Point Cloud Visualisation

Dimitrios Kapnopoulos

August 23, 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2013

Abstract

The aim of this MSc project was to implement the image-based point cloud visualisation algorithm as proposed by Paul Rosenthal and Lars Linsen. This particular algorithm performs all its computation on image space and therefore, its performance does not depend on the input data set. Furthermore, an optimisation was applied to this implementation by decomposing the image among the multiple SMs on the GPU. Every SM had a decomposed part of the image stored on the shared memory and thus, the threads inside the SMs communicate through the shared memory. The performance improvement achieved was two or three times faster depending on the memory accesses required in the original implementation. Finally, a WebGL implementation has been implemented. The architecture of the WebGL implementation is very similar to the OpenGL one but with some significant differences due to the absence of some OpenGL features. The performance of the WebGL and OpenGL implementations is very similar because the core computation is still executed on the GPU.

Contents

List of Figures

Listings

Acknowledgements

I am sincerely grateful to my supervisor, Mr. Malcolm Illingworth, for his supervision for this dissertation.

I would also like to thank my family for supporting me during my studies.

Finally, I would like to thank everyone in EPCC, who have given me an impressive experience of the MSc in High Performance Computing. I would also like to thank Alan Gray for answering questions about GPU architecture.

1 Introduction

There is a need in many fields to visualise real objects. One way to accomplish this is to create those objects using many triangles. However, when accurate detail of objects is needed, utilisation of triangles is not a feasible solution. The last few decades a lot of research has been done in the area of visualisation and interpretation of data coming mostly from 3D scanners. 3D scanners are devices that can produce information about objects such as appearance and colour. They can scan objects either by physical touch or by emitting some kind of radiation or light and determine the shape of the object as well as other properties[1]. After acquiring such information about an object, they have to be visualised or interpreted. Many algorithms exist for visualising such information. Furthermore, as the size of the data produced by the 3D scanner increases, ways for efficient storing and visualisation of this data has to be investigated and thus a lot of research has been done in this area as well. The applications of point cloud visualisation vary. It can be used in movies or video games to visualise real objects or in other areas where high fidelity is needed.

1.1 Motivation

The goal of this dissertation is to investigate point cloud visualisation techniques and implement an image-based algorithm using the Open Graphics Library (OpenGL) as proposed by Paul Rosenthal and Lars Linsen[2]. Point cloud visualisation involves a lot of processing and thus sometimes it is impractical to use it because of the low performance. The reason for choosing the image-based algorithm is because it performs all its computation on image space and that makes it unique and interesting to investigate it. Therefore, after implementing the image-based algorithm, an investigation on its performance is carried out, the bottlenecks are spotted and finally optimisation is applied.

Furthermore, there has been a trend over the last few years in running graphics applications through web browsers using the Web Graphics Library (WebGL). Thus, an implementation in WebGL has been made and a detailed analysis as well as a comparison with the OpenGL implementation is provided.

1.2 Report structure

Initially, background information about point cloud visualisation and applications is presented as well as a brief summary of OpenGL because the image-based algorithm is implemented with this library. In the next chapter, the image-based algorithm is explained in depth and the structure of the implementation is given. Output images to verify the correct behavior of the application are given in addition advantages and disadvantages of the image-based algorithm.

In chapter 4, the performance of the image-based algorithm is analysed using different data sets,

each with different point magnitude. Because both the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU) are used in graphics applications, both the CPU and the GPU code were profiled and the bottlenecks were identified. After having the bottlenecks spotted, an optimisation was proposed and applied. Then, the output of the optimised implementation is shown as well as the performance improvement for different data sets.

The WebGL implementation is discussed in chapter 5, how it was implemented, differences in performance with the OpenGL implementation and possible improvements that can be made regarding the performance in the future.

Finally, In chapter 6, possible future work is proposed for the OpenGL implementation as well as the WebGL one. For the OpenGL is proposed a system with multiple GPUs which could benefit the performance by distributing the points among the GPUs. Furthermore, the utilisation of Octrees as a data structure for storing the points is proposed. For the WebGL implementation, the possibility of using WebCL is explained. Also, a comparison with an implementation made with the Point Cloud Library (PCL) library is proposed.

2 Background

2.1 Point Cloud Visualisation

Many point cloud visualisation algorithms exist. They all have different properties or they might depend on the input data. They can be classified according to the input data. The algorithm might not have any proof of the shape of the object and the only information that they hold is the spatial position through the input data. Alternatively, some algorithms can take into account other information about the object such as breaklines, open and closed surfaces etc. The most common steps in point cloud visualisation algorithms include pre-processing, mesh generation and finally post processing. The pre-processing steps usually apply noise reduction, data sampling and holes filling. The core part of these algorithms is the mesh generation. This step converts the input points to triangles and these points might be exactly the same points as the input or additional points for optimal mesh generation. Finally the post-processing steps correct the generated mesh by splitting or inserting triangles or even reducing them.[3]

All these algorithms perform their computation in object space. They generate a mesh and then they modify it. Paul Rosenthal and Lars Linsen have proposed another algorithm. This algorithm is completely different from the others because it performs all its computation in image space. The image space point cloud visualisation algorithm is the subject of this dissertation and is explained in detail in the next chapters.

The ability of 3D scanners to sample objects with high frequency, produces millions of points and is not feasible for the applications to visualise. This is because of either limited memory or that they visualise them with very low performance. A lot of research has been done on efficient ways of storing those points.

One data structure that is commonly used is the Octree. Octree is a regular hierarchical data structure. The first node is the root. Each node has eight children, which forms a 2x2x2 regular division of the parent node but there is no limit to the number of children. In general it could have N^3 children (in previous case N = 2). With Octrees, not all the points have to be rendered but depending on the view of the object, whole sub-trees can be discarded. However, Octrees seem to occupy more memory because of the pointers to other nodes that have to be stored. [4]

3D scanners can produce various information types, depending on what the application needs. The kind of information needed in point cloud visualisation and computer graphics usually contains the vertex coordinates that represent the position of a point, surface normals and colours. Surface normals are vectors perpendicular to a point or to a surface, and they are used in lighting algorithms. In addition, colours might be included in a file as well as other properties like the material of the vertex.

The information that is produced by the 3D scanners is stored in files. These files contain the coordinates of the points and usually other properties of the points such as surface normals and colours. There are many file formats for storing this information. The most famous formats are the Polygon File Format (PLY) and the OBJ. However, other file formats exist like X3D which uses Extensible Markup Language (XML) to represent 3D geometry. These file formats are used for a wide variety of graphics application from video games to Computer-Aided Design (CAD) programs and not only merely in point cloud visualisation.[5]

The PLY format was developed by the Stanford University. It consists of a header that includes information about how many vertices the file contains, what properties each vertex has (if any) and finally the number of the faces. Faces are lists of vertices expressing which vertices to connect to create polygons. [6]

2.2 Applications

In this section, applications of point cloud visualisation are presented. There is a large range of applications and every application that needs detailed representation of objects can utilise point cloud visualisation techniques.

2.2.1 Medicine

Point cloud visualisation can be extensively used in the field of medicine because there is a demand for visualising different body parts and internal human organs. A 3D visualisation system has been proposed for the case of location and navigation for atrial fibrillation ablation operation. This system mainly consists of an electrode catheter which produces data about the position of the heart and outputs this data to an algorithm. The first step of this algorithm is to reconstruct the surface using the Poisson surface reconstruction algorithm. Then some noise points produced by the operation and breathing are discarding. After this step a model with enough details is produced and by applying additional processing, further information can be collected and parts of the heart can be picked up.[7]

2.2.2 Geology

In many cases when geologists want to investigate the surface of the earth, point cloud visualisation systems have been used. In particular, light detection and ranging (LiDAR) has been employed to collect information about the surface of the Earth. This process involves flying an airplane over the land and LiDAR, which is a scanner, directs laser pulses towards the land and produces topological information. In many cases, geologists want to investigate the land for scarps and examine areas for earthquake risks and thus they use point clouds.[8][9]

2.3 OpenGL

OpenGL is an application programming interface (API) which can access the graphics hardware. It includes more than 500 functions for manipulating the GPU and sending data to it. OpenGL is portable and can run even entirely on software if a GPU is absent. However, it does not include support for reading images from files or describing three dimensional objects. Instead, objects are constructed from primitive geometric types like points and lines.

The latest version of OpenGL (version 4.3), has changed dramatically and has become more flexible giving more control to the programmers. The common way that OpenGL works is:

- Send the data for constructing the objects to the GPU.
- Write shader programs, which are executed on the GPU, to manipulate this data.
- Finally, render the points using the shader programs mentioned above.

2.3.1 The OpenGL pipeline

Every time we render data passed by our program to the GPU, a number of stages are executed known as the OpenGL pipeline. Figure 1 shows the stages that are executed. The orange boxes can be programmed but the green ones cannot. The vertex and fragment shaders are mandatory whereas the tesselation and geometry shaders are optional.

