

*School of Electronics and Computer Science  
Faculty of Engineering, Science and Mathematics  
University of Southampton*

*Joshua England  
28/04/11*

## **3D Graphical Output of Voxel Data from Gait Experiments**

*Project supervisor: Dr. J.N. Carter  
Second examiner: Dr N.G Green*

**A project report submitted for the award of Computer  
Science**

## Abstract

This report explores the design and development of a voxel viewing system which uses Ray Tracing in order to produce an image and executes on GPU hardware. The voxel data was collected from a Gait research tunnel and pipeline at Southampton University. The system developed produces a graphical representation of human subjects in this tunnel, from any position or angle set by a virtual camera controlled by the user.

Although the software was developed around this specific scenario, the techniques described in this report and used in the implementation can be generalised. The key design feature was using multiple GPU threads in order to trace Rays in parallel. The tracing process used the Fast Voxel Traversal Algorithm in order to traverse the data in a constant time until an active voxel was found.

Using a GPU in this way gave a noticeable acceleration to the application over CPU execution. The Ray Tracing process was able to take place in real-time with a frame rate of 30fps on a 512x512 pixel window using standard consumer hardware.

## Contents

Abstract.....	i
Acknowledgements .....	2
Originality Statement.....	2
1 Project Description.....	3
2 Background Reading.....	7
3 Design Basis.....	9
3.1 Ray Tracing Algorithm.....	9
3.2 Fast Voxel Traversal Algorithm.....	12
3.3 Using parallelism to achieve time complexity acceleration.....	12
3.4 Real Time Ray Tracing using GPU Hardware.....	13
4 Development Process.....	14
5 Final System State.....	17
5.1 System Layout.....	17
5.2 Work Flow.....	20
6 Project Management.....	26
6.1 Time Spend Management.....	26
6.2 Problem Solving during Development.....	28
7 Critical Evaluation.....	31
8 Future Work.....	33
References.....	35
Glossary of Terms.....	37
Appendix A : Control Flow Diagrams.....	38
Appendix B : Gantt Charts.....	44
Appendix C : Screen-shots.....	48
Appendix D : Agreed Project Brief .....	52
Appendix E : Achievements of Requirements.....	53
Appendix F : Contingency Planning.....	54
Appendix G : CD Contents.....	55

## Acknowledgements

I am grateful to my supervisor John Carter for his suggestion about the project theme. Although I wished to work in area of graphics, it was he who used the existing Gait research infrastructure at Southampton University in order to create a realistic scenario.

## Originality Statement

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at ECS or any other educational institution, except where due acknowledgement is made in the report. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

## 1 **Project Description**

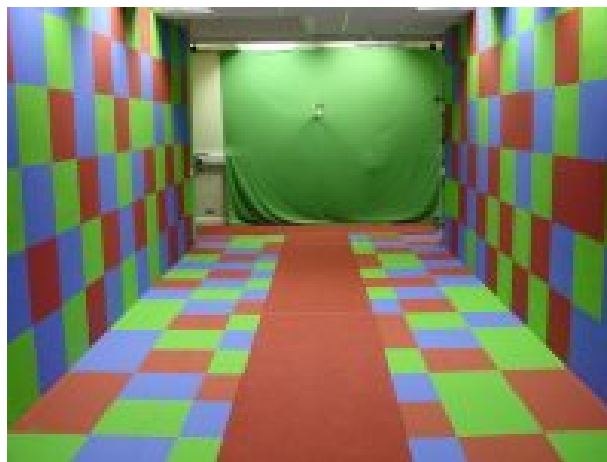
The aim of this project was to make a system which could produce a graphical representation of voxel data in real-time. This was achieved using Ray Tracing to discover active voxels with computation being performed in real-time using GPU hardware.

The following section introduces the scenario behind the project, formally specifies the goals and requirements before describing the GPU programming paradigm which is key to the implementation as it provides a lightweight interface to mass parallelism.

### **Introduction to the Problem**

Researchers at Southampton University have been studying the biometric properties of a person's walk or Gait[1]. As part of this research, a tunnel has been constructed in order to collect data to analyse. This tunnel can be seen in Figure 1.1.

Subjects are asked to walk through the tunnel whilst multiple cameras track their movements. The pictures taken are then fed to an existing processing pipeline where image processing and computer vision techniques are applied – one of which is triangulation in order to produce 3D voxel data.



*Figure 1.1: Gait research tunnel at Southampton University*

Voxels are the 3D equivalent of pixels and describe a small volume in 3D space. The data produced from the pipeline shows the volume which the subject is occupying in the time frame. The data is represented as 1 byte per voxel with a binary classification so show which voxels are active and contain the subject.

The main aim of this project is to produce an graphical representation of the subject using this data. This must be done in real-time at 30 frames-per-second so that the viewer can receive data in memory from the existing pipeline in the future. The system must also allow the user to be able to choose their view point by changing the position and orientation of the virtual camera. This will allow the user to view subjects from any point rather than from the fixed cameras or along the axis.

The existing pipeline uses GPU technology in order to accelerate processing as the hardware allows for multiple lightweight threads to be controlled. This means complex or repetitive work can be done in parallel. This project also uses the hardware to be able to produce the representation in real

time. GPU programming is a relatively new paradigm and is discussed in more detail later.

More information about the aims & goals of this project can be found both informally in the 'Agreed Project Brief' (see Appendix D) and requirements section below.

### Project Goals and Requirements

The following is a formal definition of the goals and requirements of the project and software produced. These are then used to evaluation section later in this document. Goals are defined as high-level statements for what this project would achieve. Requirements relate directly to the implementation of the voxel viewing system. They have been split into *functional*, things the program must do, and *non-functional* properties of the system.

#### Project Goals

- To provide a graphical representation of voxel data generated from the Gait research tunnel
- To provide a system which could easily integrate with the existing processing pipeline
- All work is to be done within 400 hours during the academic year 2010/2011

#### Functional Requirements

- Must be able to read, store and manage voxel data either from files or the existing pipeline
  - All voxel data forms a data-volume describing the state of the tunnel
  - Each voxel represents a small part of the tunnel volume
  - The tunnel is divided into 77x367x201 voxels
  - Each voxel is represented by 1 byte
  - If a voxel has a value  $> 0$  then the subject is in that part of the volume
- Use a windowing system display a graphical representation of the subject described by the voxel data
  - Graphical representation must be as accurate as possible
  - No need for lighting or colour calculations
  - Windowing system to allow for instant display
- Allow the user to interact with the system, to view the scene from different positions or angles
  - Capture user inputs and use them to change the parameters of the virtual camera which controls the view-port

#### Non-Functional Requirements

- The program must be able to execute on a desktop computer running Linux due to legacy code from the existing pipeline
- The program must be able to produce an image in real time, at 30 frames per second as this is the target data rate of the existing pipeline

## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

- Future integration work should allow for a subject to be viewed virtually whilst they are still within the physical tunnel
- Nvidia CUDA is to be used as the API for GPU programming, in as the existing pipeline does
  - Target architecture is 9600 series of GPUs using CUDA 1.1

As the report will show, these requirements have been met by the voxel viewing system produced. One key non-functional requirement is that the Nvidia CUDA API should be used to allow GPU programming to be used. The next section explores this paradigm and how it applies to this project.

### GPU Programming

The ability to use GPUs for general computation is a relatively new paradigm, introducing new concepts and terminology. This section aims to show the benefits of GPU programming, explain some of the jargon and show how this project uses the technique.

Traditionally GPU hardware has been used for graphics processing, through APIs such as OpenGL[2] and DirectX[3]. The demand for better frame-rates and more complex scenes in computer games has led to many advances in this field, chiefly the ability to perform many operations in parallel. This ability allows for multiple graphical primitives to be rendered in fractions of seconds.

In the past, these APIs specified a fixed pipeline which was largely abstracted into function calls. As the technology progressed, the process became more flexible. Shader technology (such as GLSL[4]) was introduced to allow for short programs to be written to perform certain operations. For example pixel-shaders can be written to simulate lighting at every visible point in the 3D scene. These programs are executed in parallel during the rendering process.

The concept of a programmable GPU has been pushed further to allow for general computation to take place on GPU hardware. APIs such as CUDA[5] and OpenCL[6] provide a simple way to write flexible shader programs (known as kernels) which access the parallelism, previously used for graphics, so it can be used for other tasks.

OpenCL is analogous to other open standards, such as OpenGL and OpenAL. It is supported directly by AMD, Apple and ATI whilst also receiving driver support from Nvidia. It also aims to go beyond the scope of GPU programming by allowing programmers to utilise other processing power available in hardware.

However, the requirements of this project force the use of CUDA which is a proprietary technology developed by Nvidia. It is available on most GPUs later than the 8000 series and is currently in version 4.0.

CUDA uses the concept of kernels, grids, blocks and threads in order to perform work on the GPU. These are explained below:

- Kernels are similar to graphics shaders: short subroutines or programs written in a C like language which are executed in parallel on the GPU by a grid of threads
- A block is a group of threads that can cooperate, they are synchronized and can share memory
- Threads execute on multi-threaded Streaming Multiprocessor cores

Other terminology often used is:

- Device : the GPU hardware where kernels are ran
- Host : the CPU-controlled hardware where the main program is executed

To run a GPU program, the kernel is executed in parallel on a grid of multiple threads on the device. Each thread executes independently but can share memory either with its neighbours (in the same block) or globally with the grid. Each thread is assigned a small piece of work, in this project this is to track one ray. Once all threads executing the kernel have terminated, control flow returns to the host/CPU. Figure 1.2 shows this process pictorially.

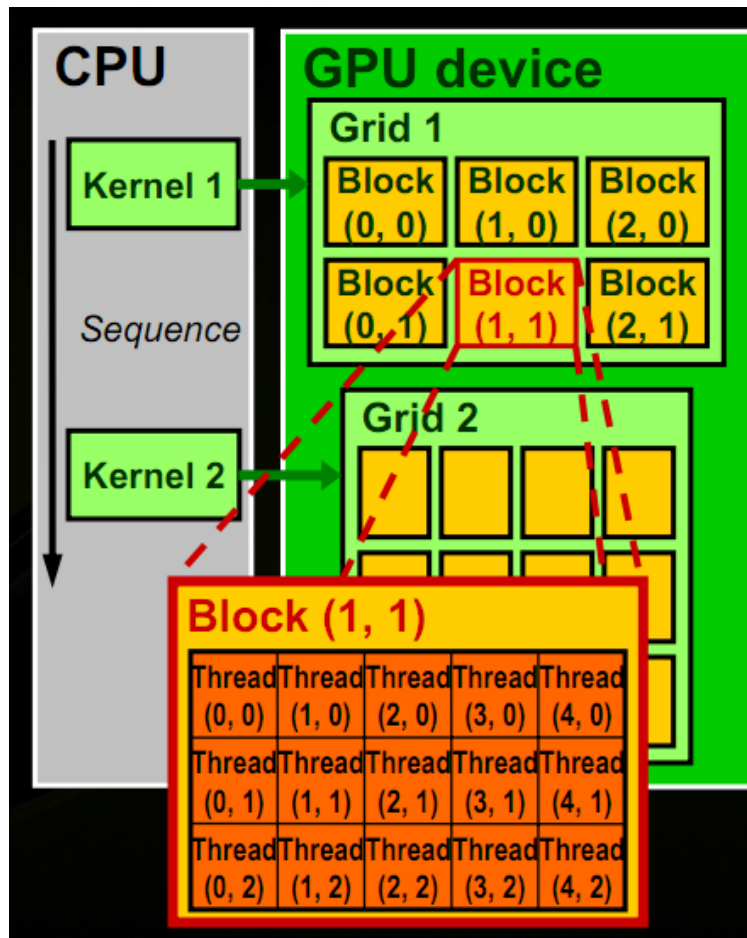


Figure 1.2: How CUDA runs the kernel in parallel over multiple threads in the GPU hardware

Due to the large number of threads, available with limited overhead (especially compared to distributed systems) CUDA has been used in fields such as Medical Imaging, Computational Fluid Dynamics and Environment Science. As GPU programming continues to develop, it is likely to become much more common place. For more information about CUDA please refer to the CUDA by Example Book[7], CUDA Programming Guide[8] or other online resources[9][10].

This project uses GPU programming to performing ray tracing. Traditionally, this graphics technique has been renowned for having a poor time complexity. However, by performing tasks in parallel images can be rendered in real time. Section 3 explains this concept further and explains some of the design decisions made in order to take advantage of it.



## **2 Background Reading**

After assessing the problem and requirements, background reading swiftly began. This started with the graphics field, specifically Ray Tracing, using various APIs including CUDA for GPU programming but also included more general development topics. This section describes this reading process in more detail and how it affected the project.

### **Using Ray Tracing and Voxels**

The first area of reading was the graphics field. Traditionally, pixel-based methods have been used to display multiple primitive shapes which make up the scene. Voxel data can not easily be displayed in this way as it does not describe shape, only volume. Therefore a different graphics technique, Ray Tracing, was investigated.

In Ray Tracing, the computer tracks the path of a virtual light ray travelling in a virtual world. When it collides with an object more rays can be produced to represent reflections & refractions then after extra computation (i.e. for lighting) a colour can be written to the output buffer. If the ray leaves the scene then a background colour is written. This process is described in more detail in section 3.1.

Unfortunately Ray Tracing has a bad time complexity so reading focused on finding methods to resolve this. Most papers centred on techniques involving primitives rather than voxels so they did not directly apply to the problem. Some suggested using complex acceleration data structures[11] [12], such as Octrees. Although some of these could apply to using voxel data, these papers were aimed at computations done on a CPU with easy access to vast amounts of memory whereas its limited in GPU hardware. Also, Ray Tracing papers as a whole tended to focus on static scenes whereas this problem involved the possibility of changing the scene up to 30 times per second as data is received from the pipeline.

Further reading lead to the Fast Voxel Traversal Algorithm[13]. This directly applied to the scenario due to its application to Ray Tracing with voxels. The algorithm shows a simple way to model a ray and traverse it through voxel data in constant time making it ideal to be run in parallel by multiple GPU threads. The FVTA is described in more detail in section 3.2.

After considering the Fast Voxel Traversal Algorithm and Ray Tracing, a rough solution emerged. Multiple GPU threads, executing in parallel, would be used to track Rays in the system. These rays would be used to discover active voxels by traversing through the data. An image would then be formed and displayed using a windowing system. This design concept is described further in section 3.4.

### **CUDA, SDL and OpenGL**

These 3 APIs were chosen for use in the project to provide key functionality in the final system. The next few paragraphs explain the reasoning behind their choice and describe the reading which took place in order to use them although CUDA was forced due to the requirements.

SDL was chosen to handle windowing and user input tasks. This was because of its cross-platform & OpenGL support, whilst code & expertise from other projects could be re-used in order to save time and effort. An alternative library, GLUT, could have been used but is less flexible as it does not let the programmer to completely determine the control flow. The project was developed and tested using SDL1.3 which at the time was still under development.

## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

Documentation for SDL is widely available on the internet in the form of various tutorials, wikis and user forums. Most documentation focused on SDL version 1.2 as 1.3 was still in development, however, the APIs are similar so most still applied. A tutorial[14] gave a good example for using SDL 1.3 with OpenGL and was extended to form the basis of the viewing system.

OpenGL was chosen as the graphical display library as it was supported by Linux, SDL and CUDA through the interoperability API[15] allowing memory shared on the GPU. Originally it was hoped that version 3.2 could be used as it was the latest version to be supported by SDL. However, integration with the GPU code proved difficult as CUDA did not support this later version. Therefore version 2.1 was used in the final implementation.

Lots of documentation was available for OpenGL 2.1 from various websites, forums and wikis on the internet. These helped to provide a more informal explanation to various issues that arose in the project. This was in-contrast the documentation available on the OpenGL website[16] which was much more detailed and sometimes confusing. However, the function reference section here became useful when developing the graphical renderers.

An alternative graphics API was DirectX/Direct3D. This is also supported by CUDA and SDL. However, DirectX is a proprietary Microsoft technology without Linux support so using it would conflict with the requirements.

Use of the CUDA API was forced by the requirements. However, documentation for the API was easy to find and active internet forums[17][18] provided a way to get support from other programmers. The 'CUDA by Example' book[7] included many vital lessons and example code which were incorporated into various prototypes built in the development process. The 'CUDA Programming Guide'[8] was a more in-depth text and became a useful reference, however, it is aimed at more advanced users rather than those just beginning to use the API.

### Other Reading

Aside from focusing directly on the problem or APIs used, other reading centred on how to write better and more reliable C/C++ programs. This included finding good libraries for unit testing and handling command-line flags.

Mid-way through the development process, the Gflags library[19] was fully integrated into the project so that hard-coded variables (such as file-names) could be removed. Some work was done using Google-test[20] in order to test concepts. Being developed largely by Google (before being made open source) the quality of these libraries was guaranteed to be high. Also there was well written documentation available on both home websites.

A summer internship also helped to sharpen programming and problem solving skills through exposure to production level code, standards and testing. Other reading took place over the course of the project to solve specific problems as they arose.

### 3 *Design Basis*

From the background reading, it was clear that traditional graphics techniques were not suited to voxel data and that Ray Tracing should be used. Unfortunately, this process has a bad time complexity meaning it is hard to perform in real-time using standard methods.

This section explores the thought and design process behind the voxel viewing system in order to render images in real time. It begins with a description of Ray Tracing, then explores how parallelism can be used to accelerate algorithms with bad time complexities before showing how to perform Ray Tracing in real-time using GPU hardware and the Fast Voxel Traversal Algorithm.

