

Coventry University
Faculty of Engineering and Computing

Sensor Network Visualisation

by

John Robert Kemp

Student ID: 1311792

393CS Computing Project
April 2006

Supervisor: **Sarah Mount**

Declaration of Originality

This project is all my own work and has not been copied in part or in whole from any other source except where duly acknowledged. As such, all use of previously published work (from books, journals, magazines, internet, etc) has been acknowledged within the main report to an item in the References or Bibliography lists.

I also agree that an electronic copy of this project may be stored and used for the purposes of plagiarism prevention and detection.

Copyright Acknowledgement

I acknowledge that the copyright of this project and report belongs to Coventry University.

Signed:

Date:



Office Stamp

Abstract

Wireless sensors networks are a relatively new area for experimentation, utilising small devices with a wide range of onboard sensors to gather data about their environment. These devices (also called motes) are limited by reduced processing and battery power compared to larger devices, but are capable of working together to gather and analyse data over a large area and report this information to interested parties.

This project looks at the development of an application geared towards the visualisation of sensor networks in a generic and expandable way that would allow the data gathered by such a network to be more easily interpreted by the human component of the system.

An example implementation is presented demonstrating some of the expansion capabilities of the system. This example reads sensor data in from external files to provide the data required for the visualisation and is geared toward mote lateration experiments, though is shown performing terrain mapping to demonstrate that the application's adaptability isn't entirely dependant on additional code being written.

Contents

Abstract	3
Contents	4
Additional Materials on the Accompanying CD	5
Acknowledgements	6
Introduction.....	7
Overview.....	7
Project Focus.....	7
Project Client	8
Project Aims.....	8
Report Overview	8
Investigation.....	9
Previous Work	9
Networks	9
Analysis.....	11
Requirements	11
Overview of Problem Space	11
Potential Solutions	12
Development Choices	13
The Sensor Motes	13
Project Management	14
Design	15
Application Classes.....	15
Application Core	16
Graphics Subsystem.....	16
Input Subsystem.....	17
Sensor Subsystem	18
Surface Generation Subsystem	19
Network Interface Subsystem	20
Coordinate Systems	21
Mote Software.....	21
Design Mock-up.....	21
Class Interactions	22
Implementation	24
Application Core	24
Graphics Subsystem.....	25
Input Subsystem.....	25
Sensor Subsystem	26
Surface Generation Subsystem	27
Network Interface Subsystem	28
Example Implementation	28
Testing.....	33
Conclusions.....	34
Future Work	34
Project Evaluation.....	34
References.....	36
Bibliography	37

Additional Materials on the Accompanying CD

The CD accompanying this report contains the following directory structure:

```

/
|-- DirectX
|   |-- dx_sdk_apr_2006.exe           DirectX SDK installer
|   |-- directx_apr_2006.exe        DirectX runtime installer
|-- Engine                           Additional graphics and input sources
|-- Visualiser Source
|   |-- Core                         Application Core source files
|   |-- Debug                       Build output directory
|       |-- Data                   Sensor data files
|   |-- Network                    Network Subsystem source files
|   |-- Sensor                    Sensor Subsystem source files
|   |-- Surface                   Surface Subsystem source files
|-- Kemp_JR_2005.pdf                This report in pdf format
|-- Implementation Guide.pdf        Guide for the example implementation
|-- Expansion Guide.pdf            Guide to application expansion
|-- User Guide.pdf                 User guide for the application

```

The source code in the CD includes all the project files required by Microsoft Visual Studio 2003 to build the application. The Microsoft DirectX SDK (also included on the CD) must be installed before attempting to build the application or the build will fail. If the application is to be deployed on a different machine to that which it was built on then the DirectX runtime will need to be installed on that machine.

Acknowledgements

I would like to thank the staff at Coventry University, particularly Sarah Mount, Elena Gaura, and Bob Newman for their support and the time they spent with me during this project, and especially for showing patience when exposed to my time management abilities, and also Michael Allen for allowing me to use (and abuse) several of his code samples and for helping me out when he had better things to do.

I would also like to thank my friends here in Coventry for providing minimal distractions when I needed time to work, and my friends and family at home for understanding when I couldn't spend as long away from Coventry as I normally would.

And of course, thanks also go to the hard-working people at the University of California, Berkeley, without whose hardware I wouldn't have had a project.

Thank you all.

Introduction

Overview

The potential of wireless sensor networks can perhaps be best described by those people that have had a hand in creating the Mica2 motes that we use and the associates of these people.

“Wireless sensor networks have the potential to be tremendously beneficial to society. Embedded sensing will enable new scientific exploration, lead to better engineering, improve productivity, and enhance security. Research in sensor networks has made dramatic progress in the past decade, bringing these possibilities closer to reality.”

Culler et al, 2005:1

As noted here, these networks have a huge range of applications from exploration to engineering to security. This is made possible by their small size and ability to cooperate to gather and analyse data, though this same advantage in size presents several potential problems such as limited energy and low processing capability individually. These problems are becoming minimised as time passes due to our increasing ability to miniaturise components and make them more efficient, allowing more processing power in less space using less energy. This has led to motes being a feasible creation and in the future they will continue to become more capable, meeting any demands required of them.

Project Focus

Along with the increasing power mentioned above comes the problem of dealing with the increasing amounts of data the sensor motes will be capable of gathering. Currently this is usually satisfied with results printouts or custom-built applications that will be used for the course of one experiment and then discarded. However the former solution is only feasible for relatively low-volume data gathering, and the latter requires a new application to be built (or modified from a previous one) for every new experiment that is attempted. I consider this to be large disadvantage of current systems due to the restriction on the amount of data the sensors can present in the first case and the wasted expenditure of time and possibly money in the second.

Also, as “sensor networks are embedded in uncontrolled environments, a user often does not know exactly what the sensor data will look like” (Levis et al, 2005:1), and so a system that can be adapted quickly as well as show the data in a more readable form is a large benefit.

With this in mind, my project is focused on creating a general-purpose program that can be used to visualise a sensor network while required the minimum effort to allow it to function with the network. I believe this will make experimenting with networks easier and more productive as the focus of a project can shift towards the functionality of the network itself and away from having to find ways to make the data easier to understand.

Project Client

The client for this project is the Cogent Computing Research Group based at Coventry University. This group has an interest in wireless sensor networks and pervasive computing in general, and are involved in research into such projects as routing messages across a sensor network and determining sensor locations. This research means that my project will have the potential to be very useful within the group.

Project Aims

The aim of this project is to create an application that can be used to visualise a sensor network with a minimal amount of effort, with any extra programming focused on how the data is represented rather than how to get this representation in front of the viewer.

The idea of making an application that is as generic as possible in order to work with different configurations of the sensor motes certainly isn't new. For instance, in his paper on Sound Location, Michael Allen notes the following as one of his goals for the application supporting the sensor motes.

“The idea of the application is to make it as general as possible. In that way it can work with all of the different configurations of mote software, which will aid testing. For instance, if all motes are actuator/sensors, then all of them can have coordinate estimates describing their position. If however, some are set up purely as sensors, then they will never actuate, and hence their positions will never change. If the actuators aren't set up to report any distance measurements, then they cannot assist in the trilaterations, it is solely the responsibility of the sensors.”

Allen, 2005:18

While the actual application developed was geared specifically towards the trilateration experiments being conducted, it was engineered to be as generic as possible within that domain. My application will go a step further in attempting to be generic across any experiments that can be performed with the sensor motes.

Report Overview

This report is divided into sections that each focus on a particular part of the application's development. These sections are Analysis, which focuses on the requirements for the application as well as the potential problems during development and the choices I made regarding the environment in which the application would be developed, Design, which focuses on the design of the application, Implementation, which focuses on the actual creation of the application, Testing, and Conclusions. These sections of the report are arranged to follow a logical flow, from the start of development to the end.

Investigation

My investigation into the current work surrounding this area was split roughly into two parts. These were previous work that had been conducted in creating an application such as the one I was attempting to create, and the uses that are being made of sensor networks that could benefit from an application such as this.

Previous Work

My investigation into previous work done into this area brought up surprisingly little in terms of previously implemented systems, I found this strange given the potential value of such an application. I could find no examples of previously created applications designed to be able to be adapted to show a view of a variety of networks, and I can only guess that no-one has justified spending time up-front developing such an application in order to reduce time later on that would be spent writing a complete application from scratch for each network implementation. The only such application I was able to find was SpyGlass, developed by a team composed of people from the University of Lübeck and the Technical University at Brunswick. This application provides a 2D 'top-down' view of the network along with panels for textual output. Extensibility is via user-written plugins which are split into various types in order to reduce conflicts between plugins trying to do similar jobs. As mentioned, this appears to be the only application of this type that the developers have seen fit to release public information about.