- The vertex shader is issued for any vertex that is specified in the drawing command. Therefore, it includes code for manipulating a vertex, usually to transform it but it might be very simple and just pass the vertex to the next stage of the pipeline.
- The tesselation shader, if it is activated, is used for describing geometric objects by processing the vertices passed by the vertex shader. For better-looking results, it might also add more geometric primitives.
- The geometry shader includes processing of individual geometric primitives or it can create new ones.
- The rasterizer cannot be accessed by the programmer and receives the geometric primitives by the previous stage (note that some of them might be rejected if they are out of the viewport) and produces fragments. Fragments are candidate pixels, which mean that they might be rejected later on.
- The final stage of the pipeline is the fragment shader. It is used to process fragments (or individual pixels). The most common operation in fragment shader is to determine the colour of a pixel.[10]

2.3.2 GLSL

As we have seen in the previous sections, OpenGL uses shader programs to render primitives on the screen. Shaders are small programs written in a specialised programming language called OpenGL Shading Language (GLSL). GLSL is very similar to the C language with some C++ also incorporated. For every shader (vertex shader, fragment shader etc), a different program has to be written.

These programs include a `main()` function, receive some variables as inputs from previous stages in the pipeline which are declared as global and output some variables to the next stage. These shader programs have to be compiled and linked using a compiler on the GPU. The compiler does not read the programs from files but instead the programs have to be passed to special functions as C strings. Furthermore, GLSL provides many functions for manipulating vectors, matrices as well as many mathematical functions. Moreover, GLSL provides variable types for matrices and vectors. [10]

2.3.3 Points, colours and transformations

We have seen that OpenGL can only render primitives such as points or triangles that consist of points. The points are represented with four coordinates: the x, y, z and w. The first three coordinates are the position of the point in the three dimensions while the last one is used for normalisation and its value is 1.0. Also, four coordinates indicate position whereas three coordinates (x, y, z) indicate a direction. Surface normals consist of three coordinates because they need to indicate a direction.

OpenGL can only work with Red, Green, Blue, Alpha (RGBA) colour space and if the colour data is in a different colour space then it has to be converted to RGBA. Inside the shaders, the colour values are floating points numbers in the range [0 – 1.0] and they are converted to integers when they are flushed on the screen. Similarly to physical world, in computer graphics objects cannot be seen unless they are illuminated by lighting. Many models for lighting exist and usually they take into account the properties of the material of the object, the surface normal as well as the light direction.

The primary use of OpenGL is the creation of animations and the purpose of this dissertation is to be able to produce animations or to move objects interactively. The object transformations are achieved by issuing matrix multiplications. In every application there are matrices that represent the projection, the view and the object. These matrices are multiplied together and then the final matrix, the Model-View-Projection (MVP) matrix, is multiplied with every point of the object. The dimensions of each of these matrices are four by four. OpenGL does not provide functions for manipulating the matrices, although it formerly did, and an external library must be used. Therefore, the matrices are multiplied on the CPU and the result is passed to the shaders as uniform variable. [10]

2.3.4 Textures

Because in this dissertation, textures are used heavily, a brief description is provided. Textures are images that are produced either by a camera or an artist, or they are created procedurally. Many types are supported like 1D, 2D, 3D or even cube-map textures. They are composed by texels (texture elements). Textures are usually processed inside the fragment shader. They are normally used for giving high fidelity to objects by gluing textures on them. However, OpenGL can be used for image processing by processing textures just as we did in this dissertation. GLSL provides a variable type called sampler and textures must be bound to a sampler variable.

When textures are accessed inside a shader, their values are sampled. There are many parameters that set up textures. Some concern the edges of the textures and some how the texture will be sampled inside a shader. Two options can be specified for sampling, linear and nearest. In the linear sampling, the texel is read using a weighted average of the four neighbours and thus a smooth representation of the texture is given. In the nearest sampling, the nearest texel of the requested coordinates is returned and this is the parameter used in this dissertation. [10]

2.3.5 Images

The drawback of textures is that textures can only be read inside a shader and a shader cannot write to textures. In the latest OpenGL and GLSL versions (4.3), images have been introduced. Images are GLSL variable types and provide the functionality of writing to buffers and reading from them. Therefore, images can be used for general-purpose computation and both textures and buffers can be bound on images.[10]

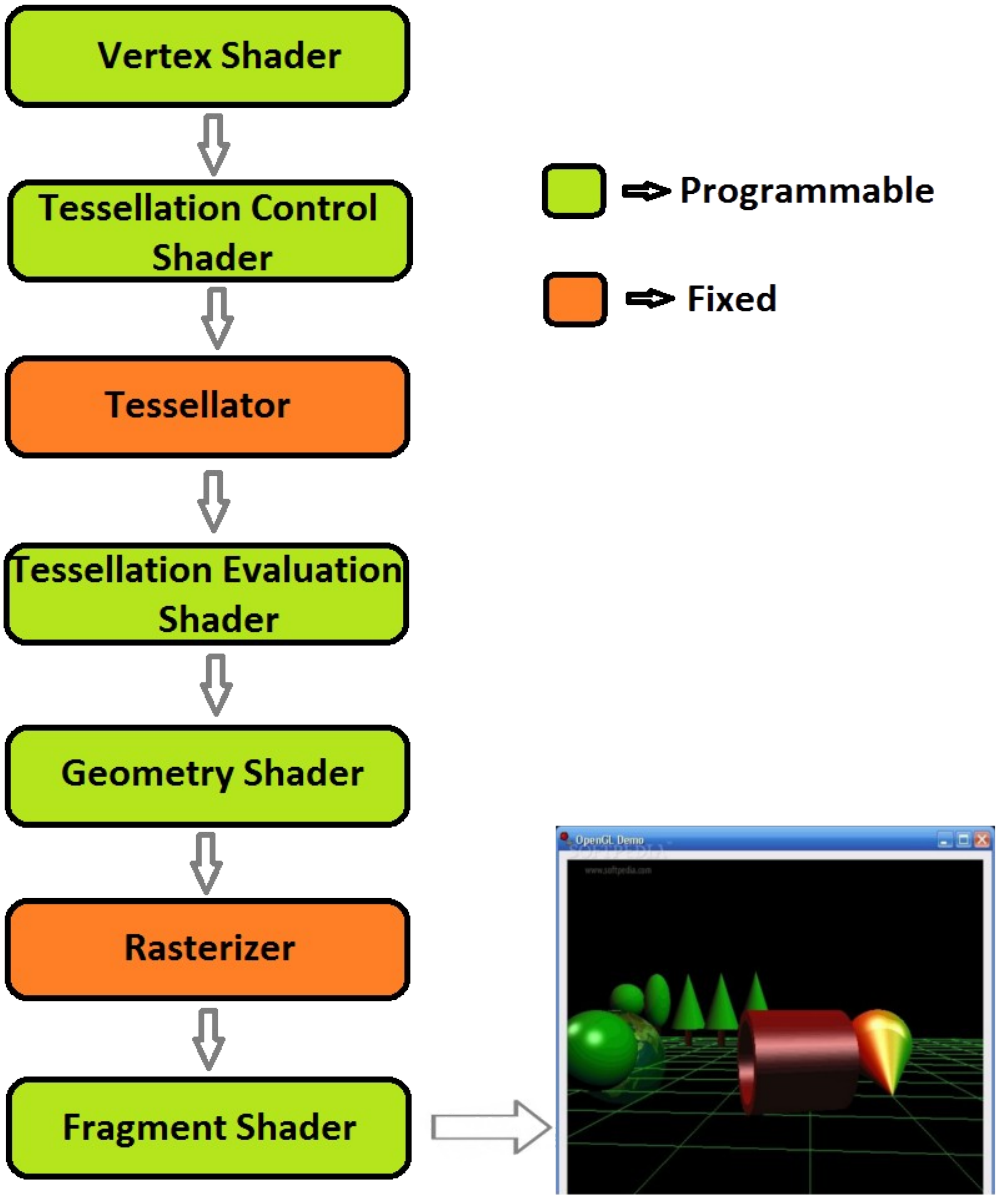

Figure 1: The OpenGL pipeline

3 The image-based point cloud visualisation algorithm

The image-based point cloud rendering algorithm consists of a few simple steps. Concisely, an image is created by projecting the points onto it and then some simple image processing techniques are performed. That differentiates this algorithm from others that perform processing in object space and also makes it easy enough to be implemented.

In particular, the points are lit using the Phong algorithm and projected onto an image. If the point data set is dense enough and every point covers a pixel, the image does not need any post-processing. However, this assumption is too optimistic. After the image has been created, some image-processing techniques must take place. The image-processing steps are required because holes are exist on the surface. There are two different types of holes:

1. Holes that background pixels are projected on them.
2. Holes that pixels belonging to a surface from behind are projected.