#### 3.1 Ray Tracing Algorithm

Ray Tracing is a graphical technique used produce a 2D representation of 3D data. The technique simulates the paths of light rays in reverse, attempting to understand their paths, refractions and reflections. It has been used successfully to render special effects in holly wood films, but the algorithm is quite inefficient and so isn't used in real-time rendering (as in video games).

In physics, light rays from a source (such as the Sun) travel, reflecting off objects and entering the eye to form an image. This would be hard to simulate as there are an infinite number of paths to track and only some would be significant. Ray Tracing instead finds the paths of rays which enter the eye by tracking them back to their originating source.

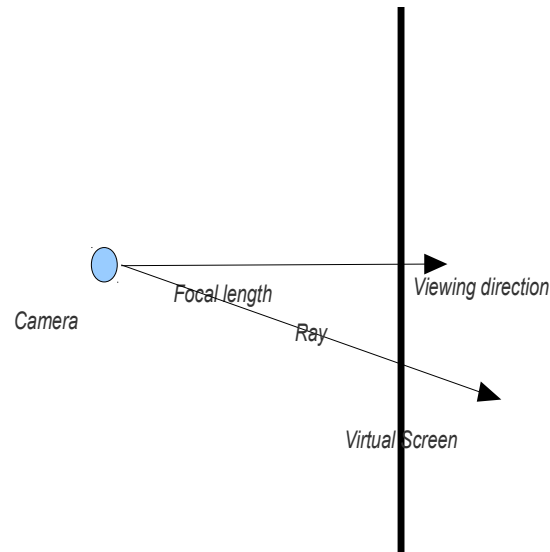
To do this, several key components need to be modelled in 3D space:

- Camera or eye position : the location which all 'viewed' rays much intersect with<sup>1</sup>
- Viewing direction : the direction which the camera is facing
- Virtual screen : represents the user's view-port in 3D space
- A focal length : the distance from the camera position and the virtual screen
- A mapping between pixels on the user's display and a point on the virtual screen
- Representation of the 3D objects to be rendered

Figure 3.1 shows the configuration of the virtual camera, screen and viewing direction from a 2D perspective. In this project, the viewing direction is modelled as a unit vector at the origin, the virtual screen is represented by 2 unit vectors to show its horizontal and vertical direction whilst a 3<sup>rd</sup> vector represents the position of the pixel (0, 0) in 3D space.

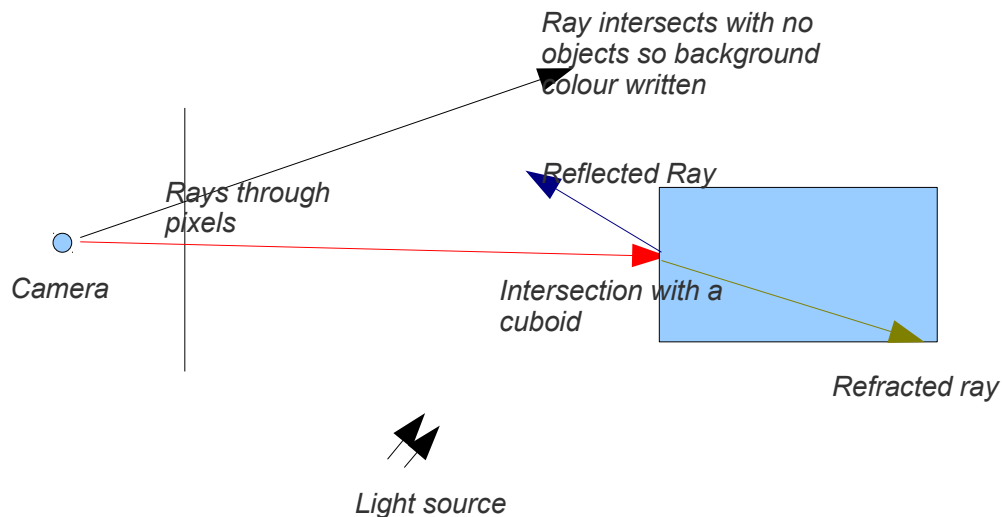
---

<sup>1</sup> Please note this is true of projection perspective ray tracing which shows depth, a parallel perspective image can also be produced



*Figure 3.1: Ray tracing components viewed from a 2D perspective*

When rendering an image, the process is considered pixel-by-pixel. For each pixel: a ray is cast from the camera, through the virtual screen and into the 3D world being represented. This ray is typically represented by a line. Objects are tested for possible intersection with this line, which indicates that light reflecting from them can be seen. If there is no intersection with any object, a background colour is written. Otherwise, further rays can be cast to represent reflections or refractions and lighting can be calculated. When extra rays are cast, their sub rays have to be fully traced and colour values taken into consideration before the colour of the pixel is set. This process is shown from a 2D perspective in Figure 3.2.



*Figure 3.2: Ray Tracing for different pixels in the screen. One intersects with no objects so a background colour is written, another hits a cuboid causing reflections and refractions.*

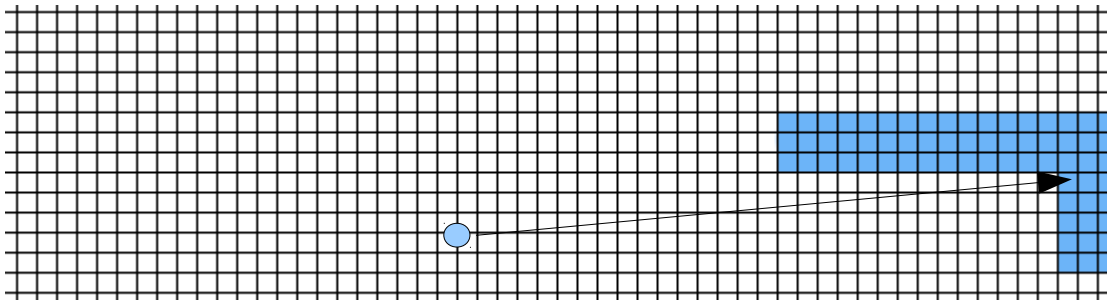
In terms of this project, it is only necessary to find when rays intersect objects so colour-related operations such as reflections and lighting will be ignored. If there are multiple intersections then only the nearest object to the origin of the Ray is taken into consideration.

## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

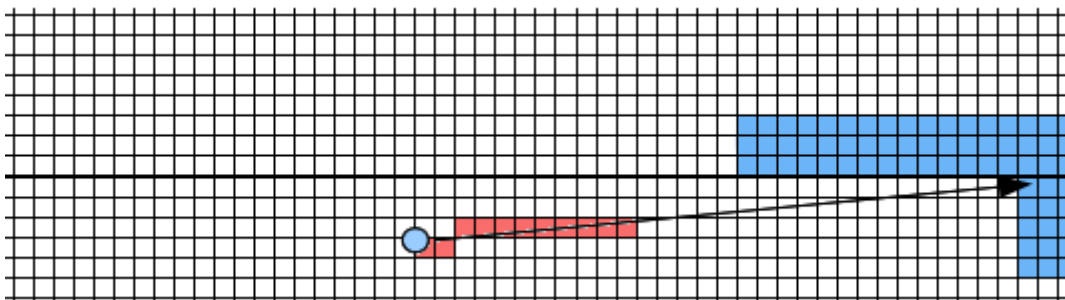
In this project, the subject is the only 3D object and is represented by voxel data. This sub-divides the 3D space into small volumes which are then represented by a binary value which shows whether the subject has entered this part of the tunnel or not.

Using voxels to represent 3D scenes makes Ray Tracing easier. This is because it is possible its possible to use 3D line-drawing algorithms in order to follow the paths of rays as they traverse the voxel data. This avoids testing the ray against multiple objects, complex maths of finding the exact intersection point with shapes and automatically finds the nearest object. This discovery process is shown below in a 2D perspective.

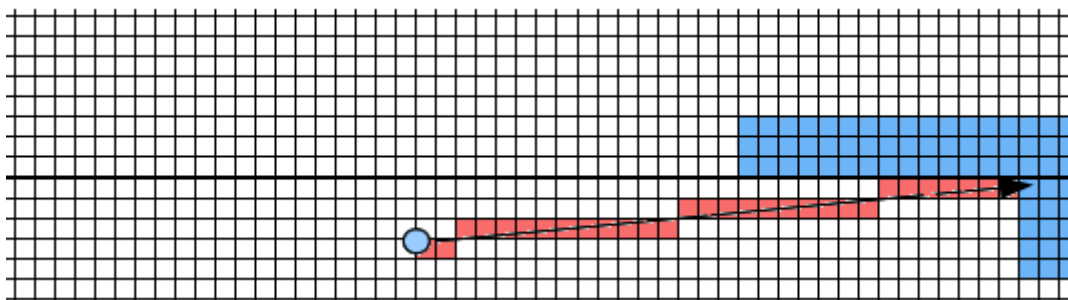
Firstly, a ray is cast for a pixel on the screen in the normal way. At this moment, only the direction of travel is known. The ray is said to be in the same voxel as the originating ray.



As the ray traverses through the scene, it passes through various voxels. This is generalisation of drawing line in graphics, except with 3D rather than 2D space.



As each new voxel is entered, it is tested for the presence of objects. If there is an object, the ray is terminated and work is done to calculate an output. When the ray leaves the data volume (scene) a background colour is written. In effect, the rays are discovering the objects rather than calculating their intersection point.



Because only the one voxel at the end of the ray is considered the complexity of testing for the intersection with multiple shapes is avoided.

In the project, the Fast Voxel Traversal Algorithm is used to traverse the ray through the data in constant time. Because each operation is simple and independent it is easy to perform in parallel using the multiple lightweight threads provided by GPU hardware. This is discussed further in the following sections.

### 3.2 Fast Voxel Traversal Algorithm

This algorithm was found during the background reading stage and clearly suited the problem due to its simplicity and direct application to performing Ray Tracing with voxel data. The following paragraphs describe it in detail and explain its application to the project.

Using the vector form for a point on a line (  $P_t = P_0 + tD$  ), the algorithm takes account of the scale (  $t$  ) in each direction and uses this to determine the next voxel to enter.

The initialisation phase is the most complex and can cause some confusion with the mathematics. There are many values which need to be set in order to perform the calculations for a ray, however, these can all be set in constant time.

- $X, Y, Z$  : the integers showing index of the voxel at the end of the ray
- $\text{step}\{X,Y,Z\}$  : these are either 1 or -1 and show the direction of movement in each axis
- $\text{tMax}\{X,Y,Z\}$  : in the formula  $P_t = P_0 + tD$  this is the maximum value of  $t$  which can be used without leaving the current voxel in each direction
- $\text{tDelta}\{X,Y,Z\}$  : this is the  $t$  which corresponds to moving exactly one voxel in each direction

The main traversing phase also has a constant time complexity, requiring at most 4 comparisons and 2 assignments. The code that implements this in the project is shown below. Note its lay out is slightly different to how it is shown in the original paper[13].

```

1 // Fast Voxel Traversal algorithm
2 __device__ void RayTraverse(Ray *ray) {
3   if (ray->tMaxX < ray->tMaxY && ray->tMaxX < ray->tMaxZ) {
4     ray->X += ray->stepX;
5     ray->tMaxX += ray->tDeltaX;
6   } else if (ray->tMaxY < ray->tMaxZ && ray->tMaxY < ray->tMaxX) {
7     ray->Y += ray->stepY;
8     ray->tMaxY += ray->tDeltaY;
9   } else {
10    ray->Z += ray->stepZ;
11    ray->tMaxZ += ray->tDeltaZ;
12  }
13  ray->dist++;
14 }

```

Figure 3.3: GPU Implementation of the Traversal phase of the FTVA

The Fast Voxel Traversal Algorithm is central to implementation of the voxel viewing system. Its direct application to Ray Tracing using voxels, low memory usage, simplicity, predictability and constant time complexity make it ideal for mass-execution on GPU hardware.

### 3.3 Using parallelism to achieve time complexity acceleration

Some algorithms and tasks have a bad time complexity when executed serially (one instruction at a time). This is often when the same sub-task has to be repeated multiple times using different data or parameters. However, by performing tasks in parallel this time complexity can be reduced as many operations can take place at the same time. Modern computer hardware allows for different paradigms for doing this: CPU-threads, distributed computing and GPU programming. However, they all follow the same principle of spreading work across multiple workers or processors.

Amdahl's Law shows how the speed advantage can be calculated. The total time  $T$  for a program to run is the sum of the time for serial-only execution  $T_s$  and the parts which could be executed in parallel  $T_p$ . Using a traditional computing model and serial execution this becomes:

$$T_1 = T_s + T_p$$

However, when using multiple processors the parallel part of the program can be executed much faster:

$$T_p = T_s + \frac{T_p}{P}$$

If  $P$  is large enough and work can be spread, then the time complexity of the algorithm becomes the time complexity of the serial part only.

$$T_\infty = T_s$$

As explored in the next section, Ray Tracing can use this concept to great advantage as there are many parts of the algorithm that can be executed in parallel – chiefly the tracing or traversal of Rays.

### 3.4 Real Time Ray Tracing using GPU Hardware

The concept behind the voxel viewing system is to discover voxels in parallel using the Fast Voxel Traversal Algorithm. Because much of the algorithm can be executed in parallel and the large number of threads available on GPU hardware, it is possible to produce an image in real time.

For the graphical image to be produced,  $n$  rays have to be cast (1 per pixel). This can travel at most through  $l$  voxels (the distance from opposite corners of the data volume). Using the Fast Voxel Traversal Algorithm there is a constant time  $k$  for each Ray initialisation and a negligible cost for transferring between voxels. This means that the time complexity for serial execution is:

$$T_1 = O(n(k+l))$$

Using CPU/serial technology with a large number of visible pixels, this time complexity would not allow for real-time execution. However, using multiple GPU hardware the time complexity reduces as the work can be done in parallel as shown in the previous section.

Using  $P$  threads the execution can take place in:

$$T_p = O\left(\frac{n(k+l)}{P}\right)$$

And as  $\lim_{P \rightarrow n} T_p \rightarrow O(k+l)$  becoming a constant time execution implying a ratio of 1:1 between Rays and Threads<sup>2</sup>. Because the GPU hardware gives access to a large number of parallel threads this ratio is achievable. Therefore high resolution representations can be rendered in real time. This is the chief principle between the voxel viewing system developed in this project.

---

<sup>2</sup> although actual execution over-heads may mean the optimum ratio is different.

## 4 Development Process

Having developed a concept of how to perform ray tracing in real time, the next stage was to make a working implementation of the voxel viewing system. Due to the experimental nature of the project, it was hard to develop a formal design or test plan. Therefore, an iterative development approach was chosen where multiple prototypes were made or extended in order to add functionality to the overall system.

Initial prototyping investigated the modelling and managing of voxel data, vector maths and using the Fast Voxel Traversal Algorithm. At this stage, Python[22] was often used to quickly test ideas due to its light syntax and scripting abilities. However, the main development took place in C/C++ to allow easy integrating with the existing pipeline<sup>3</sup> and to reduce memory & computational overheads.

Overall, several key prototypes were built including Ray Tracing renderers executing on both CPU and GPU hardware. Code from these was re-used and combined in order to form the final voxel viewing system. Later development work focused on the usability and robustness of the system including capturing user inputs and using command-line flags to set parameters.

This section explores the development process and the prototypes developed whereas the final state of the voxel viewing system is described in more detail in section 5.

### CPU ASCII Ray Tracer

The first key prototype modelled a virtual camera, managed 1 voxel data file and used the Fast Voxel Traversal Algorithm to produce an output. This output was a fake screen buffer which was then printed to the standard output as ASCII text.

It showed what information needed to be modelled, methods to change the virtual view point and showed how to use the FVTA in order to produce a graphical output. However, it was still executing serially and did not allow the user to control the view point.

Figure 4.1 shows an example output of this renderer showing a real subject standing in the research tunnel. One character is printed per pixel with '8' & '.' represent foreground & background respectively. A similar output was produced by the GPU ASCII Ray Tracer which was produced afterwards.

### GPU ASCII Art Ray Tracer

The next prototype modified the rendering code of the CPU Ray Tracer in order to use parallelism available in the GPU. This involved transferring data to the GPU, running a CUDA kernel to perform rendering then copying the resulting image back.

Unfortunately, lots of code had to be duplicated in order to be compiled by CUDA and executed on the GPU. This affected the vector maths library, ray construction & traversing, voxel-to-world translation and voxel-testing code. Although this created a lot of extra work, it was carried forward into the full GPU solution.

Another issue was the target architecture didn't support double precision floating point numbers<sup>4</sup> which the NVCC compiler defaults all maths function calls and literals to. This meant that large

---

<sup>3</sup> It is possible to use Python to interact CUDA[21], however, the API is not as established as the C/C++ versions.

