therefore read into a 2d array of the desired terrain dimensions. When the triangle vertices are created, their height is looked up in the array. The SDL Image library is used for loading the height values. First a heightmap image is loaded into memory. The height array dimensions are determined using the dimension of the image, such that each pixel represents one height value. The pixels are then iterated over and the height for each (x,y) coordinate is stored. The height is determined by how light/dark a pixel is. The lighter the pixel the higher the value. This requires the image to be in grey-scale so a conversion is first calculated using the formula:

```
grey = (red*11 + green*16 + blue*5)/32
```

This gives a height value in the range [0,255]. Further operations can be formed on the height such as scaling, where the height is multiplied by a constant, and inversion where the height is set to the maximum value minus the loaded value. For this iteration I used a Display List to render the terrain. This was easily implemented by encapsulating the rendering of the terrain within the required API calls to compile the geometry for faster rendering. For lighting operations on geometry OpenGL needs to know the normal for each triangle. The normals can be, and are by default, calculated automatically using GL_AUTO_NORMAL. Using the automatic normal generation gives each triangle one normal which is used for all three of its vertices. This makes the triangle boundaries obvious as each triangle is being coloured differently by the lighting operations.

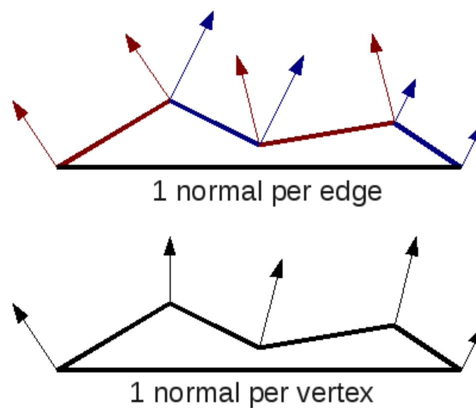


Figure 4.3: 2D example of normal averaging at vertices

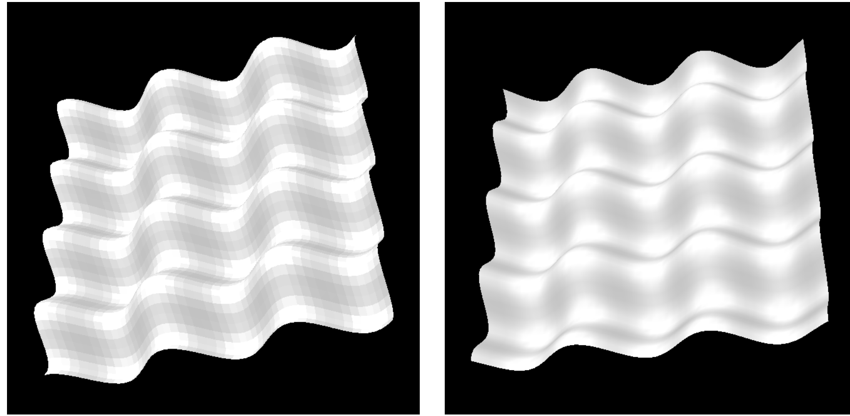


Figure 4.4: 3D render without and with normal averaging

A more continuous look can be achieved by ensuring that vertices at common points have the same normal (Figures 4.3 & 4.4). A 2d normal array can be used much like the array used to store a single height value for each (x,y) co-ordinate. To calculate the normal at each vertex of the triangulation mesh, the average of the normals of the surrounding triangles is used. The normal for each triangle therefore needs to be calculated manually, which is easily done by taking the cross product of the unit vectors $\overrightarrow{ac}$ and $\overrightarrow{ab}$ (Figure 4.5). The cross product is an expensive operation but only needs to be calculated once when the heightmap is loaded and then the normals are stored.

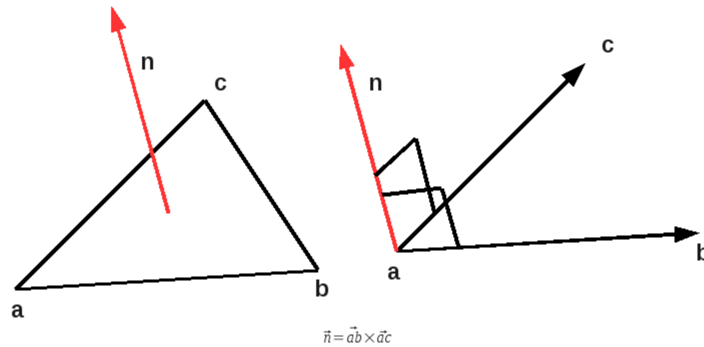


Figure 4.5: Calculation of triangle primitive normals

The terrain is generated from a heightmap image, however the map format used for the skeleton structure is a dia file and the pathing makes use of a similar dia file containing a triangulation of the walk-

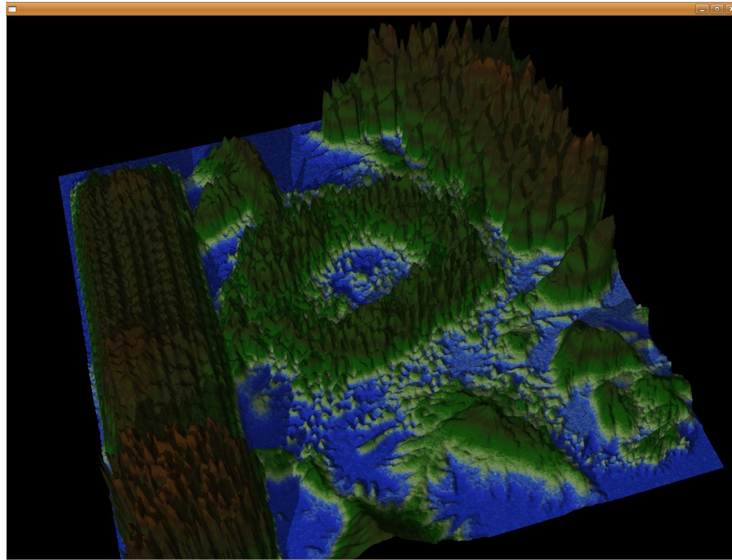


Figure 4.6: Testing of different Gimp paintbrush configurations for heightmap generation

able area. To get these into a rasterized image format, such as png, a python script making use of qt was developed. The script parses the dia file, identifies the polygons, and draws them into an image of desired dimensions. The final image is then saved as a png. The png contains a black and white image showing the walkable and non-walkable areas of the terrain. This image is then edited using an image editing program, such as Gimp (examples 4.6), to construct a realistic heightmap (Figure 4.7). The script requires two passes as the dia file does not specify a boundary for the terrain. The first pass calculates the top-left most and bottom-right most co-ordinates. All points are then taken relative to those co-ordinates and scaled to the desired dimensions.

Heightmaps are commonly textured using a 1d texture, where the height of the point determines the position in the texture. The advantage of this method is that it can be used for any heightmap that is loaded, where as a 2d texture overlayed on the terrain might not make sense. Example: A lake in one position might make sense on one heightmap but not for another. The problem with 1d textures is that it leaves the terrain looking bland and cartoon-like due to the low level of detail. To add more interest to the terrain, we used a 2d texture in a similar way in which a 1d texture would be used. The height of the point determines the y co-ordinate in the texture, and the x co-ordinate is calculated at random. More interest is created by perturbing the y co-ordinate slightly by a random amount and clamping the value to

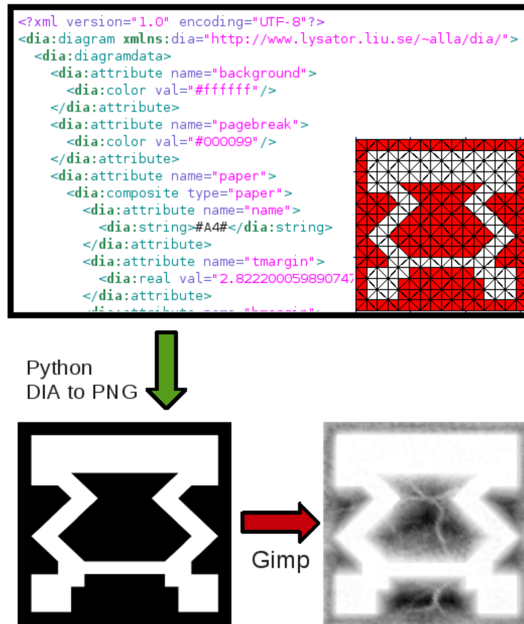


Figure 4.7: Converting a dia map to a heightmap

ensure it does not overflow.