Therefore, two processing steps take place to eliminate these holes and both include the 3x3 neighbourhood of a pixel.[2]

Filling background pixels

In order for the background pixels in the holes to be spotted, a 3x3 mask is applied. Figure 2 shows the masks that are applied. If pixels in any of the masks are identified to be background pixels, then the candidate pixel is marked as background and is not changed. This step is iterative until no pixel has to be changed. [2]

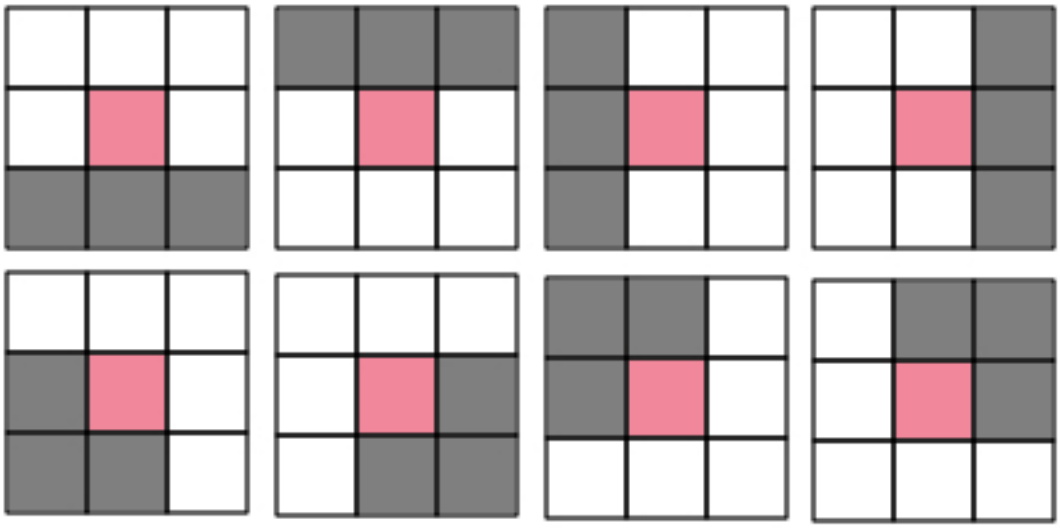

Figure 2: The masks applied in the background pixels filling processing step

Filling occluded surface pixels

This process is very similar to the previous one. In this step, pixels with a neighbourhood depth greater than a predefined depth value dmin are identified. The masks are similar to figure 2, however, the dark pixels may have any depth value and the white ones have a depth value greater than d - dmin . The dmin value is set empirically and the chosen value is 0.0001. This small but significant change in the filter masks is because of the fact that more pixels must be taken into account. In particular, we want a pixel to change if the majority of the neighbours belong to a higher surface. Similarly to the previous step, this process is iterative until no pixels have to be changed. [2]

Post-processing effects

In the original paper explaining this algorithm, it is proposed that after the two most important processing steps (background and occluded pixels filling), the image can be smoothed. However, we believe that more filters can be applied, for example someone might wish to detect the edges and thus apply that filter for edge detection or apply an object detection algorithm.[2]

3.1 The implementation

To begin with, a library for the implementation has to be chosen. This was not a difficult decision to make. The candidate library must be able to produce an animation, transform the points, apply lighting algorithms on the points and finally be able to process images. OpenGL provides all the desired functionality although in iterative image-processing algorithms it seems to have some weaknesses.

Firstly, buffers for storing the vertex coordinates, surface normals and colours of each coordinate on the GPU must be used. At the start of the program, these buffers are created and they are filled with data sent from the CPU. Throughout the execution of the program, although the points and the surface normals are transformed and the colours are modified depending on the position of the object and the light direction, the values of the buffers are not changed.

Furthermore, the core part of the algorithm is the image processing steps. In OpenGL, when points are rendered and go through the OpenGL pipeline, they appeared on the screen. However, this is not desired because an image has to be created in order to be processed and finally this image must be projected on the screen. This functionality can be achieved by doing an off-screen render. Off-screen render means that a new framebuffer is created (the default is the screen), a texture is bound on this framebuffer and the output of the OpenGL pipeline then goes to the bound texture. Therefore, one drawing command implements the off-screen render and another drawing command follows to process the texture on the default framebuffer, which is the screen.

The texture created for the off-screen rendering stores RGBA colours and the numbers stored on the texture are of float type. In the RGBA channel the RGB components store the colour of the point and the alpha component stores the depth of it as proposed by the image-based algorithm. Many types are supported for textures but floats are the ideal choice because a wide variety of numbers is covered and great precision is also supported.

Other buffers are also used in OpenGL. OpenGL needs a depth buffer to determine the depth of each point and when two points are projected on the same pixel, to decide which point to project depending on its depth. For the off-screen render, a depth buffer has to be created and bound on the framebuffer. Also producing smooth animations is achieved by having two buffers, the front and the back buffer. The front buffer is projected on the screen. In each render command, the output goes to the back buffer and when it finishes, the front and back buffers are swapped. OpenGL manipulates the front and back buffers.

Briefly, the program reads the data set from a file, creates a new framebuffer with a texture bound on it, and then processes the texture by applying the steps mentioned in previous section. Therefore, two shader programs are needed, one to project the points on the texture and one to process the texture. A Unified Modelling Language (UML) activity diagram on figure 3 shows an overview of the program.

We can see that the steps followed by the program are straightforward. First the points are read from a file and stored on the CPU's main memory. Then the points have to be sent to the GPU memory. Once they are sent to the GPU, they stay there until the endpoint of the program. Then the two

shader programs are compiled. As mentioned at the section 2.4.2, the shader programs are sent to the GPU as C strings and compiled there using a compiler on the GPU. From a programmer's perspective, all that has to be done is to call a few functions. Then a new framebuffer has to be created with a texture bound on it. In each loop in the activity diagram, we have to set either the default or the non-default framebuffer to active. Finally, two rendering commands are executed, one to render the points on the non-default framebuffer, and after that, one to process the texture created by the previous render.

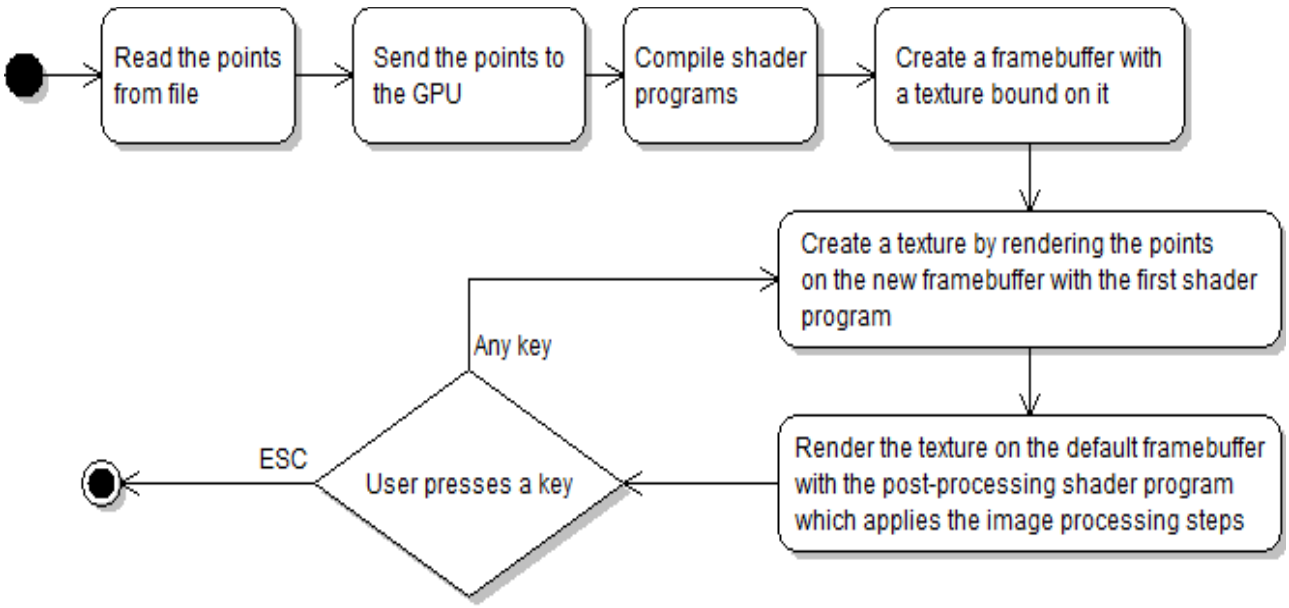

Figure 3: The UML activity diagram, brief description of the implementation

3.1.1 The architecture

The program has been written in the C++ programming language. Inside the C++ code, the C OpenGL functions are called. The program consists of two classes as shown in sequence diagram in Figure 4, the `PCObject` and the `Shader` classes.

The `PCObject` class provides methods for reading the data sets (.ply files) from files, using a library developed by Stanford University, and sending the points to the GPU. It provides a method for computing the surface normals of the points if they are not provided in the ply file using a function provided by the Stanford University library. Furthermore, it provides methods for initialising the framebuffer and binding a texture on it. Finally, it offers methods for the off-screen render and for invoking the image processing render.

The `Shader` class represents the shader programs. It provides methods for reading the shaders' source code from text files, compiling and linking them and finally methods for adding matrices to

the shaders as uniform parameters.

In the main file, there are functions for displaying the object, initialising objects and matrices and functions for capturing the user's input. Since OpenGL has changed and the fixed-pipeline has been removed, the matrices for transforming the points have to be created by the programmer and be passed as uniform variables to the shaders. For creating and transforming the matrices the GLM library was used[11]. It provides all the functionality needed for an OpenGL program and is very easy to use. Furthermore, OpenGL does not provide functions for manipulating the window system and capturing the user's input. Therefore, a window library must be used. The OpenGL Utility Toolkit (GLUT) library was used which is simple and easy to use but other libraries could also be used[12].

We can see in figure 4 that there is an object called GPU. It is not a C++ object and in fact represents the GPU itself. Therefore, it is shown that the shader programs are compiled on the GPU and an ID is returned, the points are sent to the GPU, the uniform variables are added to shader programs. These are then executed on the GPU and the rendering commands are performed on the GPU as well.

