Designing and Implementing Weather Effects in OpenGL

Stephen Tucker
BSc Computing
2002/2003

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) _____

Summary

The main objective of the project was to design and implement a realistic weather effect that can be used in the Leeds University Advanced Driving Simulator.

The report describes the specification, design and implementation of a graphical application to satisfy this objective. A study was first conducted to determine how this could be potentially achieved and then a solution was developed and implemented. The report explores all of the potential options that were available throughout the development process, such as choices regarding which technologies to use, and justifies all the decisions that were made. Any problems that were encountered are also discussed, along with how they were resolved.

The delivered application satisfies this requirement and also offers the user extra functionality, as additional methods have been implemented so the characteristics of the weather effect can be easily altered. Although the effect has not been implemented into the simulator, every step has been taken to ensure that this is certainly possible as the application was produced using OpenGL, which is a compatible technology. The effect has been fully tested and its effectiveness has been evaluated within the report.

Acknowledgements

I greatly appreciate all the help and support that was given to me by my project supervisor, Dr. Andy Bulpitt, throughout each and every stage of the project. Without all the useful feedback and suggestions that he continually offered me, I'm not convinced my project would have developed to the level it has.

I would also like to thank Hamish Jamson and Tony Horrobin, of the Leeds University Advanced Driving Simulator, for taking the time to demonstrate the simulator in action. This served as a great inspiration for my work and was an incredibly interesting and enjoyable experience.

Finally I would also like to thank my friends: Philip Aldridge, David Bedford, Alex Osmond-Brims, Becky Delbridge, Michael Fury, Lyndon Hall, Michael Heydon and Adam Thornhill, for taking the time to familiarise themselves with my developed application and supplying me with useful feedback, so I could critically evaluate it's effectiveness.

Figures

1. Project Overview

1.1. Introduction

The Leeds University Advanced Driving Simulator (LADS) is a state of the art static-base driving simulator, (see section 2.2 for further information). To ensure that the results obtained from running simulations correspond closely to the real world situations that are being modelled, the virtual environments that are rendered must appear as realistic as possible. The research staff that maintain and develop the simulator, continually strive to improve realism wherever possible and therefore suggested a final year project to design and implement a weather effect that could be used within the simulations. If well implemented, this effect would significantly enhance realism, as it would both improve the general appearance of the simulator and extend the driving conditions that could be modelled.

1.2. Project Aims and Objectives

The overall aim of the project was to design and implement a realistic weather effect that can be used in the Leeds University Advanced Driving Simulator. The objectives break down as follows:

Minimum requirements:

1. Learn how to use the appropriate graphics technologies to produce the effect, namely OpenGL or OpenGL Performer.
2. Research how similar weather effects are currently achieved using this software.
3. Design, produce and test at least one weather effect, for use in the simulation. (It will be necessary to develop a graphical environment within which this can be done).
4. Evaluate the effectiveness of the generated weather effect.

Further enhancements:

1. Include methods to adjust the appearance and properties of the implemented weather effect, to enable the user the ability to customise the effect.
2. Implement additional weather effects.

1.3. Structure of Report

The next chapter is a summary of the background research that was conducted to ensure a good understanding of the project requirements, as well as determining how these requirements might be achieved.

The third chapter explicitly states the requirements of the developed application and outlines the proposed designs that were implemented to fulfil these requirements.

The forth chapter is an explanation of how these developed designs were actually implemented. As there is often multiple ways that OpenGL can be used to implement the same functionality, supporting discussion is offered on the choices that were made.

The fifth chapter is an evaluation of the delivered application. It addresses whether or not the stated requirements were achieved and discusses how realistic the developed weather effect actually is. Potential improvements that could be made to the application are identified and from this, future enhancements that could be implemented are suggested.

Finally conclusions are offered that address whether or not the project was successful.

1.4. Project Schedule

A detailed break down of my original project schedule, which was included in the submitted mid-term report, is presented in a chart in Appendix B. A second chart is also included that shows the actual schedule that was followed and explains why deviations were made from the original plan.

2. Background Research

2.1. Literature Reviews

Throughout the course of the project, three books were constantly referenced that offered useful insights into key aspects of the project work. These were as follows:

1. ‘OpenGL - A Primer’ (2002), by E. Angel [1].
2. ‘Real-Time Rendering’ (2002), by Möller and Haines [2].
3. ‘A Book On C’ (1998), by Kelley and Pohl [3].

The first book is ‘a concise presentation of fundamental OpenGL commands’ [1]. It is generally regarded to be one of the best OpenGL resources available and has been recommended in several School of Computing modules. It provided information, explanations and examples of most OpenGL features and was an invaluable resource when implementing the application.

The second book is a best selling resource on real-time rendering, it has also been recommended in many School of Computing modules. It provides a wealth of useful information about all aspects of computer graphics and was especially useful as a source of general background information.

The third book is a guide to programming in C, and served as a useful resource when producing the application code. Many such books are readily available, however the style of the book seemed far superior to others that were examined, and there was a good balance between explanation and example code, when demonstrating the features of the language, so it was used as a primary source of information during the development process.

2.2. An Overview of the Leeds University Advanced Driving Simulator

Figure 2.1 An outside view of the Leeds University Advanced Driving Simulator

This is a static-base driving simulator, used for conducting research into areas such as transport safety, telematics and driver behaviour. The simulator has been operational since 1993 and is continually developed and supported by three University departments: the School of Psychology, the School of Computing and the Institute for Transport Studies.

The simulator consists of a complete Rover 216GTi, with all the cars basic controls and dashboard instrumentation still fully operational. A real time 3D graphical scene of a virtual environment is projected onto screens that are situated both in front of and behind the driver. The projection system consists of five forward channels, which produces a horizontal field of view of 230 degrees and a vertical field of view of 39 degrees. The rear projection provides an image that the driver can observe when using the cars rear view and wing mirrors. The scene is generated using OpenGL Performer (an Application Programming Interface (API) developed by Silicon Graphics Inc (SGI)) and this is run on a SGI Onyx2 Infinite Reality2 graphical workstation. The frame rate used for the simulation is a constant 30 Hertz, a Roland digital sound sampler is used to create believable driving sounds during the course of a simulation and realistic feedback is given to the driver by simulating steering torques at the steering wheel. Further information can be located on the official simulator website [4].

2.3. Weather Effects

After considering all the different weather effects that could be potentially modelled, the decision was made to focus on producing a rain effect. This choice was made due to the fact that rain is one of the UK's most common weather conditions, and therefore driving in rain is an activity that most drivers will invariably have to endure, (more so than say driving in hail). Therefore if this atmospheric effect can be included into the driving simulator, it will clearly help increase overall realism and believability. Background research into other driving simulators, such as the National Advanced Driving Simulator (NADS), which is based in the US, at the University of Iowa, and is the worlds largest [5], suggested that the provision of rain effects could bring a wealth of benefits to a simulator.

When rain occurs, it is often the case that the sky is dull and/or foggy, so it is worth considering these atmospheric conditions in addition. OpenGL provides facilities to easily render fog into a scene, the following website offers an overview of how this is achieved and what effects can be generated [6].

2.4. Computer Graphics Technologies

There are many Graphics Technologies available that could potentially be used to create a weather effect, these include technologies such as the open language VRML [7], which is often used for Internet based applications, or the Java based API, Java 3D [8]. However as the Leeds University Driving Simulator uses OpenGL Performer, this places the constraint of only being able to consider compatible technologies, namely OpenGL Performer, or the standard OpenGL library.

2.4.1. OpenGL

OpenGL is the "most widely used and supported 2D and 3D graphics application programming interface (API)" [9]. Since its introduction to the market in 1992, it has established itself as the industry standard for developing graphical applications. OpenGL offers many useful benefits to the application programmer, these are describes in detail on the official website [9], but can be summarized as follows:

1. The API consists of around 200 distinct commands that are used to specify all the objects and functions required to produce the graphical application. These commands are intuitive and logical which makes OpenGL fairly easy to pick up and quickly master.
2. OpenGL is an extremely powerful API as it provides a broad set of rendering, texture mapping, special effects, and many other powerful visualization functions.
3. OpenGL is platform independent, meaning applications can be easily ported from one system to another. Also it is reliable in terms of producing consistent visual display results on any OpenGL API-compliant hardware, regardless of operating system or windowing system. It includes bindings for both ANSI C and C++.
4. OpenGL is very well documented, numerous programming guides and reference books have been published, and there is a plentiful supply of example code and tutorials provided online.
5. As OpenGL has been on the market for over ten years it is considered extremely stable. Backward compatibility is ensured when new software releases are made and new additions to the specification are well controlled.

2.4.2. OpenGL Performer

"OpenGL Performer is a powerful and comprehensive programming interface for developers creating real-time visual simulation and other professional performance-oriented 3D graphics applications" [10]. It is effectively a higher-level interface that sits on top of the standard OpenGL graphics library, so offers the same set of benefits, as outlined in section 2.4.1, but with some additional higher-level functionality.

OpenGL Performer consists of two main libraries: the performance rendering library, *libpr*, which is a low level object-oriented library that provides high speed rendering functions, efficient graphics state and other application-neutral 3D graphics functions, and *libpf*, which is layered above the first library and is a real-time visual simulation environment, providing a high-performance multi-processing scene graph and rendering system that takes best advantage of SGI symmetric multiprocessing hardware. Performer also has six associated libraries: *libpfdu*, *libpfdb*, *libpfui*, *libpfx*, *libpfv*, and *libpfutil*, these are described in detail on the official Performer website [10].

2.4.3. The decision to use OpenGL

The decision was made to use the standard OpenGL library over the newer more advanced performer interface. Although Performer offers the user a wealth of new features and enhancements, the majority of these additions would not be relevant to this particular project and would not bring about many potential benefits to the application. Performer is a higher level interface and therefore the additional functionality offered is high level features, for example the package includes functions to easily specify geometric shapes, such as cubes, so the user no longer has to specify a cube from primitives such as lines and polygons, which is the required technique when using the standard OpenGL library. Although this feature and others would have undoubtedly been useful when designing certain objects in the scene, it would have been limited in use for the simple reason that a lot of the scene needed to be built up from a low level of detail, i.e. when dealing with the rain particles, these were to be represented and manipulated as simple geometric points, which are the lowest level primitives in a graphic system. Because of this, even if Performer was used, it would have been necessary to drop down a level to the standard OpenGL library to develop much of the application, and therefore the overheads in the time that would have to be invested in mastering the Performer interface, would not yield satisfactory gains to the development process.

I have had some experience of using the standard OpenGL library from computing modules that I have undertaken: 'Introduction to Computer Graphics' (SI23) [11] and 'Advanced Computer Graphics' (SI31) [12], in my 2nd and 3rd year respectively. With this previous experience gained, I was confident that OpenGL would be more than adequate for the project needs. Although some changes will have to be made to the final program to allow it to run in the Performer driven simulator, (see the discussion in the conclusion, section 6), the overheads involved in researching and making these changes, would be far less than the time required to fully learn the Performer interface.

Other factors that influenced this decision included accessibility issues. The standard OpenGL libraries can be accessed from any School of Computing Linux workstation, at any time on any day, however Performer can only be accessed on a limited number of specialist graphics workstations, which would only be available during working hours.

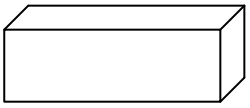
The decision was also made to produce the code in C apposed to C++. As I have had previous experience of using both languages and new that either would be satisfactory, the choice was based on the simple fact that my preliminary background research had uncovered many more useful online OpenGL resources that were written in C.

2.5. Modelling Techniques

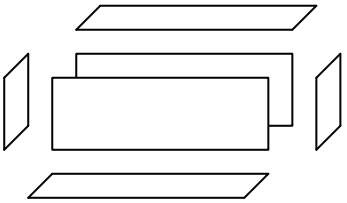
A modelling technique, in the context of computer graphics, refers to how objects in the real world can be modelled and represented in a graphical application. In the early days of computer graphics, only traditional surface based modelling was possible, however as the field has matured, computers have become more powerful, and application programmers have begun to attempt to model more of the complexities that occur in nature, these techniques have evolved significantly and new techniques, such as the use of particle systems, have emerged. D. S. Ebert discusses this technique and others, in his 1996 paper on advanced modelling techniques [13].

2.5.1. Traditional Surface Based Modelling

This refers to modelling real world objects by specifying a set of surfaces to represent those objects. The most common approach is to use polygons. A polygon is a geometric primitive that exists in space and is defined by a set of individual vertices (i.e. points in space), the second chapter of 'OpenGL – A Primer' [1] offers a detailed explanation of these geometric primitives. A set of polygons can then be built up and these can be used to represent the real world object. An example follows:

To represent a cuboid: 

One could specify the following polygons and position them appropriately in 3D space:



This is a powerful technique, as by increasing the number of polygons to be rendered and decreasing their size, it is possible to build up representations of extremely complex objects, and when combined with advanced lighting and shading techniques, the effect can produce extremely realistic models. However this technique is limited for a number of reasons, firstly because it can be computationally expensive to render a scene built up of a large number of small polygons, and more importantly to this particular project, the technique can not adequately represent objects that don't have well defined surfaces, such as the many falling rain particles that make up a rain storm. Therefore this technique will be used to represent certain objects within the graphical environment (such as roads and houses), however another technique will be needed to represent the storm.

2.5.2. Particle Systems

This is a technique, originally proposed by William T. Reeves in his 1983 paper, to provide a way to effectively model "fuzzy objects such as fire, clouds and water"[14]. Such phenomena could not be easily modelled using the standard techniques available at the time, such as using traditional surface based representations, (as described in section 2.5.1), as these objects generally do not have smooth, shiny, well defined surfaces. Therefore a more specialised technique was required to model such objects.

The basic idea behind the technique, as proposed by Reeve's, is to model these fuzzy objects as a cloud of primitive particles that define the objects volume. Once the system of particles has been initialised, more particles can be added, existing particles can move, change form and eventually die from the system. Each of the particles must be represented within the system and their associated attributes, (such as position or velocity), must be continually updated as the system runs. The result of this is the ability to effectively model motion, changes in form and complex dynamics. Such properties are essential for generating a simulated rainstorm, as the rain particles will start in the sky (at the top of the scene), they will be updated (as they fall through the scene) and after they have fallen to the ground (at the bottom of the scene) they will need to be positioned back at the top so they can fall again and produce the effect of a continuous rainstorm.

Particle systems are commonly used in many modern graphical applications. Their use is standard technique within the computer game industry, for generating atmospheric effects to include into games [15]. As the technique was specifically designed as a way to model such natural phenomena, it seemed the sensible choice for fulfilling the requirements of the project. Background research did not uncover any other techniques, which could be used instead of particle systems, and would deliver results that would be as useful in comparison, therefore the decision was made to develop the rain effect using this method.

2.6. Viewing in OpenGL

After a world has been constructed, using the modelling techniques outlined in section 2.5, the view must be set up so that the environment can be observed. Graphics systems use the synthetic-camera model to derive a two-dimensional image, which will be displayed to the screen, from the three-dimensional world that has been modelled. This emulates "what is done by most real-world imaging systems, such as cameras and the human visual system" [1], i.e. with this model two independent entities are required, a set of objects positioned in space and a viewer who observes the objects from a distinct position. After the application programmer has specified these entities, OpenGL will perform a process called projection that will produce the two-dimensional image of the graphical environment, Figure 2.2 illustrates this process.