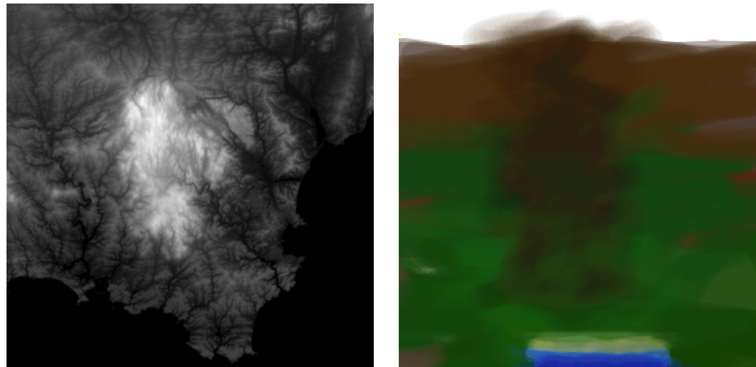


Figure 4.8: A heightmap and heightmap-texture

Multitexturing allows for adding additional textures to the terrain. It can greatly increase the realism of the terrain by adding high level detail features such as dirt, or rock features. Textures in multitexturing can be combined in different ways to give different results. GL_BLEND may be used to place a semi-transparent texture over the current ter-

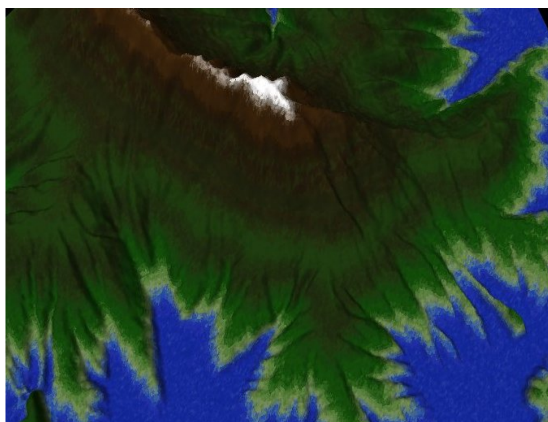


Figure 4.9: Textured heightmap

rain, or *GL_DECAL* can be used to completely replace the underlying texture with the current one. *GL_BLEND* can be used to overlay features such as dirt, rock or sand which cover the entire terrain and are not specific to a single heightmap. *GL_DECAL* can be used to overwrite areas of the terrain to add additional features which are specific to that terrain for example: rivers and lakes.

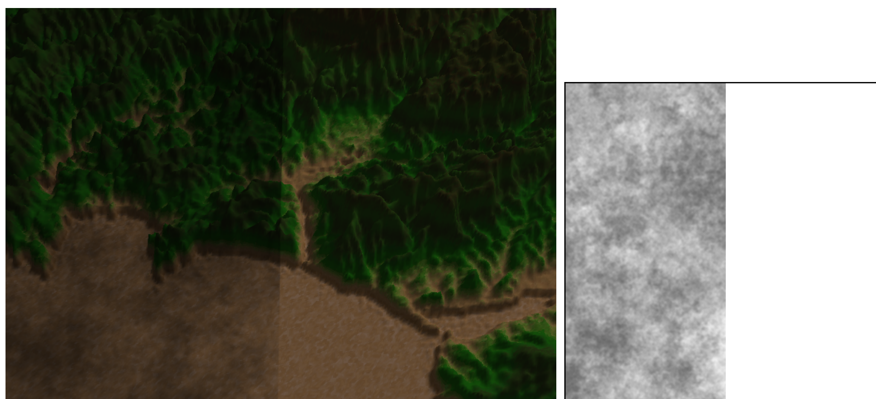


Figure 4.10: Example of multi-texturing: Dirt

4.3 Shadow Mapping

Shadow Mapping was implemented with reference to the Shadow Mapping example in the OpenGL SuperBible [21]. The code from the OpenGL SuperBible was first ported to SDL for easier referencing before being

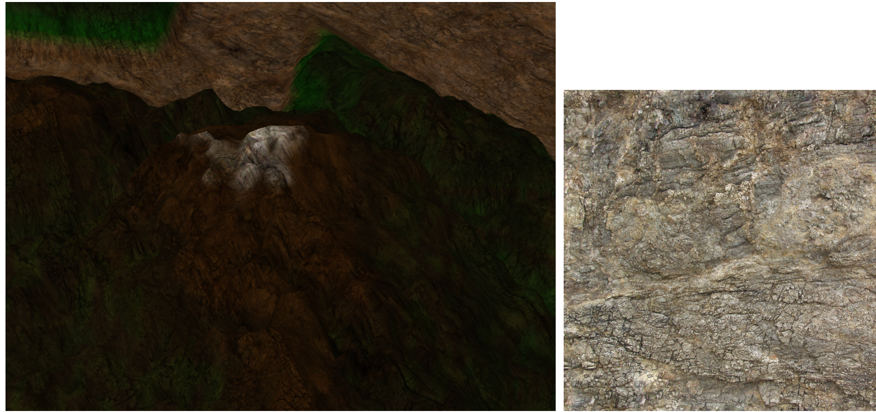


Figure 4.11: Example of multi-texturing: Rock

implemented. The code for mathematic operations on 4×4 matrices has been used directly and is referenced in the code. There are three rendering passes required to generate one render which the user will see:

1. Calculates the distance from the light source to each point in the scene that it illuminates
2. Renders the entire scene from the cameras point of view using a low ambient light. This is to render the parts of the scene which are in shadow.
3. Renders the scene again from the cameras point of view, but uses the distance of each point to the light to determine whether it is to be rendered. This render uses a brighter lighting configuration and overwrites shadowed areas which should be lit.

The scene first needs to be rendered from the light source's point of view. The difficulty with this is that you need to ensure that the scene fits entirely into the view frustum. If the scene is too small however, it will result in the shadows being at a low resolution since there will be a smaller area per pixel. To ensure that the scene fills the frustum optimally, the best frustum size is calculated using the details of the distance of the light source as well as the bounding radius of the scene. The viewport can theoretically be increased to increase the detail of the shadows however experimentation showed that the viewport gets limited to the SDL window size. When the scene is rendered only the depth information is needed and all texturing and lighting can be turned off. After the render is completed, the depth buffer (distance to the geometry for each pixel) is saved into a texture using $glCopyTexImage2D()$. A texture matrix is then calculated using the light sources position and

orientation to allow for projection of the depth texture back onto the scene in a later render.

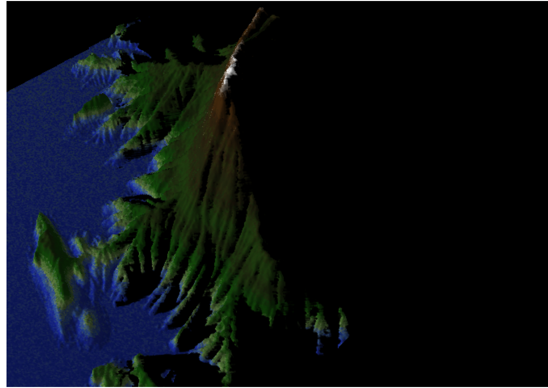


Figure 4.12: Shadow Mapping without the second pass, which renders the areas in shadow

The second pass is simply a render from the cameras point of view with a low ambient lighting condition as well as texturing turned on. Without this pass, areas in shadow would not be rendered and will appear as holes in the terrain. Hardware which supports the $GL_ARB_shadow_ambient$ extension can skip this pass and incorporate it into the next pass. This extension was not supported on any of the development hardware and is therefore not used.

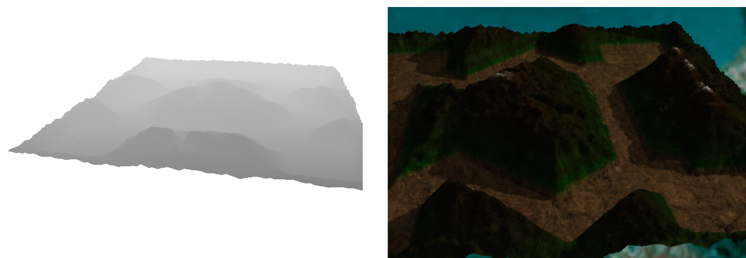


Figure 4.13: A Shadow Map with its corresponding rendered terrain with shadows

The final pass renders the scene again, but overwrites areas which are to be lit. The texture holding the depth information from the first pass is used to determine if a pixel is to be lit or not (Figure 4.13). For each pixel in the rendered scene its distance to the light source (B) is compared to the distance stored in the texture (A). If B is greater than A, there must be geometry obscuring the pixel from the light source and the pixel is not rendered. If B is not greater than A the pixel is

"lit" and is therefore rendered. The comparison to the texture is done using $GL_COMPARE_R_TO_TEXTURE$. The texture is projected onto the scene for the comparison and the scene is finally rendered using the "lit" lighting conditions.

For better performance shadowing can be turned off. Without shadows only one rendering pass needs to be done, where the entire scene is rendered once with the "lit" lighting conditions. If only static geometry is to cause shadows and the light position is static, an optimisation can be made. During each render from the light source, the terrain won't be changing. This render is therefore only performed once at the beginning of the application. The shadow map texture is still used, but does not change during its execution. for the initial generation of the shadow map, the bots are not rendered else their shadows will be shown in their starting positions even after the bots have moved.