The closest other type of application I could find were the network simulators, such as the SenSOR application by members of the Cogent Research Group and NAB (Network in A Box) developed at the Ecole Polytechnique Fédérale de Lausanne in France, which simulate the operation of a wireless network, though these often did not provide a graphical interface, or did provide a graphical interface albeit an unintuitive one for those that have not used the application before, and still require programming knowledge in the appropriate language in order to adapt them for a particular network.

Networks

I found a wide variety of networks have been designed and/or implemented, with purposes ranging from experiments in efficient routing algorithms for messages within the network to working out the location of a sniper based on the sound of the gunshot (Simon et al, 2005). Obviously, this represents a huge range of possible applications, but they all have basically the same type of operation:

- Sensors on the nodes receive data
- This data is processed (either within the network or within the client application)
- The results of this processing are presented to the user

This commonality of data flow is because this is the underlying idea of the sensor networks, to receive data and present it to interested parties in a meaningful way.

Even in cases of networks which are entirely orientated towards pure data collection with no user interaction, they must usually transmit this data to a PC or other device with a large storage capacity due to the limits of the onboard memory on the motes. When this is occurring anyway, it would make sense to put in a little extra effort to provide a user interface in case someone does wish to view the network. In fact, this type of network would probably be widely distributed, so an interface would be even more important as the user may not even know where all the sensors actually are.

Analysis

This section of the document details the decisions that went into the development of the application before the design itself was started; this includes the requirements and the potential problems in designing an application such as this.

Requirements

The requirements for the project were established by meetings with members of the Cogent Research Group in which the main expected uses for the application were discussed. The requirements that came out of these meetings are summarised below:

- The application should be able to interface to a sensor network to show data in real-time
- The application should provide a graphical display of the network
- The application should provide a method of interpolating/extrapolating the terrain that the sensor motes are on
- The application should be able to display data on a per-mote basis
- The application shouldn't be restricted to a particular type of network or sensor mote
- The application should be able to provide additional information such as sensor density

These requirements set out the basic functionality that is required of the application in order to be useful to the client for the intended usage. For me to consider the project successful these requirements must be met in the final product, or at least accounted for in such a way that allows them to be met with minimal effort in the future.

Overview of Problem Space

There are several problems to take into account when creating an application such as this one, these are summarised below.

- Network functionality
The number of things that can be done with a sensor network is huge and steadily growing. The application should be able to account for the largest subset possible of the networks and network functions that could be encountered and shouldn't be biased towards a particular network setup. The functionality of individual sensor motes should also not be restricted to particular models or types.
- Development platform
Any platform the application is developed on will not necessarily be in use by whoever wishes to make use of the application, this is unfortunately a problem with almost all software and can only be circumvented to a certain extent, especially in the case of utilising 3D

graphics and I/O operations, where the calls to external code is rarely compatible across platforms.

- Data display

The application should allow the data to be displayed in a form that allows easy interpretation by the user.

These problems combine to make an application such as this a challenge to produce, though not impossible. The scope for customisation of the application must be open enough to be able to account for a wide variety of cases without having to change code deep within the application.

Potential Solutions

The application can account for future expansions to functionality by allowing for the easy replacement of certain pieces of code, such as the code to interface to the sensor network itself. This can be achieved by a modular class-based approach where each class has a defined set of methods it must provide. In this situation, a class can be completely rewritten, but as long as it provides the appropriate methods which produce the desired results it can replace the old class with no problems. If the replaceable classes are partitioned into their own source files, modifying the program can be a simple matter of dropping the functionality you want into the folder containing the source files and performing a recompile. There are things that could require code to be changed in the core of the application itself, but the code can be written to minimise this as much as possible.

Ensuring that the application is not biased towards particular mote types or experiments is a hard task due to the large potential variety. The first obvious steps would be to allow for any number of onboard sensors taking any number of readings over the course of the experiment. Further, you can allow the data from a sensor to be interpreted either as a list of all the readings, as the average of the readings, or simply using the last reading taken. The same can be allowed for any positional readings taken by the motes.

The issues related to the development platform used could be alleviated by making use of a cross-platform type language that is either interpreted or compiled to an intermediate form. This would introduce a speed penalty however, and additionally such languages usually do not provide libraries for 3D graphics programming. The user would also have to obtain and install an interpreter or virtual machine for the language, which may not be possible on (for instance) a machine that lacks internet access. While a scripting language such as Python would raise the possibility of the user being capable of changing the code to account for new functionality without having to perform a complete recompile, I don't consider this a large enough benefit given the disadvantages mentioned.

The data display issues can be mainly solved by virtue of not restricting the application as a whole to a particular network or sensor mote type. Easy interpretation of the display usually requires it to be displayed graphically and in a form that the human brain can easily relate to reality, in this case the obvious solution would be a 3D representation of the motes within the network.

Development Choices

My choices for the software and languages I would be using for this project were based on the problems and potential solutions discussed above. To write the application I decided to use a compiled language with good support for 3D graphics programming, both in terms of available libraries and documentation. My own familiarity with the language was also an issue as a language I had used before would speed up development time considerably. My choice eventually was C++ as it is capable of interfacing with the DirectX libraries (which are in fact specifically designed for use with the language, though they do work with others) and while I have not had as much experience with the language as I have with others I do have access to the Microsoft Visual Studio development environment. Additionally, C++ supports the class-based approach that I wished to use, unlike C, which requires the programmer to create support for classes themselves.

My choice of 3D graphics library was DirectX, specifically its Direct3D component. This was because I had access to more tutorials and examples of using it than I had for other graphics libraries (such as OpenGL) and because I had already been exposed to its documentation so I had a grasp of the concepts behind it.

My choice of using Direct3D as my choice of graphics library also implies the choice of Microsoft Windows as the development platform. This is the platform I have had most experience with, and so the one that was most logical to choose.

For communication with the motes I decided to use one of the many helper classes written to help people use a PC's serial ports. The most common method of receiving data from a serial port is to create a separate thread which can check for incoming data and buffer it, this is because of several reasons. The first is that it is impossible to guess when data will arrive, and polling the serial port in the main application thread slows down the responsiveness of the application. Second, the internal buffer on a serial port isn't huge, and if the application waits too long before checking for data then some may be lost due to the buffer overflowing and data being overwritten or rejected. Third, Windows does not provide an event driven system for the serial port wherein the application would be informed of incoming data via a standard message inserted into the application's message queue. Helper classes can provide this functionality.

The Sensor Motes

The sensor motes I will be working with for the purposes of this project are the Mica2 motes developed by UC Berkeley (shown below with detached sensor board). This is due to them being the most stable and worked upon platform available within the Cogent Research Group at the current time. The Mica2 motes measure 58x32mm (excluding battery pack) and are powered by their own AA batteries. They are equipped with a radio operating in the 900MHz band for communication and a sensor board which allows them to use a microphone, sounder, magnetometer and accelerometer, along with light and temperature sensing capabilities. They are programmed via a PC's serial port and a base station that one mote must plug into, with the base station and its serial port connection also being used to forward data from the network to the PC.



Image courtesy of Michael Allen

Project Management

Before I started coding the application I decided on a series of deadlines that I should stick to during development in order to complete the project on time. I did not apply additional strict project management techniques to this project as I considered them unnecessary given the lack of need to coordinate with and organise a team and the additional overhead in terms of time required to apply these techniques. The deadlines I set for this project are summarised below.

- 16th December 2005
Complete background reading and research, as well as checking requirements with the client and organising a working environment.
- 6th January 2006
Complete design of application and begin writing the report.
- 10th March 2006
Complete first draft of the main sections of the report and minimal application functionality.
- 31st March 2006
Complete draft of entire report with required layout and additional pages (cover, contents, etc). Complete majority of final code and being customisations for demonstration functionality.
- 20th April 2006
Complete report in final form and ensure code meets the client's requirements.

Design

This section of the document deals with the design of the software. It starts with a general overview of the subsystems incorporated into the design and then discusses each one in more detail. Following that there are sections detailing various other aspects of the application that are not directly code related.