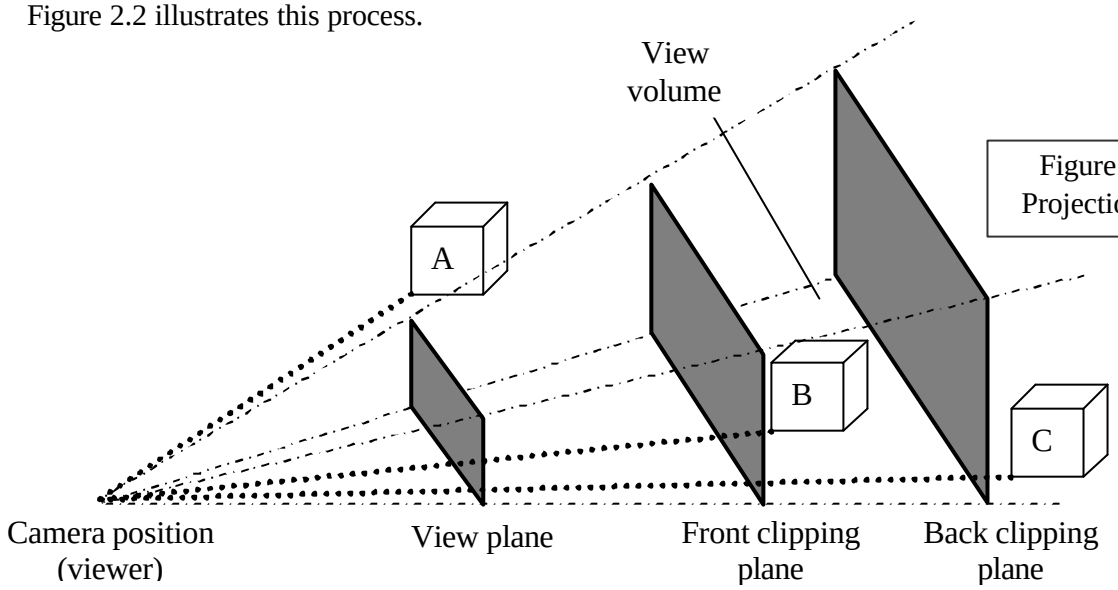


Figure 2.2. The Projection Process

In this example the application programmer builds a graphical environment containing three objects A, B and C. He/she specifies the position from which the scene will be viewed (camera position) and also the front and back clipping planes. OpenGL will draw lines called projectors from all of the scene objects to the viewing position, (represented by the bold dotted lines). The point on the viewing plane that is bisected by an objects projector, is the position on the screen where that particular object will be rendered, therefore the view plane can be thought of as being a grid of pixels that make up the screen, with each pixel corresponding to one or more projections. From this example, the projectors from object A will not pass through the view plane, meaning the object is not in the field of view when this particular camera position is used to view the environment. The projectors from objects B and C will both pass through the view plane, however only B will be displayed on the screen, as C does not fall within the view volume (i.e. the area between the front and back clipping planes) and only objects within this volume are ever displayed. At points on the view plane where there is no bisecting object projector, OpenGL will shade the corresponding pixel on the screen a default colour that the application programmer can specify.

3. Design

When designing the application, there were three primary areas that needed to be tackled, firstly the design of the graphical environment, secondly the design of the particle system that would simulate the rain effect, and thirdly, how the application user would interact with these elements.

3.1. The Graphical Environment

As the rain effect will not be developed within the actual driving simulator, it was first necessary to construct a graphical environment using OpenGL, within which the effect could be developed.

As the effect is intended for use in a driving simulator, the graphical scene was designed to mimic this as closely as possible, therefore a design was constructed that contained a network of roads with road markings and signs, surrounding grassy areas, road side houses and a clear blue sky overhead, with the optional addition of scene fog, of a colour specified by the user, that can be toggled on and off. The surface of the environment is perfectly flat, at a height of y = 0, and figure 3.1 illustrates the developed design:

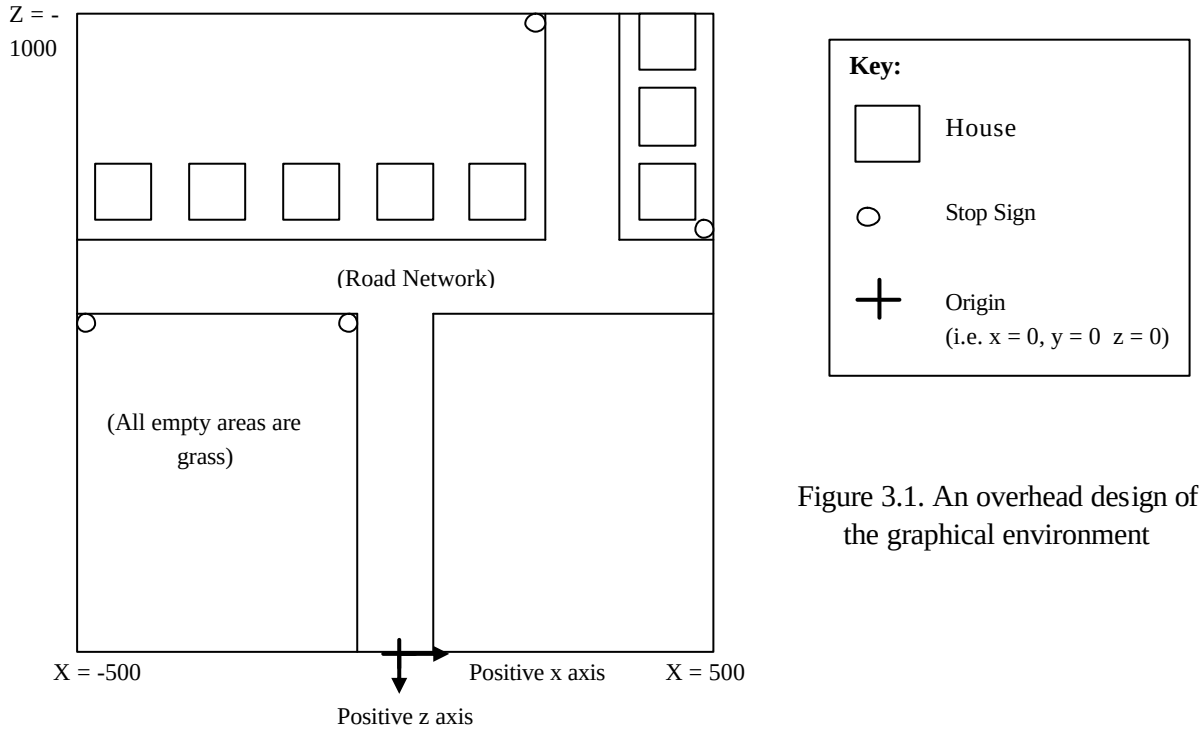


Figure 3.1. An overhead design of the graphical environment

All of these scene objects were designed using surface based modelling (see section 2.5.1). Further information about how this was actually achieved can be found in the implementation section (see section 4.2).

3.2. The Particle System

As previously mentioned, the decision was made to model the rainstorm using a particle system (see section 2.5.2). The requirements for this system were as follows:

Basic Requirements (to fulfil minimum objectives):

1. The rain effect must be implemented in a way so that it can be switched on and off, whenever the user wishes.
2. The particle system will contain a fixed number of particles, so these particles must be initialised and updated in a way that convincingly produces the effect of a continuously falling rainstorm.
3. The particles will occupy points in 3D space and it is these points that must be drawn in the scene, in a specified colour, to display the rain.
4. As it would be extremely inefficient to model the rainstorm falling over the entire scene, (as the user will only ever view a subset of the scene at any one time), the particle system must only run in a subset of this area, and then methods must be included so the particle system is translated around the environment as the viewpoint changes.
5. When the user exits the program, the memory that was used to store all the information about the particles must be de-allocated.

Additional Requirements:

1. Methods must be included to allow the user to specify how many particles will be used in the system and to control the rate at which these particles fall. This will enable to the user to specify both the density and the severity of the rainstorm.
2. Methods must be included so the user can control the size of the falling particles. This should be implemented in a way that will allow the user to perform this operation as the application is running, and will enable the user to alter the appearance of the falling raindrops.
3. Methods must be included to allow the user to specify the colour that the particles will be displayed in. This will allow the user to create different visual effects, i.e. grey particles will look like rain, whilst white particles will give the effect of snow.
4. An option must be included to allow the user to display the rain particles as lines, as well as simple points. This should be implemented in a way that will allow the user to change this setting at any time when the application is running, and will provide two very different looking rainstorm effects.

To fulfil these requirements, the system of particles must be continually updated, so rain always appears to be falling when the system is switched on. All the particles should exist in a specified three-dimensional volume and this volume should be translated around the environment as the viewpoint changes, to ensure that the particles always appear to be falling overhead, which will give the illusion that rain is falling across the entire scene, regardless of the viewpoint. This is illustrated in Figure 3.2. The arrows represent two different viewpoints and the surrounding cubes illustrate the volume in which the rainstorm will occur, when these viewpoints are used. The rain will fall from the top face of the cube to the environment surface and after the particles reach the surface, they will be moved back to their original starting height, so they can fall through the scene again and create the continuous rainstorm.

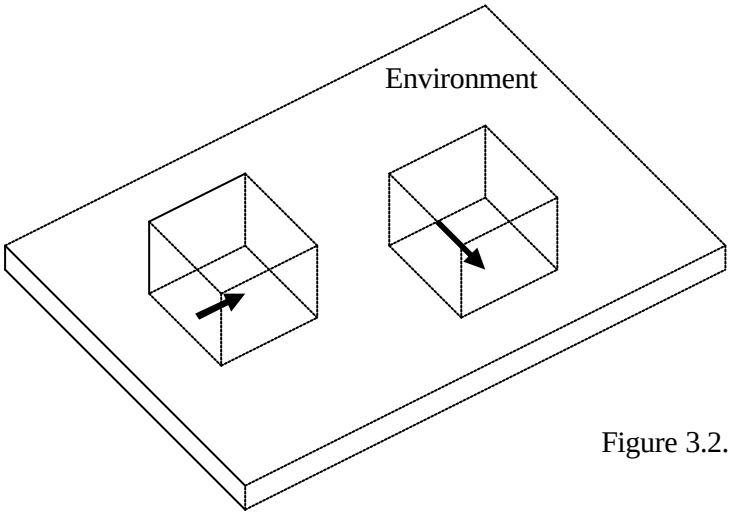


Figure 3.2. Particle System Design

3.3. User Interaction

The user must be able to efficiently interact with all elements within the application, some of these interactions will be performed in an initialising menu, which can be entered when the application is first executed, and others will be done by key presses and mouse inputs, when the application is actually running.

Interactions with the environment:

1. When the application is first run the scene will be viewed from a default position, however the user must be able to change this viewpoint, so the entire environment can be explored effectively. Therefore the user must have 360-degree control of the view direction, so he/she can fully look around, and also be able to move forward and backward in this direction, to allow full exploration.
2. The user must be able to specify the colour of the scene fog and be able to toggle it on and off when required.
3. The user must be able to easily exit the application.

Interactions with the particle system:

The user must be able to…

1. Specify how many particles will be used to represent the rainstorm, (i.e. the density of the storm).
2. Specify the rate at which the particles descend (i.e. the speed of the falling rain)
3. Specify the colour that the particles will be displayed in.
4. Toggle the rain on and off.
5. Toggle between the rain particles being drawn as points or lines.
6. Alter the size that the particles are drawn in the scene, (when the raindrops are drawn as points).

4. Implementation

4.1. OpenGL Initialisation

Before the developed designs could be implemented, it was first necessary to initialise OpenGL. This is achieved by setting up the application properties in the programs 'main()' function, (see page 3 of Appendix C for the code and additional details).

The function 'glutCreateWindow()' creates a window that the graphical application will run within and the command 'glutFullScreen()' instructs OpenGL to display this window on a full screen. The display mode is set up and double buffering is enabled, meaning that two buffers will be used when refreshing the graphical scene, so that when the contents of one buffer is being displayed by the graphics hardware, the second one will contain the next frame to be drawn, so that it can be immediately switched over when the new frame is required, this ensures that any animation will appear to happen smoothly.

Also within the 'main()' function, callback functions for the application are specified. These define how the program should react to events that may occur, such as a user making an input. When an event is detected, for example a keyboard entry being made, the event is placed in an event queue and then these events are processed sequentially from the queue, by the callback functions written by the application programmer. In this example, the callback function that manages keyboard inputs, 'glutKeyboardFunc()', will call a subsequent function 'Key()' that will process this input appropriately. If there are no events to process in the queue, the 'glutIdleFunc()' will be called and appropriate actions will be taken. Finally the function 'glutMainLoop()' is called, this causes the program to enter an event-processing loop, so that the system will always continuously monitor for events and react to them appropriately. These functions are explained in detail in Angel's OpenGL Primer [1].

Another important function that is called within the 'main()' function is 'glClearColor()'. This specifies what colour should be displayed to the screen at any position where there is no objects to be displayed, this was set to sky blue and was used to create the sky effect over the environment. Other methods called in the 'main()' function are explained by comments in the code on page 3 of Appendix C.

4.2. The Graphical Environment

After OpenGL had been initialised, it was then possible to begin to develop the graphical environment.

4.2.1. General Construction

All the objects that make up the graphical scene were built up from simple lines and polygons. These geometric primitives were then translated, scaled and rotated, by use of the 'glTranslatef()', 'glScalef()' and 'glRotatef()' OpenGL functions, to position them appropriately in three-dimensional space, to create the desired objects and environment. OpenGL provides functions to easily produce these primitives, the function 'GL_LINES' takes two points in 3D space, (points can be specified by the function 'glVertex3f(x, y, z)'), and renders a straight line between them, and with the subsequent functions, 'glLineWidth()' and 'glLineStipple()', the width and style of the line can also be specified. The function 'GL_QUADS' takes four points and renders the rectangular area that the points specify, and similarly, the function 'GL_POLYGON' can take any number of points and will render the described polygon. These functions were used to build up the objects within the scene and then these primitives were rendered in the appropriate colour, by the use of the function 'glColor3f(r, g, b)', which accepts three numbers as arguments, corresponding to the desired RGB value. When constructing a geometric primitive, for example a polygon, the functions 'glBegin(GL_POLYGON)' and 'glEnd()' were used to specify the beginning and end of that primitive. The following code helps illustrate these concepts:

```
glBegin(GL_POLYGON);                // Create a new polygon.
{
        glColor3f(1, 0, 0);         // Colour the polygon red.
        glVertex3f(-0.5, -0.5, 0);
        glVertex3f(-0.5, 0.5, 0);   // Specify four points in three dimensional space,
        glVertex3f(0.5, 0.5, 0);    // (i.e. a square polygon with midpoint at (0, 0, 0)).
        glVertex3f(0.5, -0.5, 0);
}
glEnd();                            // End the specification of the polygon.
```

The optional fog was included into the scene by using the OpenGL fog functions, this code and a supporting explanation, can be seen on page 3 of Appendix C. Figure 4.1 is a screenshot of the developed graphical environment, it contains all of the static objects that were developed for use in the application. Figure 4.2 shows the same scene as Figure 4.1, but with the addition of scene fog.