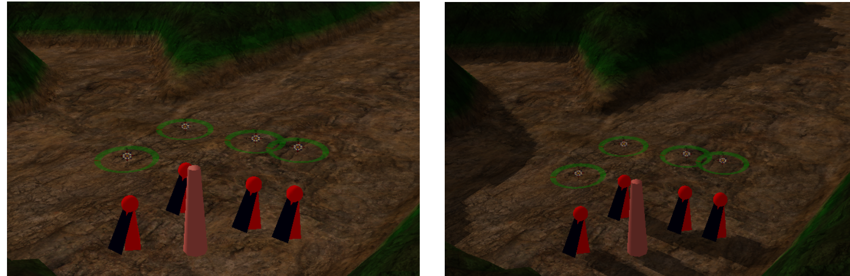


Figure 4.14: Comparison between scene rendered with and without shadows

4.4 Simulation Control and GUI Components

A difficulty encountered is that there are 3 processes happening in the application at different rates and with different conditions. Rendering needs to happen as fast as possible, The engine needs to update at a steady rate, and the simulation needs to update at a steady rate which is different from the engine and must also be dynamic. The rendering process is naturally dynamic since it will be happening as fast as possible. The engine needs to be at a constant speed to ensure that camera movement, user interaction, etc. remains consistent regardless of the hardware or speed at which the simulation is executing. The simulation speed needs to be consistent to ensure that it is not dependent on the hardware, and also needs to be dynamic so that the user can speed up and slow down the simulation.

These features where achieved by having timing related variables for the engine and simulation update processes. The time for the next

call of the process is stored and on each iteration of the game loop, the process is called if the time has been reached. if a process is called, the next update time is recalculated. On every iteration of the loop the rendering process is called.

A problem however occurs when rendering is slow, as a lag effect is carried over onto the engine update and simulation update processes. The solution implemented to get around this is to allow for multiple updates of the processes during a single game loop iteration. The functions are called until they have caught up to their required time.

```
double next_sim_update = SDL_GetTicks();
double next_engine_update = SDL_GetTicks();
double current_time = SDL_GetTicks();
while (running)
{
    current_time = SDL_GetTicks();
    if (current_time > next_sim_update)
    {
        sim->update();
        next_sim_update += SIM_UPDATE_RATE;
    }
    if (current_time > next_engine_update)
    {
        update();
        next_engine_update += ENGINE_UPDATE_RATE;
    }
    render();
}
```

For the implementation of the GUI Components is was easier to have independent, specialized classes than to use inheritance. The finer details for the different components makes the implementation using inheritance difficult. A problem encountered is the co-ordinate system to use. the screen space co-ordinate system would make sense however problems occur due to the fact that it is dependent on the window size, which is dynamic.

The solution implemented is to give the GUI Components their own co-ordinate system. The co-ordinate system is x = [0,1] and y = [0,1]. When mouse commands are passes to the components the mouse co-ordinates are transformed into the new space from the screen space by dividing by the screen width and height. Using a different size window or resizing the window during execution, results in the components scaling by the correct amount.

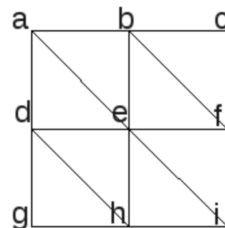
Button components are placed at the edges of the screen and their mouse-over callback functions are set to the required camera movement functions. This provides panning of the camera by moving the mouse to the edge of the screen.

4.5 Vertex Arrays and Chunking

A Vertex Array requires a 1d array holding the vertices of the triangles from the triangle mesh. The array therefore needs to hold $width \times height \times 2 \times 3 \times 3$ values as each grid block is subdivided into two triangles. The original implementation was modified to pack a vertex array with the vertices instead of performing the calls to $glVertex3f()$. Drawing the triangle mesh is then achieved by calling $glDrawArrays(GL_TRIANGLES, 0, 3)$.

This method was then improved upon by using the fact that many of the triangle vertices are common in the triangle mesh. OpenGL provides functionality where a set of vertices can be given and then when the geometry needs to be rendered, the indices of the required vertices is given. This is done by calling
$glDrawElements(GL_TRIANGLES, number_of_vertices, GL_UNSIGNED_INT, indices)$
where indices is an integer array of indicies of the required vertices. This increases performance as less vertices need to be uploaded to the graphics card. Example: For a 100×100 heightmap, 180000 ($100 \times 100 \times 2 \times 3 \times 3$) values need to be sent to the graphics card for the $glDrawArrays()$ method, and only 20000 ($100 \times 100 + 100 \times 100$) would need to be sent using the $glDrawElements()$ call.

Sending texture co-ordinates and normal vectors for the vertices is done similarly. The values are stored in a 1d array, of size $width \times heights$, and is then indexed by an index array. The index array for the texture co-ordinates and normals are the same, requiring only one array to be stored for both.



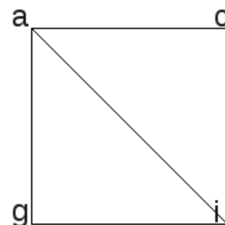


Figure 4.15: Example of jumping over vertices to render at a lower level of detail

Chunking is achieved by splitting the terrain into cells. Each cell is then able to be rendered at different levels of quality. An advanced technique would involve down sampling the area covered by the cell and creating multiple vertex arrays. We implemented a simpler yet more memory efficient method. Instead of recreating a vertex array for each level of detail, only a different index array is used. The index array is smaller and skips over vertices stored within the array of vertices. A difficulty experienced in the chunking implementation is gaps between the chunks. The chunks need to be extended over their required area and then clamped back to ensure there are no gaps.

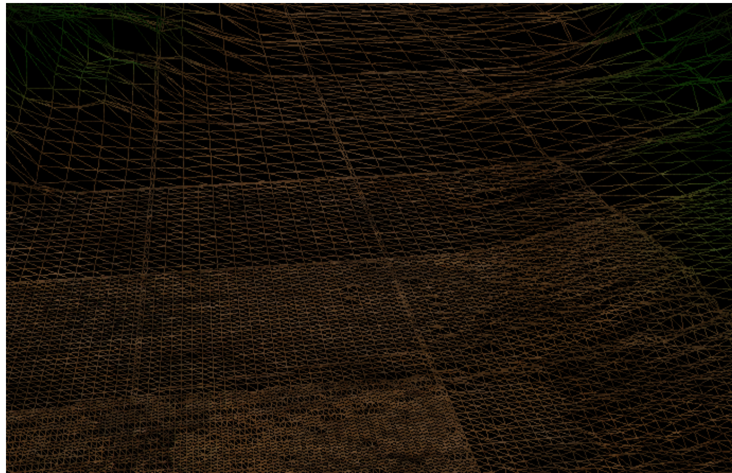


Figure 4.16: Chunking in wireframe mode to show the change in detail at different distances

The level of detail at which a chunk is to be rendered is determined by the distance of the chunk to the camera position. The closer the distance, the greater the detail. The level of detail system is used for all three passes of rendering, this allows for better quality shadows closer to the camera position.

The level of detail used for different distances could be made dynamic such that a given frame rate is guaranteed, however this is complicated and is rather left as an extension.

4.6 Integrating Prototype Code Into Single Application With All Features

The prototype from iteration 4 (vertex arrays and chunking) was modified to include all the features of the other prototypes including shadows, GUI components and user interaction.

4.7 Integrating Code From This Part of The Project With The Other Parts

The rendering and visualization only needs to interact with the Simulator. The Simulator first needs to be initialized with the simulation settings and then an update function gets called from within the game loop. The renderer accesses the simulator to get the arrays of bots and their information.

A difficulty encountered with interfacing the two sections was that the co-ordinate systems do not match. The renderer uses the rasterized map with user specified dimensions, whereas the simulator uses the original dia file. The co-ordinate system of the dia file can be arbitrary and start and end at any position and be of any scale. A further problem is that the dia and world co-ordinates reflect each other across the y axis. The solution implemented was for the simulator to store the starting x and y co-ordinate as well as the width and height of the map, when it is loaded. These values are retrieved from the simulator after initialization and are used for transforming between the two co-ordinate systems. Basic bots were rendered to ensure that the co-ordinates were correct and for testing of the simulator.

Conversion to DIA co-ordinates

```
dia_x = world_x* dia_width/world_width - dia_x_offset
dia_y = (world_height - world_x)* dia_height/world_height+dia_y_offset
```

Conversion to world co-ordinates

```
world_x = (dia_x - dia_x_offset)* world_width/dia_width
world_y = world_height-(dia_y - dia_y_offset)* world_height/dia_height
```

4.8 Visualisation, User Interaction and Additional Features

In this iteration the bot models were improved to look more realistic. They are rendered as a frustum for the body, a sphere for the head and a cloak indicates the direction the bot is facing. The bots can be in one of six states: Attack, Flee, Explore, Camp, Sneak and Defend. This state is indicated to the user as an icon above the bots head. To ensure the icon is visible, it is rotated firstly around the z-axis and then the x-axis such that is always faces the camera.