Application Classes

The first task in creating class based software is to decide what the different classes within the application should be and how they should interact. Having a well defined interface between classes is the key to allowing their easy replacement when changing the functionality of the application. Before I designed the application, I split it into several subsystems which account for separate areas of functionality. These are:

- **Application Core**
This is responsible for tying all the other subsystems together.
- **Graphics Subsystem**
This controls the displaying of the data to the screen, handling tasks such as setting up the graphics environment when the program starts and informing renderable elements when they should send their graphical data to the DirectX libraries.
- **Input Subsystem**
This handles any input from the user, such as key presses and mouse movement.
- **Sensor Subsystem**
This is responsible for maintaining the application's internal representation of the network and the sensor motes contained within.
- **Surface Generation Subsystem**
This is responsible for generating any surfaces that need to be displayed, such as guesses at the terrain that the network is distributed across or visualisations of the incoming data.
- **Network Interface Subsystem**
This is responsible for interfacing with the external network and parsing the incoming data.

The graphical code within the application will not be completely separable from the Sensor and Surface Generation subsystems due to complexity and performance reasons, but this shouldn't be a problem unless the application has to be modified to use different graphics libraries (such as a conversion to use OpenGL), and even then the required recoding should be mostly limited to the graphics subsystem anyway. In a similar way, the code to handle user input is contained within the Input subsystem, but the code that actually determines what to do with this input must be contained

within the Application Core in order that it may directly modify the operation of the application.

Application Core

This subsystem acts as the code equivalent of glue, ensuring that all the other subsystems work together without problems and that everything happens in the correct order.

When the program first starts, the core must first initialise the Graphics Subsystem, then the Input Subsystem, and then the real-time data capture provided by the Network Subsystem. The Sensor Subsystem is set up by the Network Subsystem as data from the network comes in and nodes are discovered by the application, and the Surface Subsystem is set up on request by the user.

During the running of the application, the core must determine how to react to user input via the keyboard and mouse as well as informing the Network subsystem to process new data (and potentially recreate any surfaces that rely on this data), and informing all renderable elements to send their data to the Graphics Subsystem.

Graphics Subsystem

This subsystem is responsible for the actual rendering of the 3D network representation to the screen. It is based on a simple 3D graphics engine I developed separately which serves the requirements placed upon it by this application and can easily be extended if more advanced features are required in the future. This engine implements several classes vital to the graphical environment.

cgGraphics

An instance of this class will set up the graphical environment for the rest of the application as well as performing such tasks as setting the transformation matrices and flipping the back buffer.

cgVertexBuffer

This stores information about a set of vertices that will be rendered.

cgCamera

This encapsulates behaviour related to the user's view, and provides methods to move and rotate the viewpoint. This allows other code to simply say, for instance, "move the camera 5 units to the right" rather than calculating the new view transformation matrices needed to accomplish this.

cgFreeCamera

This implements a free-floating camera that can be moved relative to its own axis' rather than those of the 'world'.

cgTransform

This will calculate the transformation matrices required to, for example, move an object 10 units up and scale its size up by 2x.

cgText

This allows the easy rendering of text to the screen.

cgMesh

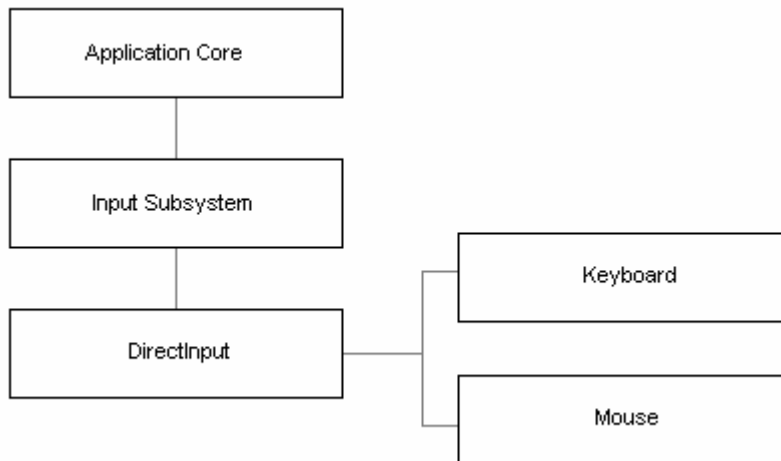
This stores data about an object that needs to be rendered to the screen within the 3D world. It incorporates several other classes and ties their functionality together to provide a single class that can be used for this purpose.

The two main purposes of using this graphics engine are to abstract away the need to deal with transformation matrices and to replace the details of getting an object onto the screen with a simple call to a method such as `cgMesh.Render()`. Transformation matrices represent the mathematical operations that must be applied to a coordinate in order to move from, for instance, its position in the imaginary 3D world being rendered to its position on the screen. A typical transformation will include matrices describing the scaling, rotation and translation to be performed on a set of points, which are then combined to form the final transformation matrix. Luckily the Direct3D component of DirectX includes methods to make the creation of these matrices much easier, allowing you to supply “real-world” values in order to create them, such as in the case of the projection matrix, which describes the transformation from 3D world coordinates to screen coordinates, you can supply the field of view and aspect ratio required for the user’s view of the world.

Input Subsystem

This subsystem is responsible for checking for user input and turning it into a form that can be used by the application. Input for the application will be of two basic types: navigation and control. Navigation input will allow the user to move around within the simulation to inspect different areas of the network and will be based upon the standard control method in most first-person games, with the keyboard controlling movement in the six directions (forward, back, left, right, up, down) relative to the current viewpoint and the mouse controlling rotational movement (looking around). Control input modifies the way in which the application is operating, for instance creating new surfaces and changing which sensors are displayed. This type of input will be via a ‘console’ type system where the commands are typed rather than binding commands to particular key presses. I decided to take this route as it allows for greater flexibility in cases where, for instance, there are several variations of the same command which would otherwise require a key binding each.

Input is checked for by making use of the `DirectInput` component of DirectX, which allows devices to be monitored without having to account for variations between devices of the same general type from different manufacturers. The Input subsystem provides another layer of abstraction which allows the Application Core to call a simple set of functions that check if certain keys are pressed and how much the mouse has moved by since the last check. These levels of abstraction are shown in the following diagram:



The Input Subsystem provides methods to read the current state of the device, keyboard or mouse, being monitored, as well as methods to fetch information about this state, for instance which buttons are held down or how far the mouse has moved.

Sensor Subsystem

This subsystem handles the application's internal view of the network and the sensor motes contained within. It stores data about each mote including position and readings, as well as data about the network as a whole such as the number of motes and mote density.

The basic data that needs to be held about each mote is its position (which can be absolute, relative, or completely abstract depending on how it will be used) and any readings it has taken. Additional data that may need to be stored includes things like how many iterations have been attempted in the case of the example network I will be writing the customisable parts of the application for, or in the case of other networks may include data such as the remaining battery life or some measure of the reliability of the mote. This need for different mote types is handled via inheritance. There is a base class that the Sensor class is derived from that provides the functionality common to all sensor motes, such as adding a new reading or retrieving the mote's position. The Sensor class derived from this base class adds any functionality specific to the motes in use in the particular network being visualised. The basic methods required for the SensorBase class is shown below.

SensorBase

```

Setup
AddPosition(x : double, y : double, z : double)
AddReading(source : unsigned long, value : double)
GetPositions : PositionsList
GetReadings : AllReadings
Render
DisplayData(display : boolean)
  
```

The 'source' parameter to `AddReading` refers to the particular sensor that the reading originated from, for example the Mica2 motes carry light and acceleration sensors among others. `DisplayData` updates whether the `Render` method should write the mote's data to the screen along with sending the marker to be rendered.

The readings recorded by a sensor mote are stored on a per-sensor basis (sensor in this case referring to the actual sensors built into the sensor mote). The application may freely dictate how many sensors a mote has onboard, the limit (assuming the application is run on a 32-bit Windows machine) would be approximately 4 billion sensors, assuming a long integer is used to index them, or as many as the PC's memory is capable of storing information for, whichever happens to be smallest. Obviously, this will not be a limitation for the foreseeable future.

The `SensorNetwork` class keeps a list of all the `Sensor` instances created along with additional data about the network. This additional data includes statistics such as the density of the motes across the whole network or within a particular radius of a location.

Surface Generation Subsystem

This is responsible for generating and managing any surfaces that the user requests, which are used for such tasks as guessing the terrain that the sensor motes are scattered across or providing conceptual maps of the incoming data, for instance a map of temperature within a room. Due to my limited experience in this area, the implementation of this class is mainly left as an exercise for someone with more experience, though a sample implementation is provided. The following methods should be supported by the class.