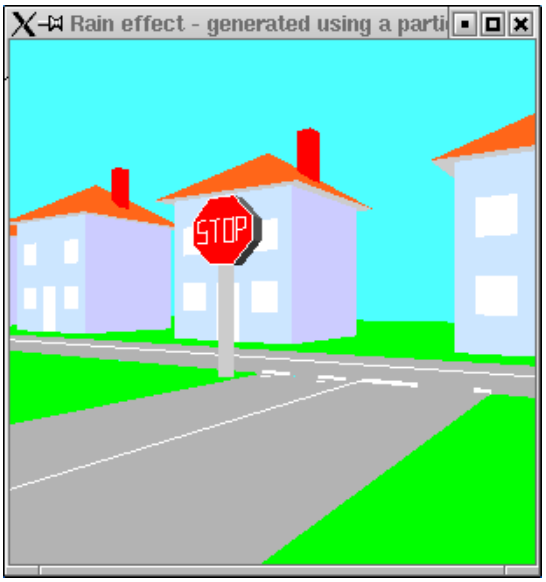


Figure 4.1. A screenshot of the developed graphical environment

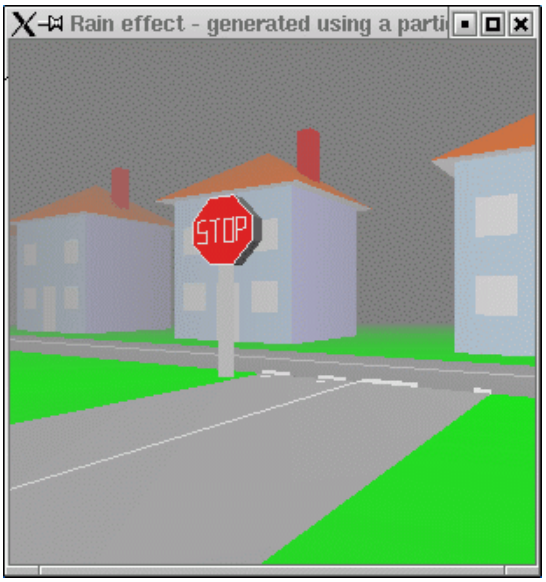


Figure 4.2. A screenshot of the developed graphical environment in fog

4.2.2. Display Lists

As it can be seen from the original environment design (Figure 3.1) and the previous screenshots (Figure 4.1 and 4.2), some of the scene objects, such as the house, were to be rendered multiple times across the environment, therefore a method was required so that the geometric primitives for that object could be specified only once, yet the object could be rendered more than once in different positions. The alternative to using such a method would be to specify the primitives for 8 distinct houses, 4 distinct stop signs and 2 distinct sets of junction markings, which would obviously be very time consuming and an extremely inefficient solution.

Fortunately this desired functionality could be achieved through the use of OpenGL Display Lists. Here it is possible to specify an object, in the normal fashion, but through the use of the function 'glNewList()', it is possible to store all of the geometric primitives that make up that object, in a list that can be accessed at anytime. Therefore these primitives can easily be taken and drawn to the scene as many times as required, and through the use of the transformation functions mentioned earlier, they can easily be moved around the environment. Display lists were not needed for all the objects that were only rendered once, such as the individual grassy areas and roads, as their use would not have brought about any benefits to the application. The function that defines the stop sign object, 'DefineSign()', makes use of a display list, this code can be seen on page 13 of Appendix C.

4.2.3. Viewing

Whilst the development of the graphical environment was underway, it was also necessary to specify how the scene should be viewed, so the program could be compiled and the environment could be observed. Initially a fixed view was set up, (discussed in section 2.6), the function 'gluLookAt()' was used to achieve this, as follows:

gluLookAt(CameraX, CameraY, CameraZ, focusPointX, focusPointY, focusPointZ, 0, 1, 0);

The first three arguments are variables that store the position of the camera within the three-dimensional environment, the next three store the position that the camera was focused on, and the last three specify the direction of the up vector. The initial camera position is (0, 20, 0) and the focus point is (0, 20, -2000), therefore the scene is being viewed from a height 20 above the environment surface, (to give the perspective of being in a car), in the negative z direction. The function 'glFrustum()' was then used to specify the position of the front and rear clipping planes, to control the users field of view, so the scene could be viewed appropriately. This initial viewpoint is illustrated in Figure 4.3.

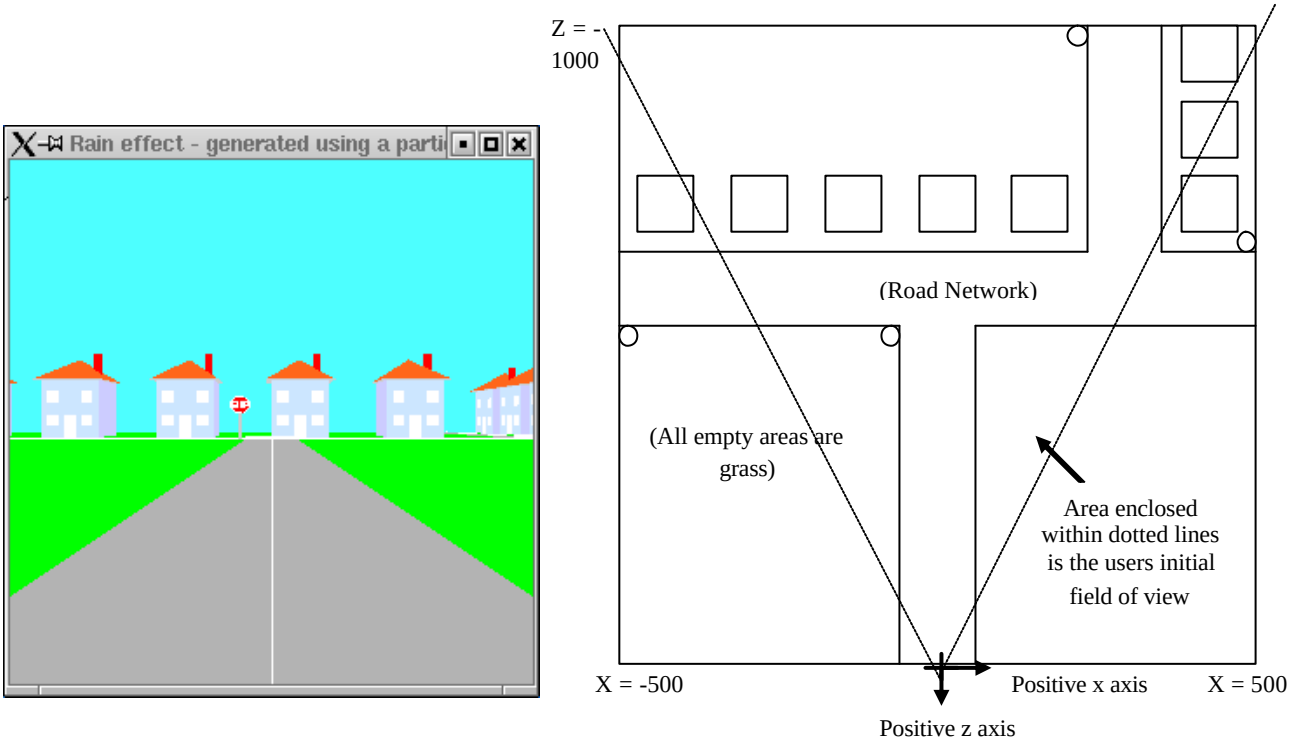


Figure 4.3. The initial viewpoint when viewing the graphical environment

4.3. The Particle System

After the graphical environment had been developed, it was then possible to implement the particle system that would simulate the rainstorm.

4.3.1. Data Types

It was first necessary to construct the data types that would store the particle system data, two new types were required: One to represent the individual particles, containing all the necessary particle attributes, and another to represent a system of particles. The following definitions were used:

```
///// Definition of a particle
typedef struct                      // Define a new structure
{
        GLfloat posA[3];            // The current position of the particle
        GLfloat posB[3];            // A secondary position for the particle
        GLfloat Vx, Vy, Vz;         // The particles current velocity in each direction
        GLfloat colour[3];          // The current RGB value of the particle
        Glfloat size;               // The current size of the particle
}Particle;                          // This new data type is called 'Particle'

///// Definition of a particle system
typedef struct                      // Define a new structure
{
        Particle *array;            // A pointer called array that points to a Particle type
        int total;                  // The total number of particles in the system
        int update;                 // The number of particles to update per Frame
        GLfloat Rx, Ry, Rz;         // The distance the particles travel in each direction
        GLfloat Vx, Vy, Vz;         // The global velocity of the particles in the system
        GLfloat red, green, blue;   // The default particle colour in the system
        GLfloat defaultSize;        // The default particle size in the system
}ParticleSystem;                    // This new data type is called 'ParticleSystem'
```

These two new data types form the basis of the particle system. Instances of these new types will be declared and the attributes will be continually updated as the system runs, to generate the desired dynamic particle system. John van der Burg's paper [15] was a useful resource when deciding which particle attributes would be needed in this particular system.

4.3.2. Initialising the System

When the application is executed, the function 'InitParticles()' is executed, (see page 5 of Appendix C for the code and further explanation). This initialises the system of particles so that it is set up correctly, ready for when the user toggles the system on. The function takes the following arguments:

InitParticles(&psys, start, end, particleColour, particleSize, numOfParticles, numOfFrames);

The first argument supplied must be of type 'ParticleSystem' and this is the particle system that will actually be initialised. The second and third arguments specify the mean start and end position of all the particles within the system. The fourth variable specifies the colour that the particles will be displayed in and the fifth specifies their size. The sixth specifies how many particles will be used within the system and the final argument specifies how many frames will be used to display a particles descent. (Note: There must be more particles in the system than frames used to display their fall, and that the number of particles divided by the number of frames must leave no remainder. The reason for this will be apparent in the discussion of how the particle system is updated – section 4.3.3).

The function computes the distance that the particles will fall in each direction and stores the result, (calculated from the supplied start and end points). It then works out the velocity that the particles must be assigned to fall these distances, given the amount of frames that will be used to display this descent. The number of particles that need to be updated per frame (number of particles / number of frames) is then calculated, (more information on the reason for this can be found in section 4.3.3). Memory is then dynamically allocated using the 'malloc' command, (see pages 259-261 of reference [3]), to store all of the particles that will exist within the system, and then these particles are all assigned a random primary position (posA) in a pre-specified three-dimensional rectangular area, which is directly above the initial viewpoint and is illustrated in figure 4.4. A secondary particle position is also specified (posB) that is at a random position, up to the fall distance per frame, below the primary position. This is used when the user specifies to display the rain as lines instead of points and is explained in section 4.3.4.

The random positions are generated by using the 'drand48()' function, which returns a random number in the range zero to one, (the code and a supporting explanation for this can be found toward the end of the 5th page of Appendix C). The particles are also assigned a velocity of zero, as they will not move until the user toggles the system on and the update function is called. Figure 4.5 is a screenshot of a particle system that has been initialised, (the particles have been displayed), the red arrow shows the initial viewpoint.

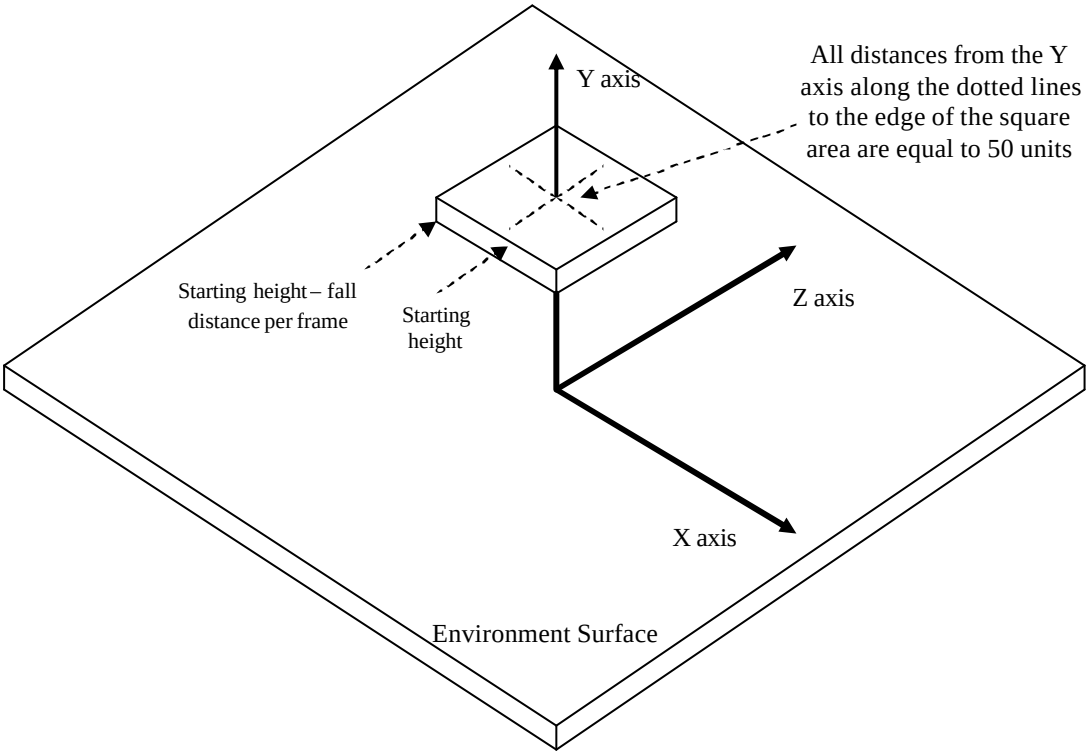


Figure 4.4. Particle System Initialisation

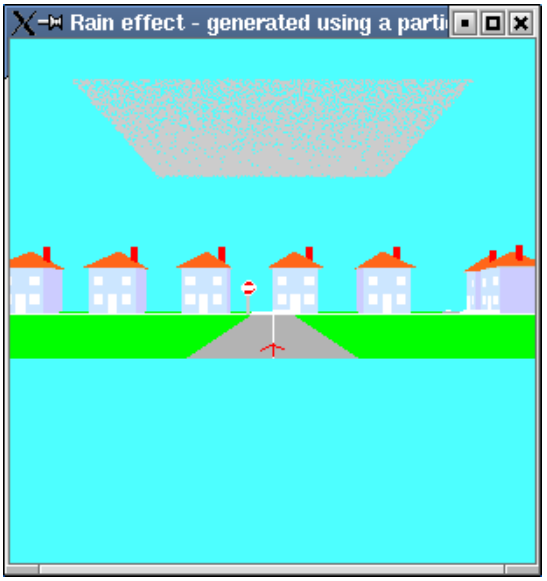


Figure 4.5. A screenshot of an initialised particle system

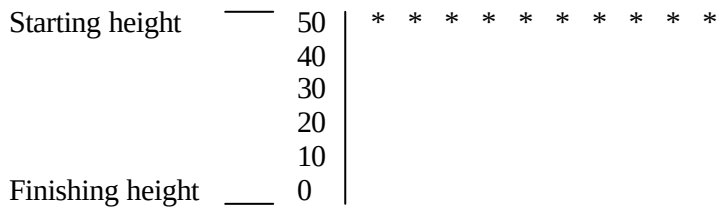
4.3.3. Updating the System

As soon as the user has activated the rain, the 'ParticleUpdate()' function will be called, (see page 6 of Appendix C for the code and additional information). The function works in two phases, firstly it assigns all the particles a velocity so they will begin to descend from their initialised position towards the ground, and after this has been completed the second phase is entered, where the fallen particles are positioned back up to their original starting height, so they can fall through the scene again and create the effect of a continuously falling rainstorm.