A bots health is indicates as a coloured ring around its status icon. Green indicates health that the bot has, and Red indicates health that

Figure 4.17: Bot health and status icon. The icon pivots such that it is always facing the camera

the bot has lost. The ring needs to be rendered as a series of triangles with an inner and outer radius. For semi transparent objects in OpenGL they are required to be rendered in the order from furthest to closest to the camera to ensure the correct blending. Since the status icon is semi transparent and the health indicator wraps around the icon, the health indicator is split into two halves. First the back half is rendered, then the status icon, and finally the front half. The paths which the bots are planning on taking are rendered using GL_LINES, as well as the trace of where the bots have been. All of the indicators which are rendered in the world space are not rendered during the first pass (building the shadow map) of the rendering process. This ensures that there are no unwanted shadows cast in the environment.

Figure 4.18: Status Icons for each state of the bots

For simplicity the user is only able to select a single bot at a time. Bot selecting is done by double left clicking near the bot you want to select. If there are bots within a set distance, the closest bot will

be selected. If no bots fall within the threshold, no selection will be done. The position of the click in world space is calculated using the $gluUnproject()$ function. Using the function a vector can be calculated that represents the mouse within the view frustum. The position on the terrain is calculated by intersecting the vector with the environment plane.

Double clicks are determined by having a timestamp for each button which can double click. Whenever a click is made, the timestamp is checked to see if it lies within a set interval. If it is within the interval it is counted as a double click, else it is counted as a single click and the timestamp is reset.

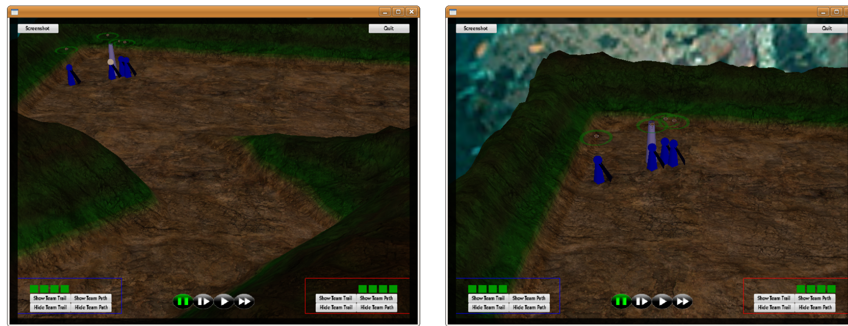


Figure 4.19: Before and after pictures of centering the camera on a point without changing the orientation of the camera

Some more advanced interaction features for camera movement were added. The first is that double right clicking pans the camera, keeping its angle constant, such that it is viewing the world point that was clicked (Figure 4.19). This is achieved again by using the $gluUnproject()$ function to find both the point that the camera is currently facing and the point that was clicked. The vector between these two points is then recorded and the camera is moved along the vector over a set time period. At the end of the camera movement the camera is facing the clicked point.

Another method for looking at a point can be done by double middle clicking. This method changes the camera angle, and not the camera position (Figure 4.20). It is done using trigonometry and the horizontal and vertical distances between the camera and clicked position.

A camera movement that allows the user to circle strafe around a point was also implement. When the user clicks and drags with middle mouse button the camera will rotate around the point which it was originally looking at. The distance from the point remains constant and the camera is always looking at the point. The position of the camera is determined using the horizontal distance to the point and which direction the mouse was dragged. The theta angle of the camera

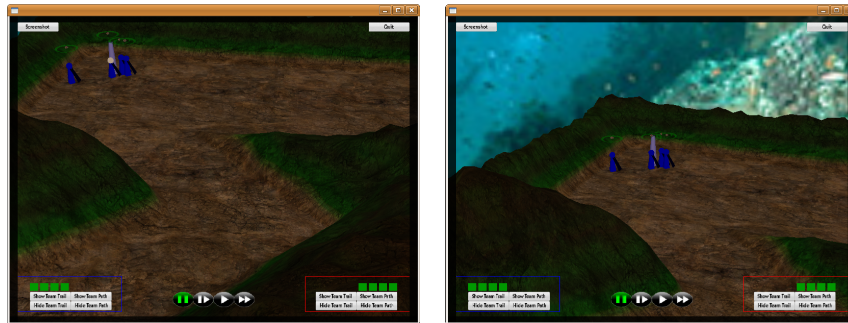


Figure 4.20: Before and after pictures of centering the camera on a point without moving the camera

is determined using the horizontal vector from the camera to the point of focus, and the phi angle remains constant.

During camera movement, the speed at which the camera moves is dynamic. Movement is initially slow and speeds up as you continuously move. When movement stops the movement speed slows down to its initial speed again. This was implemented by having a speed variable that increases every time a move happens, and decrements every engine update. The value is clamped to a maximum and minimum speed, and the speed increase and decrease rates are set for easy movement control.

The state of the simulation can be seen and interacted with using controls often associated with media players.

Figure 4.21: Controls for controlling the simulation

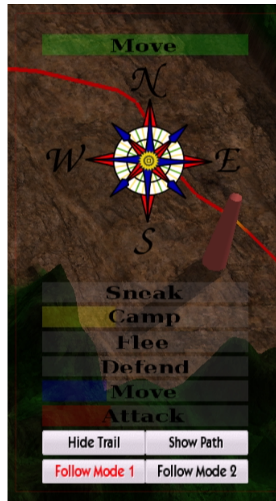


Figure 4.22: Data and menu options which are displayed when a bot is selected

When a bot is selected, additional data and menu items are made available to the user (Figure 4.23). A larger health bar for the bot is displayed in screen space for easier reading. The bot's status is also displayed in screen space both as an icon as well as in text. The interface for the bot displays the percentage if time the bot has been in each state. Menu buttons are made available for turning on data visualisations pertaining to that specific bot.

Figure 4.23: Paths showing the routes which the bots plan to take to get to the flag

The visualisations that can be turned on are for the rendering of the bots current desired path (Figure 4.23) and a trace of where the bot has moved (Figure 4.24). The trace is colorised depeneding on the state of

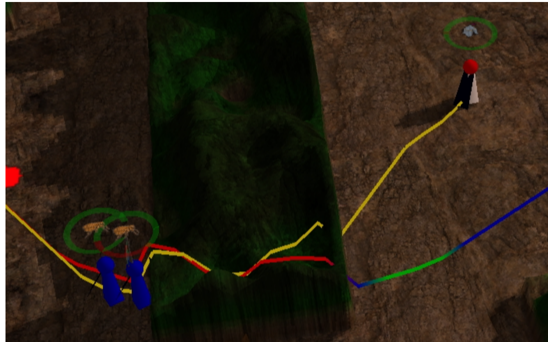


Figure 4.24: Trail showing the the route taken by the bot as well as its state at each point

the bot at each point. This helps with the interpretting of the data as the user can easily identify what states the bot has been in and where.

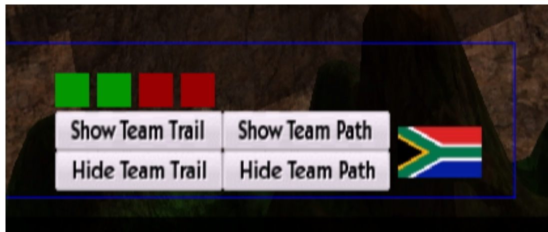


Figure 4.25: The team menu. Only 2 of 4 bots are still alive and the team currently has the flag

Each team also has a team menu which is always visible (Figure 4.25). It displays information and buttons which apply to the whole team. The team menu indicates how many bots are still alive and whether the team is in possession of the flag.

Two important types of data which have a spacial component are areas of the map where bots have been killed and have killed from. This data is shown to the user via multi-texturing the information onto the terrain surface. Different colours represent different locations. Red was used for locations where bots were killed and green was used for locations where kills took place from 4.26. OpenGL does not provide functionality to write to the pixels of a texture. For this reason SDL was used for modifying a surface object which can then be converted into a texture when needed and then reuploaded to the graphics card for use by OpenGL.

A bots intention to attack someone is shown as a line from the attacker to the bot they wish to attack4.26.