Surface
Setup(cg : cgGraphics *)
CreateHeightMap(positions : Positions, readings : Readings, mapType : unsigned long, source : unsigned long)
Colour(positions : Positions, readings : Readings, mapType : unsigned long, source : unsigned long)
Recreate(positions : Positions, readings : Readings)
Render
Smooth(method : unsigned short)

`CreateHeightMap` will create the actual surface to be displayed upon a call to `Render`. The `mapType` parameter informs it whether to use the actual mote height or one of the sensor's data readings for the height at the various points on the surface. The `source` parameter defines which sensor's readings to use in the latter case.

`Colour` will change the colouring of the surface based on the parameters passed, it essentially does a similar job to `CreateHeightMap` except colouring the surface rather than setting its height.

Recreate will recreate the height map using the parameters previously supplied, with the exception of requiring new position and readings data as these changing would be the main reason for requiring the surface to be recreated.

Render tells the Surface instance to send its data to the Graphics Subsystem to be rendered.

Smooth will smooth the surface in the case that the implementation used doesn't do this automatically. Whether this smooths the surface a predefined amount, or one iteration is up to the particular implementation. In the case of mine it will perform one smoothing iteration each time this method is called.

Network Interface Subsystem

This subsystem is composed of two layers using a simplified version of other more well-known network stacks as inspiration. These are:

- Layer 1 (Reception)
Receives data packets from the network and forwards them to layer 2 when requested.
- Layer 2 (Data)
Checks the packet header for validity and forwards the payload to the Updater.
- Updater
While this is not technically a layer in the stack, it is very important as this is the part responsible for using the data to actually change the application's view of the network.

This is similar to the way that most network stacks are implemented (or designed) and provides advantages in cases where, for instance, a different packet format is adopted in which case it would only require the code for layer 2 to be modified.

In this design, the Reception layer watches for incoming packets from the network and passes any recently received packets to the Data layer for processing. The Data layer takes a pointer to a buffer containing the received packets, splits them up and checks them for validity. It then passes the actual data contained within each packet to the Updater. This uses the passed data to inform the Sensor subsystem of any changes that have occurred within the network.

From the application's point of view, only the Updater is visible. A method is provided which tells it to update the network view; this causes the Updater to request new data from the Data layer, which in turn requests any new data that may have arrived from the Reception layer. A simple way of visualising this is that the request for data travels down the stack and the data that has been received is passed back up it with some processing at each step to make it understandable to the layers higher up in the stack.

A packet from the sensor motes being used by the Cogent Research Group is 36 bytes in total length with a maximum payload of 29 bytes and is of the following format:

Field Name	Length (bytes)
Destination Address	2
Packet Type	1
Group ID	1
Payload Length	1
Payload (padded)	29
CRC checksum	2

This structure is defined by the development environment used with the sensor motes. The data payload is up to 29 bytes in length and if it is shorter than this size it is padded out to the full 29 bytes. Payload length is set so that the receiving application knows how much of the payload is valid data.

Coordinate Systems

The display of the network within the application must use some form of coordinate system in order to allow the Graphics subsystem to work out where to display items on the screen. This coordinate system must be related some real-world or imaginary coordinate system in order that the display have any meaning.

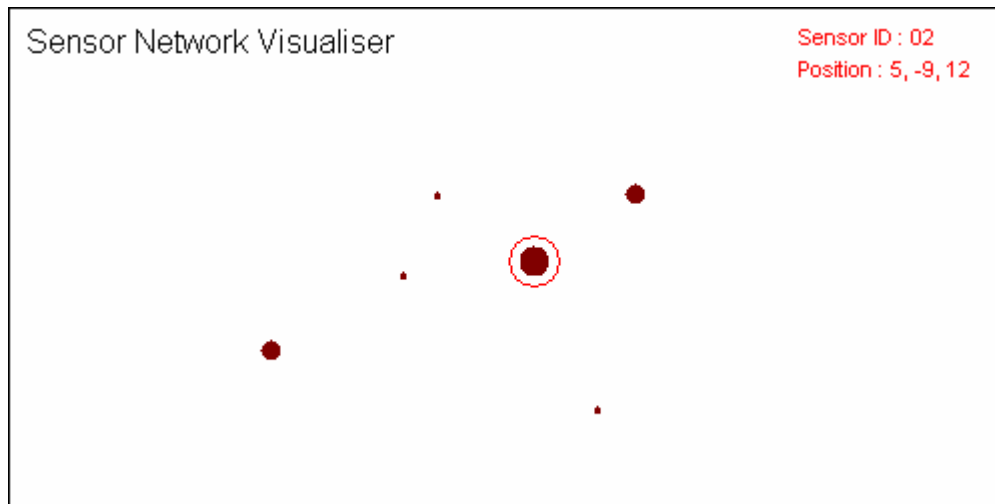
For the purposes of the network the application will be customised for, the coordinates of the sensor markers within the 3D space that is mapped onto the screen directly correspond to the coordinates of the sensors in the real-world (relative to some arbitrary point).

Mote Software

The motes will be loaded with pre-made software that attempts to measure the distances between the motes and reports this back to the base station, and thus the PC, where this is used to perform laterations in order to determines the positions of unknown motes.

Design Mock-up

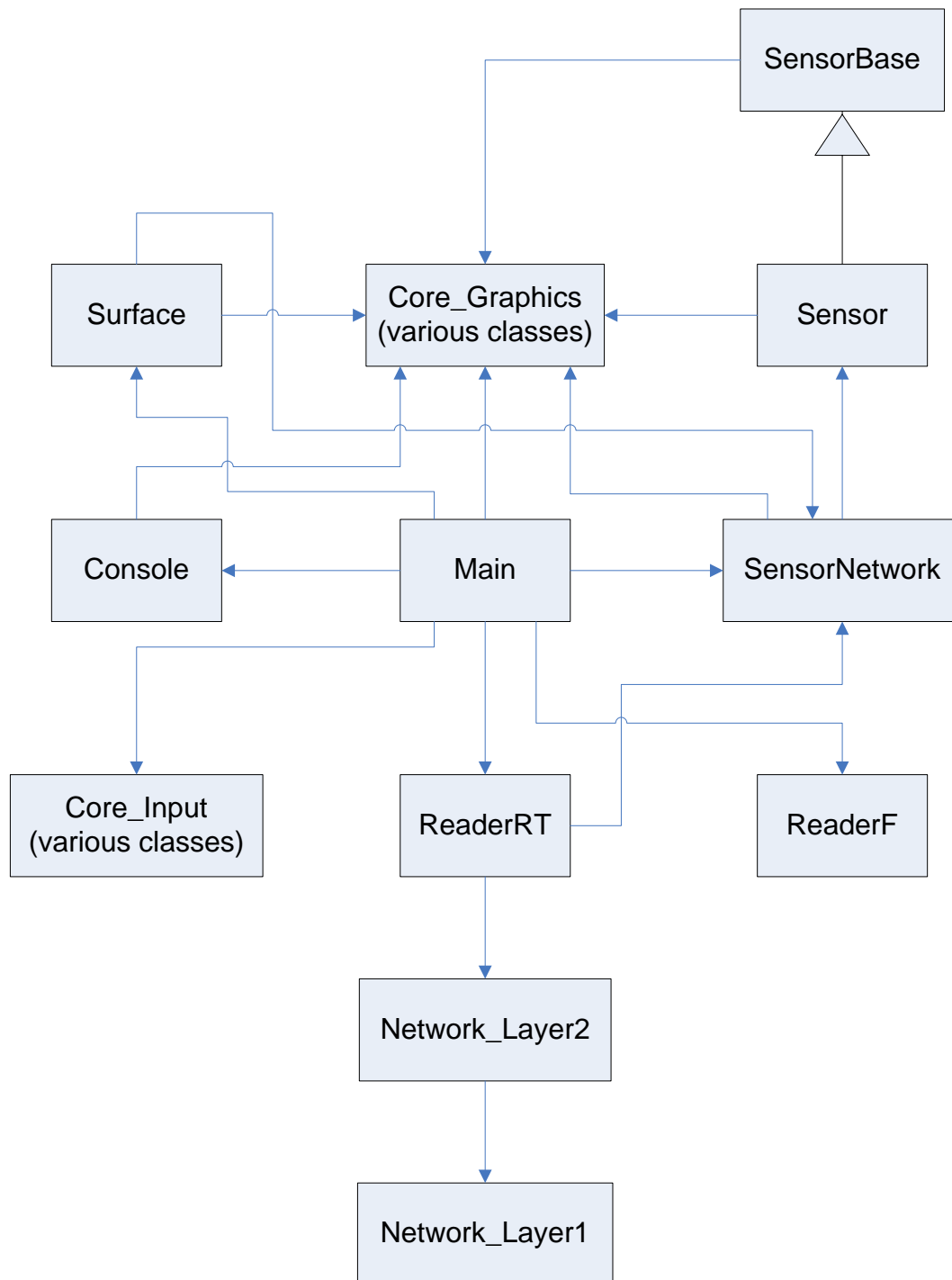
During the design I created a mock-up of the possible layout of the graphical interface of the application in order to guide my work later on when implementing it. This is shown below.