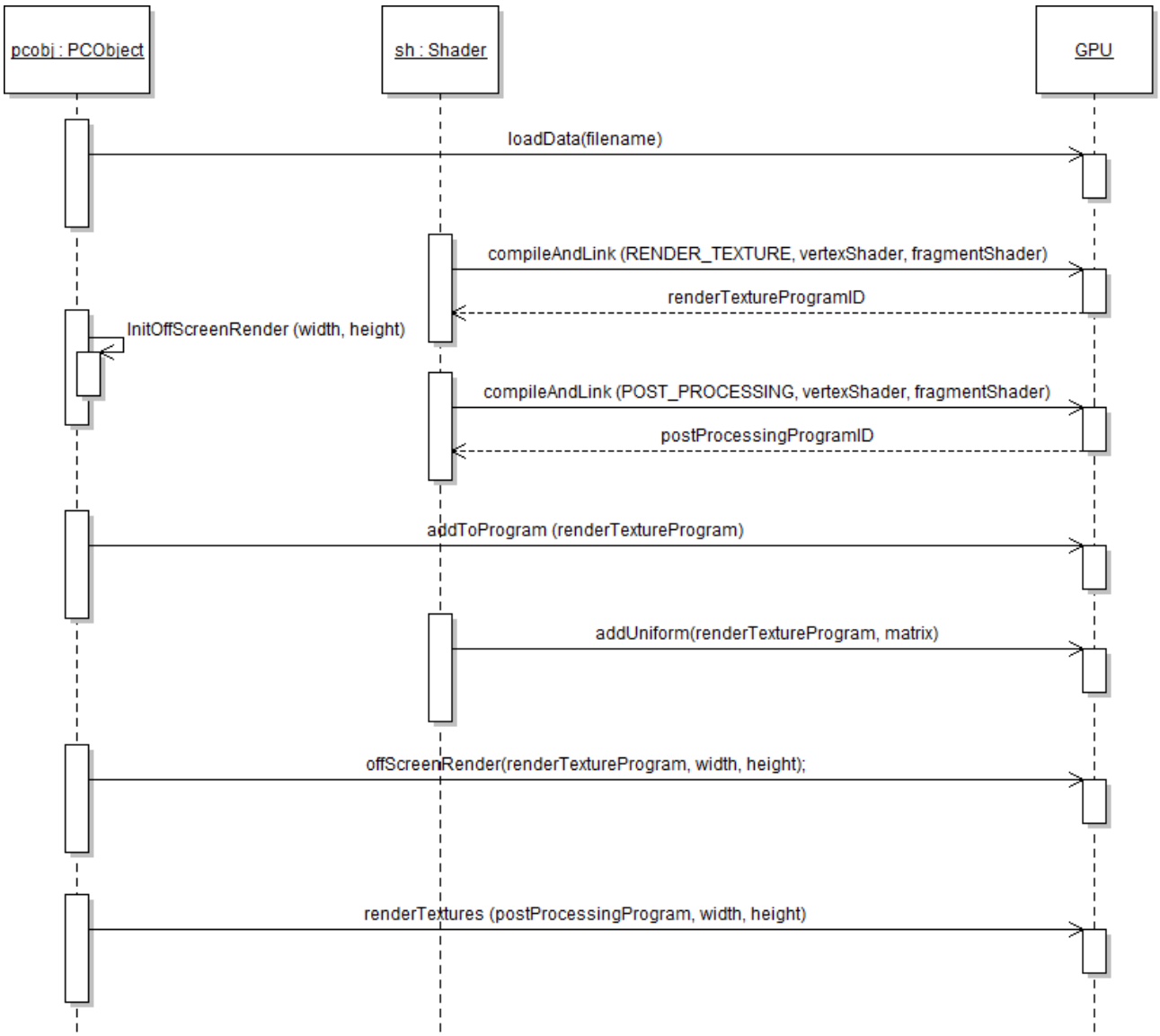

Figure 4: The UML sequence diagram of the implementation

3.1.2 The shader programs

In this application, two shader programs exist, one for rendering the points on the non-default framebuffer and one for doing the image processing. As mentioned in section 2.3.2, every program must have a vertex and a fragment shader. Thus, we have two vertex and two fragment programs one for each shader program. The first shader program, which renders the points on the non-default frambuffer, is straightforward. Every point is transformed using the Model-View-Projection matrix in the vertex shader and then in the fragment shader, the fragments are lit with the Phong model.

In the post-processing shader program, the vertex program does not do anything except to pass the coordinates of a rectangle where the image is going to be projected. However, the fragment shader is very important because it performs the image-processing steps. Note that as many fragment shaders are executed as the total number of the pixels. Therefore, The code inside the fragment shader manipulates only one pixel. OpenGL parallelises the program very efficiently. The fragment shader is shown below in C-like pseudocode.

```
pixelsFilledBefore = 0
pixelsFilledAfter = 1

while (pixelsFilledBefore < pixelsFilledAfter){
        pixelsFilledBefore = pixelsFilledAfter
        fillBackgroundPixels()
        if (pixel has been filled)
                addOneToAtomicCounter(backgroundpixe
                ls)
        pixelsFilledAfter = readAtomicCounter(backgroundpixels)
}

pixelsFilledBefore = 0
pixelsFilledAfter = 1

while (pixelsFilledBefore < pixelsFilledAfter){
        pixelsFilledBefore = pixelsFilledAfter
        fillOccludedPixels()
        if (pixel has been filled)
                addOneToAtomicCounter(occludedpixel
                )
        pixelsFilledAfter = readAtomicCounter(Occludedpixels)
}

smooth()
flush()
```

Listing 1: The post-processing fragment shader pseudocode

We can see that the program consists of two loops which are the two mandatory image-processing steps, the smoothing function and finally a function which just outputs the pixel to the framebuffer. In order that the multiple shader instances know when to stop, they have to communicate each other. The only way that the multiple instances can communicate is through an atomic counter. The atomic counter is a hardware-implemented counter with limited operations. The only operations that can be done on it are increase by one, decrease by one and read the value that holds. Therefore, when a pixel changes, the atomic counter is increased. When the counter stops getting increased, this means that no other changes to the image are going to be made and the program exits the loop.

It is very important to explain how the communication is achieved through the different shader invocations. The shader invocations are assigned to the multiple cores of the GPU and thus the communication is achieved through the memory. In the previous versions of OpenGL and GLSL, the texture pixels could only be read by the shader invocations and writing directly to textures was impossible. However, the application has been written in the latest OpenGL version and the image variable has been used.

Therefore, only one texture is used and inside the shader program, pixels are both read and stored on the texture. The function for reading the neighbours of a pixel as well as the pixel itself is shown below. The size of the neighbours array is nine and is used to store locally the pixels. The function imageLoad takes as arguments the image variable and an integer vector of size two with the coordinates of the pixel to be read and returns a vector of size four with the value of the pixel.

```
void getNeighbours(){

   neighbours[CURRENT] = imageLoad (image, ivec2(gl_FragCoord.xy));

   neighbours[UP] = imageLoad (image, ivec2(gl_FragCoord.xy) +
ivec2(0.0, 1.0));

   neighbours[DOWN] = imageLoad (image, ivec2(gl_FragCoord.xy)
+ ivec2(0.0, -1.0));

   neighbours[LEFT] = imageLoad (image, ivec2(gl_FragCoord.xy) +
ivec2(-1.0, 0.0));

   neighbours[RIGHT] = imageLoad (image, ivec2(gl_FragCoord.xy)  +
ivec2(1.0, 0.0));

   neighbours[UP_LEFT] = imageLoad (image, ivec2(gl_FragCoord.xy) +
ivec2(-1.0, 1.0));

   neighbours[UP_RIGHT] = imageLoad (image, ivec2(gl_FragCoord.xy) +
ivec2(1.0, 1.0));

   neighbours[DOWN_LEFT] = imageLoad (image, ivec2(gl_FragCoord.xy)
+ ivec2(-1.0, -1.0));

   neighbours[DOWN_RIGHT] = imageLoad(image,ivec2(gl_FragCoord.xy)
+ ivec2(1.0, -1.0));
}
```

Listing 2: The getNeighbours function used to read the pixels

Also every time when a pixel is changed, this value has to be written to the texture. Below the

function for writing a pixel to the texture is shown. The only difference between the imageLoad and imageStore functions is that the latter takes a third argument which is the value to be written to the texture.

```
imageStore (image, ivec2(gl_FragCoord.xy), result);
```

3.2 Results

In this section, the output of the image-based point cloud rendering algorithm is shown. The data sets have been taken from the Stanford University website and permission for their use in research projects is given. For these sections we use the hand data set which consists of 327K points. This data set provides the points coordinates and surface normals but no colour is provided. Therefore, we give a colour that we choose out of preference.

Figure 5: The hand data set without any processing

Figure 5 shows the hand data set without any image processing. As we can observe, only points are rendered and they are lit with the Phong lighting model. Many holes exist and it is the responsibility of the post-processing steps to fill them. There are two kind of holes. Holes that background colour is projected on them and holes that pixels from a occluded surface is projected on them. Occluded surfaces are behind the surface the camera sees.

Figure 6: The hand data set after filling holes with background pixels

Figure 6 shows a screen-shot of the program at a stage when the holes projecting background pixels were filled. The algorithm spots the background pixels that are surrounded by non-background pixels according to the masks explained in section 3, and fills those pixels with the colour of the closest to the camera neighbour pixel. However, there are still holes on the objects projecting pixels from occluded surfaces. Figure 7 shows the result after applying both the background pixels filling and the occluded surface pixels filling steps. All the holes that the object had, have been correctly filled, and it is ready to be used in other applications or to apply further image-processing algorithms.