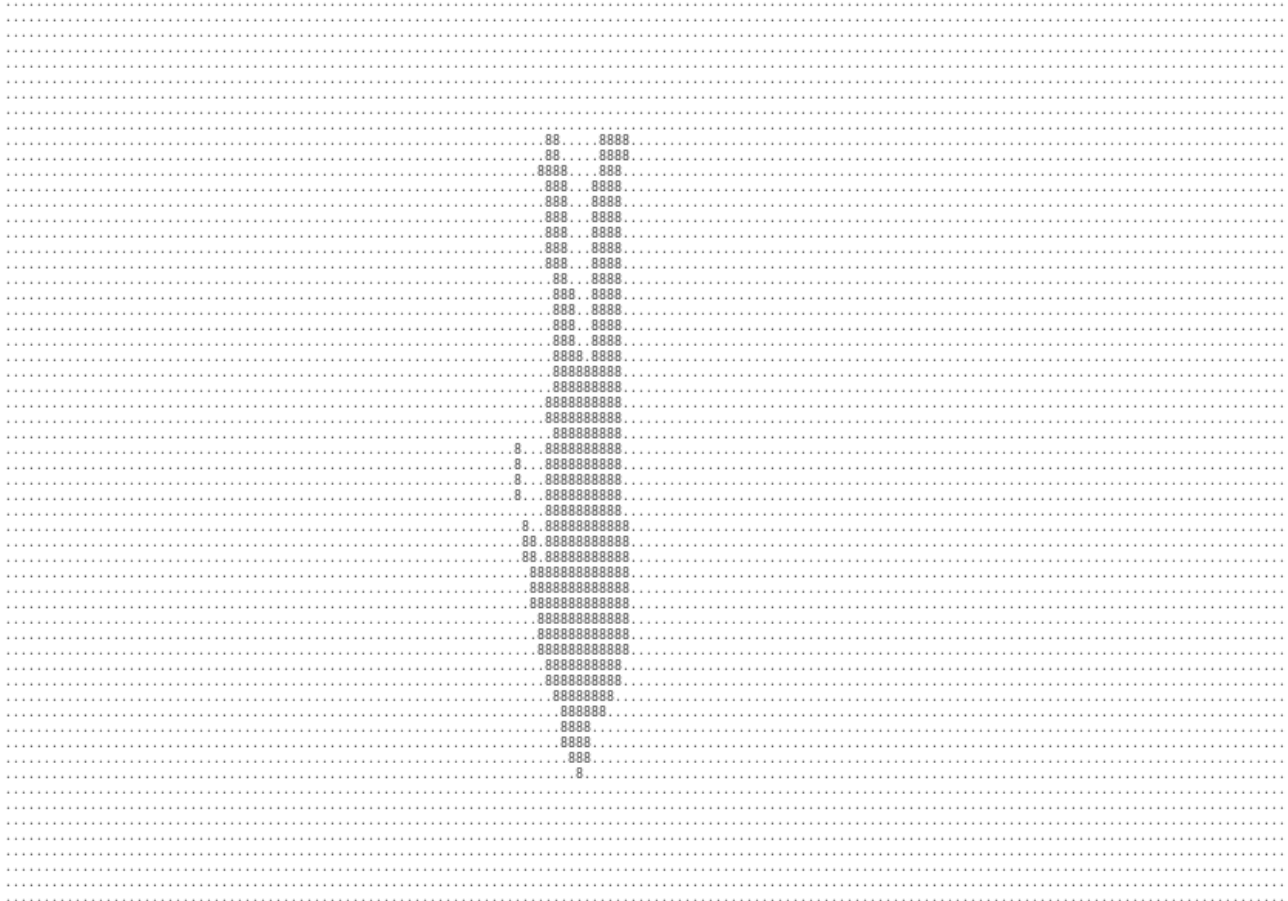
<sup>4</sup> Doubles : IEEE 64 Bit double precision floating point numbers



parts of the code had to be inspected to ensure everything was forced into single precision mode<sup>5</sup>.

Like the CPU solution, the results were written into a fake screen buffer which was rendered as ASCII text. Because rendering tasks were executed in parallel there was a very noticeable speed increase with the computation being in the order of seconds rather than minutes.

This prototype highlighted some of the difficulties of moving from CPU executed code to GPU programming but it also showed that the time complexity acceleration shown in section 3.4 was possible. The time to produce a frame dropped from in the order of seconds to the order of hundredths of seconds. This meant that a real-time graphical renderer could be built.



*Figure 4.1: An example output of the CPU and GPU ASCII Art Ray Tracer with 1 character per pixel. The voxel data used with a real output of the gait experiment tunnel.*

### SDL Displayer with CPU Ray Tracer

This prototype was much more advanced and formed the basis of the final voxel viewing system. It integrated the CPU renderer into an OpenGL context to produce a graphical, rather than text, output. Also extra components were added to capture user inputs and manage voxel data better.

Code developed for other projects was reused giving a structure for managing an SDL window, creating an OpenGL context and capturing/processing user inputs. By re-using code in this way, many hours of work were saved and could be put to use developing the full solution rather than producing supporting systems.

---

<sup>5</sup> Single/Float : IEEE 32 Bit single precision floating point numbers

After the integration work was complete, the renderer gave a stable frame rate of 15fps with a 200x100 pixel window using a desktop computer<sup>6</sup>. This output is similar to the output of the GPU Ray Tracer shown in Figure 4.2.

Further development on this code increased the software quality by using namespaces, STL containers, checking file pointers & file lengths and splitting the support code into de-coupled components. Also the Gflags[19] library was introduced in order to handle command-line flags so that parameters could be set and static/hard-coded content could be removed.

The development of this prototype meant that many of the complexities and engineering aspects needed to fulfil the requirements were met aside from a GPU renderer. The software quality and robustness of the system was also improved. This meant that less work was needed to modify this prototype into the final voxel viewing system with a GPU renderer.

### SDL Displayer with GPU Ray Tracer

The last prototype combined the GPU renderer with the SDL-based prototype described above in order to fulfil all the requirements. Unfortunately many problems were encountered during this stage.

This included the slowness of copying data between the GPU & CPU multiple times, segmentation faults from the CUDA-OpenGL interoperability API and incorrect thread-to-pixel mapping. These problems are expanded upon in section 6.2.

Nevertheless, the work done by the previous prototypes meant that the errors were concentrated around the rendering code only and so were easier to locate and solve. Once working, the prototype was able to render frames at 30fps on a 512x512 pixel window, an example of which is shown below.



*Figure 4.2: An example output of the GPU render with OpenGL integration*

This prototype was then modified and adapted in order to create the full voxel viewing system as described in section 5.

---

<sup>6</sup> CPU: AMD Athlon X2 3Ghz and 2 GB DD2 RAM

## 5 Final System State

During the development phase various prototypes were made and modified. The last prototype met all the requirements of a real-time rendering system which allowed for user controls of the viewing point. However, work continued to produce a more rounded solution. This section describes the final voxel viewing system in detail: exploring the main components, the control flow of the system and examining some source code. Further documentation is provided in the source code itself in the form of comments and by the Doxygen system. Both are included on the CD provided with this document.

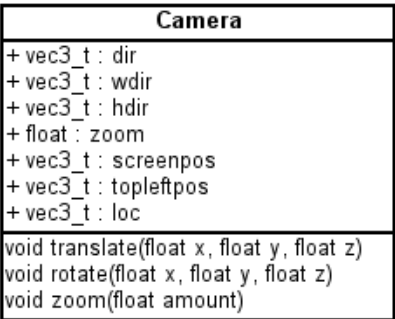
### 5.1 System Layout

The voxel viewing system features several, mainly decoupled, components linked together by a utility component which allows for the high-level work flow to be controlled. Normally these components would be implemented as classes using Object Orientation techniques such as interfaces. However, CUDA is a C-based language and has no classes. Therefore, components are implemented using a struct as the data object then a series of functions to represent methods.

Due to the experimental nature of the project the system layout was not formally designed but evolved during the iterative development process. Therefore, restructuring and other improvements could be made to improve the software. However, an attempt was made to use good practice and techniques such as de-coupled components and patterns.

#### Main Components

To adhere to good software engineering principles, the system has been split into several components. Each of these control a different aspect of the system such as managing voxel data or rendering. Table 1 lists the main components of the system, describing their key responsibilities, referencing their location in the source code along with a class diagram if appropriate.

Component	Files	Responsibilities	Class Diagram
Camera	camera.h camera.cc	Holds the state about the position & orientation of the virtual camera and screen. This information is shared with renderers through the RayTraceInfo component.	 <pre> classDiagram     class Camera {         +vec3_t : dir         +vec3_t : wdir         +vec3_t : hdir         +float : zoom         +vec3_t : screenpos         +vec3_t : topleftpos         +vec3_t : loc         +void translate(float x, float y, float z)         +void rotate(float x, float y, float z)         +void zoom(float amount)     }           </pre>

### 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

Component	Files	Responsibilities	Class Diagram
Ray	ray.h ray.cc cuda_code.cu	Holds state about rays as needed by the Fast Voxel Traversal Algorithm to traverse rays through the volume. Methods/functions require 2 implementations for both the CPU and CUDA renderers.	<pre> <b>Ray</b> + int : x, y, z + char : stepX, stepY, stepZ + float : tMaxX + float : tMaxY + float : tMaxZ + float : tDeltaX + float : tDeltaY + float : tDeltaZ void Init() void Traverse() </pre>
World	world.h world.cc	Manages the voxel data. This is either read from file or from the pipeline.	<pre> <b>World</b> + vector&lt;VOXEL*&gt; : all_worlds + int : index + vector&lt;string&gt; : file_names &lt;&lt;typedef&gt;&gt; VOXEL uint8_t void Init(vector&lt;string&gt;&amp; filenames); int GetSize(); void SetWorld(int index); string GetFilename(); VOXEL* GetWorld(); int GetIndex(); bool ObjectPresent(int i, int j, int k); void Close(); </pre>
RayTraceInfo	raytraceinfo.h	A method for passing information to different renderers using a common interface.	<pre> <b>RayTraceInfo</b> + vec3_t wdir; + vec3_t hdir; + vec3_t loc; + vec3_t topleft; + int pixelw; + int pixelh; + int screenw; + int screenh; </pre>
Controls	controls.h controls.cc commands.h	Handles SDL events from generated from user inputs. Contains mappings so that relevant commands can be found. Can also display current mappings to the user.	<pre> <b>Controls</b> - map&lt;SDLkey, Command&gt; keyMap - map&lt;Command, string&gt; descMap void Init() void GetCommand(SDLKey key) </pre>
Commands	commands.h commands.cc	Handling commands after user input events. This could involve using other components to load voxel data or change the camera properties.	<pre> <b>Commands</b> - mat4 rotmatrix void HandleCommand(Command com) </pre>

## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

Component	Files	Responsibilities	Class Diagram
CUDA Wrapper	cuda_code.h cuda_code.cuh	Provides a wrapper around CUDA code so that other components can use the GPU hardware.	<div style="border: 1px solid black; padding: 5px;"> <p style="text-align: center; margin: 0;"><b>Cuda Code</b></p> <pre> - uchar : dev_buffer - int : dev_buffer_size - VOXEL* : dev_world - GLuint : pixel_buffer - RayTraceInfo* : dev_rinfo - int : device  void Init(int width, int height) void RayTrace(RayTraceInfo* info) void LoadWorld(VOXEL* data) void ShowFrame() void Close() </pre> </div>

*Table 1: Component responsibilities, file reference and make up*

### Other Components

The system also contains several other components with smaller responsibilities. These include libraries for command-line flags and vector maths. Some had to be duplicated in order to be executed on GPU hardware. The components are described in the table below:

Component	Files	Responsibilities
Gflags Library	gflags/gflags.h gflags/config.h gflags/mutex.h gflags/gflags_completions.h gflags/gflags.cc gflags/gflags_nc.cc gflags/gflags_reporting.cc gflags/gflags_completions.cc	Handles user inputs through command line flags. An Open-source Google library which has been lightly modified.
Mathlib	mathlib.h mathlib.cc cuda_mathlib.cuh	Maths library providing support for 3D vectors, homogeneous coordinates and other functions. Based on code from id software[23] but heavily modified. Lightweight CUDA implementation also needed to be made.
Bounding Box	boundingbox.h boundingbox.cc cuda_boundingbox.cuh	Allows the user to view the data from outside the data volume. A CUDA implementation needed to be constructed for the GPU renderer.
Voxel Translation	voxel_translation.h voxel_translation.cc cuda_code.cu	Translates between problem coordinates and voxels. Used mainly in the initialisation of the FVTA

*Table 2: Other, more minor, components in the system*

## 5.2 Work Flow

The following sub-sections aim to describe, in fairly high-level terms, how the voxel viewing system renders images through the use of UML flow diagrams. These show the start-up phase, the loading of voxel data from files, the operation of renderers and how user inputs are handled. The diagrams also reference functions or files in the source code which can be found on the accompanying CD.

Please note that these diagrams show the process of using data read from files rather than receiving data from the pipeline. The proposed control flow for a fully integrated viewer is shown in Appendix\_A 6 and described further in the Future Work section. Also, all flow diagrams shown can be found in Appendix A at a larger scale.

### High level overview

The high-level execution of the program is fairly simple: a start-up phase to read data, a while-loop in which frames are displayed and user input is handled, then a shut-down sequence to free allocated resources. This is similar to the high-level control flow in many 3D applications such as computer games especially those using the SDL library. Figure 5.1 shows this as a flow diagram.

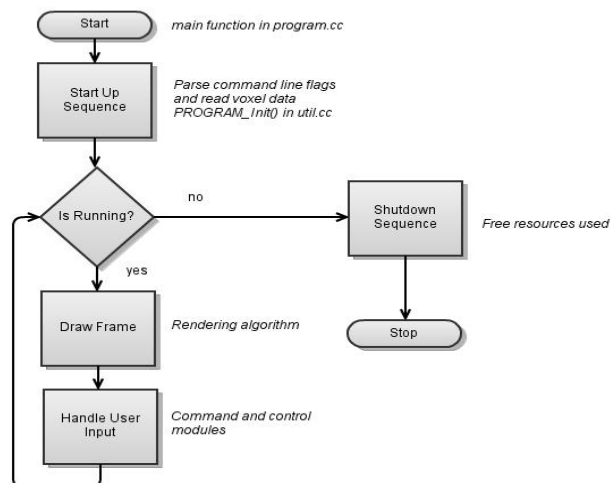


Figure 5.1: High-level overview of control flow in the voxel viewing system

The actual implementation has this high-level control flow built into the main-function of the system. A sample of this source code can be seen in Figure 5.2.

```

1 int main(int argc, char *argv[]) {
2     viewer::PROGRAM_Init(argc, argv);
3     while (viewer::PROGRAM_IsRunning()) {
4         viewer::PROGRAM_DrawFrame();
5         viewer::PROGRAM_HandleUserInput();
6         viewer::PROGRAM_WaitForFrame();
7     }
8     viewer::PROGRAM_Close();
9     return 0;
10 }

```

Figure 5.2: Point of entry 'main function' of the voxel viewing system. This can be found in program.cc in the full source code.

All control flow in the system stems from this point. This means that all the following diagrams represent sub-components in the high-level diagram shown in Figure 5.1.

### Start up sequence

During the start-up sequence, command-line flags are parsed and voxel data read into the system from files specified with the `--files` flag. Figure 5.3 shows how the system does this. Initially the *util* module does the work of parsing flags. Then work transfers to the *world* which reads voxel data from file and then manages it. Note that there is 1 byte set for every voxel and if no voxel data is read then the system exits.

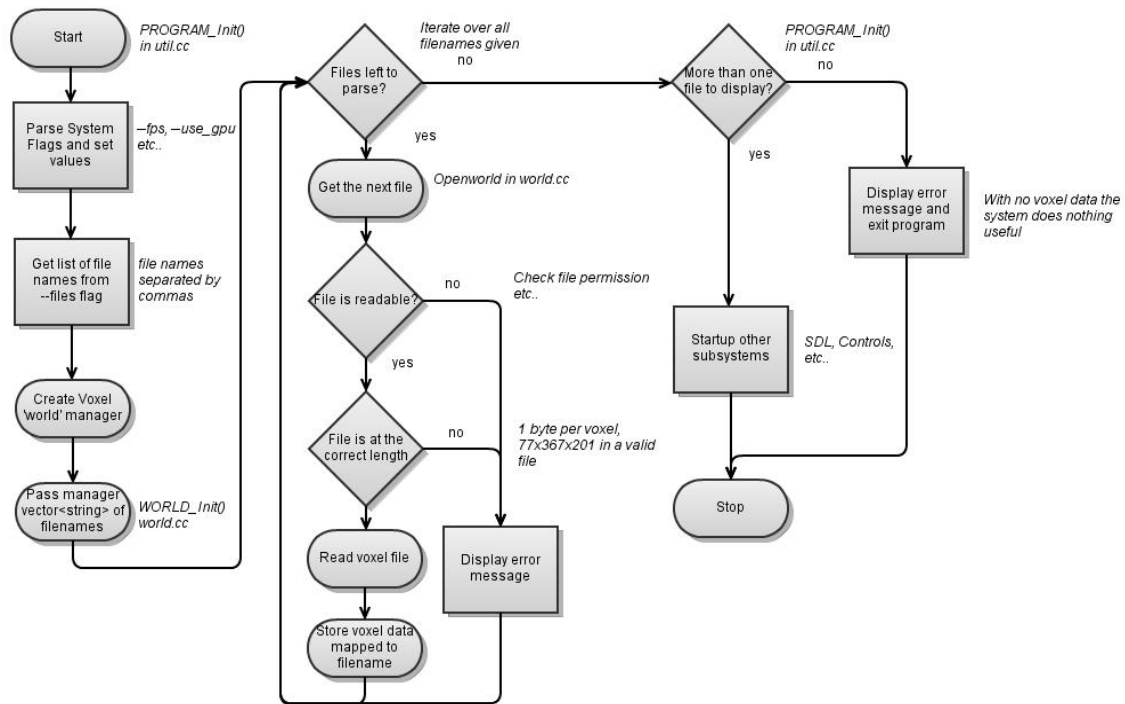


Figure 5.3: Flow diagram showing the start-up sequence of the voxel viewing system focusing mainly on how voxel data files are read into the system

As mention above, the implementation of this diagram is mostly in the *world* module which can be found in the *world.{h,cc}* files. When files are being accessed they are tested for readability, existence and size. A mapping is also made to the originating file-name allowing for a helpful message to be displayed when the voxel data changes so the user can see its origins. Once the data is loaded, it is possible for the GPU or CPU renderers to produce their images.