The sensor markers, shown at various sizes to represent the 3D nature of the program where some will be further away than others, are drawn into the main area of the application. When a sensor is selected, in this sketch shown by the red ring around it, its information is shown in the top-right corner of the display. I decided to work with this layout in mind because the display of the network is the most important aspect, and so has the most screen space, while the information about the sensors should be easily noticeable without having to switch to an alternate view or a different display. If there is a large amount of information to display about a sensor then one of these alternate approaches would potentially be much better suited, but for information such as the position of the sensor and its most recent readings this type of layout should be suitable.

Class Interactions

Given the above descriptions, the interactions between the classes are as shown in the diagram presented on the next page. The arrows show where a class makes use of another and the large arrow denotes inheritance.



Implementation

This section discusses the final implementation of the application based on the designs presented in the previous section.

As noted in the Analysis section of this report, the application was implemented using the C++ programming language with the DirectX graphics and input libraries and designed to run on the Microsoft Windows operating system. The application was built up a subsystem at a time, with 'stubs' providing minimal functionality for classes that had not yet been implemented where other classes relied on their existence. This was made possible by the fact that I had already designed how each subsystem and class would interface with each other. Unfortunately the Network Subsystem could not be fully implemented at the time of the submission of this report due to problems reading from the serial ports on the machines being used for testing, however a replacement class capable of reading input from files instead is provided where testing of the remainder of the application is required. Due to the problems being localised within Layer 1 of the network stack, the rest of the subsystem was implemented and simply awaits the dropping in of a functional Layer 1 class in order to work.

Application Core

This subsystem was implemented in two source files, one containing the main core of the application, the other containing the code related to the console interface for the user. This was, as with the other separation into classes within the application, to both allow for ease of modification and ease of maintainability. The console was implemented in a relatively simple manner, using text rather than graphics. This was partly to allow more time for development of the other parts of the application that are more directly related to interfacing with the network and displaying the information about it. The console class provides methods to check for input being typed, toggle the display of the console, and actually render the console to the screen.

The main core sets up the various other subsystems used in the application and then enters the message pump. Under Windows, an application is informed of events by messages being sent to it and placed on a queue that it can, and should, retrieve them from. The message pump is a standard part of almost any application, it simply loops, retrieving messages from the queue and processing them, any that it does not know how to handle are passed off back to Windows which has a series of default handlers. If there are no messages to process, control is instead passed to a procedure named 'frame', so called because in most games it is the part of the application that generates the frames to render to the screen. In my application this procedure carries out that job among others; it is where the rest of the application is controlled from, including checking for new incoming data packets and interpreting user input. When new data is received from the motes any surfaces that have been created are also recreated to use the new data, though this is staggered as otherwise there could be a substantial and very visible pause as all the surfaces are recreated simultaneously.

An option I added to allow for a more visually pleasing view is the ability to allow the camera to automatically rotate around the view of the network, to show it from all angles and from slightly above. This allows for ease of viewing while requiring no user input to manually move the viewpoint.

The general flow within the core during the generation of each frame is as follows:

- If the automatic camera option is enabled then work out the new position
- Handle keyboard input
- Handle mouse input
- Check for new sensor readings
- If new readings were obtained then regenerate the required surfaces
- Instruct the graphics libraries that the rendering of a frame has begun
- Set the new camera position
- Write any text required to the back buffer
- Check if the view is centered on a marker, and if so instruct it to display any relevant data
- Render the sensor mote markers to the back buffer
- Render any surfaces to the back buffer
- Render the console if required
- Instruct the graphics libraries that rendering of the frame has ended
- Send the back buffer to be displayed

Graphics Subsystem

I do not consider the graphics engine code to be a part of this project as such as it is based on code from the game engine developed by me separately to this project, therefore I will not go into great detail as to its workings (such detail would also be quite lengthy). As noted before, the Graphics Subsystem provides several classes which allow other parts of the application to easily render to the screen. It relies quite heavily on the DirectX graphics libraries as it uses them to greatly simplify the entire process of getting the data to the screen. The libraries mean that my code does not need to worry about what type of graphics card is installed, what its capabilities are, or even if there is a 3D accelerated card installed. Obviously, machines with better capabilities will enjoy faster performance, but the specifics are no longer a huge issue for my code.

Input Subsystem

Similar to the above, the Input Subsystem is implemented by means of code from the game engine I implemented separately to this project and I therefore do not consider it to be a part of the project, thus I will keep the level of detail as to its inner workings low and focus the discussion on the interface it provides to the application.

This subsystem provides the `ciInput` class, which is used to set up `DirectInput` (part of the DirectX package) to be used by the application, and the `ciDevice` class which is used to receive input from an actual device. A lot of the implementation details are hidden by the `DirectInput` libraries and more are hidden by the `ciDevice` class, so

interfacing to the keyboard and mouse become a very simple task. Along with some housekeeping methods, the ciDevice class provides the following.

Create(ci, type)

This sets up the device using the parent (previously initialised) ciInput class instance. The 'type' parameter sets whether it should be set up to read from a mouse or keyboard using one of two constants provided in the header file: CI_MOUSE and CI_KEYBOARD.

Read()

This reads the current state of the device and stores it in a buffer within the class instance. No data is passed back to the application at this time.

GetKeyState(char num)

This returns a value representing the state of the requested key (up, down, or held down).

GetXPos()

GetYPos()

GetXDelta()

GetYDelta()

These return data about the position of the mouse, either relative to the point it was at when the first call to Read() was made, or relative to the point it was at last time Read() was made (ie, how much it has moved since then).

These methods make it very simple to read data from the devices as it consists of merely calling the Read() method and then whichever method returns the value that needs to be known by the application. The only caveat is that GetKeyState requires as its parameter one of the key codes known by the DirectInput libraries. This means you must for instance call GetKeyState(DIK_A) rather than the more intuitive GetKeyState('A').

The application makes use of two instances of the ciInput class, one for reading from the keyboard and one for reading from the mouse. The Application Core makes use of both of them to check for navigation type input and the console makes use of the keyboard device to check for commands being typed if the console is active at the time.

Unfortunately there is a bug in the handling of movement and rotation in the free floating camera class which means that this feature is severely limited, this is however a bug that should be easily fixed given time to track it down accurately and as this code is entirely separate to the main application code this bug does not affect any other part of the application.

Sensor Subsystem

The Sensor Subsystem was implemented within three classes. Two are designed to not need modifications except where the network functionality is wildly different to the norm and one is designed to be replaced as needed. The two fixed ones are called SensorBase and SensorNetwork, the former implementing functionality common to all sensor motes and the latter implementing functionality related to the network as a whole.

Along with various housekeeping methods, SensorBase provides methods to set the mote up (including initialising a simple model to render to the screen), add readings and position data, render itself to the screen, and display information about itself to the screen. The data display, in keeping with the layout concept design, is located in the top-right corner of the screen, though the Sensor class can override this if it is desired.

Again ignoring the usual housekeeping methods, SensorNetwork provides methods to do all of the operations stated for SensorBase (individually in the case of adding readings and position data, and to the whole network or subset thereof in the case of rendering) as well as providing information about the network, such as the boundaries of the network (lowest and highest X, Y and Z values).

Each mote is allocated an ID when it is created which is supplied by the code creating the mote, not the SensorBase code itself. This is to allow for easier tracking, for instance where the real motes being interfaced with do not have consecutive identifiers starting at 0.