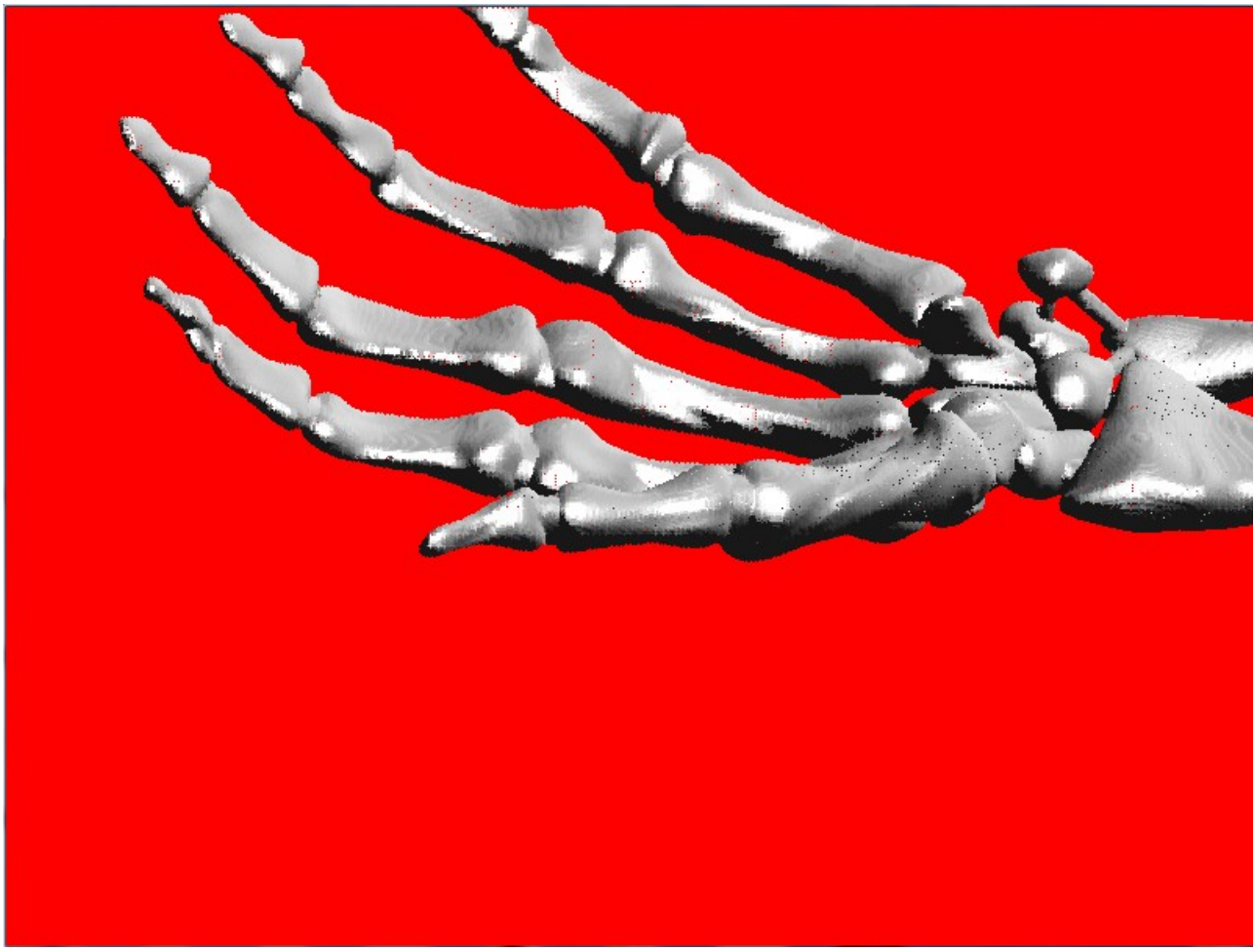

Figure 7: The hand data set after filling the holes of occluded surfaces

However, although this algorithm seems to behave correctly and produce results with great detail, it suffers when the points are sparse which might occur when we look closer at an object or the data set is sparse itself. Figure 8 shows a screen-shot from the hand data set when the camera is close to the object. In particular, the background pixels filling step cannot cope with sparse points. The reason is that while the points become sparser, even more masks contain background pixels. As explained at the beginning of chapter 3, when a mask contains background pixels, the pixel is not filled and keeps the value of background colour.

A way to improve the quality of the image is to make a small change in the background pixels filling algorithm. Instead of stipulating that for a pixel to be filled all the applied masks must not contain background pixels, we could say that even if one or two masks contain background pixels, the current pixel can be filled. That could essentially improve the output but it will probably have a bad impact at the edges of the object. In particular, it might extend the edges of the objects for a few pixels.

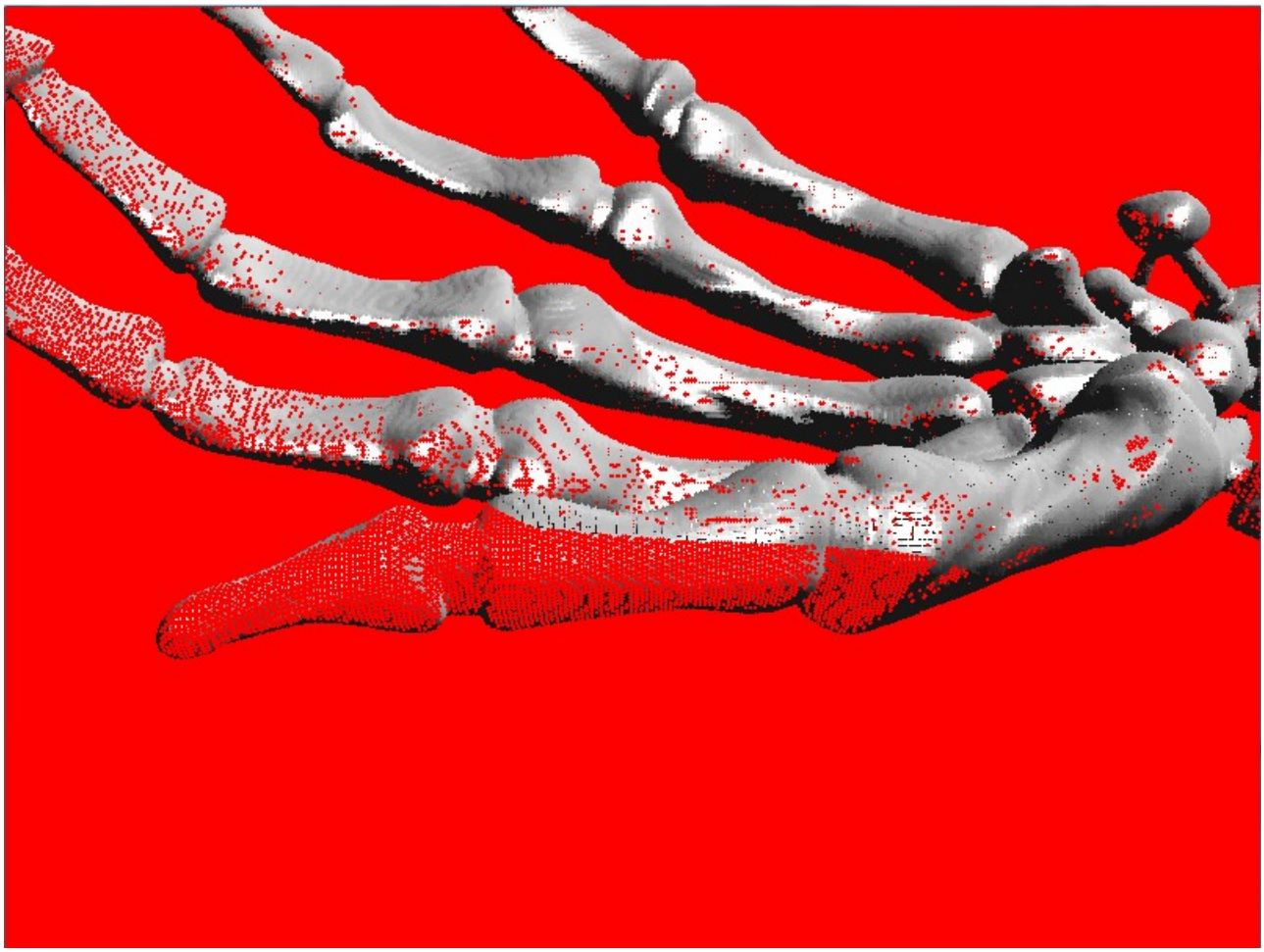

Figure 8: The hand data set, zooming too much so that the holes cannot be filled

3.3 An alternative implementation

Our implementation relies heavily to the latest OpenGL features. However, how would someone implement this algorithm with an older OpenGL version? The latest OpenGL features used are the atomic counters and the GLSL image variables. Unfortunately, there is no way to replace the functionality of the atomic counters. Therefore, the number of iterations in the image processing steps has to be fixed.