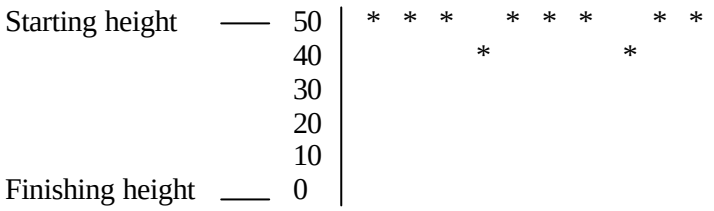
When this first phase is entered, the function staggers the assignment of velocities so that the volume of initialised particles will fall in distinct bands, creating the effect of smoothly falling rain. If this was not done and all the particles were assigned a velocity at the same time, they would all fall together and this would not create the required effect. The following example illustrates how this assignment of velocities creates the desired effect:

* Total number of particles used within the system = 10
* Number of frames used to display a particles descent = 5
* Therefore number of particles that must be updated each frame = (10 / 5) = 2
* Starting height of the initialised particles = 50
* Height of the particles at the bottom of their descent = 0
* Therefore total distance that the particles must fall = (50 – 0) = 50
* Therefore velocity that the particles must have to fall this distance in the specified number of frames = (-50 / 5) = -10 (note: negative because the particles are falling downwards)

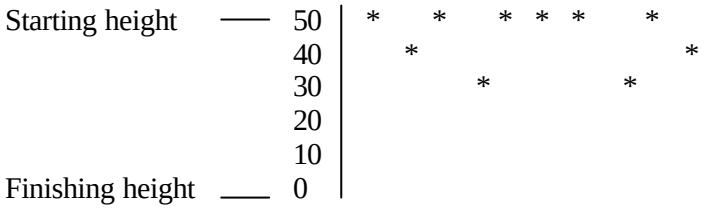
1. After initialisation all particles are at the top of their descent and have zero velocity, i.e.



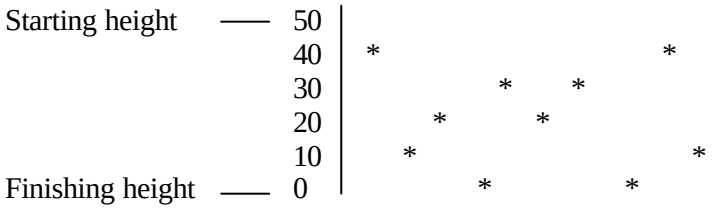
2. After the user has toggled the system on, 'UpdateParticles()' will be run for the first time and this will assign the first 2 particles a velocity so they can fall a 5^{th} of the total distance, (as they have 5 frames to fall the total distance) i.e.



3. Similarly, after ‘UpdateParticles()’ has been run for the second time, the next 2 particles will be assigned a velocity. Remembering that the first 2 still have their velocities assigned, we have the following situation:



This process continues until eventually ‘UpdateParticles()’ has been run 5 times and all the particles have had a velocity assigned, i.e.



4. At this stage the system will be in a position where the first particles to be assigned a velocity will just be finishing their descent and the last ones will just be starting. Here the first phase of ‘UpdateParticles()’ terminates and the second phase is entered. Figure 4.6 is a screenshot of this first phase of the function in action.

The second phase continually moves all of the particles that have reached the bottom of their fall back up to their original starting height, (this is achieved by simply adding on the vertical displacement to the Y component of the particles position). These particles will simply begin to fall through the scene again, as they still have a velocity assigned, and it is this looping that creates the effect of a continuously falling rainstorm. This can be seen in figure 4.7.

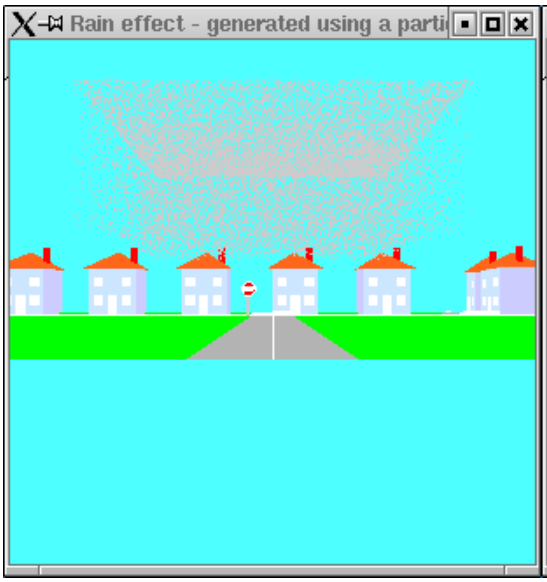


Figure 4.6. A screenshot of the first phase of the particle system update

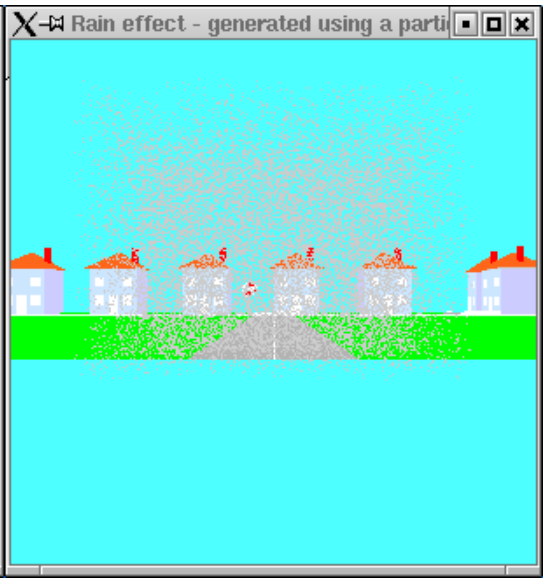


Figure 4.7. A screenshot of the second phase of the particle system update

4.3.4. Displaying the System

The function ‘DisplayParticles()’ is used to display the particles within the graphical environment, (page 7 of Appendix C). After the user has started the system and the ‘UpdateParticles()’ function has been executed for the first time, the ‘DisplayParticles()’ function will then iterate through all the particles within the system and draw a point in the environment, (in the colour and size that was supplied to the ‘InitParticles()’ function), where each of the particles are positioned (i.e. posA). As long as the rain effect is switched on, ‘DisplayParticles()’ will be called after each execution of the ‘UpdateParticles()’ function, this will therefore continually redraw the new particle positions within the environment, as they are updated, and will produce the effect of falling rain.

When the user specifies that the rain should be displayed as lines instead of points, the function will execute a different piece of code, through the use of ‘if statements’ (see page 7 of Appendix C), and will instead render a line between the primary and secondary particle positions (posA and posB). These two display methods produce very different effects, as illustrated in figures 4.8 and 4.9.

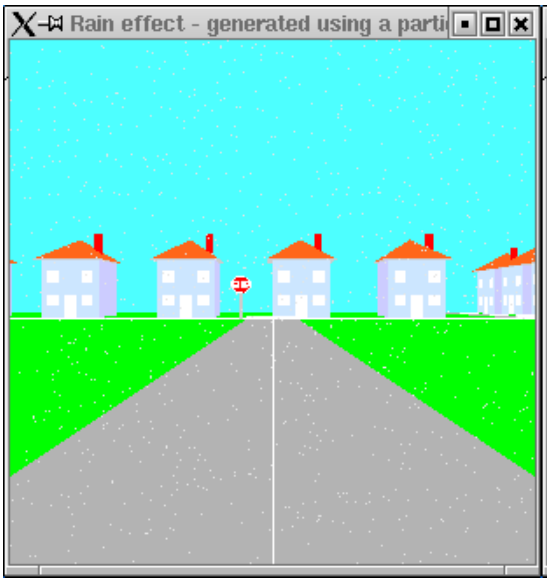


Figure 4.8. A screenshot of the rainstorm displayed as points

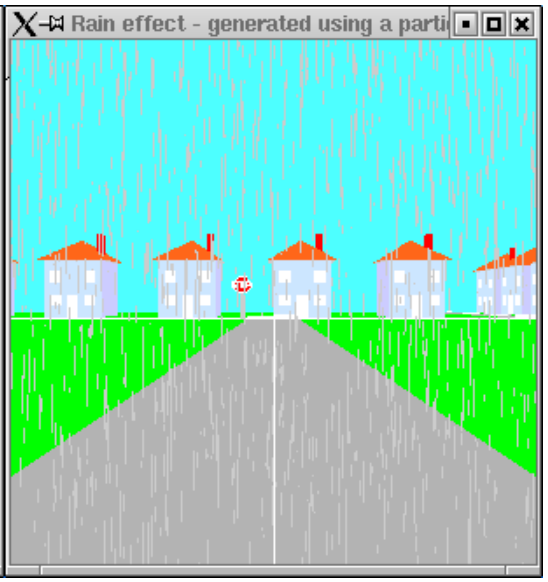


Figure 4.9. A screenshot of the rainstorm displayed as lines

4.3.5. Translating the System

As the system of falling particles is initialised directly above the initial viewpoint, the system must be translated around the environment when this viewpoint changes, to ensure that it is always positioned directly above the current viewpoint. This is achieved using the 'MoveRain()' function, (page 7 of Appendix C).

This function is called whenever the viewpoint is changed. It iterates through all of the particles within the system and amends the position of each one (posA and posB), by the exact displacement in the X and Z directions that the camera experienced. This ensures that the system is always poisoned directly above the current viewpoint and creates the desired visual effect of rain falling across the entire environment, irrespective of viewpoint.

4.3.6. Removing the System

When the user exits the application the function 'RemoveParticles()' is called, (page 6 of Appendix C). This de-allocates all the memory that was used to store the particle system data, (allocated during the 'InitParticles()' function call, see section 4.3.2). This is achieved through the use of the 'free()' function, (see pages 259-261 of reference [3]) and assures that this memory, which is no longer in use, will be made available to other applications.

4.4. User Interaction

The required user interaction is achieved using both an initialising menu, which can be entered when the program is first executed and is used to set up the overall properties of the system, and a combination of key presses and mouse inputs, which can be made when the application is running and alter the state of the system.

4.4.1. Initialising Menu

When the program is executed, the first function to be called within the 'main()' is 'initialization()', (page 4 of Appendix C). This firstly outputs to the screen a description of the controls that can be used within the application, and then presents the user with the option to either enter the initialising menu, so he/she can customise the system settings, or to instead bypass this menu and run the application with the default system settings. Figure 4.10 is a screenshot of the developed menu.

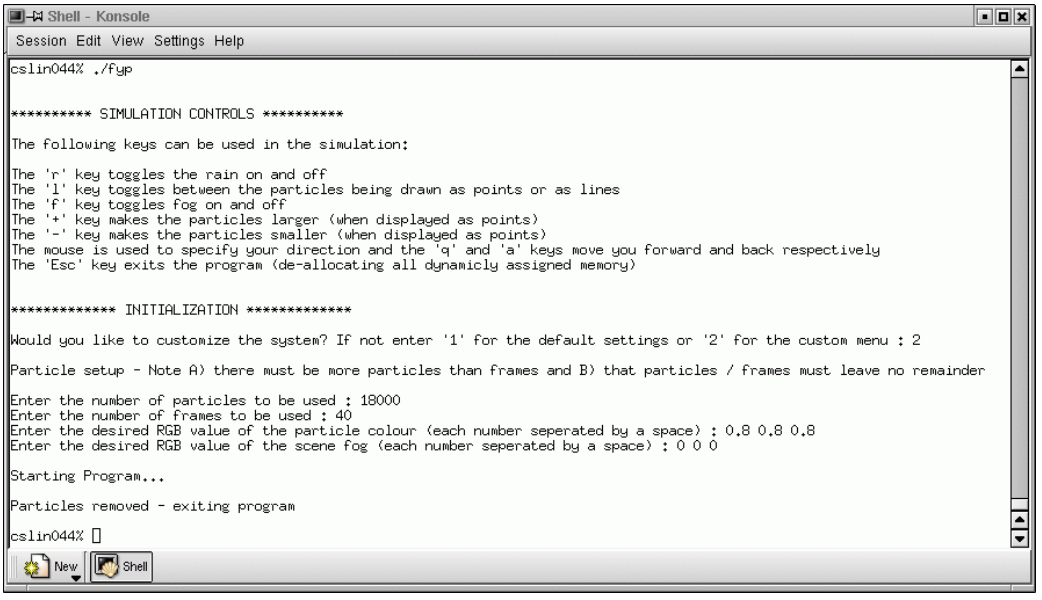


Figure 4.10. The initialising menu

If the menu is entered, the user can alter the general properties of the particle system by A) stating the number of particles that will be used within the system, (i.e. the number of raindrops that will be visible), B) the number of frames that will be used to display a particles fall (i.e. the more frames used, the slower the rain will fall), and C) the colour that the particles will be displayed in, (supplied as an RGB value). Here the user will also have the opportunity to specify the colour of the optional scene fog. These values are then stored as variables within the program, overwriting the default values included in the code, and are then taken as arguments by the functions described in section 4.3, to produce the desired system effects.

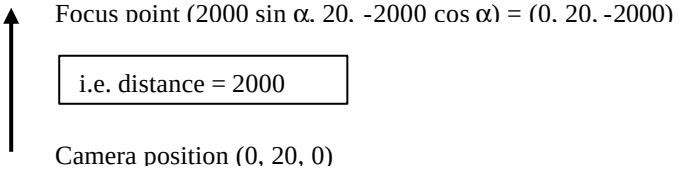
4.4.2. Interactions within the running application

After the user has specified the system settings in the initialising menu, or instead has selected the use of the default system values, the application will then run. The user is then able to interact with the application through the use of subsequent mouse movements and keyboard strokes. When a mouse input is made, the event will be detected and the 'glutPassiveMotionFunc()' callback function will be invoked, this will then call the 'Mouse()' function, (page 12 of Appendix C), which will process the input. Similarly when a keyboard input is detected, 'glutKeyboardFunc()' will be invoked and this will call the 'Key()' function, (page 10 of Appendix C), to carry out the necessary processing.

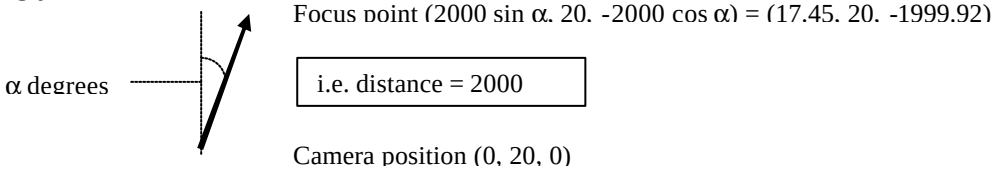
Mouse Inputs:

The mouse is used to change the direction of the current viewpoint. When the user moves the mouse into the left quarter of the application window, the view direction will rotate to the left by a fixed angle of half a degree, and similarly, if the mouse is moved into the right quarter, the view direction will rotate to the right by half a degree. If the mouse is positioned in the central area of the window, the current view will not be altered. When an input is made the actual camera position is not changed, instead the focus point, which always lies at a distance of 2000 away from the camera (see section 4.2.3), will have it's X and Z components amended to take into account the new viewing angle, (calculated using basic trigonometry equations [16]). This process is illustrated in the following example:

The initial viewpoint has viewing angle $\alpha = 0$ degrees, (this direction looks straight down the negative Z axis), i.e:



Next the mouse is moved into the right side of the application window, so the viewing angle changes by 0.5 degrees in that direction (i.e. α now equals 0.5). The focus point will therefore be updated accordingly:



This process is then repeated every time a new mouse position is registered and the method allows the user to have 360 degree control of the view direction.