### CPU Renderer

The final voxel viewing system includes a both a GPU and CPU renderer, both performing the same work in parallel and serial respectively. The CPU renderer begins by allocating memory to store the output image to display later. Looping over every pixel, rays are generated and using the Fast Voxel Traversal Algorithm they traverse the voxel space.

Once a ray finds an active voxel, a colour is written to the output image. If the ray leaves the volume then black is written. After a value for every pixel has been calculated, the image is displayed on the screen using OpenGL. This process is shown pictorially in Figure 5.4.

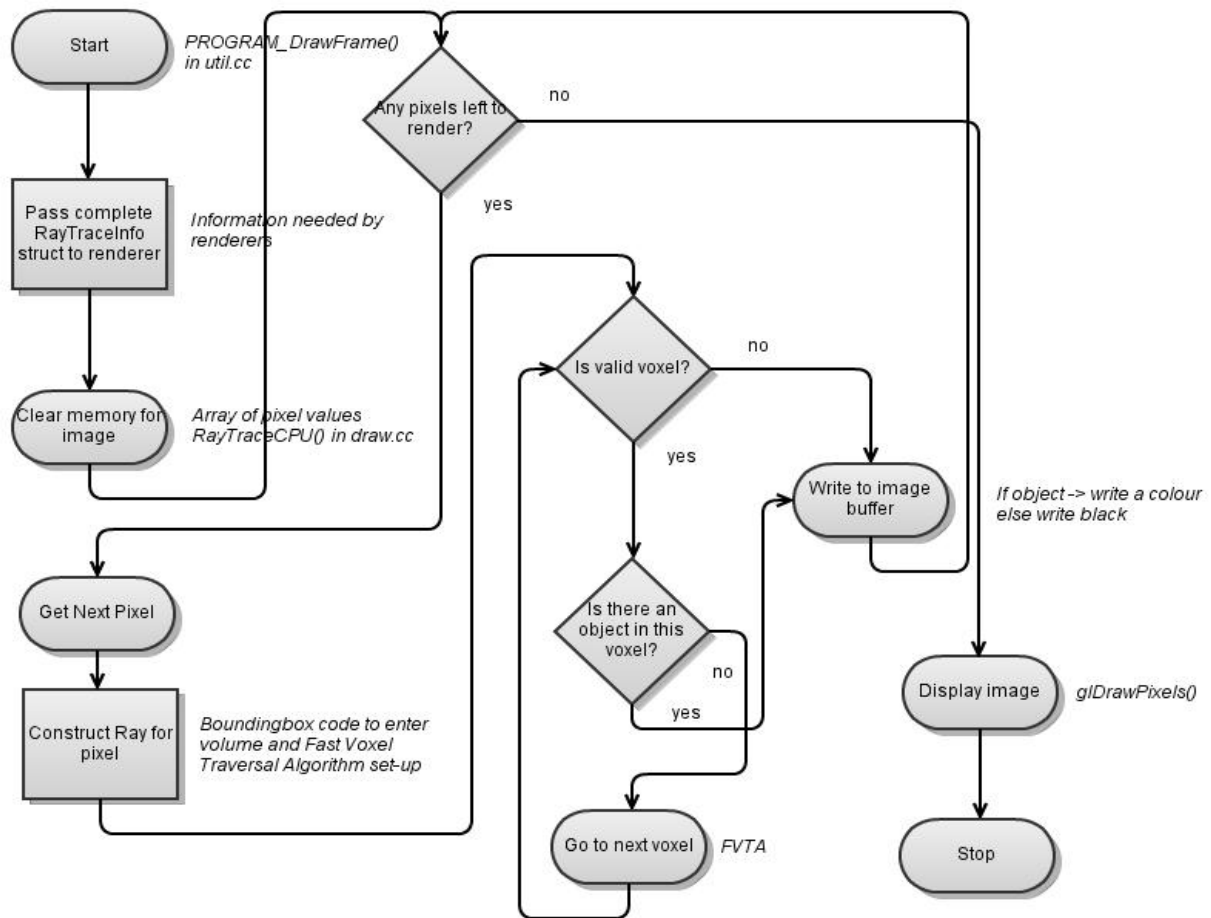


Figure 5.4: Flow diagram showing how the CPU ray tracer works

Please note that the implementation uses traditional Ray Tracing techniques to allow the user to move the virtual camera outside the data volume so that they can view the voxel data from any position. The process that allows for this is described later in this section.

## GPU Renderer

Aside from the parallelism, the GPU renderer is simpler than the CPU equivalent. The process begins by passing allocating memory on the GPU into which parameters will be passed as the output image will written into. This image buffer is shared between OpenGL and CUDA which helps to reduce memory traffic. The ray trace algorithm is then run for each pixel in parallel before the resulting image is displayed. This high-level process is shown on Figure 5.5.



## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

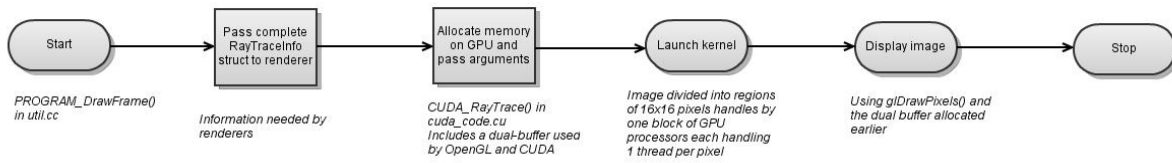


Figure 5.5: GPU render start-up phase, showing memory being allocated on the GPU and parameters passed before the kernel is launched in multiple threads to render the image

The kernel itself follows a process similar to the inner loop of the CPU renderer. Unfortunately, lots of code had to be duplicated because of how CUDA works so there is little shared functionality between the two renderers.

In the kernel executing on a GPU, the renderer first maps the thread to a pixel in the image. Then constructs the ray going through this pixel on the virtual screen and traverses it through the voxel data until either it hits an object or leaves the data volume when a colour is written. This is done for all pixels in the image in parallel. The control flow of the kernel is shown on Figure 5.6.

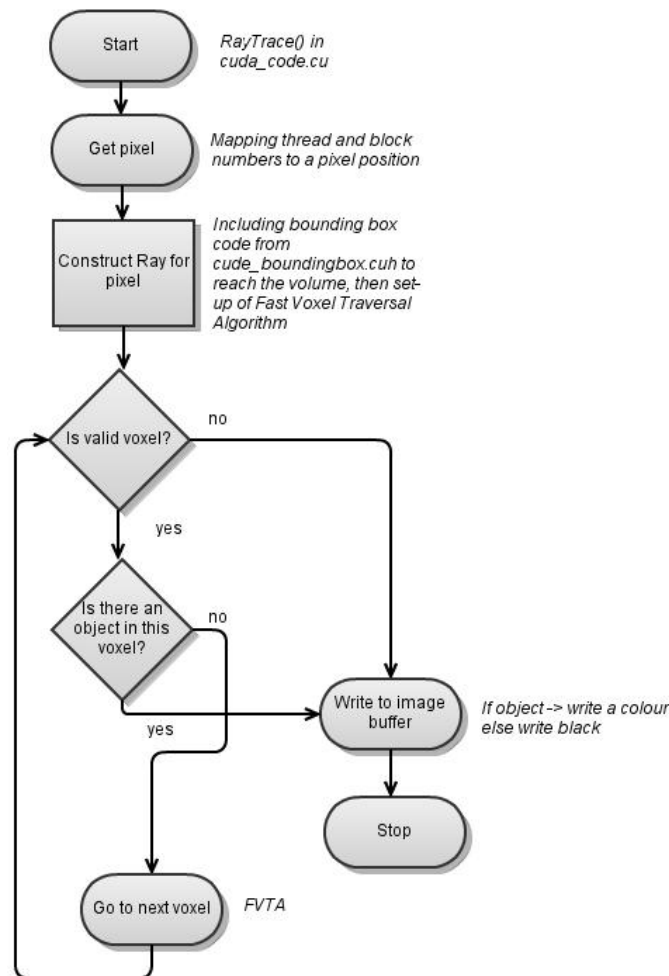


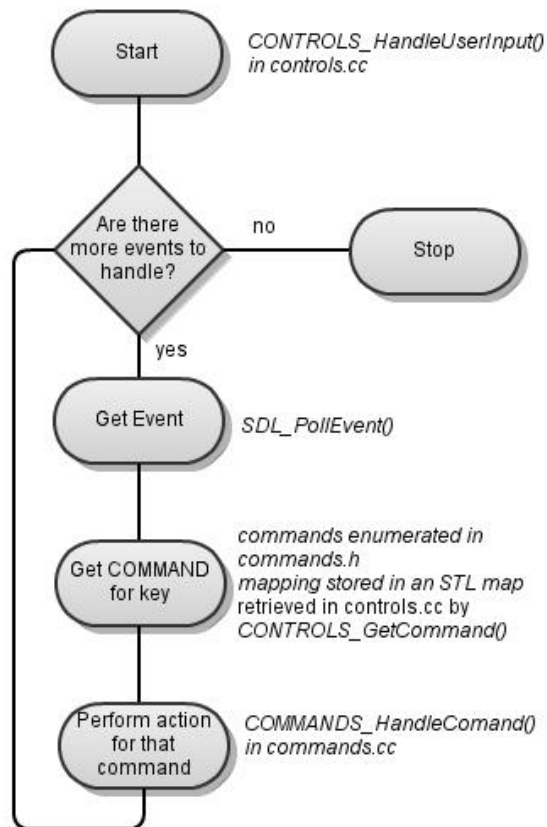
Figure 5.6: Control flow of the kernel function performing the ray tracing algorithm to produce a graphical output

The implementation of the GPU renderer can be found in *cuda\_code.cu*. Like the CPU renderer, it also uses traditional ray tracing techniques to allow for more camera movement in a process described later in this section.

### Handling of user inputs

The ability to handle user inputs fulfils the requirement of allowing users to view the voxel data from any angle or position. This is done by adjusting the position/orientation of the virtual camera. The process has been generalised to allow for other controls/commands such as printing help and could be extended in the future to allow for screen-shot capture and other desirable features.

There are 2 phases: converting user events into commands and then performing actions based on these commands. This is done until there are no more user events (in the current queue) to handle. The high level control flow is shown in Figure 5.7.



*Figure 5.7: Control flow for the processing of user input events*

Commands are enumerated in the file *commands.h* and then given a default binding in *controls.cc*. This binding is stored as key-value pairs in an STL map object to enable fast look-up. Once a valid command is found, functions in *commands.cc* perform the action(s) associated with it.

### Ray-Data Volume intersection

One of the key requirements is to allow the user flexibility in choosing the position from which the representation of the voxel data is generated from. This means that they should be allowed to move the camera out-side the space spanned by the voxels read from file (the data volume).

However, to do this an extra calculation must take place to find where camera Rays intersect the data volume and this becomes their origin rather than the point at which they intersect the virtual screen. This process is similar to how Ray tracing with (graphical) primitive shapes takes place.

The process treats the volume as a box made by 6 intersecting planes. Using the backface-culling technique, faces of the box are eliminated from the search if the Ray won't cross through them. If all faces are eliminated then the ray never enters the data volume a background colour is written. Otherwise, the intersection point with each plane is found for the ray. It is tested to see if its valid on that side of the box by summing the angles to each vertex which should form a complete circle. The valid intersection point is then used as the place where the ray 'begins' in the real rendering process. Pseudo code to describe this process can be seen below.

```

1 Ray {
2   vec3 loc
3   vec3 dir
4 }
5
6 // Finds where the Ray enters the data volume
7 func RayEntry(Ray ray) :
8   vec3 PT, P0, TDIR, NORMAL, output
9   float tmin = INFINITY
10  for side in box :
11    P0 = side.vert(0)
12    NORMAL = side.normal;
13    float t = (dot(P0, NORMAL) - dot(Ray.loc, NORMAL)) / dot(dir, NORMAL)
14    if (t >= 0) :
15      PT = Ray.loc + t*Ray.dir
16      if IsValidOnSide(PT, side) :
17        if (t < tmin) :
18          output = PT
19          tmin = t
20  Ray.loc = output

```

*Figure 5.8: Pseudo code of the bounding box algorithm used to find when rays enter the data volume*

The implementation of the process can be found in *boundingbox.{h, cc}* for the CPU renderer. The GPU renderer required complete duplication in order for the code (especially constant arrays) to be recognised by the CUDA compiler. Its implementation can be found in *cuda\_boundingbox.cuh*.

## **6 Project Management**

With a project this size many problems, conflicts of interest, time constraints and other issues were always going to arise. This section discusses how well the project was managed by examining the time spent and showing how different problems were solved.

### **6.1 Time Spend Management**

A total of 400 working hours were set aside for this project to be used over 7 months whilst other work needed to be done. This meant it was crucial to effectively plan and organise the work done. This section discusses this time management thought out the project, often referencing to Gantt charts (made at key points in the project) which can be found in Appendix B on page 44.

#### **Original planning**

Despite being a rather experimental project, background reading allowed for the main stages and milestones to be organised and a plan was made early in the project. These stages are showing in Appendix\_B 1 and described below:

- Background Reading : finding solutions to the problem
- Experimentation : very basic prototyping and testing
- Project Progress Report : doing the first major documentation
- CPU Ray Tracer design and development : building a solution running on CPU hardware
- GPU Ray Tracer design and development : building a solution running on GPU hardware
- Final Documentation : writing this report and other documentation
- Integration : building the code made for this project into the existing pipeline

The original plan also included contingencies to help deal with problems. Some of the contingencies are listed in Appendix F.

#### **Time spend at the progress report**

The project progress report marked a rough half way point and gave an opportunity to review the project time management. The following paragraphs describe the work done and attempt to compare it to the original plan.

The 3 weeks originally set aside for background reading was a good prediction. However, reading wouldn't just take place at the beginning of the project but would continue throughout albeit in a more limited manor. For example reading took place before the writing of the progress report which focused the CUDA API.

The experimentation time set in the original project plan was actually used to develop a prototype ray tracer. This had similar functionality to what had been expected to be built after the progress report deadline and much of the work work done could be used in the final solution.

When comparing the original Gantt chart (Appendix\_B 1) and the mid-point Gantt chart (Appendix\_B 2) it can be seen that they are similar. This part of the project went according to the plans made, with main difference that the experimentation took the form the ASCII Art Ray Tracer

and that extra background reading took place. Nevertheless the original predictions for this stage of the project were correct.

### **Planning after the progress report**

As part of the progress report, a plan made for the next major stage of the project leading up to this document. Appendix\_B 3 shows this in Gantt Chart form with the stages being described in more detail in the paragraphs below.

Like the original plan, a break was scheduled for the Christmas vacation and Semester 1 exams. This was to take account for the importance of revision and the time it would draw away from the project. However, there was an expectation that some limited CUDA reading and experimentation would take place in this time.

Due to the CPU ASCII Art Ray Tracer being developed 3 weeks early, more time could be used to produce the GPU version and it was hoped that testing could take place before the final report was written.

However, this plan did not describe the stages of development for the GPU solution, making it very vague. This was mainly due to the experimental nature of the project, meaning it was unclear how the final system would be reached so a detailed plan was hard to develop.

### **Time spend at the final report**

Because of this vagueness, the work done after the progress report varied from the plan as there was much more complexity than had been shown. This meant that more prototypes were developed to help to build understanding and test concepts; rather than directly building the full GPU solution.

The work done during this time can be seen in the final time spend Gantt chart in Appendix\_B 4. The following paragraphs describe the stages shown and why this part of the project varied so much from the planning made.

According to the plan, the next stage of the project was to develop a GPU version of the ASCII Art Ray Tracer. Unfortunately, this involved re-writing large amounts of code in order for it to be accessed from CUDA wasting much time, although this was code was brought forward to the final renderer.

After this, time was spent investigating a distortion problem which occurred with rotating the virtual camera. Solving this involved analysing a MATLAB implementation[24] of the Fast Voxel Traversal Algorithm and re-writing the initialisation stage the renderer.

Next the CPU renderer was integrated into an existing SDL display framework. This was not planned for and was not strictly a GPU implementation issue. However, it served as a useful prototype and allowed for the modularisation of the renderer so that it could be changed easily.

Further work helped ensure code quality through the use of command-line flags, de-coupling and enforcement of the code style guide. Again, this was not part of the plan but it did help to ensure the quality of the final solution as the work was re-used later.

The final stage involved integrating the GPU renderer with the SDL display code in order to produce the final solution. Due to earlier prototypes and the de-coupling work done, the changes were limited to the rendering code only. However, many problems were encountered some of which are discussed in section 6.2.

### **Working Habit**

For the duration of the project, there was a day in each where no lectures took place. This gave 8 clear working hours (9:00-17:00) to dedicate to the project. This was supplemented through work done on campus (using a laptop computer) and in the Easter vacation. Also, 30minute meetings have taken place most weeks with the project supervisor Dr J Carter.