The replaceable class is named Sensor is derived from SensorBase and adds any extra functionality needed by the particular sensor motes being used, which will usually be by overriding the relevant methods defined in SensorBase. It is usually also a good idea to override the Render() method in Sensor as while this is implemented in SensorBase, it is a very limited implementation simply designed to display the marker. While the framework is in place for displaying data about the sensor, it is not meaningful to do this when the reason for using the application is unknown. As an example, the Sensor class implemented as a demonstration by me overrides Render() to make the marker rotate and to display information about the number of laterations performed by the mote when appropriate, or to display the message "Fixed position during lateration" if the mote was one with a known position. Very little other customisation was required as the functionality required was mainly accounted for by the things already implemented both in the SensorBase class and the rest of the application.

Surface Generation Subsystem

My implementation of this class was quite simple compared to what could be produced by someone with more experience in this area, but it does the job required of it. It creates the surface as a grid where each point on the grid is set to the height of the nearest sensor mote, the appropriate reading from that mote, or the local density of the motes depending what the mapType parameter is set to. This surface can then be rendered to the screen when required by the Application Core. The smoothing operation I implemented performs a single linear smoothing iteration each time it is called. This works by taking the heights of all the points within a certain grid size centered on the current point, working out the average height of these points, and then setting the height of the current point to this average. This is not a very efficient nor advanced technique, but for the intended purpose of the application it is sufficient. The resolution of the grid and the area over which the averaging for the smooth operation is performed can be altered within the code as required. The method which handles colouring the surface takes essentially the same parameters as creating the height map except that it colours the points on the surface based on the data rather than setting the heights of the points, as indicated by the name of the

function. This is unfortunately an expensive operation in terms of time taken and could do with further optimisation. There are additionally several possible changes that could be made to make the calculations use a “best guess” technique rather than begin precisely accurate; this would provide speed improvements.

Network Interface Subsystem

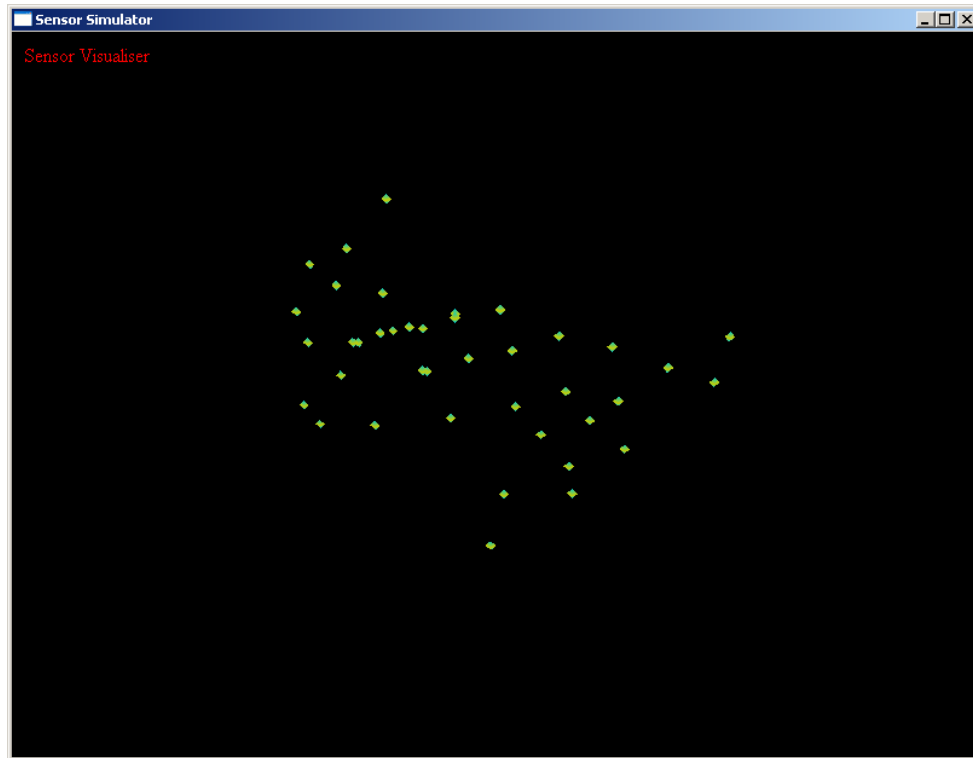
The Network Interface Subsystem implemented in this design was quite simple due to the simple structure of both the packets used by the motes, defined by the TinyOS header files, and the data with the payload of each packet. Unfortunately due to time constraints and several problems reading data in from the serial port I was unable to complete a working layer 1 for the network stack, though this should not be a problem given further development time.

The layer 2 I implemented simply requests packet-sized amounts of data from layer 1, extracts the payload and passes this on. There is very little error checking involved as the messages are quite simple, there is only one type of message so different handlers are not required and only the messages that are supposed to be received by the PC are transmitted to the base station for relaying.

The Updater I implemented takes the incoming readings and stores them until it has collected ones that it is capable of using to calculate the position of one of the motes which has an unknown position. It then updates the SensorNetwork class, and thus the Sensor instance itself, with this new data, which will be reflected in the user's view of the network next time the display is rendered. The lateration itself is performed via an algorithm derived from code generously supplied by Michael Allen, which was in turn derived from a description of “Multilateration Mathematics” produced by Prabal Dutta and Sarah Bergbreiter of the UC, Berkeley (see bibliography). This piece of code demonstrates another advantage of using the DirectX libraries in this project, which is their support for matrices and the various matrix operations which are used. The incoming readings are stored in a vector (which is essentially a dynamically resizable array) on a per-mote basis and when four readings are received which relate to a particular mote its position is worked out and the SensorNetwork instance is informed. I say “four readings [...] which relate to a particular mote” rather than just “four readings from a particular mote” because a reading from a different mote which gives the distance to the mote in question is just as useful.

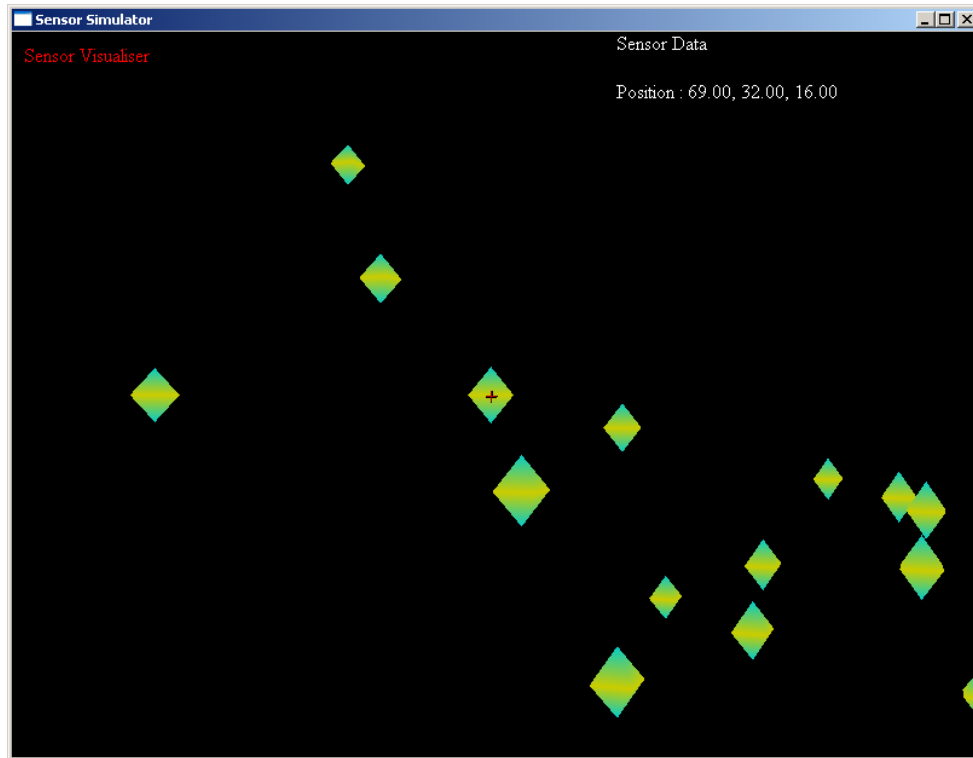
Example Implementation

My example implementations of the customisable areas of the application are described in their relevant sections and this discussion will not be repeated unnecessarily here, except to note that for the purposes of the example shown here, the only data to be shown was positional data, so the extra information output regarding the number of lateration attempts was not required, and thus the implementation shown is of a slightly simpler nature than the one discussed previously in the report. The only further thing to add is screenshots of the application while running.