The most important part of this algorithm is the image processing steps and because they gradually improve the image, the state of the texture in each iteration has to be kept. GLSL image variables solved this problem by providing the flexibility of writing to textures. An alternative way of keeping the state of the texture in each iteration is two use to textures. In every iteration of the image processing steps, the application reads from one texture and outputs the result to the other texture and vice versa. Now the loop goes outside the shader program and actually wraps the

drawing command. Remember that when we create a framebuffer and attach a texture on it, the output of the shader program goes to that texture.

4 Performance and optimisation

In this chapter we are going to present information about the performance of this particular algorithm as well as the performance of the applications itself. Detailed performance evidence will be given for the most time consuming parts of the application. Furthermore, an in-depth explanation of the GPU architecture is given as well as the bottlenecks of the shader programs. Finally, the most important part, the optimisation of the application, follows.

The image-based point cloud visualisation as mentioned previously differs from the other algorithms because it performs all its computation on image space. Before even implementing the algorithm we expected the performance to depend only on the size of the image and the properties of the data set (density). Therefore, if the points are dense, the image-processing steps will run fewer times to fill the holes.

4.1 Performance

All the experiments were run on the student's laptop which has an Intel i3 CPU and an Nvidia GT 525M GPU with 96 shader units. The performance of the application is measured in frames per second (FPS) and the data was collected over period of time. As mentioned above, the performance is expected to depend on the image size and have a low dependency on the number of points. Figure 9 shows the performance of three different data sets on a 512x512 viewport. The hand data set consists of 327K points, the dragon data set consists of 3.6M points and the statuette one contains 5M points.

The hand data set runs in about 50 frames/second. When the camera comes very close to the object, the frames/sec drop at 20 because the image-processing algorithms have to run more times to fill in the holes. It is particularly worthwhile to note that when the object is behind the camera and the image contains only background pixels, the performance reaches at 75 frames/sec. The performance of the rest two data sets behaves similar to the hand but with lower speed.

Furthermore, although the point magnitude difference of the hand and the statuette is enormous, the difference in the performance does not correspond to this magnitude. The hand runs two time faster than the statuette but the statuette consists of 5M points and the hand of 327K points. In addition, the statuette contains 1.4M more points than the dragon but the performance is very similar to it.

As expected, the performance depends on the image size or the window size. Figure 10 shows the difference in the performance of the hand data set between a 512x512 and a 1024x1024 window size. The model on the 512x512 window size runs more than two times faster than the model on a 1024x1024 window. However, the performance was expected to be four times faster. The reason the

512x512 window does not run four times faster than the 1024x1024 window is that the object does not occupy the whole window. Therefore, the performance does not depend entirely on the window size but on the size that an object occupies within the window.

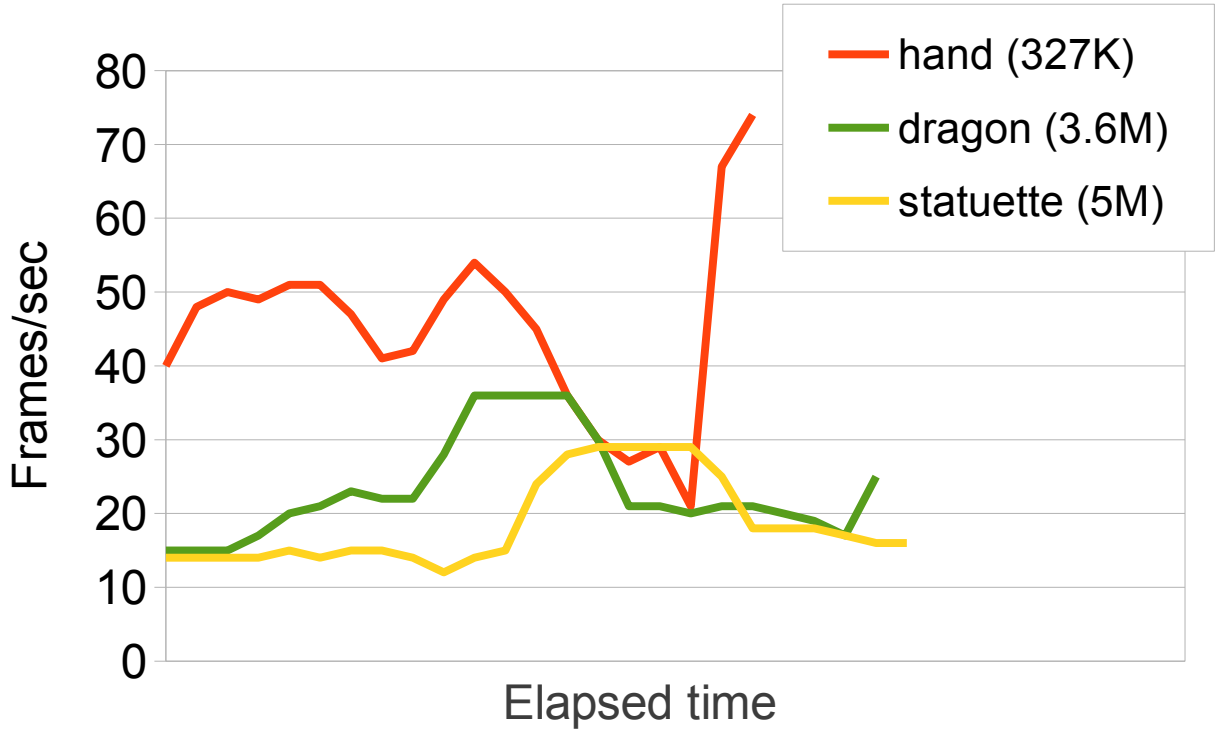

Figure 9: The performance of three data sets with different point magnitude on a 512x512 viewport

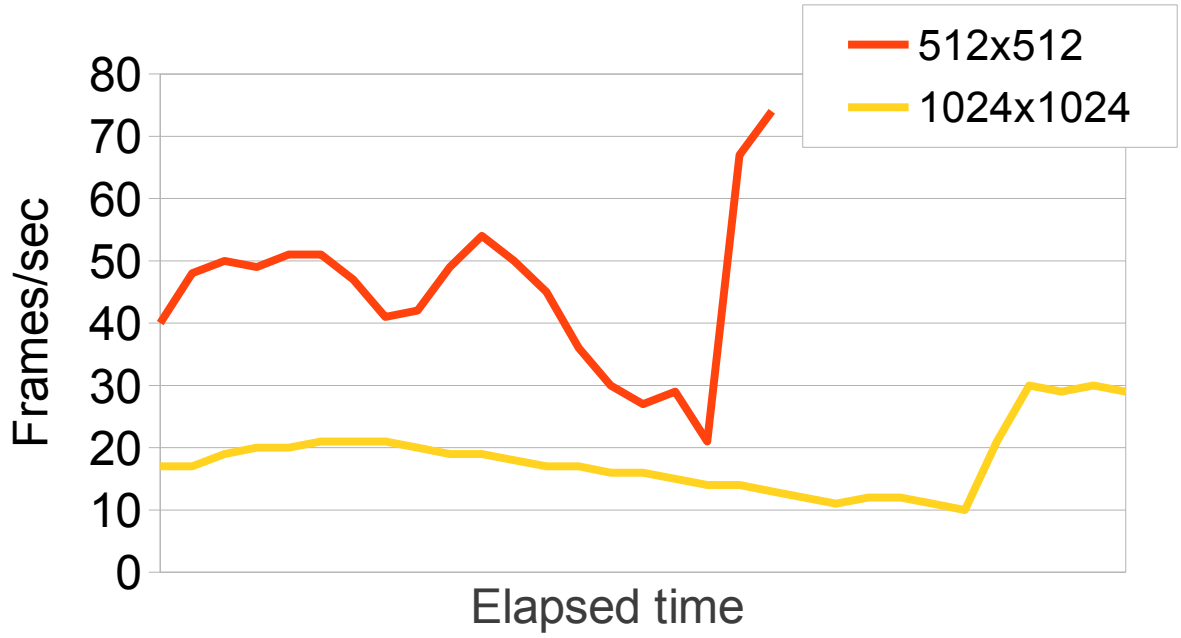

Figure 10: the performance of the hand data set (327K points) with different viewports

In particular, OpenGL creates as many fragment shader instances as the size of the window. Every instance runs for a specific pixel. Therefore, the shader instances run as long as changes are made to the image. Even if a shader instance does not follow the entire image processing functions and does not apply all the computation steps, it causes two type of bottlenecks. Firstly, every shader instance, reads the neighbour pixels in every loop and thus it causes traffic on the interconnect. However, the GPUs are very good at scheduling threads and while a shader instance waits for data to come from the memory, another thread might run on this particular core. Nevertheless, some overhead exists. The second bottleneck occurs when a shader instance reads the atomic counter to see if a change has been made to the image. Although the atomic counter is implemented on the hardware, obtaining a lock and reading the value of the counter causes some overhead.

As an experiment, the program was run two times on a 512x512 window size. The first time the object size was normal. The second time the object was scaled down by 50% and thus the size was half the normal size. Figure 11 shows the difference on the performance. We can see that when the object is in half size, the performance increases and becomes double the performance of the normal size. On the 1024x1024 window size, more pixels are of the background type. Although the shaders executing the background pixels cause some overhead, they do not apply any computation at all and thus the speed is not four times slower than the 512x512 window size but faster.