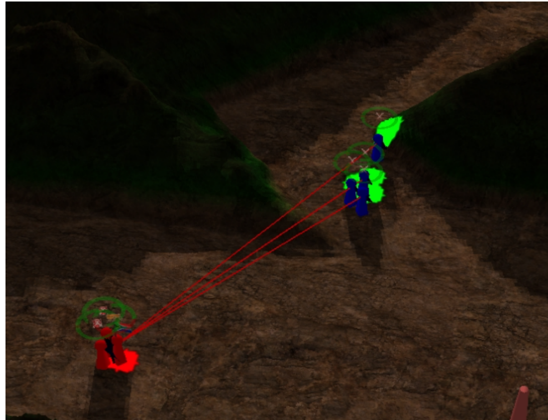


Figure 4.26: Red areas are potentially dangerous as a bot has died there. A Green area is possible advantageous since a bot has been killed from that location

Chapter 5

Testing and Evaluation

5.1 Introduction

The components which need to be tested for this project are the efficiency of the renderer and the benefit gained for the user from the data visualisations. The renderer needs to be able to render at a minimum of 30 frames per second on low end hardware with the only requirement being a graphics card. Shadows can be turned off to increase performance where necessary. The application being developed would be used by people who are interested in developing rule systems for virtual environments or games. Another potential user is someone who is interested in developing maps for games which produce a desired behaviour in the players. An example would be the design of a map which benefits a camping strategy. The data visualisations need to inform the user about what is happing during the simulation. This includes information about each bot specifically as well as information related to an entire team.

5.2 Required User Testing

5.2.1 Rendering Efficiency Testing

A frame counter was implemented such that the frame rate can be recorded while the simulation is running. It records the amount of time in milliseconds between the last 100 frames and then calculates an average to give the current frame rate. The frame rate is output to the terminal at a much lower rate to ensure that writing to the output stream does not slow down the rendering. The level of detail system reduces the number of vertices which are to be rendered for a relatively larger increase in frame rate. The increase in frame rate can accurately be recorded however the amount of detail lost is highly subjective. To

measure the amount of detail lost, user testing was used. The users were shown an image rendered at full detail and then have to decide at what percentage detail other images were rendered at. The other images include the terrain rendered at different detail levels as well as using the dynamic level of detail system.

5.2.2 Data Visualisation Testing

To measure the benefit gained from the data visualisations, user testing needs to be implemented. The target user for the application would not posses any special skills for reading the data visualisations. The data visualisations should also be simple and clear enough such that no special skills are required. It was therefore decided that we can perform user testing on average people. The goal of the test is to find out whether the data visualisations help the user in acquiring information about the simulation. For that reason the user testing needs to test the use of the system both with and without the data visualisations. The test with out the visualisations serves as a control to see if any significant benefit is added.

If no data visualisations are displayed it would be impossible to make any decisions, so some basic elements are provided for the control. Health bars and status icons have been included. These elements are the traditional elements which are expected to be seen in real time strategy games and cannot be seen as ”new” or adding benefit to the user. The test provides a series of tasks for the user to perform. They include identifying positions as well as state information about the bots. For each test an average score can be calculated which reflects the users ability to use the system both with and without extra visualisations.

Another factor which is being tested is the users confidence with their answers. A user may have gotten an answer correct, but only because they had a free guess.

It was decided that each user would perform tests with and without the extra visualisations. This greatly increased the complexity of user testing as the ordering in which they do the tests could affect the outcome. Another potential problem is when performing the second test, the user would have the advantage of already seeing the simulation scenario. To get around these problems two scenarios are tested and the order of whether it is with or without visualisations is alternated. The result is that four versions of the test need to be implemented.

A user’s ability to interact with the application can have an effect on their ability to read the information about the simulation. For example a user with gaming skills could potentially move around the map easier and therefore get a better score. For this reason the affects of user interaction was removed, by making use of videos and images oppose

to the use of the application.

The final user test which was carried out was constructed as a presentation. It was split into two sections, Data Visualisation and Rendering Efficiency. The user was required to view the presentation and complete the necessary tasks on a printed answer sheet. The presentation displayed the instructions as well as any images which needed to be viewed for the test. Videos for the test were linked to from within the presentation. A total of 12 people took part in the user test ensuring an equal number of sample points for each varying factor as wells as for a statistically relevant analysis of results.

5.3 Expected outcomes

The relationship between the detail observed in an image to the number of frames per second achieved would be expected to be linear and inversely proportional. i.e. As the detail decreases the frame rate should increase in a linear fashion. The result for the level of detail system should however not follow this pattern as it theoretically provides a better trade of between detail and performance.

For the data visualisation testing, it is expected that the additional data visualisations would give the users an improved average score. The amount of improvement, however is not known. There should also be an increase in the confidence experienced by the users when using the visualisations. The amount of this improvement is also unknown.

5.4 Results And Analysis

To test the efficiency of the rendering, the application was run on a Pentium 4 with a nVidia 9800GT graphics card. A larger than standard size map of 900 by 900 pixels was used. The average number of frames per second was recorded as 86 frames per second. This is much higher than the required 30 frames per second for a standard map size of 300 by 300 pixels. The renderer is also capable of running on a PC with no graphics card, but at a relatively slow 10 frames per second.

The full set of captured results from the user testing is provided as an appendix (C.1) and this section will focus on the analysis of the data.

The results of the trade-off between detail and performance for the Level of Detail scheme were unfavorable. The graph of detail vs performance is as expected, however the Chunking system doesn't manage to show extraordinary performance for a given detail level. The reason for this could be that the frame rates recorded for the comparison were recorded on a computer graphics card. It is more likely that the problem is that the terrain size is just not big enough to see a significant

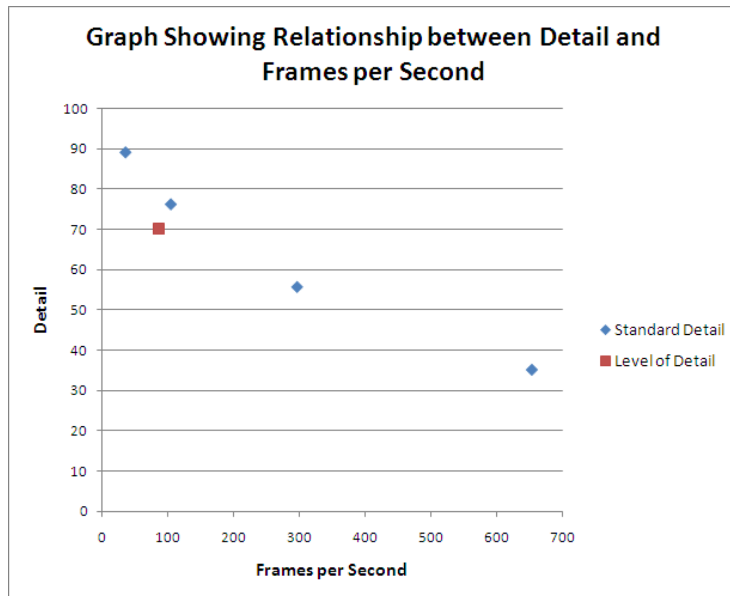


Figure 5.1: Graph showing the trade-off between Detail and Frames per Second

performance increase. A larger terrain favors level of detail systems as there is more geometry for the system to cut out, which would otherwise be rendered.

To test if the users score improved during when doing the test with the data visualisations it needs to be compared to their score when they had no data visualisations. If users performed better when using the visualisations, the mean score value of the test with visualisation will be larger than that of the mean from the test without visualisations. It is not good enough for the mean to just be larger. It needs to be statistically significantly larger. ie. There needs to be a less than 5% chance that the mean was larger just by pure chance. The test used to compare two sets of data, which follow a normal distribution, for similarity is called a t-test. There are two varieties of the t-test.

Unpaired There are no associations between samples of the first data set with those of the second

Paired Each sample of the first data set has an association with a sample in the second data set

The data being compared for our user testing is paired as each user is performing both versions of the test. To prove that the data visualisations benefited the users in making decisions about the simulation,

we disprove the that they performed equally well with and without the visualisations. The null hypothesis is therefore: *Users perform equally well at making decisions about the simulation regardless of whether they have extra data visualisations.*

Paired t-test on scores with and without visualisations

	Mean	Standard Deviation	Standard error of mean	N
Score without	6.33	2.57	0.74	12
Score with	7.75	1.6	0.46	12

Intermediate values

t	2.38
df	11
Standard error of difference	0.6

Confidence interval

Difference of mean values	1.42
95% confidence interval	0.1 to 2.73

Statistical significance

P Value	.0367

Figure 5.2: Paired t-test of the Scores achieved with data visualisations turned on and off

The results of the t-test (5.2) show that the two data sets are not statistically significantly close enough to each other for them to be similar. This disproves the null hypothesis, and the data visualisations did in fact benefit the users when tying to identify features of the simulation. A ”P Value” of 0.0367 states that it can be said with 99.6% certainty that the data visualisation improved the users performance in the test. The ”95% confidence interval” implies that on average 95% of the the time a users score will be between 10% and 27.3% better than their score without the visualisations. The difference between the two data sets means is 14.2%.