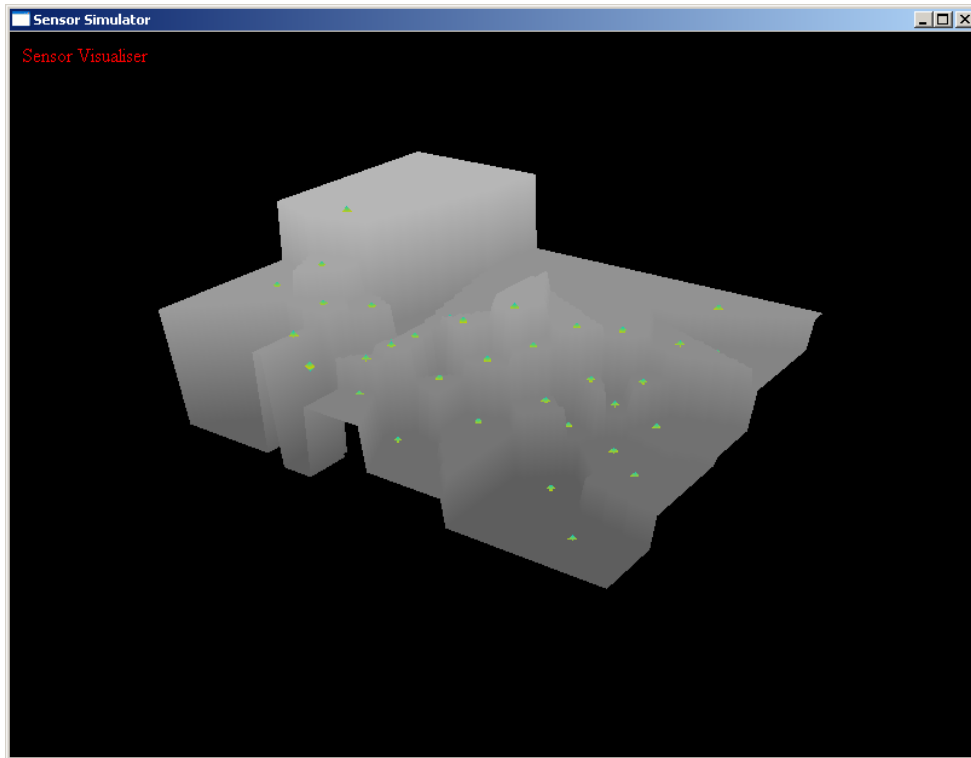
Shown above is the application running using positional data gathered from a model of a mountain range constructed by a member of the Cogent Research Group in order to generate realistic test data for applications such as this. No extra functionality within the application is being used as this point, so no surfaces have been generated to aid in the visualisation of the positions and no information about a particular sensor mote is displayed.

The next screenshot show the results of focusing on a particular sensor mote, thus causing it to display information about itself. If this screenshot is compared against the concept diagram of the screen layout shown earlier then the similarities can be easily seen.

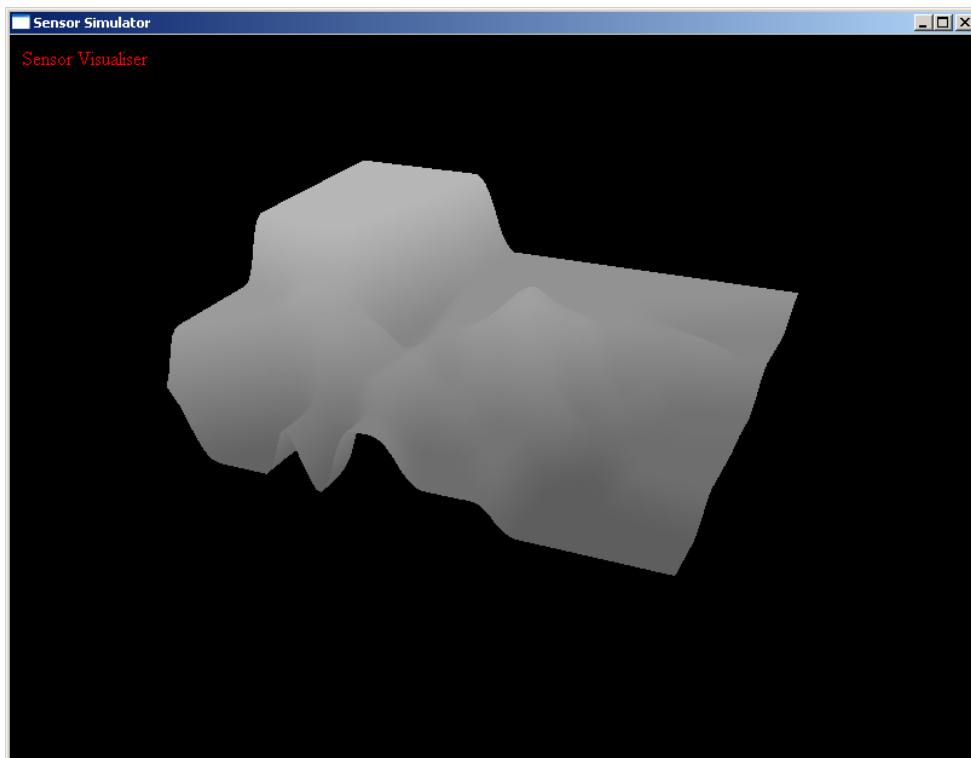


This display, however, still does not give any indication of the layout of the sensors, as the cloud of markers is not very easily interpretable, this is where the surfaces come in.

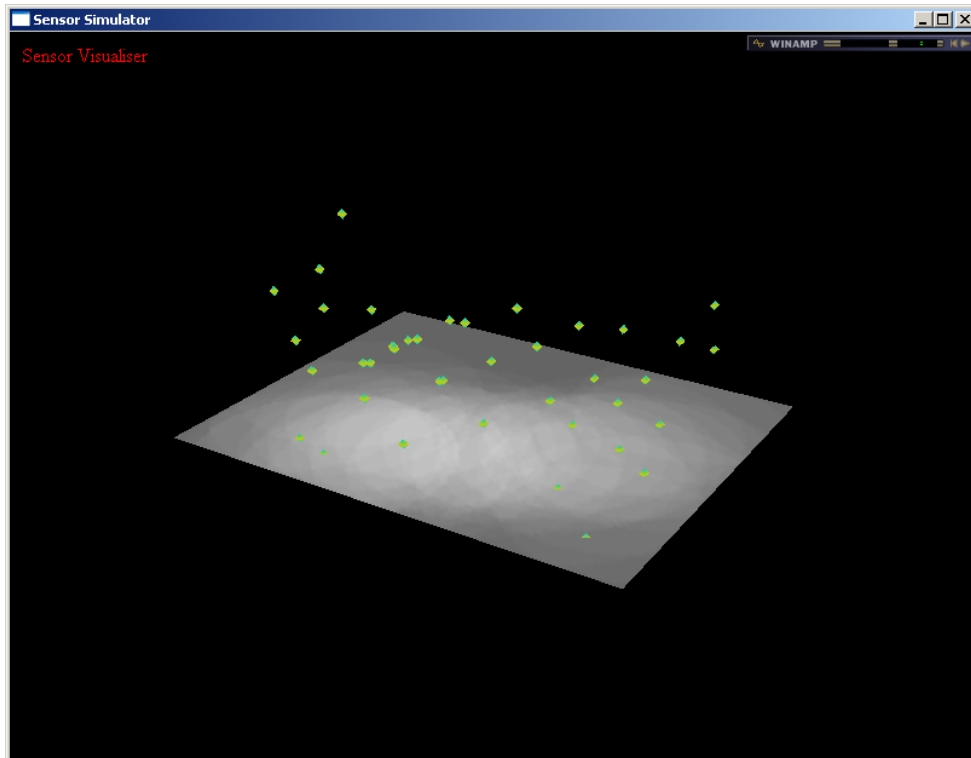
The next screenshot shows the result of creating a surface based on the sensor points shown. At this point the surface has not been smoothed, and so each point is simply set to the height of the nearest sensor mote, hence the jagged and angular appearance. Note however that with the surface having been added to the display, even taking into account that it has not been smoothed in any way, it is now much easier to see roughly what the area of terrain beneath the sensor network looks like and where the majority of the sensors are located. It is much easier to spot, for instance, that most of the markers are located on the side of the terrain that is shown facing us in the screenshot, while the back has almost no points mapped.