Keyboard Inputs:

Figure 4.11 is an overview of all the keys that can be used within the application and their functions.

Key	Function
‘q’	Moves the viewpoint forward a fixed distance in the view direction.
‘a’	Moves the viewpoint backwards a fixed distance in the view direction.
‘f’	Toggles the scene fog on and off, (default is off).
‘r’	Toggles the rain effect on and off, (default is off).
‘l’	Toggles between displaying the particles as points or lines, (default is points).
‘+’	Increases the size of the displayed particles, (when they are drawn as points).
‘-‘	Decreases the size of the displayed particles, (when they are drawn as points).
‘Esc’	Exits the application, returning the user to the command prompt.

Figure 4.11. A description of the application keyboard controls

The ‘q’ and ‘a’ keys allow the user to change the camera position. These keys move the camera forward and backward a fixed distance, (specified in the code to be 3 units, on page 1 of Appendix C), in the current view direction (selected using the mouse). This is achieved by calculating the X and Z displacements that will occur as a result of the movement (dependant on the movement magnitude of 3 units and the current viewing angle), and then adding or subtracting these displacements to both the camera position and the focus point, to move the viewpoint forwards or backwards respectively.

Whenever the ‘f’, ‘r’ or ‘l’ keys are pressed, a corresponding Boolean variable, either ‘fog’, ‘rain’, or ‘line’ (that are all defaulted to false and are declared globally at the beginning of the code, page 1 of Appendix C) changes value. These variables are used to control the execution of appropriate pieces of application code. When ‘f’ is pressed, indicating that the user wants to turn the fog on, an ‘if statement’ becomes valid and the fog function is enabled (‘glEnable(GL_FOG)’). When the key is pressed again, this statement becomes invalid and an ‘else statement’ is executed that disables the fog (‘glDisable(GL_FOG)’). When ‘l’ is pressed the event is managed in a similar way, (see code for further explanation). When ‘r’ is toggled on, the ‘ParticleUpdate()’ function is called, (see section 4.3.3). After this function has been executed, it continually runs until the application is terminated. When the user presses ‘r’ for a second time, the ‘DisplayParticles() function is disabled so that the particles are not drawn in the environment, however as ‘ParticleUpdate()’ is still operating, when ‘r’ is pressed again the falling particles will be displayed once more.

When the ‘+’ and ‘-‘ keys are used, a variable that specifies the size that the particles are displayed as, is incremented and decremented respectively. This size is initially set to 1, and every key press adds or subtracts 0.5 to this value. A method is included so that if the user attempts to reduce this size below 0, it is set back to 0.5 so the particles will always be visible.

When the ‘Esc’ key is pressed the ‘RemoveParticles()’ function is called, this de-allocates the memory that was used to store the particle system data (see section 4.3.6). The function ‘exit(0)’ is then finally called, which causes the program to terminate and the user is returned to the command prompt.

4.5. Running the Application

After all the elements of the application had been implemented, as described in this chapter, it was then possible to compile the application code (fyp.c) and run the application. For this task a Makefile was constructed that would be used to efficiently compile the code with all the required libraries and flags. (See pages 532 – 538 of reference [3], for an overview of Makefiles). After compilation was sucessfully completed, an executable called ‘fyp’ was created and running this (by typing ‘./fyp’ at the command prompt) causes the application to execute. The code for both the application and the Makefile are included on the submitted floppy disk.

4.6. Testing

System testing was carried out during the actual implementation of the application. After a new function had been implemented in the code, it would then be exhaustively tested to make sure it functioned correctly and didn’t produce any unexpected results. Methods were also built into the code to perform various tests, such as a facility in the ‘Initialization()’ function, which checks if the user’s inputs are valid, (see page 4 of Appendix C), and the method in the ‘InitParticles()’ function, which checks to see if the system succeeded to dynamically assign enough memory to store all of the required particle data, if this was not achieved the user would be informed and the program would exit, (see page 5 of Appendix C).

5. Evaluation

5.1. The Graphical Environment

The developed graphical environment fulfilled the stated requirements. It was a convincing representation of a world containing the typical objects that would be expected in a driving simulation and served as a satisfactory environment, that from within which, the primary task of constructing a weather effect could be adequately achieved.

When comparing a typical screenshot of the developed environment, (refer back to figure 4.1), with the environment rendered in the actual Leeds University Simulator, (refer back to figure 2.1), it can be seen that the developed environment falls significantly behind in terms of believability. Improvements could have clearly been made in the implementation, to produce the model to a higher degree of realism, such as the inclusion of the following additions and amendments:

- Making use of the OpenGL texture mapping facilities, (see Chapter 5 - 'Texturing', pages 117 – 179, of reference [2]). Realism could be improved by applying appropriate textures to the surfaces that were built up to model the scene objects, instead of simply displaying each surface as a single colour. For example, instead of simply displaying the grassy areas as a green surface, a grass texture could have been applied to produce a more believable effect.
- Making use of the OpenGL lighting facilities (see Chapter 6 – 'Advanced Lighting and Shading, pages 181 – 287, of reference [2]). Realism could be improved by setting up appropriate scene lighting, perhaps using a light source positioned overhead to simulate the sun. This could have given the effect of different light intensities falling on different surfaces across the environment, which would again make the scene appear more believable when viewed.
- More scene objects could have been modelled and included into environment, such as traffic lights and trees, and the models could also have been constructed to a significantly higher level of detail, by using more polygons for the representations. This would obviously produce a richer, more believable environment than the one that was developed.
- The developed environment was perfectly flat, as this made the task of implementing the weather effect and the user interaction less problematic, however in terms of believability, if slopes were included into the environment, to simulate hills that could be driven up and down, this would be a clear improvement.

All these improvements would yield a more believable scene and are all potential future enhancements that could be made, however the decision was made not to implement them, due to the extra development time that would be required for this task. Although important, the development of the graphical environment was not central to the project, instead it was the particle system that was of primary importance, and as the developed environment was adequate, it was therefore decided that this extra time would be better spent developing the actual weather effect.

5.2. User Interaction

The implemented user interaction satisfied the requirements of the application. The user was able to fully explore the graphical environment and adequately interact with the particle system. As with the evaluation of the graphical environment, (see section 5.1), the user interaction was not of central concern to the project, therefore any potential improvements that could have been made to the user interaction capabilities, were not implemented, in favour of spending the extra time developing the actual rain effect.

Omitted potential improvements include:

- Implementing a user interaction that better simulates driving a car through the environment. The viewpoint control [20] used in the developed solution effectively delivers the sensation that the environment is being walked through. The viewpoint can only be moved at a fixed rate by each key press and this does not deliver the sensation of driving. A better solution would be to implement some sort of throttle and brake system, so the rate at which the viewpoint is changed can be increased and decreased respectively. This would clearly give the user a greater feeling that he/she is of actually controlling a car and would be a definite improvement in terms of interaction.
- Implementing a menu that can be used to re-initialise the system, which can be accessed while the application is running. An implementation where the user is able to change the system settings, which were originally set in the initialising menu, without exiting the actual application, would be advantageous. This would enable the user to change overall system settings, such as the number of particles used and the rate at which they descend, and observe the changes immediately in the running application.

5.3. The Particle System

5.3.1. Fulfilling the Requirements

The developed particle system fulfilled all of the basic system requirements, (see section 3.2) and went some way to towards fulfilling the additional requirements. The additional methods that were included indeed enabled the user the ability to modify the properties of the weather effect, by both allow him/her to specify the density of the storm and the rate at which it falls. They also enabled the user the ability to generate different weather effects, for example, by specifying that the particles should be coloured white, by enlarging their size and decreasing the rate at which they fall, this creates an effect that looks like snow falling through the scene, figure 5.1 is a screenshot of this.

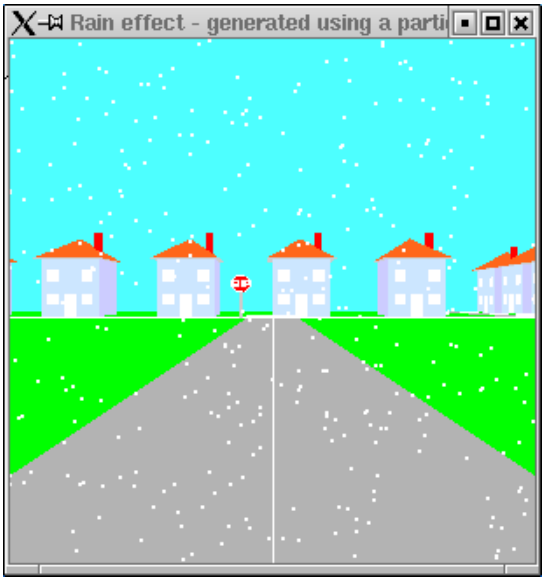


Figure 5.1. A snow effect, generated by altering the particle system settings

5.3.2. Comparing Rain Effects – A Peer Evaluation

The overall objective of the project was to produce a realistic rain effect. As the degree to which realism was achieved is a subjective measurement and different people will inevitably have different opinions on the subject, this made the task of formally setting out evaluation criteria difficult. I felt the best approach that could be taken to produce a useful evaluation, would be to ask several friends for their views on the believability of my effect, so a general sense of opinion could be obtained. I realised that this task would be difficult if the volunteers did not have any basis for comparison, so thought it would be helpful to show them a state of the art rain effect in addition to mine, so the two effects could be examined and useful feedback about the effectiveness and limitations of my effect could be gathered. The latest driving game that I had access to, which contained a rain effect, was used for this comparison. This was ‘Grand Theft Auto 3’ [17], which was developed by ‘Rockstar

Games' [18] and was released in 2002. Figure 5.2 is a screenshot of the rain effect that is used within the game.

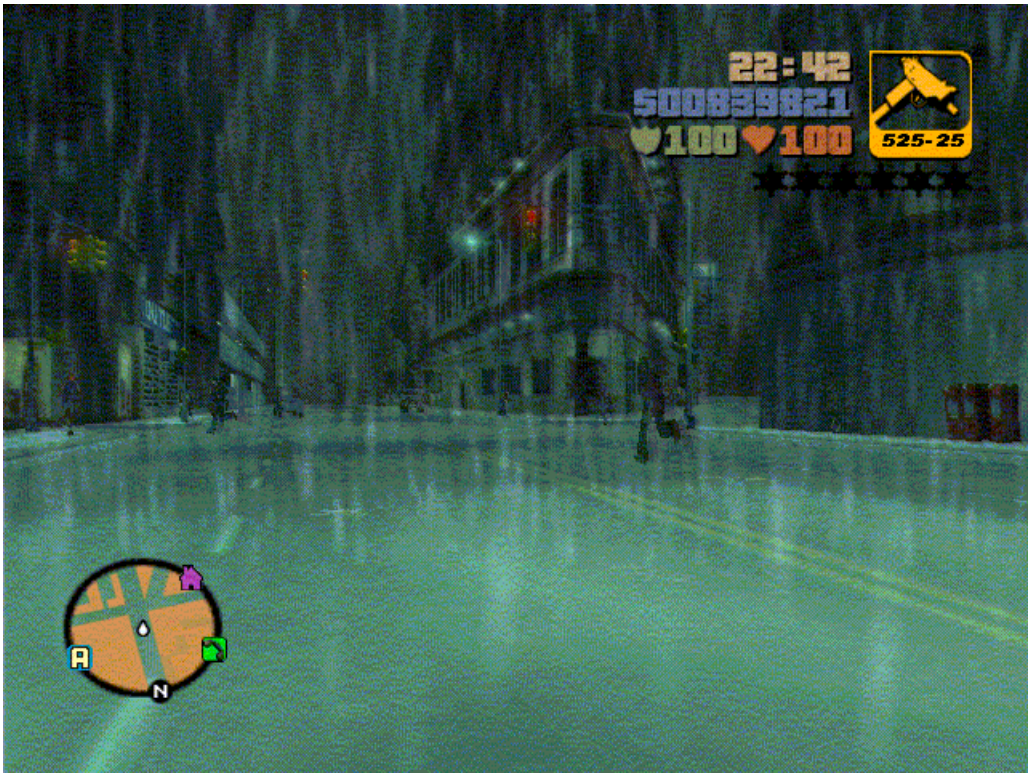


Figure 5.2. A screenshot of the rain effect used in the computer game 'Grand Theft Auto 3'

Eight friends spent approximately ten minutes using the computer game and observing the simulated rainstorm in action. Next I demonstrated the functionality offered by my developed application and they then each spent a similar amount of time using it for themselves. Finally I asked for each of their opinions regarding which aspects of the effect seemed realistic, which aspects were not, whether or not anything was missing from the implementation and then invited them to offer any further constructive comments.

General feedback from the peer evaluation:

Compliments:

- Almost all of the volunteers commented that the algorithm used in the update function seemed to produce a very believable storm. Some mentioned the fact that when the rainstorm was first activated, the effect of the particles falling in distinct bands towards the ground was impressive.
- Generally it was felt that the system could be modified in useful ways to produce different effects. The majority felt that the most useful modifying feature was the inclusion of the ability to specify the number of particles used within the system and the ability to control their rate of fall.
- Several felt that the inclusion of the two different ways to display the particles (as points or lines in the scene), was a useful feature, as both methods produced very different effects when viewed.

Criticisms:

- The principal criticism that was made concerned how the particles were rendered in the environment. Everyone felt that the raindrops could be displayed in a far more sophisticated manor, (like the implementation in the compared computer game). Potential improvements that were mentioned included: making the raindrops transparent and blending the particle colour with the colour of the background scene, to produce a more realistic effect. Including facilities to make different raindrops different sizes, so the effect did not seem so predictable, and also adding methods that could alter the severity and density of the storm, while the storm was actually running.
- Most felt that the methods that altered the size of the particles had little use. Many concluded that the effect did not look at all realistic, after the point size used to display the particles, had been increased even by just a small amount.
- Several felt that after the rain had been switched on and then turned off, the next time it was toggled back on, the particles should have been re-initialised back up to the top of the scene, instead of the implementation that was delivered, where the particles were updated continually after the rain was first switched on, regardless of whether or not they were being displayed, so when the user toggled the rain on once more, it appears to be falling instantaneously.

5.3.3. Evaluation Analysis

I agreed with most of the comments that were offered to me in the peer evaluation. I myself had realised most of the suggested limitations of the weather effect, but feel my implementation represents a quality attempt in the time that I had available.

The general feedback suggests that the underlying data types and functions used to create the particle system were of a high quality, and because of this, a believable storm system was indeed generated. Appropriate methods were included to change the system properties and it was felt that one could adequately alter all the main characteristics of the rainstorm that would be expected, (such as the rate at which the rain actually falls).

The main limitation of the system came down to how the particles were displayed in the environment, rather than how they were represented and operated on within the system. If this aesthetic factor were improved, the realism of the effect would be significantly enhanced and the delivered solution would definitely be superior. This enhancement and others are discussed in the following section 5.3.4.

5.3.4. Potential Future Enhancements

From the results of the peer evaluation, the following list of key potential enhancements that could be made to the developed application, to improve the overall believability of the effect, was constructed. Preliminary research was also conducted to determine how these improvements could be implemented.