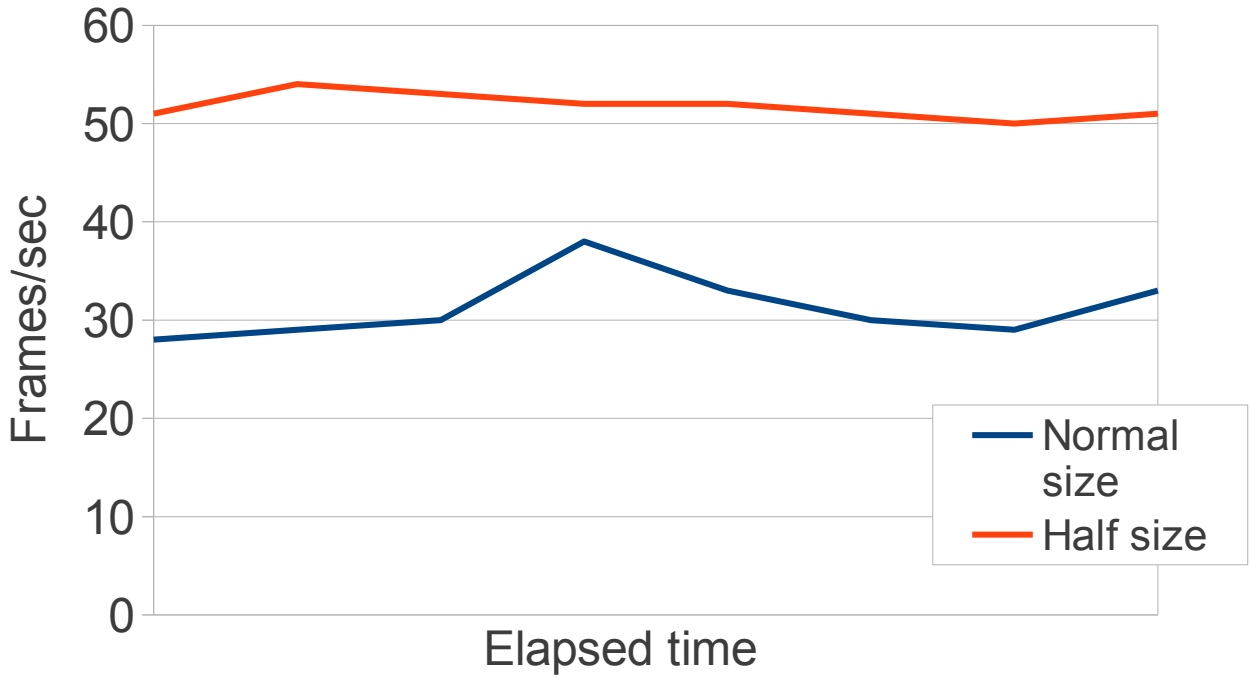

Figure 11: Normal size of the hand data set compared with the half size of it

All in all, the performance depends on the window size but decreasing the window size by 50%, the performance does not increase by 50% because of the reasons explained above.

4.2 Memory footprint

The buffers and the other aspects of the application were discussed in chapter 3. Therefore, It would be interesting to investigate the memory consumption of the application. The program gDEBugger was used to get such data as well as in the whole development of the application.

As the application runs, all the data exists on the GPU and there is no data transfer between the CPU and the GPU. When the application starts, it reads the data set from a file, stores it on a temporary buffer on the CPU main memory and then it sends it to the GPU. However, the GPU footprint does not contain only the data that the CPU sends. Figure 12 shows the GPU memory footprint for the hand data set. The numbers in the legend next to each component indicate the number of the objects of this kind that exist.

The shaders, two vertex and two fragment shaders, occupy 13KB on the memory, which is

insignificant, and their size does not depend on anything other than the number of lines of code they contain. The vertex buffer objects (VBOs) consume the most memory. The buffers included in the VBOs are three buffers for the coordinates, the surface normals and the colours and finally a buffer for storing the coordinates of a rectangle for the texture to be rendered on it. The latter consumes insignificant memory. In addition, the texture needs a rather significant memory chunk. In particular, because enough detail is needed, the texture stores 128 bits for every pixel and thus on a 512x512 texture it needs 4096KB. Similarly, the render buffer which is a depth buffer requires ¼ of the size of the texture. This is because the texture needs 4 components for every pixels and the depth buffer only one. Finally, the static buffers are some buffers needed for the rendering like front and back buffer (used in animations) and they don't depend on the point data set but they depend on the window size.

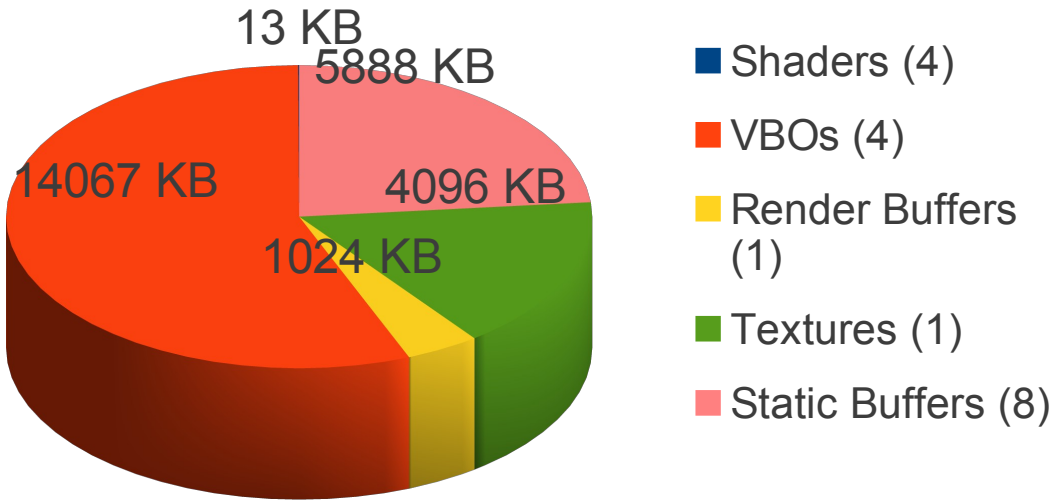

Figure 12: GPU memory footprint of the hand data set (327 points)

A good question would be, how does the memory footprint increases with the point magnitude? Figure 13 compares the footprint of three different data sets, the hand containing 327K points, the dragon with 3.6M points and the statuette consisting of 5M points. The memory consumption of the hand data set is about 25 MB and as we saw in previous section the texture as well as the static buffers require a huge amount of memory compared to the memory required to store the points. If we compare the hand data set with the dragon one, we can conclude to the fact that a lot of memory is used by the application and if the model consists of very few points, then more memory is consumed for other buffers than the buffers for the points. The dragon data set although it contains about 11 times more points than the hand data set, the memory footprint of the dragon data set is

about 7 times greater.

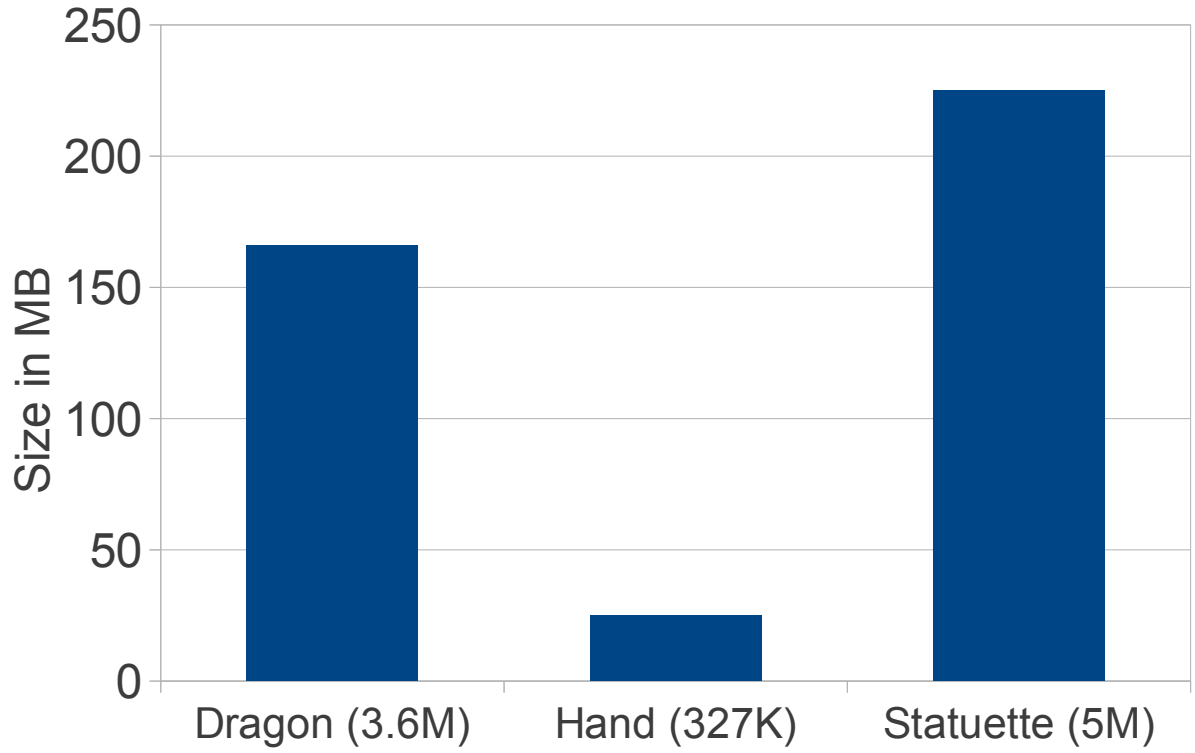

Figure 13: comparison of the total GPU memory consumption of the hand(327K), the dragon(3.6M) and the statuette(5M) data sets

4.3 Profiling

In graphics programs, there is code that runs on the GPU and code that runs on the CPU. The GPU runs asynchronously. The CPU issues commands and sends data to the GPU and then the CPU moves on to its calculations. Therefore, the first step is to investigate if our program is CPU-bound or GPU-bound. There are many ways of investigating this. One is to underclock the CPU speed by n percent and if the program speed increases by n percent then it is CPU-bound[13]. However, that might cause hardware problems and it was not done. Instead, the GPU-z program was used, which produces information about the GPU speed, load etc. Furthermore, Intel Vtune was used for profiling the CPU code (note that it profiles both the C++ and OpenGL) as well as the Nvidia's profiler that is built-in the Nvidia Nsight program.