### **Time Management Conclusions**

Despite being an experimental project which was hard to plan for, good short term time management and a disciplined working habit meant that a full solution was developed within the overall time budget.

Better medium term planning may have improved this. The plan made as part of the progress report proved to be far from actual work done. Perhaps a different methodology for planning would have been useful here, focusing on 2-week feature-based schedules (as in Xtreme Programming) rather than a full 'up-front' plan (as in the Waterfall model).

Nevertheless, work was done at a steady rate and problems overcome effectively. This lead to a full solution being developed within the overall time budget with a remainder which would be spent on improving the software quality and completing documentation to a higher standard.

## **6.2 Problem Solving during Development**

As was expected, the development stage of the project encountered many problems although none of these threatened the project. Most were solved with some creative thinking or research but they did cause a nuisance and diverted time away from developing the final solution. The following section describes some of these problems and how they were managed.

### **SVN Backup failure**

From the beginning of the project the ugforge[25] SVN service was use to provide version control and backup. However, after the Christmas 2010 no further commits could take place due to a problem with permissions. This was not solved by creating different accounts and administrators failed to respond to emails asking them to investigate.

Fearing that work would be lost, a switch was made to use DropBox[26]: a cloud-based system aimed at individuals who wish to keep files synchronized across multiple computers.

Although not specifically aimed at developers, it worked well as a replacement. Backup copies and old versions of files were kept online and could be accessed through a web-interface. Its file-system integration allowed for easy access to work without the complexities of SVN.

However, for commercial or group work it would not be suitable as it lacks some of the key features of a good version control system such as change-lists and messages. It also synchronizes whenever a file was saved rather than when a piece of work was finished, which would cause chaos when there are multiple developers changing files. Nevertheless for an individual project its lightweight interface, speed and automatic backup are ideal.

### **Distortion when rotating the camera**

After creating the initial ASCII ray tracer visual inspections showed a flaw in the implementation.

This caused a distortion in the output when the view point was rotated, giving the affect of zooming on certain regions but not on others. This proved to be difficult to be debug and took some time to resolve.

The error was caused during in the initialisation of the Fast Voxel Traversal Algorithm[13]. Specifically, incorrect voxel boundaries were being used to calculate the  $tMax\{X,Y,Z\}$  values. This meant that rays were travelling at an incorrect rate. The problem was resolved by examining a MATLAB implementation of the algorithm[24] then re-writing the renderer.

Fixing this problem did take a significant amount of time away from developing the final solution as shown on the final Gantt chart (Appendix\_B 4). However, if this problem had not been solved the graphical output would have been incorrect and a key requirement missed.

### Old versions of CUDA and Floating Point Numbers

The CUDA API has evolved over a number of years and older hardware does not always match with the latest drivers of tools. Notably the target architecture for the project (Geforce 9600 GPUs) does not support IEEE double precision floating point numbers, yet this is that the NVCC CUDA Compiler defaults built-in functions and literals to.

This lead to mysterious run-time errors rather than compile-time errors. After posting the problem on the Nvidia forums[17][18], a solution was found. However, this involved re-factoring almost every line of code so that all floating point literals were appended with  $f$  and CUDA built-in maths functions had to be forced into single precision mode i.e.  $\sin f(x)$  instead of  $\sin(x)$ .

### Confusion over programming languages

During the early stages of the project there was some confusion about whether the existing pipeline was programming in C++ or C. Since C does not include direct Object Orientation support, it became even harder to design the architecture for the viewer. Also it was difficult to find good unit-testing and command-line flag libraries written in with pure C.

Later it emerged that the pipeline was programmed using C++ but in a C/Procedural manner. This lead to some code being modified to take advantage of namespaces, singleton objects, STL, C++ casting and strings. Support for command-line flags was then added using the Gflags library from Google[19]. Because C++ supports the C programming language, all previous work could be re-used so the confusion did not affect functionality or the time spend of the project.

### Combining CUDA-compiled code with C++-compiled code

Although the nvcc compiler can handle normal C++ code (by delegating to another compiler) it rejected some of the flags and options needed to use the SDL library. Therefore, the build process was forced to use g++ and nvcc to create 'object' files and then link them together.

Background reading suggested using a header file or class to wrap around CUDA so that C++ code could be hidden from the API. Then a .cu file is written to implement these functions. All the code is then compiled into object files which are then linked together using a CUDA compiler.

This proved to be hard to implement, especially when setting compiler flags for libraries. However, after several attempts a working solution was found. An example of this is shown in the *Makefile* below.

## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

```
1 OBJECTS = program.o draw.o cude_code.o
2 SDLFLAGS = -I/usr/local/include/SDL -D REENTRANT -L/usr/local/lib -lSDL\
3 -lpthread
4 EXECNAME = myprogram
5 voxelviewer : $(OBJECTS)
6     nvcc $(OBJECTS) -o $(EXECNAME) -lm -lGL $(SDLFLAGS)
7 program.o : program.cc draw.h
8     g++ -c program.cc $(SDLFLAGS)
9 draw.o : draw.cc draw.h
10    g++ -c draw.cc
11 cuda_code.o : cuda_code.cu cuda_code.h cuda_mathlib.cuh
12    nvcc -c cudacode.cu -arch sm_11
```

*Figure 6.1: An example Makefile for combining C++ and CUDA code when using the SDL library*

### CUDA OpenGL Interoperability

Once GPU renderer had been integrated with the SDL display and user control code, it was possible to render and image using the GPU and display it using OpenGL meeting the requirements.

However, this was done using a fake screen buffer on the GPU, copying this back to CPU-controlled RAM and then displaying using the OpenGL command *glDrawPixels()* which copied the data back to the GPU. This created too much memory traffic and reduced the frame rate.

Background reading showed that it was possible to use CUDA OpenGL Interoperability[15] to share memory on the GPU and avoid this copying process. Using some example code[27] as a basis, this was introduced to the system. However, a segmentation fault was caused and this became hard to debug as there were no OpenGL or CUDA errors being shown.

Further investigation and testing revealed that the error was caused by the initialisation order. This meant that OpenGL had to be fully initialised before calling any CUDA-GL code. After the code was modified, the render worked properly and gained a speed-up by avoiding memory transfer.



## **7 Critical Evaluation**

Now that the project has ended, it is useful to review its achievements and highlight the lessons learned. This section does that by examining different stages of the project for effectiveness; checking that goals and requirements have been met, evaluating the quality of the software produced and analysing the project management.

### **Background Reading**

Background reading and research (described in section 2) mainly took place at the beginning of the project but continued through out. Most of the material found did not impact on the project as it did not apply to the scenario apart from the Fast Voxel Traversal Algorithm[13]. This algorithm proved to be very useful to the implementation of the viewer. However, a longer or more thorough search may have revealed other material which could have been of use especially once the implementation phase began.

### **Design Choices**

The design of the system was not formally planned or developed. This was due to the experimental nature of the project and the iterative development method used. Instead prototypes were built and combined together to achieve functionality. Using a formal design method, such as UML diagrams, may have lead to a better quality implementation but this did not suite the nature of the project as there were many unknown components.

The mathematical design of the system, through the use of parallelism and the Fast Voxel Traversal Algorithm, proved to be more successful. It was well thought out and planned for. The implementation used GPU threads to allow the slow Ray Tracing algorithm to be ran in real time in order to produce the graphical display, giving the acceleration expected.

### **Implementation meeting requirements**

The goal of this project was to produce a system which could provide a graphical representation of voxel data generated from Gait experiments at Southampton University. As can be seen by the screen-shots in Appendix C, this has been achieved. Also all the formal requirements of the software produced (listed in section 1) have been met. This is shown in Appendix E which details exactly how each requirement was met.

### **Implementation Quality**

As has already been discussed, there was no formal design for the implementation and this may have lowered the quality of the software produced. However, good engineering principles such as de-coupling and prototyping were adhered to in the development phase. Checks were also put into the software to deal with error cases such as wrong voxel data file sizes.

Because there was an expectation that the system would be integrated with the existing pipeline, C++ namespaces were used to help reduce conflicts with the existing code. Also the Google Code Style Guide[28] was used to produce consistent source code.

Despite only being required to run using Linux, the software would take little work to be ported to different operating systems such Windows. This is because the key libraries (SDL, OpenGL and

CUDA) are all platform independent and the project does not have any Linux-Specific dependencies.

A negative point is that the software has not been tested thoroughly. As prototypes were developed, their functionality was tested informally by inspecting verbose outputs or operating user controls. However, there was no formal unit testing or testing plan. Test-driven development would have improved the quality of the software produced and problems (such as the incorrect initialisation of the FVTA) could have been solved quicker. However, the experimental nature of the project meant that the functionality or existence of many components was unknown so tests would have to be frequently adapted to take account of changes. This would have meant a bigger development time spend and may not have been as beneficial to the project as a whole.

### **Project Management**

Time management for the project was good, it was finished on time with all the requirements met. However, the long term planning was no effective as there were many hidden complexities. Perhaps a different planning methodology would have helped here. Nevertheless, contingencies and good short term planning helped to deal with job interviews, examinations and other activities which took time away from the project. Also, problems were often solved quickly through the use prototypes and background reading.

### **Conclusion**

This project has reached its goal of producing a graphical representation of voxel data. GPU hardware has been used to perform this task in parallel so that the renderer can operate in real time. The software produced meets all its requirement and is in a good position to be integrated into the existing processing pipeline for the Gait research tunnel.

The software engineering aspects of the project could have been improved through using formal design methods and by better testing. This may have helped when problem solving and debugging. However a working solution was still produced.

The project was also managed well, aside from long term planning. Problems were dealt with swiftly and effectively. Work was carried out in a disciplined manor on a week-by-week basis, with extra time being dedicated if necessary. Perhaps a different work scheduling methodology would have helped to organise the project better. Nevertheless all the necessary work was completed within the original 400 hour time budget.

Overall the project can be judged as successful. All the goals and requirements were met within the time budget. Planning, design and testing could have been improved; a lesson for future project work perhaps. Nevertheless a working system was successfully produced as can be seen in several videos on the CD accompanying this document and will be demonstrated further at the viva event.

## 8 **Future Work**

Having shown how a working solution was produced, this section of the report focuses on future work. The immediate work is to integrate the software with the existing pipeline and Gait research tunnel. Other work could also be done in the future, perhaps as part of another project, to optimise the software and to add features such as a full GUI to make the system easier to use.

### **Full integration**

A goal of this project was to create a system which could easily integrate with the existing Gait processing pipeline. Full integration would allow users to view subjects in real-time whilst they are still within the tunnel.

To do this, modifications must be made to the existing pipeline code and tests done with the associated hardware. Previous extensions[29] have proven to be difficult to integrate, therefore, this work will be done after completing this document and before the project viva/demonstration.

To integrate, the high-level control flow will be modified so that it polls for new data in memory from the rest of the pipeline – managing as its made available and transfer it to the GPU as appropriate. There may be a need to delete old voxel data at this point. The viewing system should run on a separate (CPU) thread so that it does not interrupt other work being done. However, aside from a different management process, the system will act in the same way. The proposed control flow can be seen in Appendix\_A 6.

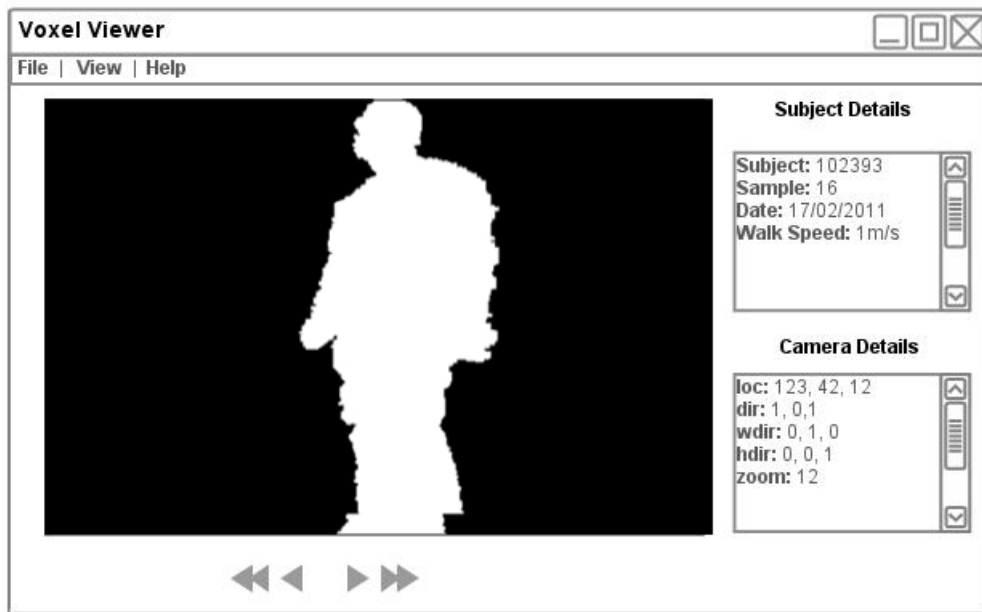
### **Optimisation**

Although the viewer meets the current requirements, these may change in the future and require the system to be optimised. For example, the pipeline could be improved to produce higher resolution voxel data. This means the amount of data needed to be considered in the ray tracing operation would be much larger, using more GPU memory and taking a longer time to traverse. One way to solve this problem could be to use a data-structure to store voxel data.

For example an Octree could be used. These combine voxels with the same value into single nodes which could be used to compress the amount of memory used. Octrees were suggested by various materials during the background reading phase[11] but there was no clear way of representing the tree using GPU hardware. However, later versions of CUDA include kernel support for *malloc()* and *free()* operations which could be used to create the data-structure directly through a kernel. However, a downside to Octrees and other complex data-structures is that the time complexity of testing voxels could increase due to the traversal process.

### **Full GUI**

Another improvement could involve developing a full GUI for the voxel viewer or the full pipeline. This might be easier to use than a command-line based system. Components could be used to display information more clearly to users, such as information about different measurements of attributes of the subject. It could also allow for researchers to load different voxel files or connect to different pipelines without closing the viewer program. As basic prototype for a GUI is shown in Figure 8.1.



*Figure 8.1: A prototype GUI for the voxel viewing system which could be used to convey more information to the user*

As part of this GUI, Mouse Look[30] (aka free-look) could be used to improve the interaction between the user and the system. This allows the user to use the mouse rather than the keyboard to change the camera orientation, giving more precise control. It is a common technique implemented in 3D applications such as computer games. However, it is a complex process hence why it wasn't implemented in this project.

## References

- [1] M.Nixon, Automatic Gait Recognition for Human ID at a Distance, 2003, <http://www.gait.ecs.soton.ac.uk/>
- [2] Anon, OpenGL - The Industry Standard for High Performance Graphics, 2011, <http://www.opengl.org/>
- [3] Anon, Learn about DirectX - Games For Windows, , <http://www.gamesforwindows.com/en-US/directx/>
- [4] Anon, OpenGL Shading Language, 2011, <http://www.opengl.org/documentation/glsl/>
- [5] Anon., What is CUDA?, 2011, [http://www.nvidia.com/object/what\\_is\\_cuda\\_new.html](http://www.nvidia.com/object/what_is_cuda_new.html)
- [6] Anon, OpenCL, 2011, <http://www.khronos.org/opencl/>
- [7] E.Kandrot and J.Sanders, CUDA by Example: An Introduction to General-Purpose GPU Programming, 2010
- [8] Nvidia Corporation, Nvidia CUDA Programming Guide, 2006-2010, [http://developer.download.nvidia.com/compute/cuda/3\\_0/toolkit/docs/NVIDIA\\_CUDA\\_ProgrammingGuide.pdf](http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf)
- [9] R.Johnson, Workshop on Automatic Tuning: GPU Computing with CUDA, 2007, <http://cscads.rice.edu/workshops/july2007/autotune-slides-07/RichardJohson.pdf>
- [10] D.Kirk, CUDA Programming Mode, 2006-2008, <http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf>
- [11] R.Shroud, PC Perspective - John Carmack on id Tech 6, Ray Tracing, Consoles, Physics and more, 2010, <http://www.pceper.com/article.php?aid=532>
- [12] J.Gunther et al, Realtime Ray Tracing on GPU with BVH-based Packet Traversal, 2007
- [13] J.Amanatides and A.Woo, A Fast Voxel Traversal Algorithm for Ray Tracing,
- [14] Anon., Tutorial1: Creating a Cross Platform OpenGL 3.2 Context in SDL (C / SDL) - OpenGL.org, 2011, [http://www.opengl.org/wiki/Tutorial1:\\_Creating\\_a\\_Cross\\_Platform\\_OpenGL\\_3.2\\_Context\\_in\\_SDL\\_\(C\\_/SDL\)](http://www.opengl.org/wiki/Tutorial1:_Creating_a_Cross_Platform_OpenGL_3.2_Context_in_SDL_(C_/SDL))
- [15] Nvidia Corporation, NVIDIA CUDA Library: OpenGL Interoperability, 3022, [http://developer.download.nvidia.com/compute/cuda/2\\_3/toolkit/docs/online/group\\_\\_CUDART\\_\\_OPENGL.html](http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/group__CUDART__OPENGL.html)
- [16] Anon., OpenGL API Documentation, 2011, <http://www.opengl.org/documentation/>
- [17] Nvidia Corporation, NVIDIA Forums -> CUDA GPU Computing, 2011, <http://forums.nvidia.com/index.php?s=47b76ff39e9702cb0a785475a20cdc0f&showforum=62>
- [18] Nvidia Corporation, NVIDIA Developer Forums -> Parallel Nsight for Graphics Developers, 2011, <http://developer.nvidia.com/forums/index.php?s=92dc04a5fc03166c75abc818d16473a5&showforum=45>
- [19] Google Inc. & Open Source Community, google-gflags - Project Hosting on Google Code, 2011, <http://code.google.com/p/google-gflags/>
- [20] Google, googletest - Google C++ Testing Framework - Google Project Hosting, 2011, <http://code.google.com/p/googletest/>
- [21] A.Klöckner, PyCUDA | Andreas Klöckner's web page, 2011, <http://mathematician.de/software/pycuda>
- [22] Anon., Python Programming Language - Official Website, 2011, <http://www.python.org/>
- [23] iDSoftware, Koders Code Search: mathlib.h - C - GPL, 2005, <http://www.koders.com/c/fid9CC0B4D22CC450C1B3038DAA8884821C0A56591E.aspx>
- [24] Jesús P. Mena-Chalco, A fast voxel traversal algorithm for ray tracing, 2010,