- Making the raindrops transparent and blending their colour with that of the obscured background objects. OpenGL provides facilities that could be used to make this improvement possible. When specifying the colour of an object, three values that correspond to the desired RGB value were used, however a forth value can be enabled that is used to specify the desired alpha level. This corresponds to the opacity of the object, it takes a value between 0 and 1, where 0 means the object will be fully transparent and 1 means it will be fully opaque. This feature could be used in conjunction with the function 'glBlendFunc()', which can be used to perform the required colour blending, and this could ultimately produce a more believable rain effect. E. Angel's book offers an overview on these OpenGL features [1].
- Incorporating more random elements into the system to make the effect look less predictable. Although some stochastic elements were included in the system, (when specifying the position that the particles were initialised at), additional methods could be included to increase believability. One potential inclusion could be to randomly vary the displacements that different particles experience in the X and Z directions during their fall. In the delivered implementation this did not vary at all, only the y position of a particle ever changed for a fixed viewpoint. This inclusion would make different partic les follow slightly different trajectories and would likely increase realism significantly, as it would be a better model of the random nature of a rain.
- Other methods that could be implemented, to improve the appearance of the rendered system of particles, are explored in Reeves second particle system paper [19]. Amongst other things, it discusses approximate and probabilistic algorithms that can be used to efficiently shade the system, to give realistic effects. Some of these more advanced concepts could be implemented within the developed system.

6. Conclusion

The overall aim of the project was ‘to design and implement a realistic weather effect that can be used in the Leeds University Advanced Driving Simulator’. As described in the evaluation chapter, (chapter 5), the delivered application fulfils this objective and also offers additional functionality, as methods were included to enable the user the ability to modify the characteristics of the delivered effect. Although potential future enhancements that could improve the believability of the effect have been identified and discussed, the delivered implementation represents a quality solution when considering the timescale of the project.

The weather effect has not been incorporated into the driving simulator, as I did not have access to this research tool, however every step has been taken to ensure that this is possible and represents an area for potential future work. The application was produced using OpenGL and this is compatible with the OpenGL Performer technology that drives the simulator, therefore further research is required to establish how the code could be modified to work in this system.

References:

[1] Edward Angel, (2002), 'OpenGL – A Primer', published by Addison-Wesley.

[2] T. Akenine-Möller and E. Haines, (2002), 'Real-Time Rendering' 2nd edition, published by A.K. Peters.

[3] A. Kelley and I. Pohl, (1998), 'A Book on C – Programming in C' 4th edition, published by Addison-Wesley.

[4] Official website for the Leeds University Advanced Driving Simulator: (http://www.its.leeds.ac.uk/facilities/lads/) [Last accessed on the 21/04/03]

[5] Official website for the National Advanced Driving Simulator: (http://www.nads-sc.uiowa.edu/) [04/12/02]

[6] OpenGL fog tutorial: http://www.opengl.org/developers/code/sig99/shading99/course_slides/ShadowsTransparencyFog/sld051.htm [26/03/03]

[7] Information on VRML: (http://www.3dsite.com/n/sites/3dsite/cgi/VRML-index.html) [25/03/03]

[8] Official Java3D website: (http://java.sun.com/products/java-media/3D/) [25/03/03]

[9] Official OpenGL website: (http://www.opengl.org) [28/04/03]

[10] Official OpenGL Performer website: (http://www.sgi.com/software/performer/)

[11] Leeds University, School of Computing, SI23 webiste: (http://www.comp.leeds.ac.uk/kwb/si23/) [21/04/03]

[12] Leeds University, School of Computing, SI31 website: (http://www.comp.leeds.ac.uk/royr/si31/index.html) [21/04/03]

[13] David S. Ebert, 'Advanced Modelling Techniques for Computer Graphics'. ACM Computing Surveys, Vol. 28, No. 1, Pages 153-156, March 1996.

[14] William T. Reeves, 'Particle Systems – A Technique for Modelling a Class of Fuzzy Objects'. ACM Transactions on Graphics, Vol. 2, No. 2, Pages 91-108, April 1983.

[15] John van der Burg, 'Building An Advanced Particle System', June 2000. Paper published on Gamasutra.com, (http://www.gamasutra.com/features/20000623/vanderburg_pfv.htm) [03/02/03]

[16] An introduction to trigonometry - tutorial: (http://www.ping.be/~ping1339/gonio.htm) [16/04/03]

[17] Official 'Grand Theft Auto 3' website: (www.gta3.com) [28/04/03]

[18] 'Rockstar Games official website: (www.rockstargames.com) [28/04/03]

[19] William T. Reeves and R. Blau, 'Approximate and probabilistic algorithms for shading and rendering structured particle systems'. ACM SIGGRAPH Computer Graphics, Vol. 19, No. 3, Pages 313-322, July 1985.

[20] R. Ruddle, (2002), 'Advanced Computer Graphics (SI31/AGR) Lecture outline, OHPs and supporting papers', School of Computing, University of Leeds.

[21] A. Watt, (1993), '3D Computer Graphics' 2nd edition, published by Addison-Wesley.

Appendix A – Personal Reflection

Overall I found the experience of undertaking my project both rewarding and enjoyable. I have always had a keen interest in computer graphics and very much enjoyed having the opportunity to really apply the theory that I have acquired over the past three years at University. I feel I have truly developed both my skills and understanding as a direct result of the work and feel far more confident in my abilities.

This project represents the largest single piece of work I have ever attempted and although I initially felt daunted by this, it has really taught me a lot. In particular I have realised the fundamental importance of effective time management when attempting such a sizeable assignment and this is sure to be a valuable lesson that I will take with me for my future work. As the process involved juggling so many different tasks that all needed to be completed, without managing your time effectively, it is possible to get yourself into a situation where you end up so far behind you will struggle to catch up. I regrettably found myself in this position towards the end of the final semester, as I had somewhat neglected certain areas of my work prior to this, so I would advise anyone attempting a similar project, to start making plans and taking action early, to avoid any unnecessary stress later on.

I would also advise spending a substantial amount of time on choosing your project and really exploring straight away what will be involved. I did this and as a result choose a title that really interested me and I truly enjoyed. This made the task of motivating myself to complete the work far easier than it would have been if I had merely chosen any old project that really didn't appeal to me.

The greatest difficulty I encountered was implementing the method that would update the rain particles within my system. I initially spent at least a whole week struggling with this part of the code and didn't make any real progress, which meant that over this period, all other aspects of my work were also neglected. I would therefore advise anyone undertaking a similar assignment, to avoid wasting too much time on any one task that you are finding difficult, instead it is much more effective to move on and come back to it later on, when you have clearer head and are in a much better position to deal with the problem. It also makes sense to seek help from your supervisor as soon as you experience difficulties, to help you overcome any problem sooner rather than later.

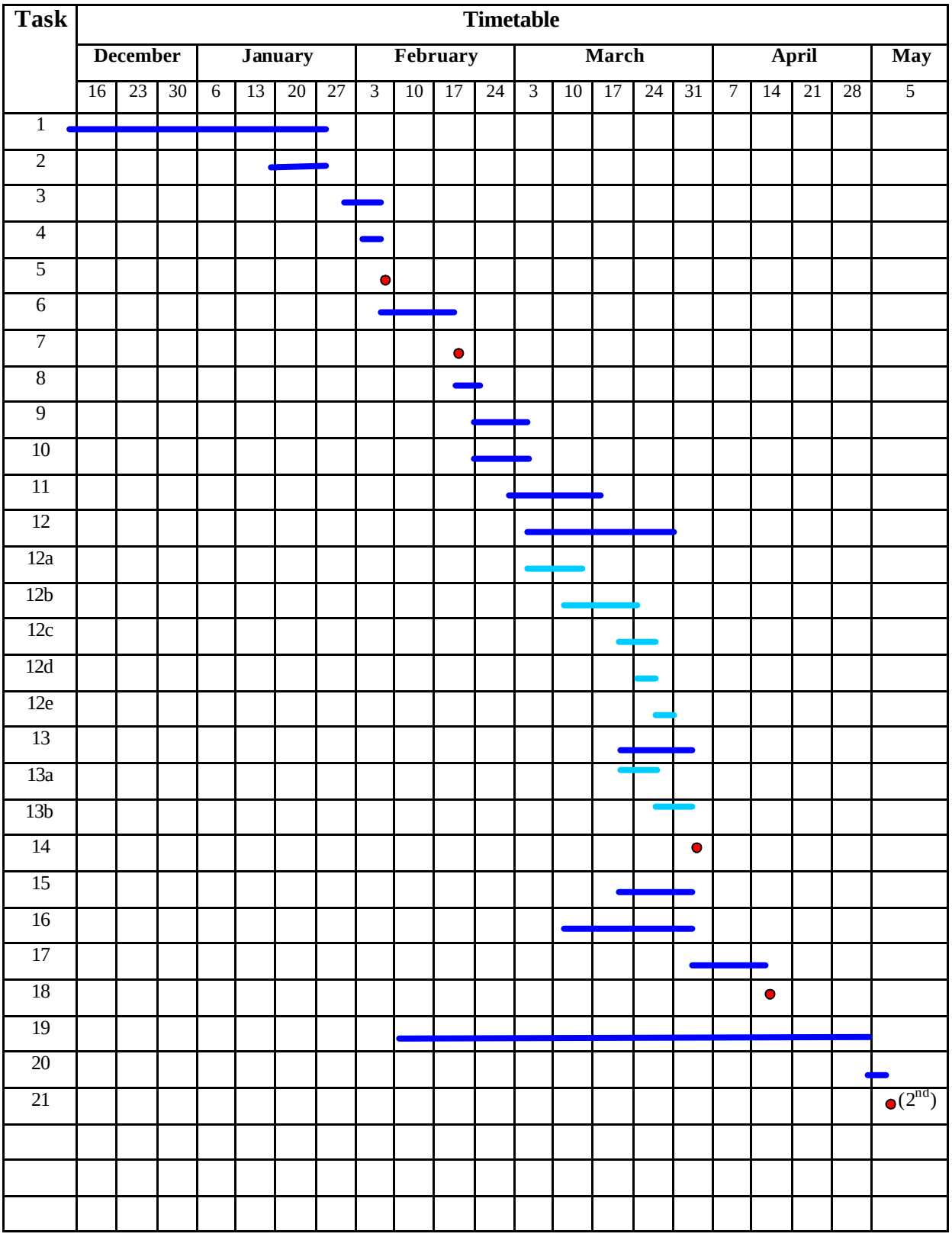
Due to these difficulties I experienced in the implementation stage, I found myself quite pushed for time towards the end of the project. I would therefore advise devising a project schedule with worse case estimates for the times that you feel all the different tasks will take, then when the inevitable happens and a difficulty is encountered, it should not mean that you will find yourself in a position with too little time to deal with the unexpected problem.

Appendix B – Project Schedule

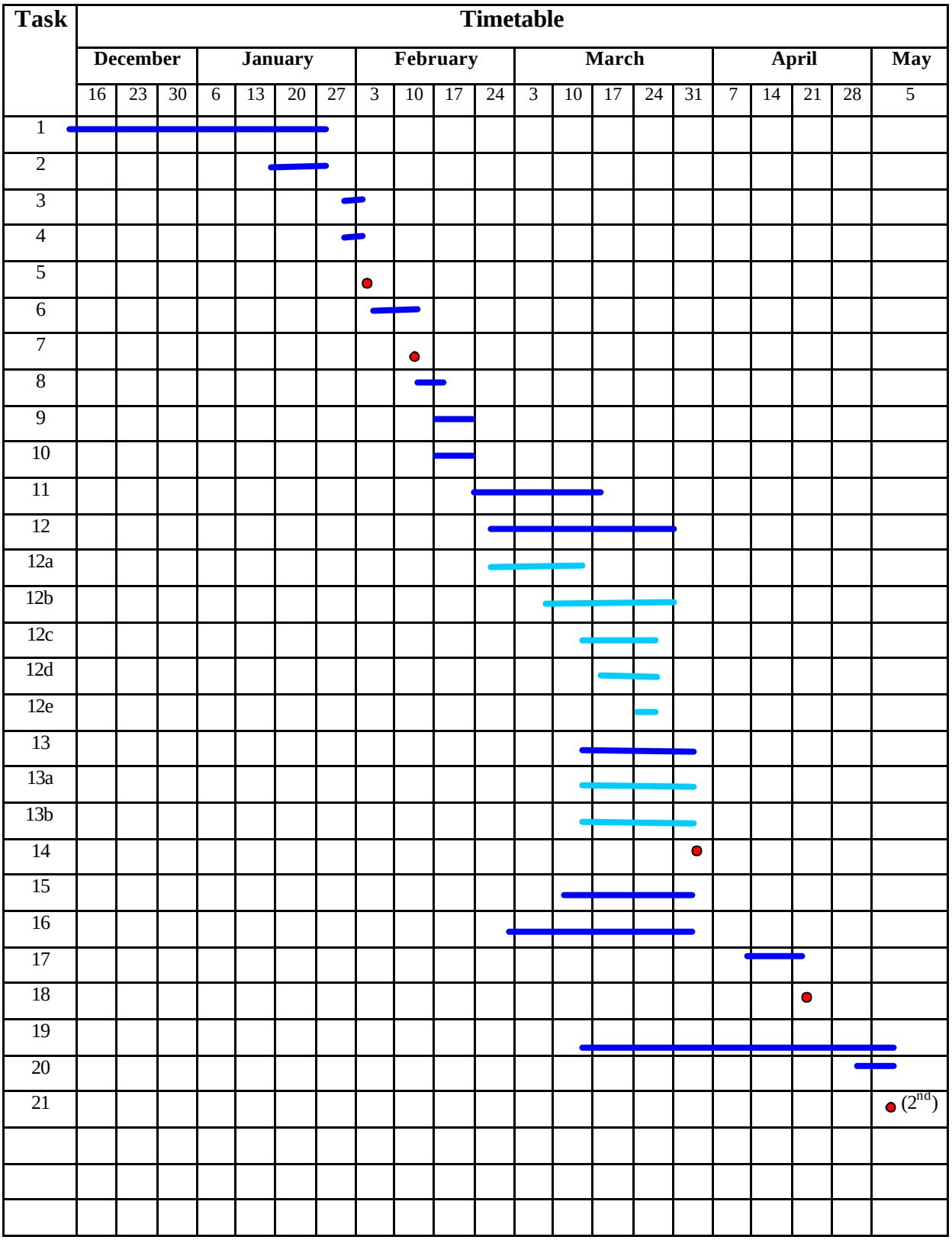
The following is a numbered list of all the tasks that needed to be completed over the course of my project. Note that the list does not include the background research that I was carrying out in the first semester.

1. Exam revision
2. Sit exams
3. Learn how to use the OpenGL API
4. Understand how to initialise OpenGL
5. *Competition of minimum objective 1*
6. Research how a rain effect can be implemented using this software
7. *Completion of minimum objective 2*
8. Design the graphical environment
9. Design the weather effect
10. Design how the user will interact with the system
11. Construct the graphical environment
12. Construct the basic weather effect
    12a. Initialising the effect
    12b. Updating the effect
    12c. Displaying the effect
    12d. Translating the effect
    12e. Removing the effect
13. Implement the user interaction
    13a. Implement the in application interaction
    13b. Construct the initialising menu
14. *Completion of minimum objective 3*
15. Implement the advanced features of the weather effect
16. Test the implementations
17. Evaluate the delivered solution
18. *Completion of minimum objective 4*
19. Write up report
20. Write personal reflection
21. Submit report (project complete)

The chart on the following page shows the original schedule that was developed to carry out all of these essential tasks. The page after contains a second chart that shows the actual scheduling that was performed, and following that is a supporting explanation of why the original schedule was deviated from.