The confidence the user had in their answer is expected to increase when aided with the data visualisations. To compare the two data sets of confidence with and without the data visualisations another t-test is performed. The test is again a paired t-test. The null hypothesis is: *Users are equally confident in their ability to identify features within the simulation regardless of whether they have extra data visualisations.*

Paired t-test on confidence with and without visualisations

	Mean	Standard Deviation	Standard error of mean	N
Confidence without	0.6817	0.0985	0.0284	12
Confidence with	0.855	0.106	0.0306	12

Intermediate values

t	4.8927
df	11
Standard error of difference	0.035

Confidence interval

Difference of mean values	0.1733
95% confidence interval	0.0954 to 0.2513

Statistical significance

P Value	0.0005

Figure 5.3: Paired t-test of the Confidence experienced with data visualisations turned on and off

The results of the t-test (5.3) show that the two data sets are not statistically significantly close enough to each other for them to be similar. This disproves the null hypothesis, and proves that the data visualisations increase the users confidence when trying to identify features within the simulation. The "P Value" for the confidence t-test is 0.0005 implying that the users were more confident due to the data visualisations. 95% of user can expect to see an increase in confidence within the range of 9.5% to 25%. The difference in mean values for the two data sets is 17.1%

After completing the user test both with and without the additional visualisation, the users were asked to express how useful they thought the visualizations were. The mean value out of 10 (with 10 being the maximum) was 9.08.

Chapter 6

Conclusion

In this paper a lightweight rendering engine was developed to ensure easy interfacing with the Spatial Awareness Framework and allow for full customisation to implement data visualisations. A requirement of any renderer is to be computationally efficient and achieve high frame rate. The renderer implemented achieves this goal and is also able to run on low end hardware. The renderer implements advanced techniques such as Shadow Mapping and Chunking to achieve greater realism and faster frame rates.

The Chunking was shown to not provide a significant increase in performance due to the small size of the environments. The Chunking functionality does however allow for some scalability of the system if larger terrains are required.

User testing showed that the data visualisations helped users of the system to make accurate decisions about what is happening within the simulation. The confidence of the user was also shown to increase when the visualisations are included. The visualisations would help potential users of the system to develop and analyse rules for sets of bots. This analysis will help with developing a desired behaviour of the bots within the environment. The system could also be used to build environments that require bots to implement desired strategies in order to win.

Bibliography

[1] OpenGL - The Industry Standard for High Performance Graphics. http://www.opengl.org/.

[2] Simple DirectMedia Layer. http://www.libsdl.org/.

[3] The freeglut Project. http://freeglut.sourceforge.net/.

[4] P. Atherton, K. Weiler, and D. Greenberg. Polygon shadow generation. *ACM SIGGRAPH Computer Graphics*, 12(3):275–281, 1978.

[5] P. Baker. Paul's Projects - Shadow Mapping Tutorial. http://www.paulsprojects.net/tutorials/smt/smt.html.

[6] S. Ben. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. Of the 1996 IEEE Symposium on Visual Languages, IEEE Computer Society, Washington, DC*, pages 336–343, 1996.

[7] P. Buono, M. Costabile, and F. Lisi. Supporting data analysis through visualizations. In *Proceedings of the International Workshop on Visual Data Mining*. Citeseer, 2001.

[8] F. Crow. Shadow algorithms for computer graphics. *ACM SIGGRAPH Computer Graphics*, 11(2):248, 1977.

[9] L. De Floriani and E. Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Transactions on Graphics (TOG)*, 14(4):363–411, 1995.

[10] P. Dickinson. Instant Replay : Building a Game Engine with Reproducible Behavior, 2001. http://www.gamasutra.com/features/20010713/dickinson$_$01.html.

[11] H. Jones and M. Snyder. Supervisory control of multiple robots based on a real-time strategy game interaction paradigm. In *IEEE INTERNATIONAL CONFERENCE ON SYSTEMS MAN AND CYBERNETICS*, volume 1, pages 383–388, 2001.

[12] B. Larsen and N. Christensen. Real-time terrain rendering using smooth hardware optimized level of detail. *Journal of WSCG*, 11(2):282–9, 2003.

[13] J. Nielsen. Ten Usability Heuristics. `http://www.useit.com/papers/heuristic/heuristic_list.html`.

[14] S. Perkins, D. Jacka, J. Gain, and P. Marais. A spatial awareness framework for enhancing game agent behaviour. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 15–22. ACM New York, NY, USA, 2008.

[15] T. Ulrich. Rendering massive terrains using chunked level of detail control. *SIGGRAPH Course Notes*, 3(5), 2002.

[16] L. Valente, A. Conci, and B. Feijó. Real time game loop models for single-player computer games. In *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, pages 89–99. Citeseer, 2005.

[17] L. Williams. Casting curved shadows on curved surfaces. *ACM SIGGRAPH Computer Graphics*, 12(3):270–274, 1978.

[18] U. Wiss and D. Carr. An empirical study of task support in 3d information visualizations. In *Proceedings of the International Conference on Information Visualisation (IV)*, pages 392–399, 1999.

[19] K. Witters. deWiTTERS Game Loop Article. `http://dewitters.koonsolo.com/gameloop.html`.

[20] A. Woo, P. Poulin, and A. Fournier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–32, 1990.

[21] R. Wright and B. Lipchak. *OpenGL superbible*. Sams Indianapolis, IN, USA, 2004.

Appendix A

User Test Presentation

User Test

- This user test has two sections:
  - Data Visualisation
  - Rendering Efficiency
- You are required to go through the slide show and answer the questions on the provided answer sheet.
- When you are finished call me and I will pay you. You may then leave.

Data Visualisatioin

- For this section you will be shown multiple videos and you will have to identify certain things within the videos.
- Each video is shown twice
  - During the first time you may not answer the questions
  - During the second time (and afterwards) you may answer the questions
- The videos consist of competing bots within a virtual environment

Data Visualisation

- The videos may or may not include additional data visualisation to aid the answering of the questions
- Pay careful attention to the video during the first viewing so that during the second viewing the questions are easier to answer

Video 1

- In this video you need to identify the following:
  - Two places where bots are killed
  - Two places where bots are killed from
- Finally you must give a rating of how confident you feel with the answers you have given
- Mark the locations on the answer sheet
  - Cross for a location where a bot was killed
  - Circle for a location where a bot was killed from

Video 1 – with visualisations

- Red is drawn on the map where bots have been killed and green is drawn where bots have been killed from
- First viewing
  - Rember not to answer
  - Click to begin
- Second viewing
  - You may answer during and after this viewing
  - Click to begin

Video 1 – without visualisations

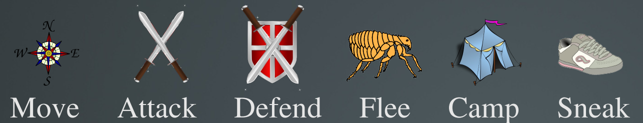
- You need to watch and remember the locations where the bots are killed and killed from.
- First viewing
  - Rember not to answer
  - Click to begin
- Second viewing
  - You may answer during and after this viewing
  - Click to begin

Video 2

- For this video you need to perform the following:
  - Draw the path that the bot follows
  - Identify whether the bot is in the Attacking or Moving state more often
  - Write out the order of the states of the bot
- Finally you must give a rating of how confident you feel with the answers you have given

Video 2

- The states of the bot is identifiable by the icon above its head



(You may refer to this slide at any time)

Video 2 – with visualisations

- The path is traced out by the bot and is colourised depending on the state. The percentage of time a bot was in each state is shown at the side
- First viewing
  - Rember not to answer
  - Click to begin
- Second viewing
  - You may answer during and after this viewing
  - Click to begin

Video 2 – without visualisations

- You need to rember the path as well as the states the bot was in, in order to answer the questions
- First viewing
  - Rember not to answer
  - Click to begin
- Second viewing
  - You may answer during and after this viewing
  - Click to begin

Image 1 – without visualisation

- For this image you need to perform the following:
  - Identify who a bot is attacking
  - Identify whether a bot is attacking or moving towards the flag
  - Draw the path that a bot is planning on taking to get to the flag
- Finally you must give a rating of how confident you feel with the answers you have given

Image 1 – with visualisation

- For this image you need to perform the following:
  - Identify who a bot is attacking
  - Identify whether a bot is attacking or moving towards the flag
  - Draw the path that a bot is planning on taking to get to the flag
- Finally you must give a rating of how confident you feel with the answers you have given

Additional questions

- You have seen the system with and without visualisations
- How useful would you say the visualisations are for identifying features within the simulation?
- Any comments??

Rendering Efficiency

- For this section you will be shown a series of images. Each image will show a rendered environment rendered at different amounts of detail.
- The image which was rendered at full detail (100%) will be identified for you. You need to say at what percentage detail you think the others were rendered at (0-100%)

Image 1 – Detail 1

Full Detail (100%) ?