### 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

<http://www.mathworks.com/matlabcentral/fileexchange/26852-a-fast-voxel-traversal-algorithm-for-ray-tracing>

[25] Anon., ECS : Students Projects Forge, 2011, <https://forge.ecs.soton.ac.uk/>

[26] Dropbox, Dropbox - Online backup, file sync, and sharing made easy., 2011, <https://www.dropbox.com/>

[27] R.Farber, CUDA, Supercomputing for the Masses, 2010, <http://drdobbs.com/architecture-and-design/222600097>

[28] G.Eitzmann et al , Google C++ Style Guide, 2011, <http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml>

[29] C.Spicer, Real time 3D reconstruction for gait recognition using graphics processing hardware, 2010,

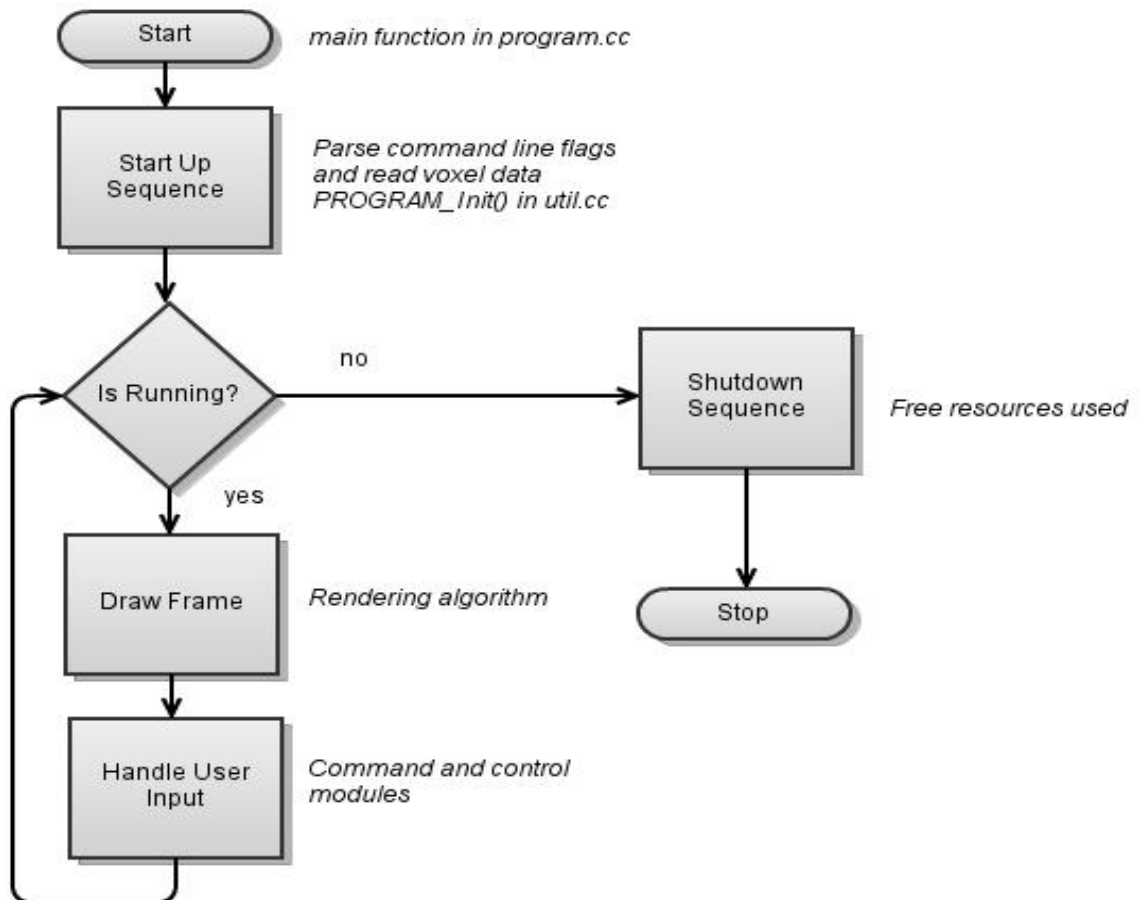
[30] Anon., Free look, 2011, [http://en.wikipedia.org/wiki/Free\\_look](http://en.wikipedia.org/wiki/Free_look)

## Glossary of Terms

Active Voxel	A voxel which a subject (being analysed) has entered
Block	Smaller group of threads which can cooperate
CPU	Central Processing Unit which is the hardware most computer programs use
CUDA	Compute Unified Device Architecture – an Nvidia API for interacting with GPU hardware to perform general computation
Data-volume	The volume of the tunnel being described by the voxel data this system receives
FVTA Fast Voxel Traversal Algorithm	Algorithm used to traverse rays through the voxel data in order to produce a graphical output
Gait	A description of a person's walk
GPU	Graphics Processing Unit, the hardware used by this project
Grid	Describes a group of threads which execute a kernel program on GPU hardware
Kernel	Similar to graphics shaders: short subroutines or programs written in a C like language which are executed in parallel on the GPU
OpenGL	A standard specification defining a cross-language, cross-platform API or writing applications that produce 2D and 3D computer graphics
Ray Tracing	A technique for generating an image by tracing the path of light through pixels and simulating the effects of its encounters with virtual objects
Research Tunnel	Researchers at Southampton University have built this tunnel to help them study Gait
SDL	Simple DirectMedia Layer a cross-platform multimedia library designed to provide low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video Frame-buffer
Thread	Works which execute on multi-threaded Streaming Multiprocessor cores to run kernels
Voxel	Volumetric pixel. Describes a small volume in the 3D space. Often used in the visualisation of medical and scientific data.

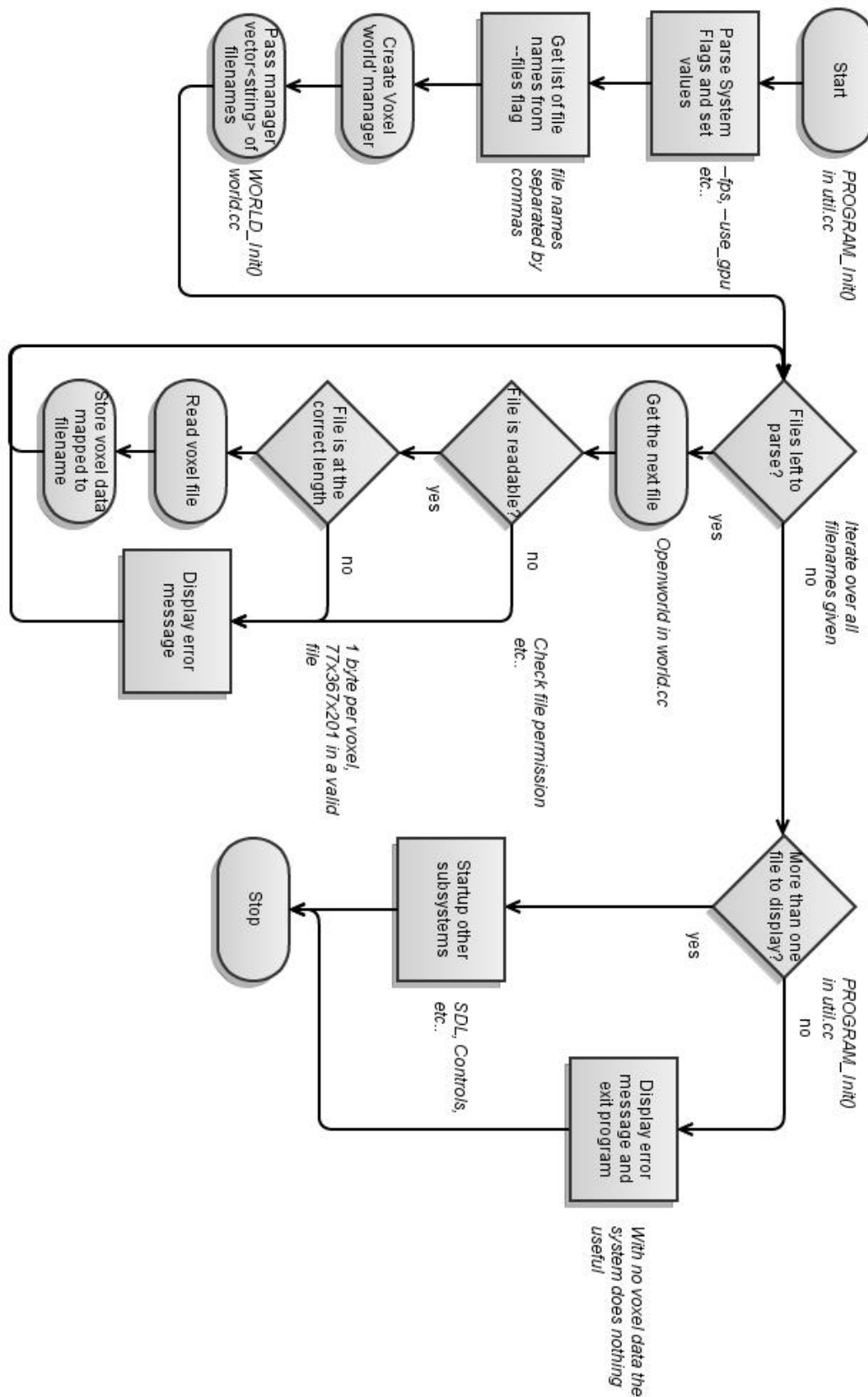
## Appendix A : Control Flow Diagrams

*Various diagrams to show the control flow throughout the voxel viewing system.*

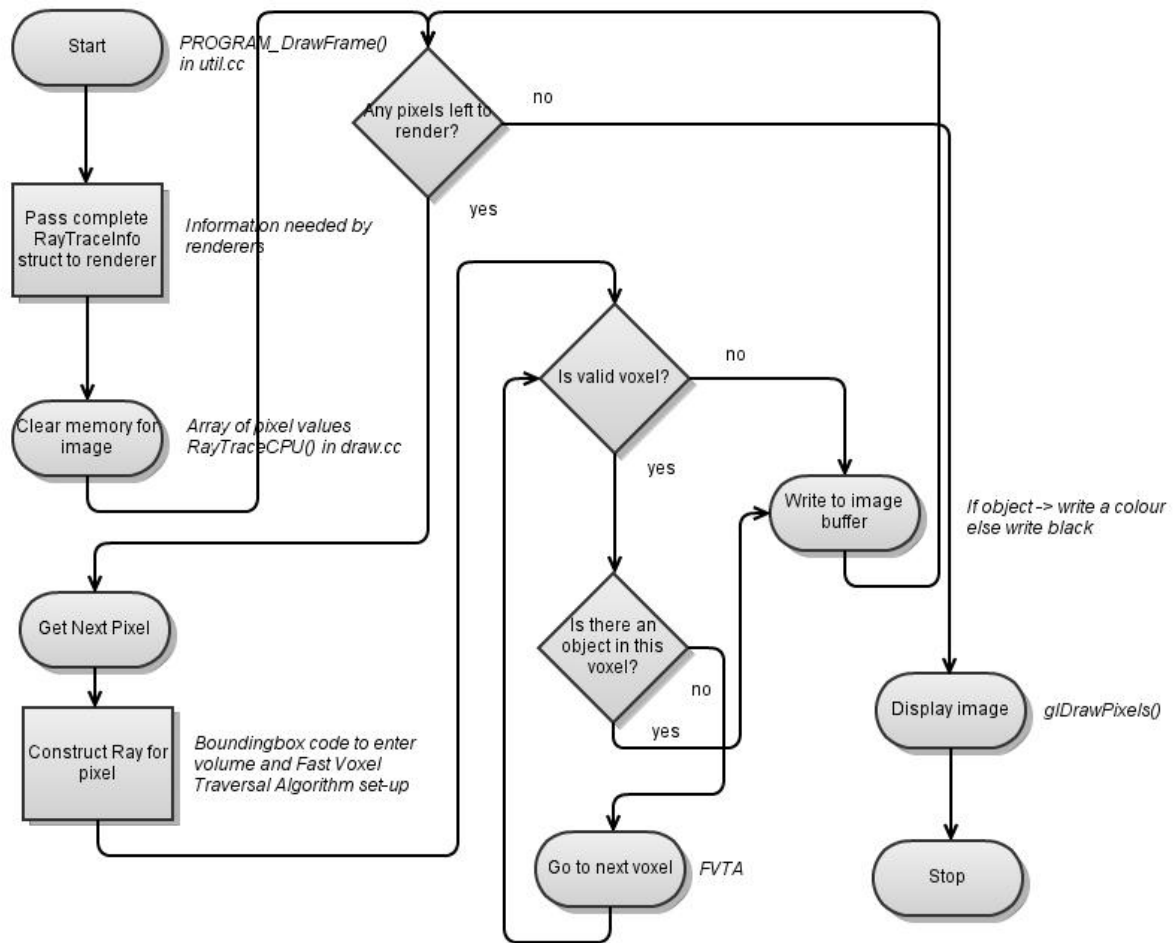


*Appendix\_A 1: High level control flow of the voxel viewing system*



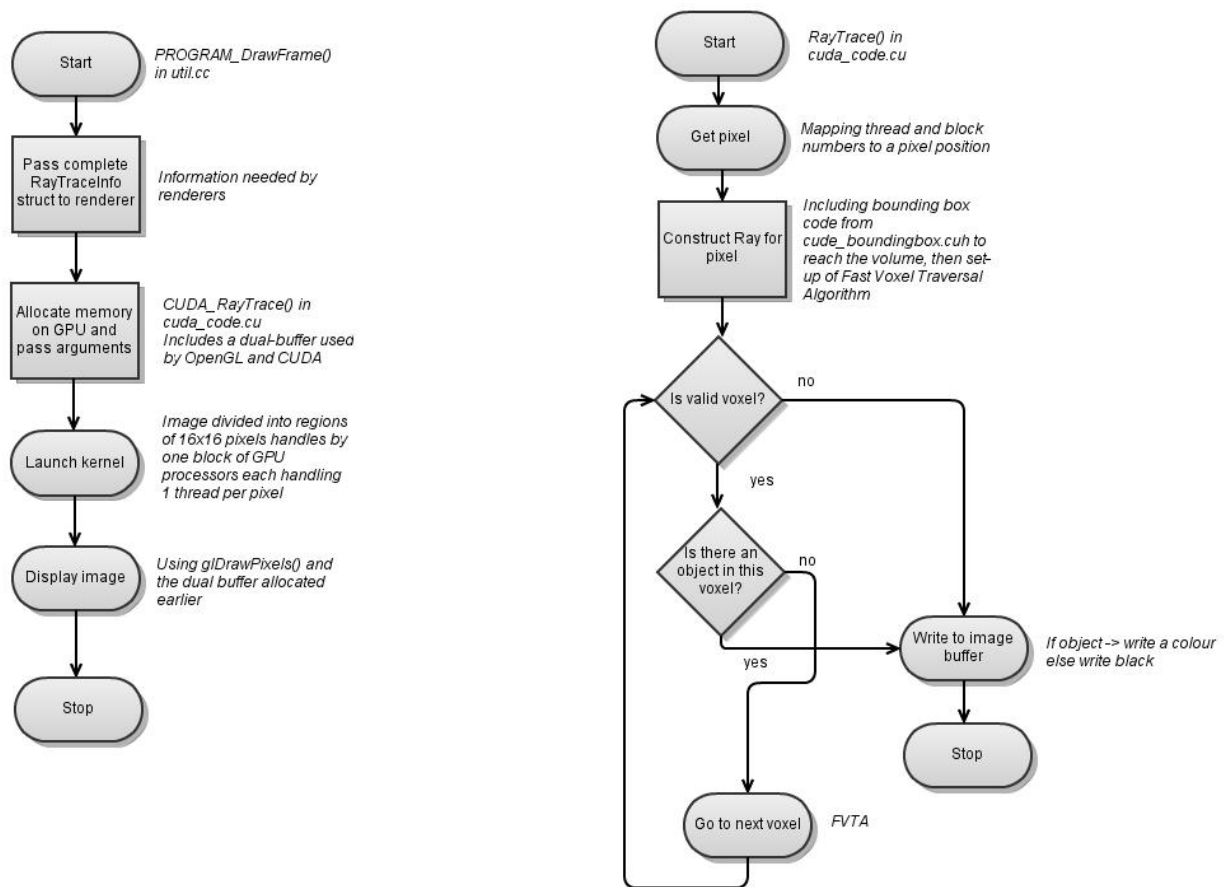


Appendix\_A 2: Control flow when the system begins, including the reading of voxel data from file