The original proposed project schedule



The actual project schedule

Explanation:

The first tasks that required completion were actually achieved before schedule. When familiarising myself with OpenGL (tasks 3 and 4), I soon realised that less time would be needed for this, as it soon became clear that the computer graphics modules I had previously undertaken, (SI23 [11] and SI31 [12]), had already taught me most of the OpenGL features that I would need to use. The next task of researching how a rain effect could be implemented, (task 6), also progressed quicker than planned. I had originally expected to find many potential techniques that I could use for this task and therefore thought that a lot of time would be required to determine which one should be selected. However it soon became obvious, upon discovering the particle system modelling technique, that this method was somewhat of an industry standard for generating such effects and was clearly the sensible choice.

It was fortunate that these early stages progressed well, because I had begun to experience significant development problems in the implementation stage, (task 11 onwards), and would need the extra time to work on these elements of the project. I soon realised that my original estimates concerning how long these tasks would take, were far too optimistic, and because of this, I found myself being quite pushed for time. I had to spend much more of my time juggling the various implementation tasks, to get enough of my program working to demonstrate in my progress meeting on 20/03/03, whereas my original plan was to be able to concentrate on these different aspects one at a time. Because of these difficulties, I had to use some of the time that I had originally scheduled for writing up, to overcome these problems and finish the implementation. This subsequently meant that I had less time to complete the evaluation and to produce the actual report, however fortunately this remaining time still proved adequate.

Appendix C – Program Code

Index

```
//////////////////////////////////////////////////////
// fyp.c
// Produced by Stephen Tucker (ctxsrt)
// Code to create a rain effect for my final year project
//////////////////////////////////////////////////////


////////////////// Required include statements //////////////////

#include <stdlib.h>    // for the standard library.
#include <math.h>      // for the maths library (needed for sin, cos e.t.c).
#include <stdio.h>     // Allows standard input / output operators.
#include <GL/glut.h>// includes the GL/glut libraries.


////////////////// Initializing Variables //////////////////

int windowWidth = 300;          // The default width and height of the viewing window.
int windowHeight = 300;

GLfloat cameraX = 0;            // The x, y and z position of the current viewpoint (camera).
GLfloat cameraY = -80;
GLfloat cameraZ = 0;

GLfloat focusPointX = 0;        // The x, y and z position of the current focus of view (target).
GLfloat focusPointY = -80;
GLfloat focusPointZ = -2000;

GLfloat angle = 0;              // The current viewing angle (relative to the z axis). I.e. when the scene is initialised the viewing direction
                                // is at an angle 0 radians, meaning the viewer is looking directly down the z axis.

int movementMagnitude = 3;                      // The distance that the viewpoint will move when either the 'q' or 'a' key is pressed.

GLfloat start[] = {0, -25, 0};                  // The global start and end points for the particles, i.e the difference between the two
                                                // y values is the distance the particles will fall through.
GLfloat end[] = {0, -100, 0};

GLfloat particleColour[] = {0.8, 0.8, 0.8};     // The default colour of the particles (dark grey).

GLfloat particleSize = 1;                       // The default particle size.

int numOfParticles = 18000;                     // The default number of particles.

int numOfFrames = 40;                           // The default number of frames used to display a particles fall.

GLfloat fogColour[] = {0.5, 0.5, 0.5};          // The default fog colour (grey).

int choice;                                     // A variable to store keyboard inputs.


///// Boolean variables used for simulation control
GLboolean initComplete = GL_FALSE;              // Has the system initialisation been completed? Initially not.
GLboolean rain = GL_FALSE;                      // Is the rain enabled? Initially not.
GLboolean lines = GL_FALSE;                     // Are the particles displayed as lines? Initially not, they are displayed as points.
GLboolean fog = GL_FALSE;                       // Is the fog enabled? Initially not.
GLboolean waitingForRain = GL_TRUE;             // Are we waiting for the rain to be started for the first time? Initially yes.

///// The names of the display lists that will be constructed
GLuint Sign;
GLuint House;
GLuint JunctionMarkings;

///// Declare enumeration types
enum {X, Y, Z};         // This enables the 1st element of an array to be called as X, the 2nd as Y and the 3rd as Z. It is necessary
                        // so the different elements that make up a particles position (i.e. X, Y, Z), which have been stored in a
                        // single array for efficiency, can be easily accessed. This is used in the InitParticles function (see page 5).

enum {R, G, B};         // As above - but used to refer to red, green and blue values that are stored in an array.
```

```
////////////////////// Define the particle data types ////////////////////

////// Define a particle
typedef struct
{
        GLfloat posA[3];    // The current position of the particle in the 3D scene.
        GLfloat posB[3];    // A secondary position for the particle that will be slightly below posA (used when rendering lines instead
                            // of points, i.e a line segment will be drawn from one point to the other and this will represent the particle)

        GLfloat Vx, Vy, Vz; // The particles current velocity in each direction.
        GLfloat colour[3];  // The current RGB value of the particle.
        GLfloat size;       // The particles current size.

} Particle;

////// Define a particle system
typedef struct
{
        Particle *array;                // A pointer called array that points to a Particle type.
        int total;                      // The total number of array elements.
        int update;                     // The number of particles that will be updated per frame.
        GLfloat Rx, Ry, Rz;             // The reset vector, stores the total distance the particles fall in each direction.
        GLfloat Vx, Vy, Vz;             // The global velocity of particles.
        GLfloat red, green, blue;       // The default particle colour.
        GLfloat defaultSize;            // The default particle size.

} ParticleSystem;


ParticleSystem psys;                    // Create a variable called psys of type particleSystem.

void (*updateptr) (ParticleSystem * sys);   // the pointer updateptr points to a function.



////////////////////// Function Prototypes ////////////////////

void Initialization();                                  // Initialises the system.

void InitParticles(ParticleSystem * sys, GLfloat * start, GLfloat * end, GLfloat * particleColour, GLfloat particleSize, int
numberOfParticles, int numberOfFrames);
                                                        // Initialises the particles in the specified particle system.

void RemoveParticles(ParticleSystem * sys);             // Removes the particles from the specified particle system.
void ParticleUpdate(ParticleSystem * sys);              // Updates the particles in the specified particle system.
void DisplayParticles(ParticleSystem * sys);            // Displays the particles in the specified particle system.
void MoveRain(ParticleSystem * sys, GLfloat x, GLflo at z);     // Moves the specified particle system to be centred above the viewpoint.
void DrawScene();                                       // Draws the scene.
void Redraw();                                          // Manages the redrawing of the scene.
void Key(unsigned char key, int x, int y);              // Manages keyboard inputs.
void Mouse(int x, int y);                               // Manages mouse inputs.
void Idle();                                            // Redisplays scene when nothing else is being done.
void Update();                                          // Updates viewing.
void Reshape(int wid, int ht);                          // Reshapes the window.

///// Prototypes of the functions that define the scene objects:
void DefineSign();
void DefineHouse();
void DefineJunctionMarkings();

///// Prototype of the functions that draw the scene objects:
void DrawSign1();
void DrawSign2();
void DrawSign3();
void DrawSign4();
void DrawHouse1();
void DrawHouse2();
void DrawHouse3();
void DrawHouse4();
void DrawHouse5();
void DrawHouse6();
void DrawHouse7();
void DrawHouse8();
void DrawJunctionMarkings1();
void DrawJunctionMarkings2();
```

```
////////////////// Main function //////////////////

int main()
{
	// Calls the initialising function that will prompt the user to specify the system settings (see page 4).
	Initialization();

	// Initialise the application window size.
	glutInitWindowSize(windowWidth, windowHeight);

	// Initialise the display mode (double buffering is used so the animation appears smooth).
	glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL | GLUT_DOUBLE);

	// Create the application window (give it the title that has been supplied).
	glutCreateWindow("Rain effect - generated using a particle system");

	// Display the application window on a full screen.
	glutFullScreen();

	///// Specify the callback functions to be used that will process any events that may occur:
	glutDisplayFunc(Redraw);
	glutReshapeFunc(Reshape);
	glutPassiveMotionFunc(Mouse);
	glutKeyboardFunc(Key);
	glutIdleFunc(Idle);

	///// Appropriately set the viewing frustum for the application:
	glMatrixMode(GL_PROJECTION);
	glFrustum(-5, 5, -5, 5, 10, 1000);
	glMatrixMode(GL_MODELVIEW);

	// Enable the depth test facility to ensure that objects in the foreground are always drawn in front of objects in the background.
	glEnable(GL_DEPTH_TEST);

	// State the clear colour for the scene - sky blue so that it gives the appearance of sky.
	glClearColor(0.3, 1.0, 1.0, 1.0);

	///// Initialize the fog:
	glFogi(GL_FOG_MODE, GL_LINEAR);   // Specify the type of fog calculation to be used - i.e linear.
	glFogfv(GL_FOG_COLOR, fogColour); // Specify the fog colour.
	glFogf(GL_FOG_DENSITY, 0.30);     // Specify the density of the fog.
	glFogf(GL_FOG_START, 0);          // Specify that the fog start plane will be the x axis, 200 behind the viewpoint.
	glFogf(GL_FOG_END, 500);          // Specify that the fog end plane will be the x axis, 300 in front of the viewpoint.
	glHint(GL_FOG_HINT, GL_NICEST);   // Indicates that the fog calculation should be made to a high accuracy.

	InitParticles(&psys, start, end, particleColour, particleSize, numOfParticles, numOfFrames);   // Initialise particles
	updateptr = ParticleUpdate;                                                                     // Run particle update function


	glutMainLoop();   // make the program enter an event-processing loop, so that the system will always continuously monitor
	                  // for events and react to them accordingly.
	return 0;
}
```

```
////////////////// Function definitions //////////////////


///// Initialization function prompts the user for simulation settings and stores the values that are entered in the appropriate variable.

void Initialization()
{
	printf("\n\n********** SIMULATION CONTROLS **********\n\n");
	printf("The following keys can be used in the simulation:\n\n");
	printf("The 'r' key toggles the rain on and off\n");
	printf("The 'l' key toggles between the particles being drawn as points or as lines\n");
	printf("The 'f' key toggles fog on and off\n");
	printf("The '+' key makes the particles larger (when displayed as points)\n");
	printf("The '-' key makes the particles smaller (when displayed as points)\n");
	printf("The mouse is used to specify your direction and the 'q' and 'a' keys move you forward and back respectively\n\n");
	printf("\n********** INITIALIZATION **********\n\n");
	printf("Would you like to customize the system? If not enter '1' for the default settin gs or '2' for the custom menu : ");
	scanf("%d", &choice);

	while (!initComplete)
	{
		if (choice != 1 && choice != 2)		// A check to ensure that a valid input is entered.
		{
			printf("Invalid choice! Please choose again : ");
			scanf("%d", &choice);
		}

		if (choice == 1)
		{
			printf("\nStarting program...\n\n");
			initComplete = !initComplete;
		}

		if (choice == 2)
		{
			printf("\nParticle setup - Note A) there must be more particles than frames and B) that particles / frames
must leave no remainder\n\n");
			printf("Enter the number of particles to be used : ");
			scanf("%d", &numOfParticles);
			printf("Enter the number of frames to be used : ");
			scanf("%d", &numOfFrames);
			printf("Enter the desired RGB value of the particle colour (each number seperated by a space) : ");
			scanf("%f" "%f" "%f", &particleColour[0], &particleColour[1], &particleColour[2]);
			printf("Enter the desired RGB value of the scene fog (each number seperated by a space) : ");
			scanf("%f" "%f" "%f", &fogColour[0], &fogColour[1], &fogColour[2]);
			printf("\nStarting Program...\n\n");
			initComplete = 1;
		}
	}
}
```

```
///// This function initialises the particles in the particle system.

void InitParticles(ParticleSystem * sys, GLfloat * start, GLfloat * end, GLfloat * particleColour, GLfloat particleSize, int
numberOfParticles, int numberOfFrames)
{
        int i;
        Particle *part;

        sys->Rx = start[X] - end[X];      // Store the distance the particles will travel in the x direction into the reset vector.
        sys->Ry = start[Y] - end[Y];      // Store the distance the particles will travel in the y direction into the reset vector.
        sys->Rz = start[Z] - end[Z];      // Store the distance the particles will travel in the z direction into the reset vector.

        sys->Vx = -sys->Rx / numberOfFrames;    // Calculate and store the particles velocities in all three directions, i.e – Calculated
        sys->Vy = -sys->Ry / numberOfFrames;    // by taking the distance a particle has to fall in each direction (calculated above) and
        sys->Vz = -sys->Rz / numberOfFrames;    // dividing this by the number of frames that are used to display this motion. This
                                                // ensures that the particles will cover the total fall distance in the right number of
                                                // frames.

        sys->red = particleColour[R];           // Store the default particle colour - as supplied to this function.
        sys->green = particleColour[G];
        sys->blue = particleColour[B];

        sys->defaultSize = particleSize;        // Store the default particle size - as supplied to this function.

        sys->update = numberOfParticles / numberOfFrames;   // The total number of particles divided by the number of frames used to
                                                            // display a particles life will give the number of updates that are needed
                                                            // per frame. This should have no remainder and will be used in the
                                                            // particle update calculations.

        sys->total = numberOfParticles;                     // Store the total number of Particles.

        sys->array = (Particle *) malloc(numberOfParticles * sizeof(Particle));    // Dynamically allocate some space to store all of the
                                                                                   // particles in the system, (the number of particles is
                                                                                   // supplied to the function). Malloc is the command that
                                                                                   // requests this memory from the heap.

        if (sys->array == 0)            // If there is not enough memory to store the particle system, inform the user and exit the program.
    {
        printf("ERROR: Out of memory\n");
        exit(0);
    }

        for (part = sys->array, i = 0; i < numberOfParticles; i++)      // Iterate through this array of particles and initialise all the
                                                                        // attributes of each individual particle.
        {

                part->Vx = 0;                   // Set each individual particles velocity to zero, because they will not move until the
                part->Vy = 0;                   // 'FirstParticleUpdate()' function is called.
                part->Vz = 0;

                part->posA[X] = part->posB[X] = (-50 + drand48() * 100) + cameraX;      // This initialises all the particles to be
                                                                                        // positioned at random points along the x
                                                                                        // axis that are up to a distance of 50 either
                                                                                        // side of the viewpoints x coordinate.

                part->posA[Z] = part->posB[Z] = (-50 + drand48() * 100) + cameraZ;      // This initialises all the particles to be
                                                                                        // positioned at random points along the z
                                                                                        // axis that are up to a distance of 50 either
                                                                                        // side of the viewpoints z coordinate.

                part->posA[Y] = part->posB[Y] = start[Y] - sys->Vy + 2 * sys->Vy * drand48();   // Initialise all the particles to be
                                                                                                // positioned at random points
                                                                                                // along the y axis in the range
                                                                                                // from the starting height to the
                                                                                                // fall distance per frame below
                                                                                                // this.

                part->posB[Y] -= drand48() * 1.5 * sys->Vy;     // Initialise the secondary particle position to be below the primary.
                                                                // This secondary position is used when the particles are displayed
                                                                // as lines instead of points (i.e. a line is drawn between posA & B)

                part++;                                         // Increment to the next particle.
        }
}
```

```
///// This function frees the dynamically assigned memory that holds all the particle information, it is called when the program is exited.

void RemoveParticles(ParticleSystem * sys)
{
	free(sys->array);
}


///// This function updates the particles in the system, appropriately moving them through the scene.

void ParticleUpdate(ParticleSystem * sys)
{
	int i = 0;

	if (waitingForRain)		// If we are still waiting for the first lot of rain, i.e the rain has just been started, then do the following:
	{
		static int offset = 0;
		Particle *part;

		part = &sys->array[offset];

		for (i = 0; i < sys->update; i++)	// All the particles will initially be at the same starting height. This loop takes a
		{					// subset of them and gives this subset a downward velocity, therefore when the
							// particles are displayed, only the ones that have a velocity will appear to move
							// downwards.
			part->Vy = sys->Vy;
			part++;
		}

		offset += sys->update;

		if (offset >= sys->total)		// When all of the particles have been given a downward velocity they will all be
							// moving, therefore set the waitingForRain value to false so the next stage of the
							// ParticleUpdate function will be entered.
		{
			offset = 0;
			waitingForRain = !waitingForRain;
		}
	}
	else			// If we are no longer waiting for the first lot of rain, i.e all the particles are moving, then do the following:
	{
		GLfloat displacex = sys->Rx;
		GLfloat displacey = sys->Ry;
		GLfloat displacez = sys->Rz;
		static int offset = 0;

		int count = sys->update;
		Particle *part;

		part = &sys->array[offset];
		for (i = 0; i < count; i++)		// This loop takes the particles at the bottom of their fall and adds to their position
							// the total fall distance, therefore positioning them back at the starting height ready
						       // to fall through the scene again.
		{
			part->posA[X] += displacex;
			part->posB[X] += displacex;
			part->posA[Y] += displacey;
			part->posB[Y] += displacey;
			part->posA[Z] += displacez;
			part->posB[Z] += displacez;
			part++;

		}
		offset = (offset + count) % sys->total;
	}
}
```

```
///// This function is used to display the particles in the scene.

void DisplayParticles(ParticleSystem * sys)
{
	int i = 0;
		int total;
		Particle *part;

		total = sys->total;

		glDisable(GL_LIGHTING);
		glColor3f(sys->red, sys->green , sys->blue);   // Take the RGB value that the particle system is specified to be displayed in

		if (lines)                                     // If the Boolean value 'lines' is true, i.e. the user wants to display the particles as
		{                                              // lines, enable this functionality.
			glBegin(GL_LINES);
		}
		else                                           // If the Boolean value 'lines' is false, i.e. the user wants to display the particles as
		{                                              // points, enable this functionality.
			glBegin(GL_POINTS);
		}

		part = &sys->array[i];
		for (i = 0; i < total; i++)                    // Iterate through all particles within the system
		{
			glVertex3fv(part->posA);

			if (lines)
			{
				glVertex3fv(part->posB);
			}

			part->posA[X] += part->Vx;
			part->posA[Y] += part->Vy;
			part->posA[Z] += part->Vz;
			part->posB[X] += part->Vx;
			part->posB[Y] += part->Vy;
			part->posB[Z] += part->Vz;

			part++;

		}
		glEnd();
}


///// This function is used to move the particle system around the scene, ensuring it is always positioned above the camera point.
///// The supplied values x and z are the displacements that the camera has experienced in the x and z directions, so the system is translated
///// these amounts appropriately.

void MoveRain(ParticleSystem * sys, GLfloat x, GLfloat z)
{
		int i = 0;
		Particle *part;

		part = &sys->array[i];
		for (i = 0; i < sys->total; i++)
		{
			part->posA[X] = part->posA[X] + x;
			part->posB[X] = part->posB[X] + x;
			part->posA[Z] = part->posA[Z] + z;
			part->posB[Z] = part->posB[Z] + z;
			part++;
		}
		return;
}
```

```
///// This function draws the scene

void DrawScene()
{
	glDisable(GL_LIGHTING);
	glLineWidth(1);

	// Draw the ground on front left.
	glBegin(GL_QUADS);
	{
		glColor3f(0, 1, 0);

		glVertex3f(-500, -100, 0);
		glVertex3f(-30, -100, 0);
		glVertex3f(-30, -100, -590);
		glVertex3f(-500, -100, -590);
	}
	glEnd();

	// Draw the ground on front right.
	glBegin(GL_QUADS);
	{
		glVertex3f(30, -100, 0);
		glVertex3f(500, -100, 0);
		glVertex3f(500, -100, -590);
		glVertex3f(30, -100, -590);
	}
	glEnd();

	// Draw the ground at the back left.
	glBegin(GL_QUADS);
	{
		glVertex3f(-500, -100, -650);
		glVertex3f(300, -100, -650);
		glVertex3f(300, -100, -1000);
		glVertex3f(-500, -100, -1000);
	}
	glEnd();

	// Draw the ground at the back right.
	glBegin(GL_QUADS);
	{
		glVertex3f(360, -100, -650);
		glVertex3f(500, -100, -650);
		glVertex3f(500, -100, -1000);
		glVertex3f(360, -100, -1000);
	}
	glEnd();

	// Draw the 1st verticle road.
	glBegin(GL_QUADS);
	{
		glColor3f(0.7, 0.7, 0.7);

		glVertex3f(-30, -100, 0);
		glVertex3f(30, -100, 0);
		glVertex3f(30, -100, -590);
		glVertex3f(-30, -100, -590);
	}
	glEnd();

	// Draw the horizontal road.
	glBegin(GL_QUADS);
	{
		glVertex3f(-500, -100, -590);
		glVertex3f(500, -100, -590);
		glVertex3f(500, -100, -650);
		glVertex3f(-500, -100, -650);
	}
	glEnd();
```

```
	// Draw the 2nd verticle road.
	glBegin(GL_QUADS);
	{
		glVertex3f(300, -100, -650);
		glVertex3f(360, -100, -650);
		glVertex3f(360, -100, -1000);
		glVertex3f(300, -100, -1000);
	}
	glEnd();

	// Draw markings on 1st verticle road.
	glBegin(GL_LINES);
	{
		glColor3f(1.0, 1.0, 1.0);
		glVertex3f(0, -99.5, 0);
		glVertex3f(0, -99.5, -590);
	}
	glEnd();

	// Draw markings on 2nd verticle road.
	glBegin(GL_LINES);
	{
		glVertex3f(330, -99.5, -650);
		glVertex3f(330, -99.5, -1000);
	}
	glEnd();

	// Draw markings on horizontal road.
	glBegin(GL_LINES);
	{
		glVertex3f(-500, -99.5, -620);
		glVertex3f(500, -99.5, -620);
	}
	glEnd();

	DefineSign();
	DefineHouse();
	DefineJunctionMarkings();

	DrawSign1();
	DrawSign2();
	DrawSign3();
	DrawSign4();
	DrawHouse1();
	DrawHouse2();
	DrawHouse3();
	DrawHouse4();
	DrawHouse5();
	DrawHouse6();
	DrawHouse7();
	DrawHouse8();
	DrawJunctionMarkings1();
	DrawJunctionMarkings2();


	glEnable(GL_LIGHTING);
	// If rain is active draw the particles.
	if (rain)
	{
		DisplayParticles(&psys);
		(*updateptr) (&psys);
	}
	glutPostRedisplay();
}
```

```
///// This function manages the key presses that the user can make.

void Key(unsigned char key, int x, int y)
{
	GLfloat displaceX = (movementMagnitude * sin(angle));
	GLfloat displaceZ = (movementMagnitude * cos(angle));

	switch (key)
	{
		case '\033':			// When the 'escape' key is pressed the program will exit.
		RemoveParticles(&psys);		// All particles will be removed from the system before the program exits.
		printf("Particles removed - exiting program\n\n");
		exit(0);
		break;

		case 'r':			// When 'r' is pressed the rain will be toggled on and off.
		case 'R':
		rain = !rain;
		glutPostRedisplay();
		break;

		case '+':			// Press '+' to make the particles larger.
		particleSize += 0.5;
		glPointSize(particleSize);
		glutPostRedisplay();
		break;

		case '-':			// Press '-' to make the particles smaller.
		particleSize -= 0.5;
		if (particleSize <= 0)
		{
			particleSize = 0.5;
		}
		glPointSize(particleSize);
		glutPostRedisplay();
		break;

		case 'f':			// When 'f' is pressed the fog will be toggled on and off.
		case 'F':
		fog = !fog;
		if (fog)
		{
			glClearColor(0.5, 0.5, 0.5, 1.0);
			glEnable(GL_FOG);
		}
		else
		{
			glClearColor(0.3, 1.0, 1.0, 1.0);
			glDisable(GL_FOG);
		}
		break;

		case 'q':			// When 'q' is pressed the viewpoint will advance forward.
		case 'Q':
		cameraX = cameraX + displaceX;
		focusPointX = focusPointX + displaceX;

		if (cameraZ < focusPointZ)
		{
			cameraZ = cameraZ - displaceZ;
			focusPointZ = focusPointZ - displaceZ;
		}
		if (cameraZ > focusPointZ)
		{
			cameraZ = cameraZ - displaceZ;
			focusPointZ = focusPointZ - displaceZ;
		}
		MoveRain(&psys, displaceX, -displaceZ);
		break;
```

```
		case 'a':				// When 'a' is pressed the viewpoint will move backwards.
		case 'A':
		cameraX = cameraX - displaceX;
		focusPointX = focusPointX - displaceX;

		if (cameraZ < focusPointZ)
		{
			cameraZ = cameraZ + displaceZ;
			focusPointZ = focusPointZ + displaceZ;
		}
		if (cameraZ > focusPointZ)
		{
			cameraZ = cameraZ + displaceZ;
			focusPointZ = focusPointZ + displaceZ;
		}
		MoveRain(&psys, -displaceX, displaceZ);
		break;

		case 'l':				// When 'l' is pressed this will toggle if the particles are displayed as points or lines.
		case 'L':
		lines = !lines;
		glutPostRedisplay();
		break;

		default:
		break;
	}
}
```

```
///// This function is called when the scene needs to be redrawn.

void Redraw()
{
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
	DrawScene();
	glutSwapBuffers();
}


///// This function manages the mouse inputs that the user can make.

void Mouse(int x, int y)
{
	if(x > 0 && x < (windowWidth * 1/4))
	{
		angle = angle - (M_PI / 360);
		focusPointX = (2000 * sin(angle));
		focusPointZ = -(2000 * cos(angle));
	}

	if(x > (windowWidth * 3/4) && x < windowWidth)
	{
		angle = angle + (M_PI / 360);
		focusPointX = (2000 * sin(angle));
		focusPointZ = -(2000 * cos(angle));
	}
	glutPostRedisplay();
}


///// This function redisplays the scene. It is called when there are no other events to process.

void Idle()
{
	glutPostRedisplay();
	Update();
}


///// This function updates how the scene is viewed.

void Update()
{
	glLoadIdentity();
	gluLookAt(cameraX, cameraY, cameraZ,
		  focusPointX, focusPointY, focusPointZ,
	      0, 1, 0);
}


///// This function is used to reshape the scene when the window size is changed. It is needed to keep the proportions as I have stated them,
///// when the window is drawn on a full screen.

void Reshape(int wid, int ht)
{
	windowWidth = wid;
	windowHeight = ht;
	glViewport(0, 0, wid, ht);
}
```