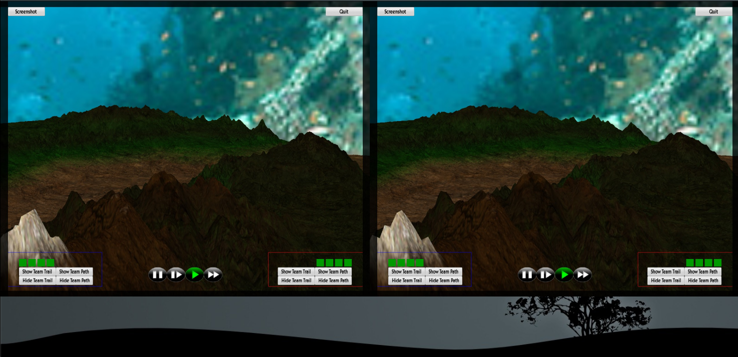


Image 1 – Detail 2

Full Detail (100%) ?

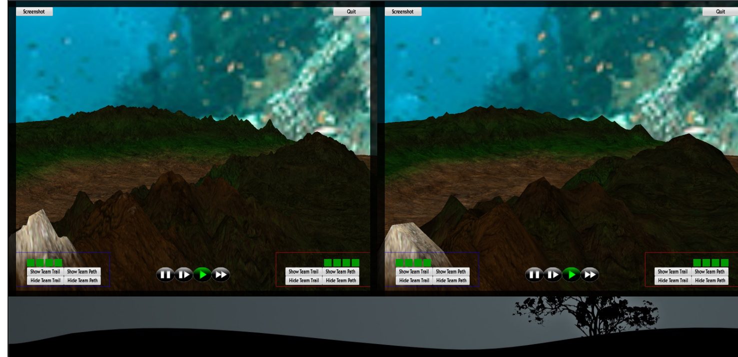


Image 1 – Detail 3

Full Detail (100%) ?

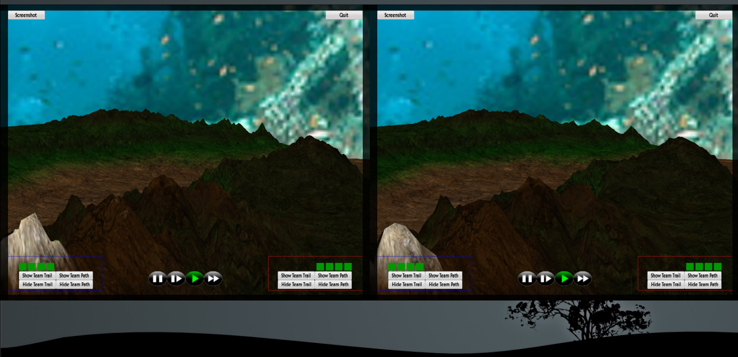


Image 1 – Detail 4

Full Detail (100%) ?

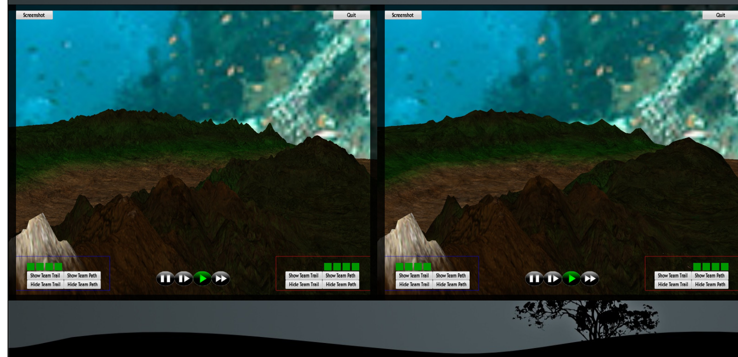


Image 1 – Detail 5

Full Detail (100%) ?

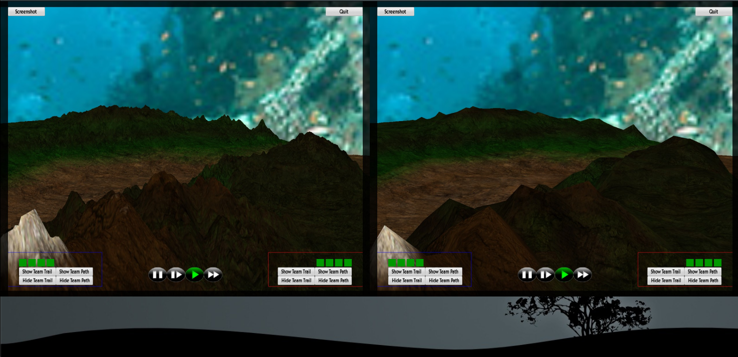


Image 2 – Detail 1

Full Detail (100%) ?

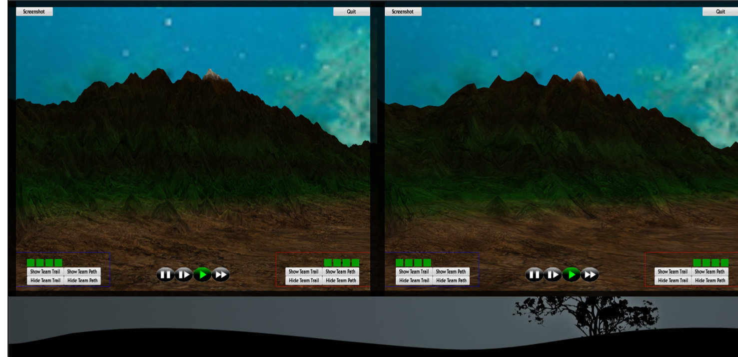


Image 2 – Detail 2

Full Detail (100%) ?

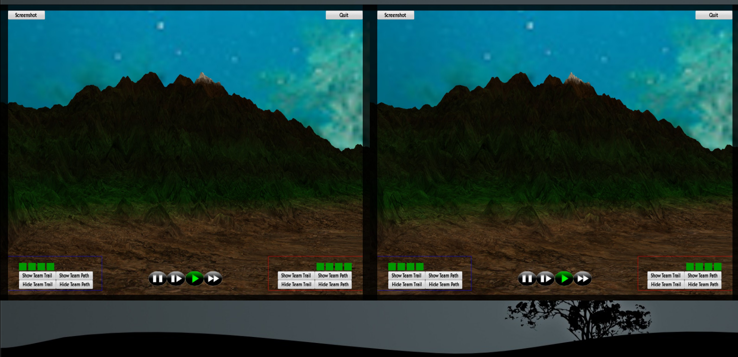


Image 2 – Detail 3

Full Detail (100%) ?

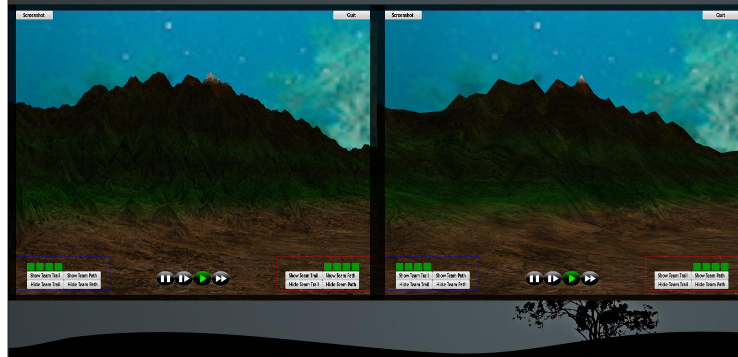


Image 2 – Detail 4

Full Detail (100%) ?

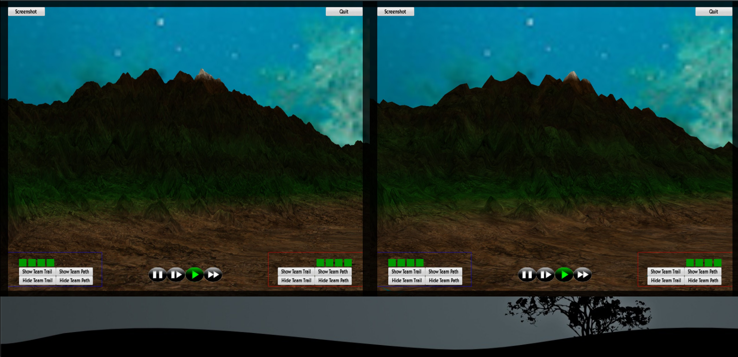


Image 2 – Detail 5

Full Detail (100%) ?

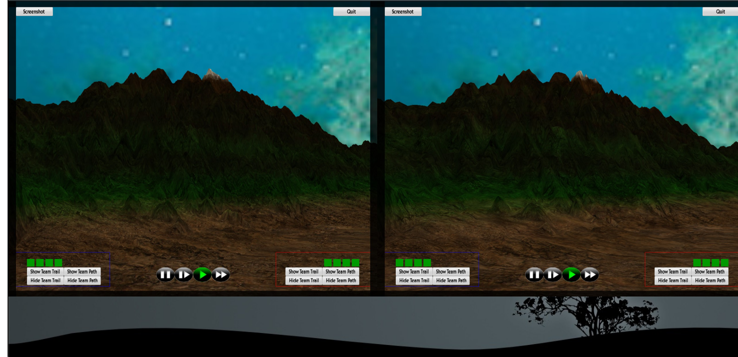


Image 3 – Detail 1

Full Detail (100%) ?