Now that the surface has been generated it makes sense to smooth it a bit to get a better idea of the shape of the terrain. The result of two smoothing iterations in the example implementation is shown below with the sensors hidden to allow for a better view of the surface. With the smoothing having been performed, it becomes easier to see what the terrain beneath the network could realistically look like, and it does in fact match the original mountain range model quite closely.



Finally, the screenshot below demonstrates the use of the surface colouring method to colour a new, flat, surface based on the local sensor density. A flat surface was used in this case as this prevents the shape of the surface from obscuring the information presented.



In order to read in data while the real-time data capture was unavailable, I created a class to read in data from plain-text files to use to create the network. This class was given a very simple interface where there is only one method that needs to be called, the appropriately named `FetchReadings`. This method parses any files in the `Data` folder located in the same folder as the application executable (or more specifically, the application's working directory) and extracts information about the position of the sensor nodes as well as the number of laterations performed by the node. The files are assumed to be in a format used by Michael Allen to export this data from his own application to MatLab, as this is the format of data file that was available for testing.

Testing

Most of the testing of the application occurred during development, with each method or piece of code being tested as it was written. This was done because it limits the scope required for testing to that particular piece of code and does not complicate testing by having to account for the interaction between different pieces of code, which could make a problem appear to be related to a different piece of code than where it is actually located.

One type of problem that can be particularly dangerous to programs is buffer overruns. These are traditionally exploited in operating system code to run other code at higher privilege levels and in non-critical applications can cause very spurious and hard to track problems. I found two locations in my application that were vulnerable to buffer overruns. These were located in the two parts of the application that require user input, the console code and the code that allows the required resolution to run at to be passed via the command line, which both used fixed size areas of memory to store the input. The first, in the console code, would accept input even when the size of the buffer was reached; this was fixed by performing a check and rejecting the input if the buffer was full, though see the next paragraph for additional information. The second, in the code accepting command line parameters, assumed that the resolution supplied had up to four digits for each dimension, if more were supplied then the buffer would be overrun. The second problem was fixed by increasing the size of the buffer slightly to allow larger resolutions and performing a check that only reads in the appropriate number of characters.

One advantage of using C++ over C is the additional classes available, particularly within the Standard Template Library (STL). Extensive use of the vector class from this library was made in order to prevent the need for pointers and manually allocating memory, which if it is done wrongly can crash the program and cause other more subtle and hard to track errors. A vector is essentially a dynamically resizable array, and the STL takes care of all the implementation details making it very easy to use. Another useful class that C++ provides is the string class. This once again reduces reliance on pointers, in this case pointers to character array ('C' strings). The above mentioned problem within the console class was removed completely when I converted much of the program to make more use of the very useful classes I mentioned, as the string class removes the vulnerability to buffer overruns by resizing the string as needed. This conversion to use the classes made available by C++ also solved several other problems within the code that had previously required ugly hacks to work around, as well as providing a general shortening of the code due to not requiring manual memory management.

One problem that could not be fixed within the space of the project was camera movement in the Input Subsystem, which does not always move in the desired direction. This appears to be a problem with the translation of the movement from the directions as viewed by the user to movement along the axis' of the simulation. Further investigation will be required to determine the particular reason this fails.

Most other problems were relatively small and were fixed as the code was being written due to my inline testing policy. These problems were therefore fixed as part of the actual code writing and didn't make it into the final code to require further fixing.

Conclusions

Future Work

This section discusses future work that could be conducted on the application, particularly in terms of making it more efficient or easier to use.

There are several ways that this application could be made to be more efficient in its operation, thus boosting the frame rate and overall responsiveness. The first is to make greater use of threads, for instance all the network data collection and usage could be performed in a separate thread rather than being synchronised with the main application loop, as could the recreation of the surfaces. This would mean that these tasks could occur alongside the rest of the application's processing, thus no longer introducing a delay into the main loop other than that required to ensure that data isn't accessed from two threads simultaneously. Another way would be to further optimise the Graphics Subsystem, which is not a task that has been performed to any large extent currently; this would provide a general benefit to all elements that must be rendered.

Additional work could also be put into writing replacement classes to be dropped in for the Network and Sensor subsystems. In particular, the Layer 1 of the Network Subsystem requires completion due to the previously mentioned troubles in its implementation.

I have also identified two additional areas that could be looked into. The first is the idea of using a separate, mainly textual, view for displaying large amounts of data when required, for instance where the history of a sensor's readings must be shown. This amount of information would not be suitable for displaying inline with the view of the network due to space limitations. The second is the use of plugins to extend the application, as is implemented in the SpyGlass network visualiser. The idea here would be to make the functionality of the application fixed and as minimal as possible, and allow the user to supply plugins which handle the receiving of data and display of information. This would potentially allow for easier extension due to there no longer being a need to recompile the source code after a change, but would require the core of the application to allow a lot more freedom and account for more possibilities even when they are very remote. Essentially the replaceable classes in the application would be removed and their functionality moved into external code, such as dynamically loadable DLL (Dynamic Link Library) files.

In addition to expanding the use of the application, there are a few bugs in the current code which are mentioned in the relevant parts of the Implementation section of this report and which require fixing given further development time, though none of them appear to be the result of deeper problems within the application, so should not require any large-scale recoding.

Project Evaluation

I believe that the overall outcome of this project is a positive step. The main requirements for the project were met, with the exception of being able to interface

with the sensor motes, and this capability is very close to completion and has an alternative already available. While this failure to interface with an actual network in the application's current state is quite crippling given that this is what the application was intended to do, it is still quite useful in circumstances where data has already been collected from a network and this needs analysing. In instance of this would be where a simulation has been conducted using an application such as SenSOR, developed by members of the Cogent Research Group, and the results of the simulation have been output to files on the PC. These can be read in by the visualiser application.

The other parts of the visualiser were implemented and meet the requirements given at the start of this report. Improvements can be made in the visual appeal of the application and, in some cases, the efficiency of it, but the functionality itself is present and intact and should not need to be changed in any major way to carry out its intended purpose.

As noted in the Future Work section, the ideas of multiple views and plugins could allow the application to even further extend its abilities, though these are not required for the current intended purpose they may be valuable at some future point.

Further development time is required for the application to be considered fully complete, but it is capable of handling a wide range of situations in its current state, therefore I consider the project to be a success overall.

References

- Allen, M. (2005) '*Sound Location in Wireless Sensor Networks.*' Unpublished PhD thesis, Coventry University
- Culler, D. Dutta, P. Tien Ee, C. Fonseca, R. Hui, J. Levis, P. Polastre, J. Shenker, S. Stoica, I. Tolle, G. and Zhao, J. (2005) '*Towards a Sensor Network Architecture: Lowering the Waistline.*' In: "*Proceedings of the Tenth Workshop on Hot Topics in Operating Systems (HotOS 2005).*" Held June 2005 at Santa Fe, NM.
- Levis, P. Gay, D. and Culler, D. (2005) '*Active Sensor Networks.*' In "*Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI).*" Held May 2005 at Boston.
- Simon, G. Maroti, M. Ledeczi, A. Balogh, G. Kusy, B. Nadas, A. Pap, G. Sallai, J. Frampton, K. (2004) '*Sensor Network-Based Countersniper System.*' In: "*ACM Second International Conference on Embedded Networked Sensor Systems (SenSys 04).*" Held November 3 2004 at Baltimore, MD. Pages 1-12

Bibliography

- Buschmann, C. Pfisterer, D. Fischer, S. Fekete, S. and Krölller (2004) '*SpyGlass: A Wireless Sensor Network Visualizer*' SIGBED Review [online] 2 (1). Available from <http://www.cs.virginia.edu/sigbed/vol2_num1.html> [8 Dec 2005]
- Dutta, P. and Bergbreiter, S. (2003) '*Multilateration Mathematics*' [online] available from <<http://www.cs.berkeley.edu/~prabal/projects/cs294-1/multilateration.pdf>> [15 Feb 2006]
- ISIS, Vanderbilt University (2005) '*Prowler Probabilistic Wireless Network Simulator*' [online] available from <<http://www.isis.vanderbilt.edu/Projects/nest/prowler/>> [9 Dec 2005]
- ISI, University of Southern California (2005) '*The Network Simulator - ns-2*' [online] available from <<http://www.isi.edu/nsnam/ns/>> [9 Dec 2005]
- MICS, Ecole Polytechnique Fédérale de Lausanne (2005) '*NAB (Network in A Box)*' [online] available from <<http://nab.epfl.ch/>> [9 Dec 2005]