////////// Definitions of the functions that define the objects in the scene //////////

///// Function that defines the stop sign object.

```
void DefineSign()
{
	Sign = glGenLists(1);
	glNewList(Sign, GL_COMPILE);		// Create a new display list, called 'Sign', that will be used to store the geometric
	{					// primitives of the sign object, so this list can be called when required.
		GLUquadricObj* Post;
		Post = gluNewQuadric();

		glBegin(GL_POLYGON);		// Post
		{
			glColor3f(0.8, 0.8, 0.8);
			gluQuadricDrawStyle(Post, GLU_FILL);
			gluQuadricNormals(Post, GLU_SMOOTH);
			gluCylinder(Post, 2, 2, 30, 30, 30);
		}
		glEnd();
		glBegin(GL_POLYGON);		// Front Sign
		{
			glColor3f(1.0, 0.0, 0.0);
			glVertex3f(4, 2, 0);
			glVertex3f(10, 2, -6);
			glVertex3f(10, 2, -12);
			glVertex3f(4, 2, -18);
			glVertex3f(-4, 2, -18);
			glVertex3f(-10, 2, -12);
			glVertex3f(-10, 2, -6);
			glVertex3f(-4, 2, 0);
		}
		glEnd();
		glBegin(GL_POLYGON);		// Back Sign
		{
			glVertex3f(4, -2, 0);
			glVertex3f(10, -2, -6);
			glVertex3f(10, -2, -12);
			glVertex3f(4, -2, -18);
			glVertex3f(-4, -2, -18);
			glVertex3f(-10, -2, -12);
			glVertex3f(-10, -2, -6);
			glVertex3f(-4, -2, 0);
		}
		glEnd();
		glBegin(GL_LINE_STRIP);		// Sign border
		{
			glLineWidth(2);
			glColor3f(1.0, 1.0, 1.0);
			glVertex3f(4, 2.1, 0);
			glVertex3f(10, 2.1, -6);
			glVertex3f(10, 2.1, -12);
			glVertex3f(4, 2.1, -18);
			glVertex3f(-4, 2.1, -18);
			glVertex3f(-10, 2.1, -12);
			glVertex3f(-10, 2.1, -6);
			glVertex3f(-4, 2.1, 0);
			glVertex3f(4, 2.1, 0);
			glLineWidth(1);
		}
		glEnd();

		// NOTE: At this point I have removed the code that defines other polygons and lines that make up the sign from this
		// appendix. This has been done because the code spanned many pages and would not have been useful. The included
		code is sufficient to understand how these objects were defined.
	}
		glEndList();		// End the definition of this display list. Therefore all polygons and lines contained will make
					// up the defined sign object.
}
```

// NOTE: At this point I have removed the functions that Define the Houses and junction markings from this appendix (DefineHouse() and DefineJunctionMarkings()). This has been done because the methods used to construct these objects were very similar to the methods used in the previous DefineSign() function, and therefore the inclusion of this code would not be overly useful.

```
////////// Definitions of the functions that draw the objects in the scene //////////


///// Draw the first stop sign.

void DrawSign1()
{
	glPushMatrix();
	glTranslatef(-35, -70, -580);	// Translate the position of the sign by –35 in the x, -70 in the y and –580 in the z direction.
	glRotatef(90, 1, 0, 0);		// Rotate the sign by 90 degrees relative to the x axis.
	glCallList(Sign);		// Call the 'Sign' display list to be used.
	glPopMatrix();

}
// NOTE: At this point I have removed the other 3 functions that draw the other signs from this appendix (DrawSign2() – DrawSign4()), as the code is very similar to that of the above function.


///// Draw the first house.

void DrawHouse1()
{
	glPushMatrix();
	glTranslatef(-450, -100, -755);
	glScalef(1.5, 1.5, 1.5);		// Scale the House by a factor of 1.5 in each direction.
	glCallList(House);
	glPopMatrix();
}
// NOTE: At this point I have removed the other 7 functions that draw the other houses from this appendix (DrawHouse2() – DrawHouse8()), as the code is very similar to that of the above function.


///// Draw the first set of junction markings.

void DrawJunctionMarkings1()
{
	glPushMatrix();
	glTranslatef(-30, 0, -590);
	glCallList(JunctionMarkings);
	glPopMatrix();
}


///// Draw the second set of junction markings.

void DrawJunctionMarkings2()
{
	glPushMatrix();
	glTranslatef(360, 0, -650);
	glRotatef(180, 0, 1, 0);
	glCallList(JunctionMarkings);
	glPopMatrix();
}
```