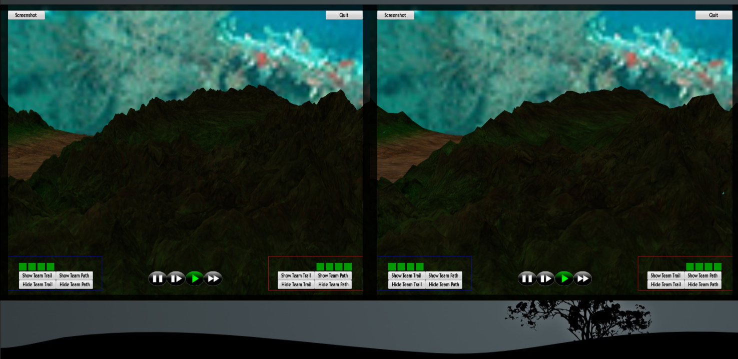


Image 3 – Detail 2

Full Detail (100%) ?

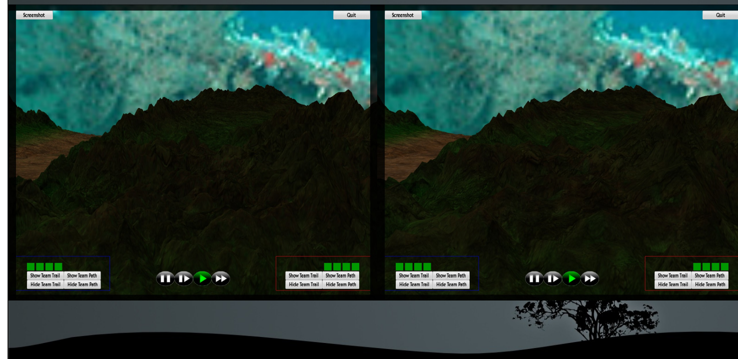


Image 3 – Detail 3

Full Detail (100%) ?

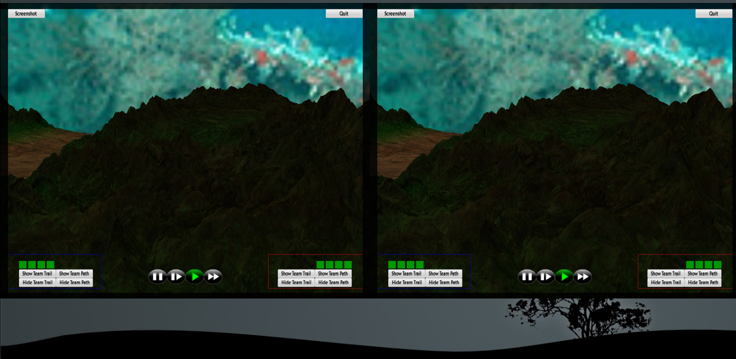


Image 3 – Detail 4

Full Detail (100%) ?

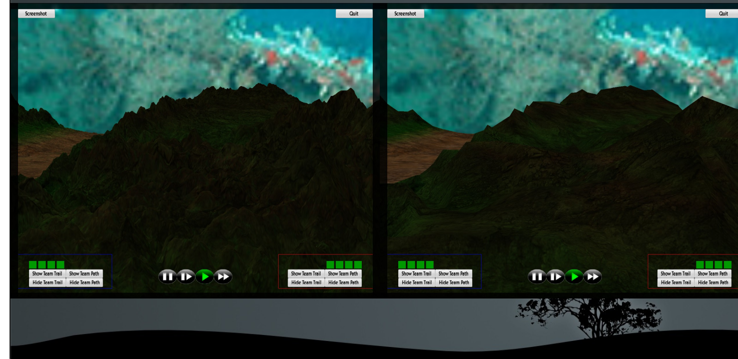
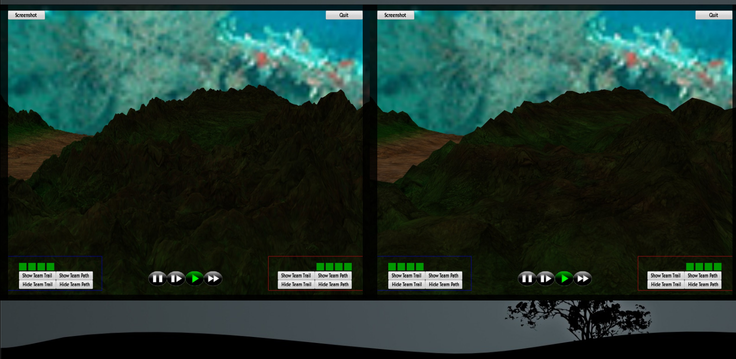


Image 3 – Detail 5

Full Detail (100%) ?



THE END

- Call me so I can collect your answers and pay you
- Thank you for your time

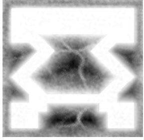
Appendix B

User Test Questionnaire

User Test Questionnaire

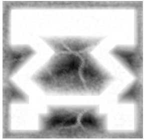
Video 1:

a) Indicate two locations on the map where bots are killed and two where bots are killed from. Draw an "x" where they were killed, and a "o" for where they were killed from.



How confident are you with your answers? (1-10) :

b) Indicate two locations on the map where bots are killed and two where bots are killed from. Draw an "x" where they were killed, and a "o" for where they were killed from.



How confident are you with your answers? (1-10) :

Video 2:

a) Draw the path taken by the bot that is being followed.

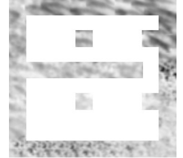
b) Is the bot being followed in an Attacking or Moving state more often?

Attacking Moving (circle one)

c) Write out (in order) the states which the bot goes through

How confident are you with your answers? (1-10) :

d) Draw the the path taken by the bot that is being followed.



e) Is the bot being followed in an Attacking or Moving state more often?

Attacking Moving (circle one)

f) Write out (in order) the states which the bot goes through

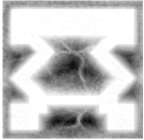
How confident are you with your answers? (1-10) :

Image 1 :

a) Who is bot 1 attacking? 1 2 3 4 5 nobody (circle one)

b) Is bot 2 Attacking or Moving? Attacking Moving (circle one)

c) Draw the path the bot 5 plans to take to get to the flag

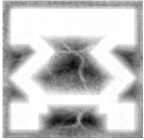


How confident are you with your answers? (1-10) :

d) Who is bot 1 attacking? 1 2 3 4 5 nobody (circle one)

e) Is bot 2 Attacking or Moving? Attacking Moving (circle one)

f) Draw the path the bot 5 plans to take to get to the flag



How confident are you with your answers? (1-10) :

Additional:

In the range from 1 to 10, how useful would you say the visualisations are for identifying features within the simulation? (10 being the most useful and 0 being useless)

Any Comments??

Image 1:

Detail 1: _______ %
Detail 2: _______ %
Detail 3: _______ %
Detail 4: _______ %
Detail 5: _______ %

Image 2:

Detail 1: _______ %
Detail 2: _______ %
Detail 3: _______ %
Detail 4: _______ %
Detail 5: _______ %

Image 3:

Detail 1: _______ %
Detail 2: _______ %
Detail 3: _______ %
Detail 4: _______ %
Detail 5: _______ %

- -

I have been paid R20 for participating in Wesley King's user test for his honours project.

First Name: _______________ Surname: __________________

Date: _______________ Signature: __________________

Appendix C

User Test Results

Test number	1	2	3	4	5	6	7	8	9	10	11	12	Mean	
Score without	3	7	3	5	10	7	9	9	5	9	6	3	6.33	
Score with	8	10	5	9	9	7	9	7	7	9	8	5	7.75	
Confidence without	0.63	0.8	0.7	0.57	0.57	0.8	0.83	0.57	0.63	0.75	0.73	0.6	0.68	
Confidence with	0.85	0.93	0.85	0.83	0.93	0.83	1	0.93	0.83	0.95	0.7	0.63	0.86	
Usefulness of visualisations	8	10	7	9	10	9	10	10	8	10	8	10	9.08	
														Average FPS
Detail level 2	260	270	275	265	240	260	288	290	260	280	270	250	89.11	35
Detail level 3	190	200	215	240	220	240	245	255	240	230	210	260	76.25	105
Detail level 4	150	110	120	215	190	150	172	205	185	180	150	180	55.75	295
Detail level 5	60	60	60	115	140	90	78	205	170	50	90	150	35.22	650
Level of Detail	210	200	190	180	220	200	168	280	215	250	200	215	70.22	90

Figure C.1: Data recorded during the user test