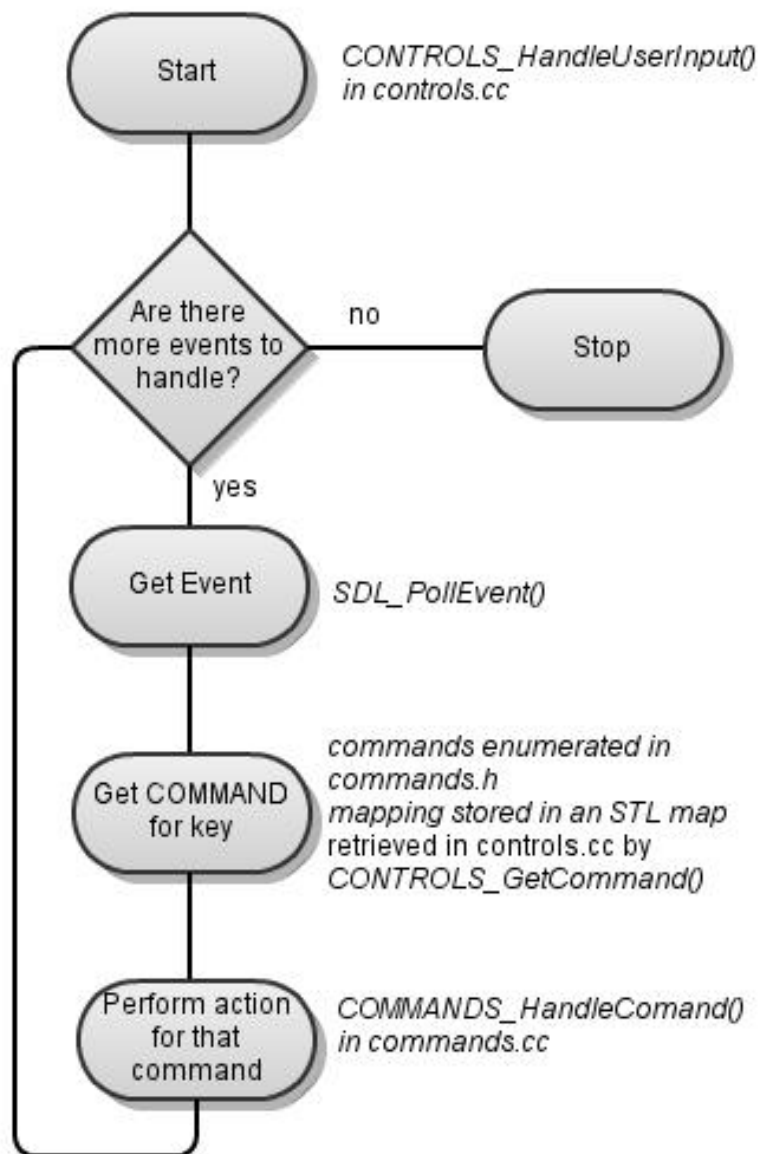


Appendix\_A 3: Control flow of the CPU Ray Tracer

## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

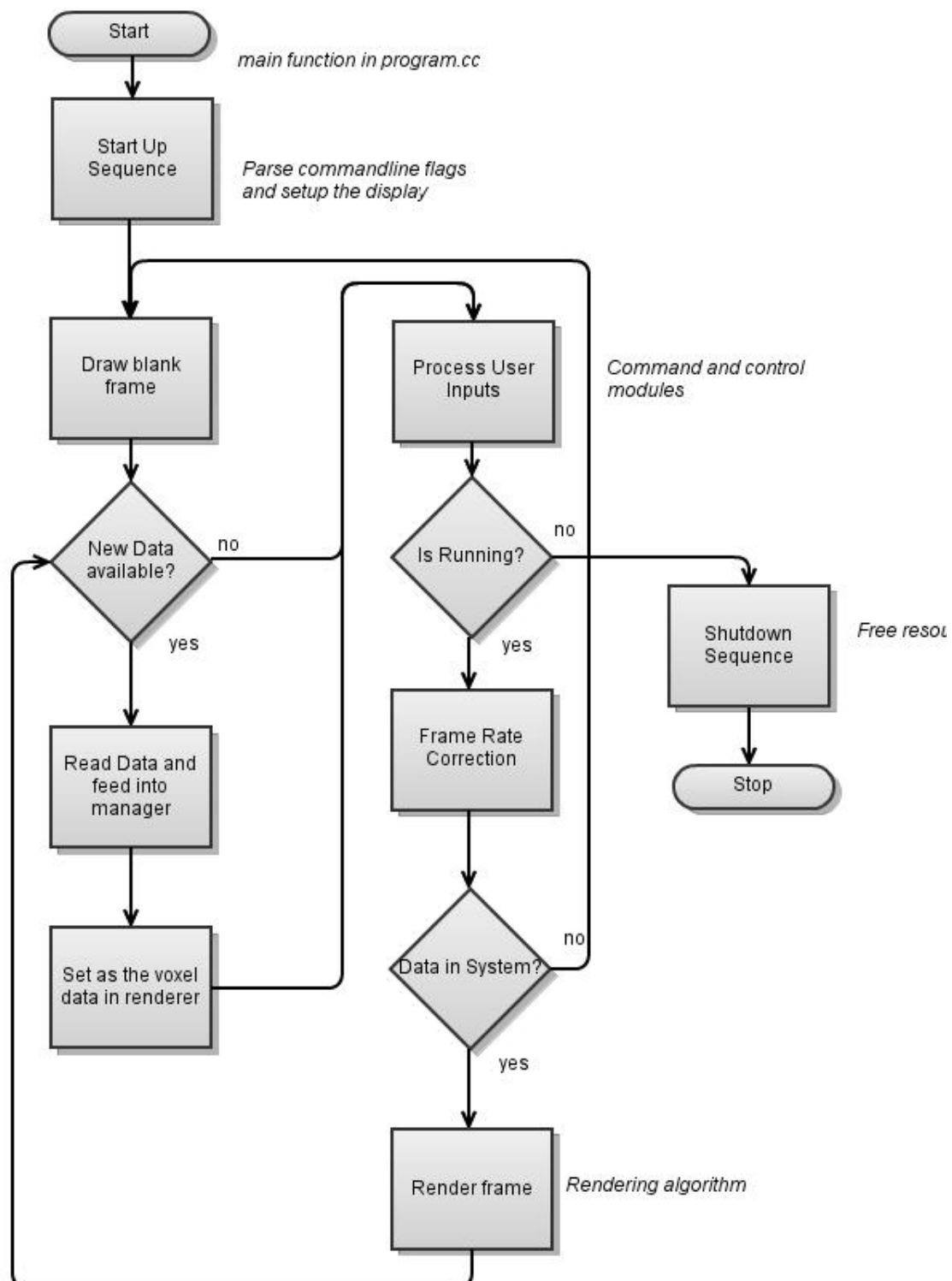


*Appendix\_A 4: GPU render start-up phase and kernel execution, showing memory being allocated on the GPU and parameters passed before the kernel is launched in multiple threads to render the image*



*Appendix\_A 5: Control flow for the processing of user input events*

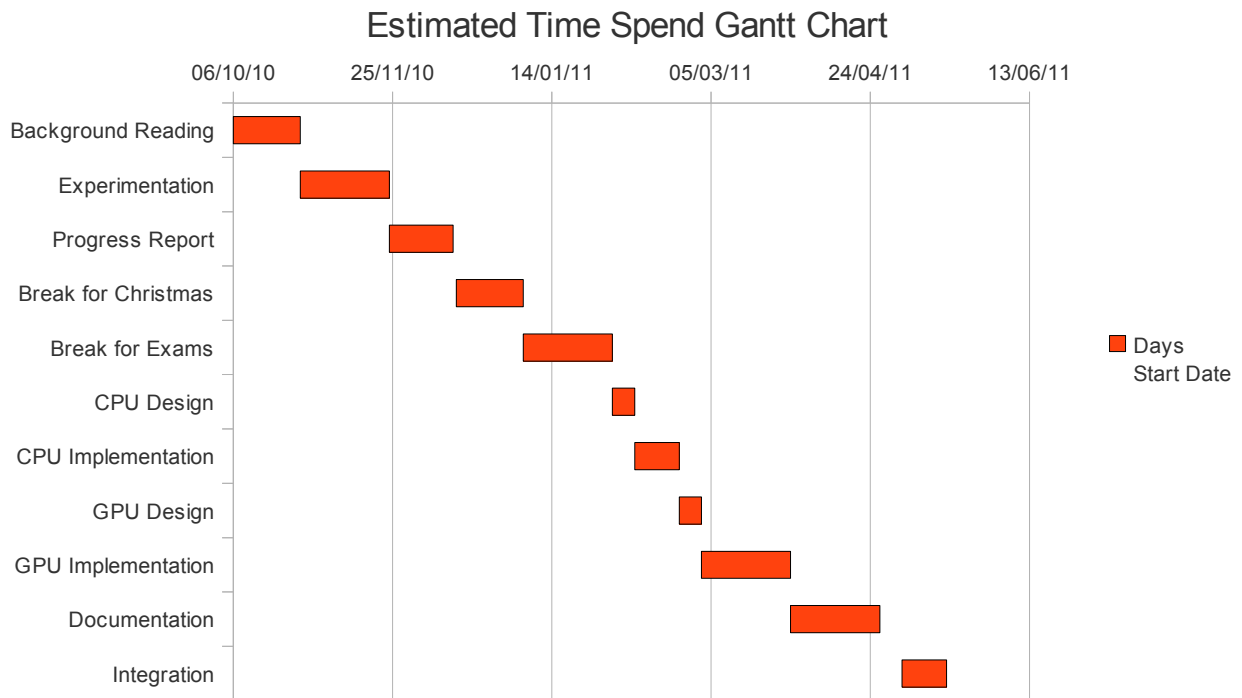
## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report



*Appendix\_A 6: The high level control flow for when the voxel viewing system is integrated with the existing pipeline*

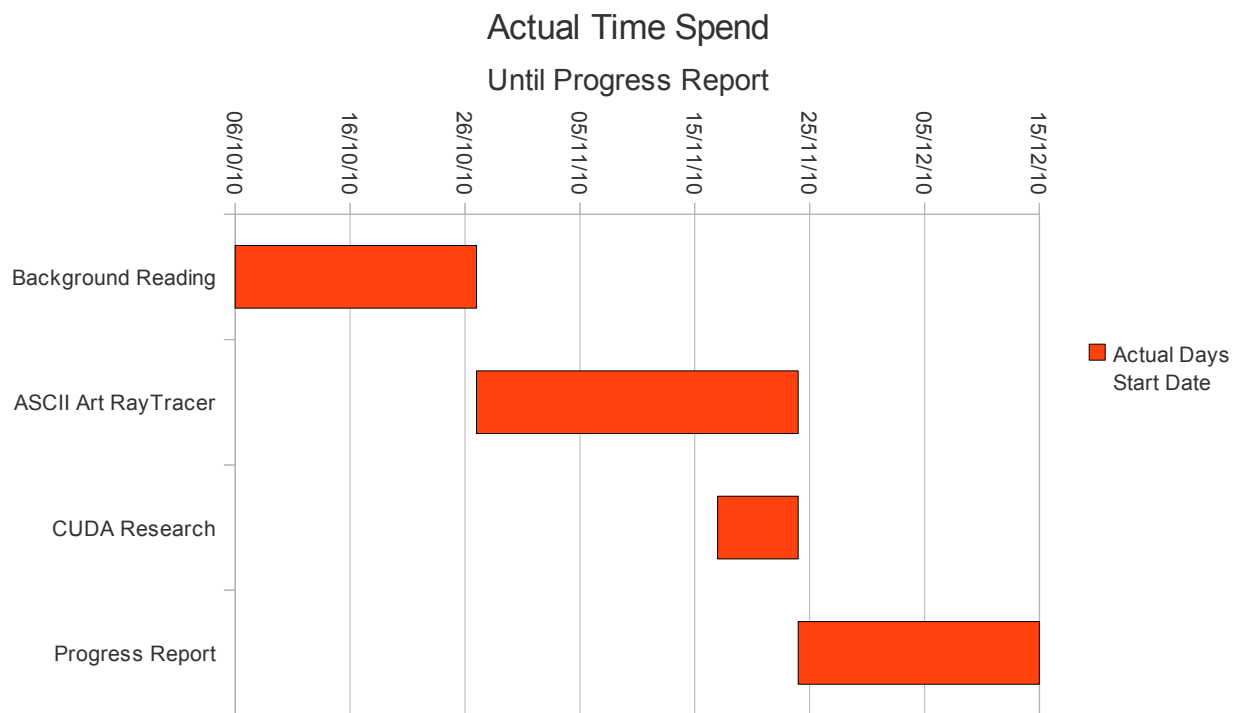
## Appendix B : Gantt Charts

*Gantt charts showing the time breakdown and project management of the project.*



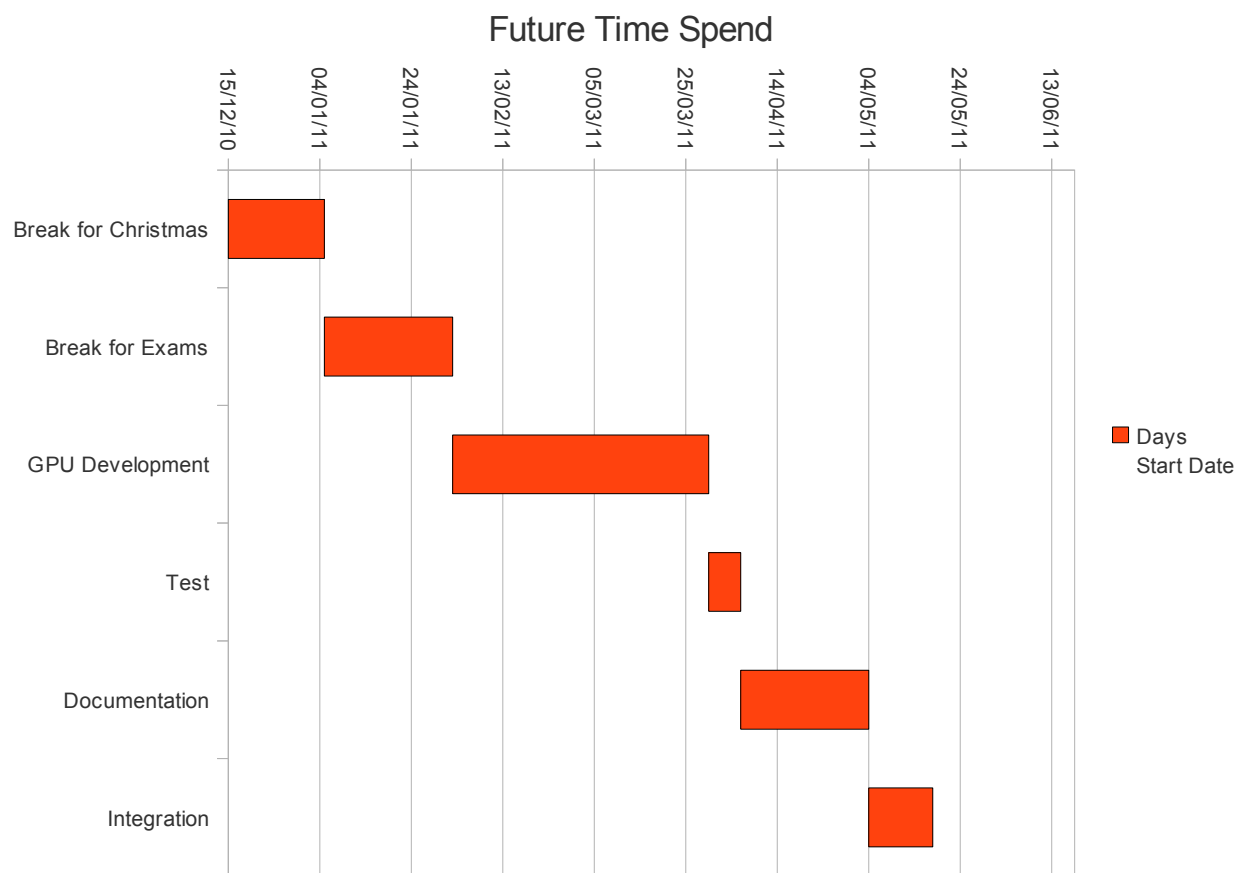
*Appendix\_B 1: Original estimate of time spend on the project*

## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report



*Appendix\_B 2: Time breakdown at the progress report hand-in stage*

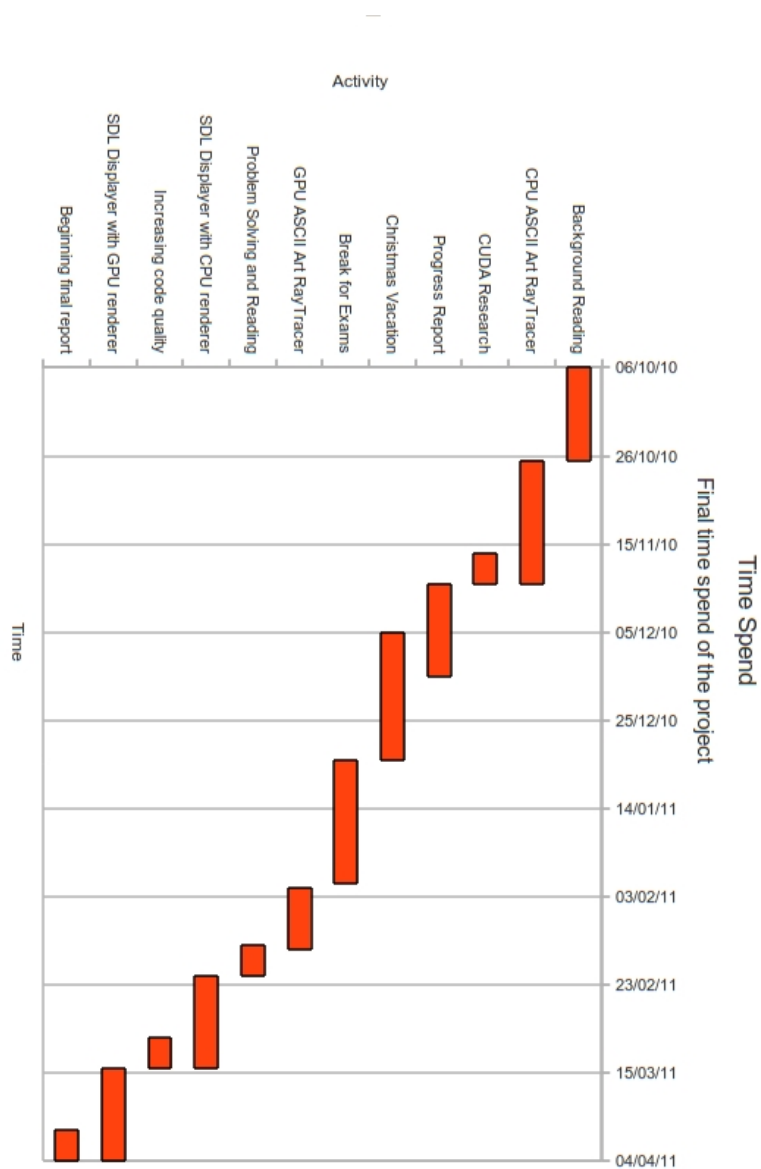
## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report



*Appendix\_B 3: Prediction of time spend between the progress report and the end of the project*



## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report



*Appendix\_B 4: Final time spend on the project from start to end*

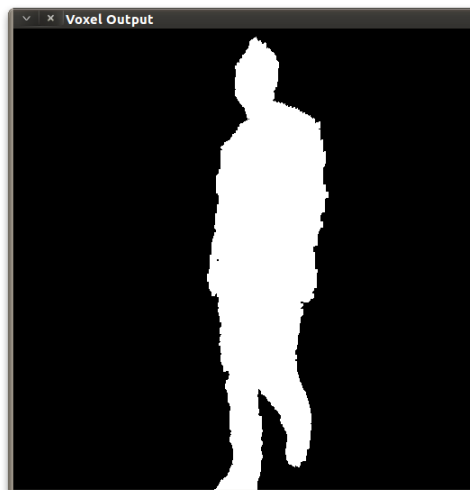
# 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

## Appendix C : Screen-shots

*The following appendix contains various screen-shots demonstrating the voxel viewing system working*

```
ethr@ethr-ubuntu64-desktop:~/Dropbox/3rd year project/vtoge/display_gpu$ make
g++ -std=c++0x -Wall -c mathlib.cc
g++ -std=c++0x -Wall -c util.cc -I/usr/local/include/SDL -D_REENTRANT -L/usr/local/lib -lSDL -lpthread
g++ -std=c++0x -Wall -c program.cc -I/usr/local/include/SDL -D_REENTRANT -L/usr/local/lib -lSDL -lpthread
g++ -std=c++0x -Wall -c draw.cc
g++ -std=c++0x -Wall -c controls.cc -I/usr/local/include/SDL -D_REENTRANT -L/usr/local/lib -lSDL -lpthread
g++ -std=c++0x -Wall -c commands.cc -I/usr/local/include/SDL -D_REENTRANT -L/usr/local/lib -lSDL -lpthread
g++ -std=c++0x -Wall -c camera.cc
g++ -std=c++0x -Wall -c ray.cc
g++ -std=c++0x -Wall -c voxel_translation.cc
g++ -std=c++0x -Wall -c world.cc
g++ -std=c++0x -Wall -c gflags/gflags.cc
g++ -std=c++0x -Wall -c gflags/gflags_reporting.cc
g++ -std=c++0x -Wall -c gflags/gflags_completions.cc
g++ -std=c++0x -Wall -c boundingbox.cc
nvcc -c cuda_code.cu -arch sm_11
nvcc mathlib.o util.o program.o draw.o controls.o commands.o camera.o ray.o voxel_translation.o world.o gflags.o gflags_reporting.o gflags_completions.o boundingbox.o cuda_code.o -o voxelviewer -lm -lGL -I/usr/local/include/SDL -D_REENTRANT -L/usr/local/lib -lSDL -lpthread
ethr@ethr-ubuntu64-desktop:~/Dropbox/3rd year project/vtoge/display_gpu$
```

```
ethr@ethr-ubuntu64-desktop:~/Dropbox/3rd year project/vtoge/display_gpu$ ./voxelviewer --files=data/voxel_0054.bin,data/voxel_0055.bin,data/voxel_0056.bin,data/voxel_0057.bin,data/voxel_0058.bin,data/voxel_0059.bin,data/voxel_0060.bin,data/voxel_0062.bin,data/voxel_0063.bin,data/voxel_0064.bin --pixelw=512 --pixelh=512
[world.cc:54] 10 worlds loaded
Initializing touch...
[cuda_code.cu:214] ID of current CUDA device: 0
[cuda_code.cu:233] CUDA Init complete
Showing file: data/voxel_0054.bin
□
```

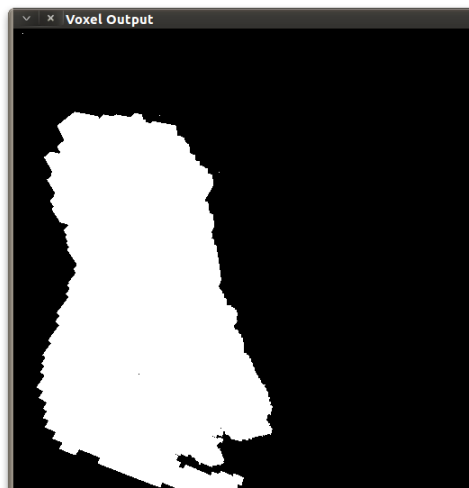


## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

```
ethr@ethr-ubuntu64-desktop:~/Dropbox/3rd year project/vtoge/display_gpu$ ./voxelviewer --files=data/voxel_0054.bin,data/voxel_0055.bin,data/voxel_0056.bin,data/voxel_0057.bin,ta/voxel_0058.bin,data/voxel_0059.bin,data/voxel_0060.bin,data/voxel_0062.bin,data/voxel_0063.bin,data/voxel_0064.bin --pixelw=512 --pixelh=512
[world.cc:54] 10 worlds loaded
Initializing touch...
[cuda_code.cu:214] ID of current CUDA device: 0
[cuda_code.cu:233] CUDA Init complete
Showing file: data/voxel_0054.bin
Showing file: data/voxel_0055.bin
Showing file: data/voxel_0056.bin
[]
```



```
ethr@ethr-ubuntu64-desktop:~/Dropbox/3rd year project/vtoge/display_gpu$ ./voxelviewer --files=data/voxel_0054.bin,data/voxel_0055.bin,data/voxel_0056.bin,data/voxel_0057.bin,ta/voxel_0058.bin,data/voxel_0059.bin,data/voxel_0060.bin,data/voxel_0062.bin,data/voxel_0063.bin,data/voxel_0064.bin --pixelw=512 --pixelh=512
[world.cc:54] 10 worlds loaded
Initializing touch...
[cuda_code.cu:214] ID of current CUDA device: 0
[cuda_code.cu:233] CUDA Init complete
Showing file: data/voxel_0054.bin
Showing file: data/voxel_0055.bin
Showing file: data/voxel_0056.bin
Showing file: data/voxel_0057.bin
Showing file: data/voxel_0058.bin
[]
```

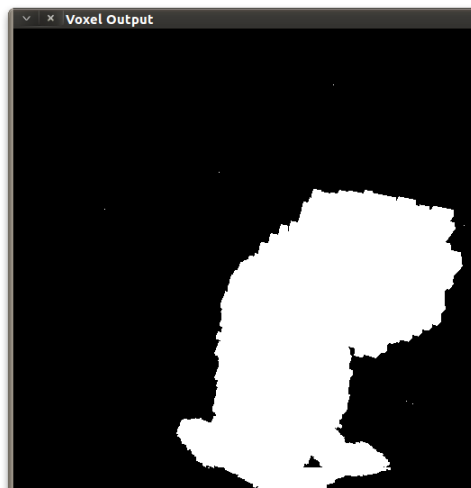


## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

```
ethr@ethr-ubuntu64-desktop:~/Dropbox/3rd year project/vtoge/display_gpu$ ./voxelviewer --files=data/voxel_0054.bin,data/voxel_0055.bin,data/voxel_0056.bin,data/voxel_0057.bin,ta/voxel_0058.bin,data/voxel_0059.bin,data/voxel_0060.bin,data/voxel_0062.bin,data/voxel_0063.bin,data/voxel_0064.bin --pixelw=512 --pixelh=512
[world.cc:54] 10 worlds loaded
Initializing touch...
[cuda_code.cu:214] ID of current CUDA device: 0
[cuda_code.cu:233] CUDA Init complete
Showing file: data/voxel_0054.bin
Showing file: data/voxel_0055.bin
Showing file: data/voxel_0056.bin
Showing file: data/voxel_0057.bin
Showing file: data/voxel_0058.bin
Showing file: data/voxel_0059.bin
[]
```

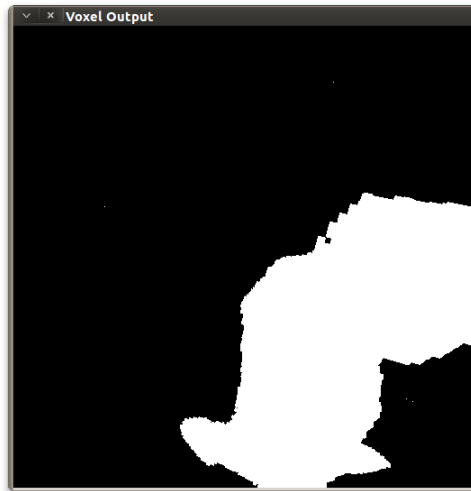


```
ethr@ethr-ubuntu64-desktop:~/Dropbox/3rd year project/vtoge/display_gpu$ ./voxelviewer --files=data/voxel_0054.bin,data/voxel_0055.bin,data/voxel_0056.bin,data/voxel_0057.bin,ta/voxel_0058.bin,data/voxel_0059.bin,data/voxel_0060.bin,data/voxel_0062.bin,data/voxel_0063.bin,data/voxel_0064.bin --pixelw=512 --pixelh=512
[world.cc:54] 10 worlds loaded
Initializing touch...
[cuda_code.cu:214] ID of current CUDA device: 0
[cuda_code.cu:233] CUDA Init complete
Showing file: data/voxel_0054.bin
Showing file: data/voxel_0055.bin
Showing file: data/voxel_0056.bin
Showing file: data/voxel_0057.bin
Showing file: data/voxel_0058.bin
Showing file: data/voxel_0059.bin
Showing file: data/voxel_0060.bin
Showing file: data/voxel_0062.bin
Showing file: data/voxel_0063.bin
[]
```



## 3D Graphical Output of Voxel Data from Gait Experiments - Final Report

```
ethr@ethr-ubuntu64-desktop:~/Dropbox/3rd year project/vtoge/display_gpu$ ./voxelviewer --files=data/voxel_0054.bin,data/voxel_0055.bin,data/voxel_0056.bin,data/voxel_0057.bin,ta/voxel_0058.bin,data/voxel_0059.bin,data/voxel_0060.bin,data/voxel_0062.bin,data/voxel_0063.bin,data/voxel_0064.bin --pixelw=512 --pixelh=512
[world.cc:54] 10 worlds loaded
Initializing touch...
[cuda_code.cu:214] ID of current CUDA device: 0
[cuda_code.cu:233] CUDA Init complete
Showing file: data/voxel_0054.bin
Showing file: data/voxel_0055.bin
Showing file: data/voxel_0056.bin
Showing file: data/voxel_0057.bin
Showing file: data/voxel_0058.bin
Showing file: data/voxel_0059.bin
Showing file: data/voxel_0060.bin
Showing file: data/voxel_0062.bin
Showing file: data/voxel_0063.bin
Showing file: data/voxel_0064.bin
□
```



## Appendix D : Agreed Project Brief

*The document below is a copy of the 'agreed project brief'. This set out the aims and goals of the project when it first began.*

### **Agreed Project Brief**

<b>Student Name</b>	Joshua England	je7g08@soton.ac.uk
<b>Supervisor Name</b>	John Carter	jnc@soton.ac.uk

### **3D Graphical Output of**

### **Voxel Data from Gait Experiments**

Researchers at Southampton University have been analysing the biometric information about how people walk to see if it is possible to identify a person by their Gait. By triangulating data from 12 cameras they are able to produce voxel data to describe in 3D how a person is walking. The aim of this project is to use this data to produce a graphical output as part of the analysis pipeline. This must be done in real time to allow for visual analysis of the data and demonstrations.

The technical challenges of this project will involve translating the 3D voxel data into a 2D graphical display in real time. To help with graphics processing, dedicated hardware will be used. This may require any algorithms used to be adapted as graphics hardware is quite different from a traditional CPU.

The most likely technique for producing the image will be a simplified ray tracing algorithm, however, this has traditionally been unsuited to graphics hardware. Some optimisations and efficiencies may have to be incorporated into the program to allow for an acceptable image to be generated in an acceptable time. For example an Octree could be used to organise the voxel data in a more structured way.

This project is an extension to an existing research so no data collection will be needed. The process of generating the voxel data has already been established. Instead this project will produce a 3D graphical output near the end of the pipeline. Because of the existing system, Nvidia graphics cards and CUDA technology will be used to produce the image on Linux operated computers.

Depending on time constraints possible extensions would allow the user to view the voxel data at any angle via a 'floating camera', produce better quality images or to provide optimisations in other parts of the pipeline.

## Appendix E : Achievements of Requirements

*The following table describes how the project requirements have been met by the implementation.*

Requirement	How it was achieved
Must be able to read, store and manage voxel data either from files or the existing pipeline	<ul style="list-style-type: none"> <li>• 'world' module for the reading, storing and management of voxel data from file</li> <li>• future work will use this module to manage information from the pipeline</li> </ul>
Use a windowing system display a graphical representation of the subject described by the voxel data	<ul style="list-style-type: none"> <li>• SDL library used to handle windowing operations</li> <li>• Within this window, an OpenGL context is created</li> <li>• Renderers then produce an image which is shown on the window using OpenGL</li> </ul>
Allow the user to interact with the system, to view the scene from different positions or angles	<ul style="list-style-type: none"> <li>• SDL libraries used to receive user inputs</li> <li>• These are mapped to commands in the 'controls' module</li> <li>• The 'commands' module then processes these commands and alters the virtual camera appropriately</li> </ul>
The program must be able to execute on a desktop computer running Linux due to legacy code from the existing pipeline	<ul style="list-style-type: none"> <li>• The system was developed using Ubuntu Linux 10.10 64 bit edition</li> <li>• Uses open source Unix tools such as g++ and make</li> </ul>
The system must operate in real time and produce an image at 30 frames per second...	<ul style="list-style-type: none"> <li>• Using the <code>--show_fps</code> flag, the GPU renderer achieves a 30fps+ frame-rate when rendering a default size voxel data file on a 512x512 pixel window</li> </ul>
Nvidia CUDA is to be used as the API for GPU programming ...	<ul style="list-style-type: none"> <li>• GPU renderer is a CUDA kernel with supporting functions</li> <li>• The building process uses the nvcc CUDA compiler</li> </ul>

## Appendix F : Contingency Planning

*The table below shows some of the contingency planning made to help mitigate problems.*

<b>Contingency</b>	<b>Avoidance</b>	<b>Response</b>
Project Progress Report becoming difficult	Giving 20 days of clear working time to complete it	Increase the per-week working time
CPU implementation becoming troublesome	Designing the experimentation prior to beginning work	Push back GPU implementation into the Easter vacation
GPU implementation becoming troublesome	Experimentation and CPU version as prototypes	Using the Easter vacation to finish the implementation
Difficulty integrating with the pipeline	Doing this work after the final report and main development phase	Demonstrate the system independently, reading voxel data from file, at the viva



## Appendix G : CD Contents

*The following table describes the contents of the disc which is attached to this report. A more detailed description can be found in the CDContents.pdf file on the CD.*

Directory	Description
data/	Example voxel data files, produced from the processing pipeline, which can be used to demonstrate the viewing system. Each file is 3110877 bytes in size, containing 1 byte per data voxel.
documentation/	Contains the main documentation for the project: Doxygen-generated source code support and the formal reports in PDF format.
prototypes/	This directory contains the source code of key prototypes developed as part of the project.
screenshots/	Various screen shots demonstrating the system and prototypes
videos/	Dynamic demonstrations of the system and prototypes developed
voxelviewer/	Full source code of the final system built