4.3.1 Profiling the CPU

As mentioned above, the first step is to investigate if our program is CPU-bound before proceeding to GPU profiling. Figure 11 shows the GPU utilisation. As we can see, although the GPU never reaches 100% load, it always has work to do. Therefore, the GPU is never idle and thus there is no CPU bottleneck. However, we would like to spot the most time consuming pieces of code that run on the CPU and optimise them. That would make the CPU to send commands to the GPU for execution faster. Furthermore, this application is not the only running on the computer but many others run like the operating system. Therefore, this application is interrupted by the operating system and swapped with other processes. By making it run faster, we eliminate the possibility of our program being CPU-bound because of the process swapping.

Thus the first step is to use the Nvidia's profiler and spot the code that is time consuming. In figure 12 the average time taken of the most time consuming OpenGL functions is shown. We can see that the function `glGetUniformLocation` takes about 3500 μs. This function is used to query the shader program about the location of a uniform variable so that data can be bound on it. In fact, all the `Get` OpenGL functions cause the CPU program to stall until all the GPU commands are finished. Furthermore, this particular function is executed in every frame because it is used to query the location of the transformation matrices within the shader programs.

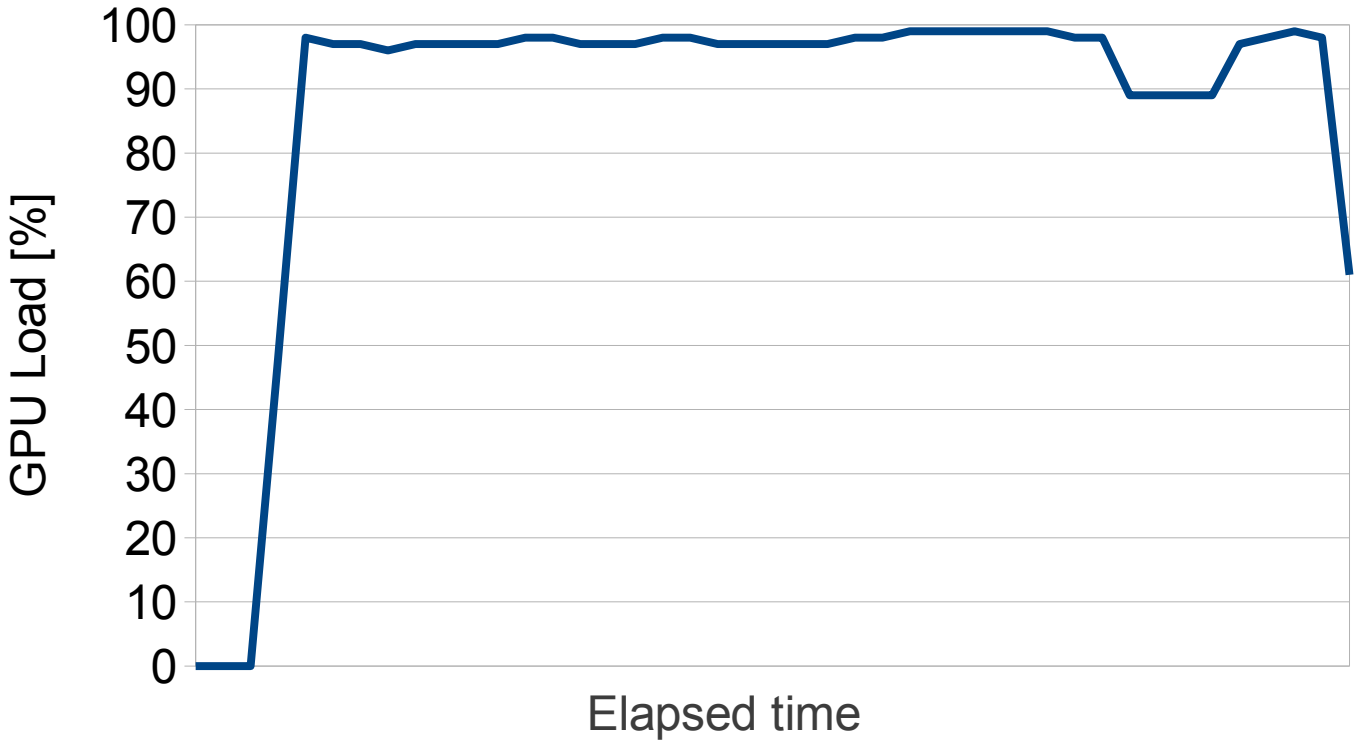

Figure 14: GPU utilisation (provided by the program GPU-z)

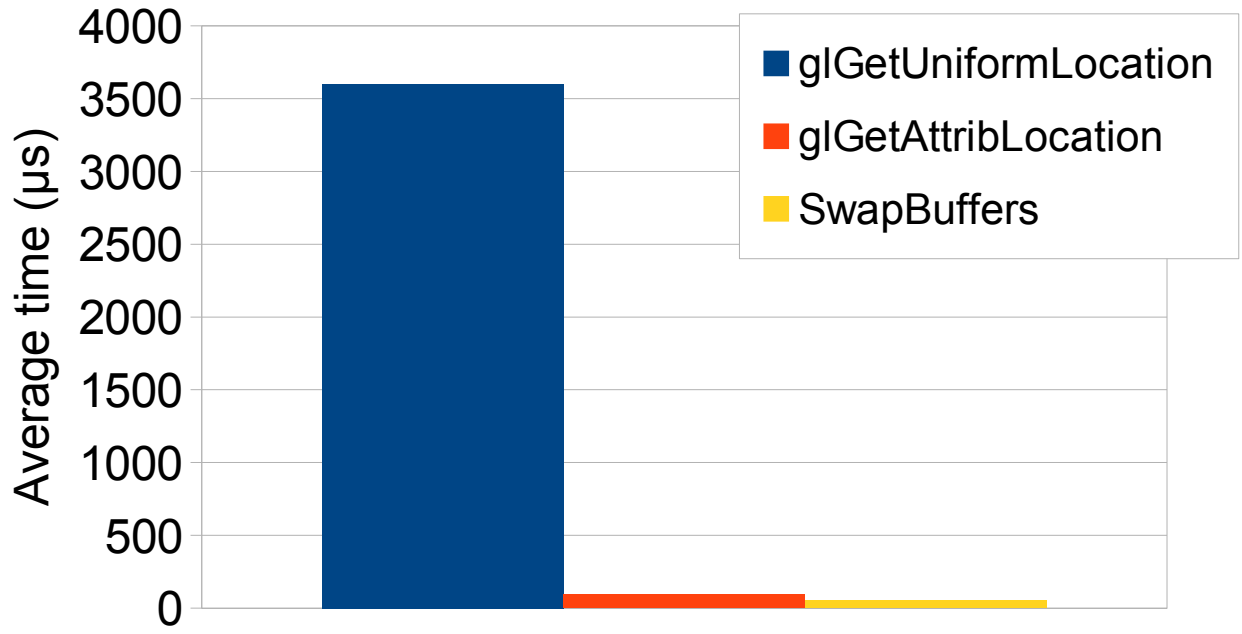

Figure 15: Average time of the most time consuming OpenGL functions

The way to optimise the most time consuming functions spotted in previous section is to avoid the glGetUniformLocation function. This can be achieved by setting a static location of the uniform variables in the shader programs and thus, no need to query the shader for the location will be needed. Although the code might be not as readable as using the glGetUniformLocation, the performance of the CPU code is expected to increase but not the overall performance. However, if the shader programs become fast enough, there will be many chances of the program to be limited by the CPU code.

4.3.2 Profiling the GPU

Apparently the key computations run on the GPU with the shaders. The shaders run every time a draw command is issued. There are two draw commands in every frame, one that renders the points and creates the texture (the off-screen render) and one that performs the image-processing steps on the texture (the post-processing). Because the shaders run asynchronously, the profiling tools cannot get their execution time. Therefore, after issuing a draw command we make the CPU to wait for the execution of the shader and the time is measured. As shown in figure 16, the off-screen rendering

shader which creates the textures runs in a static time. It does not depend on how close to the object the camera is and the execution time is much faster than that of the post-processing shader. The post-processing shader depends on how close the camera is to the object and therefore it could be interesting to investigate where the bottleneck in the shader is especially when the camera is too close to the object. Both diagrams where taken while the camera was zooming in.

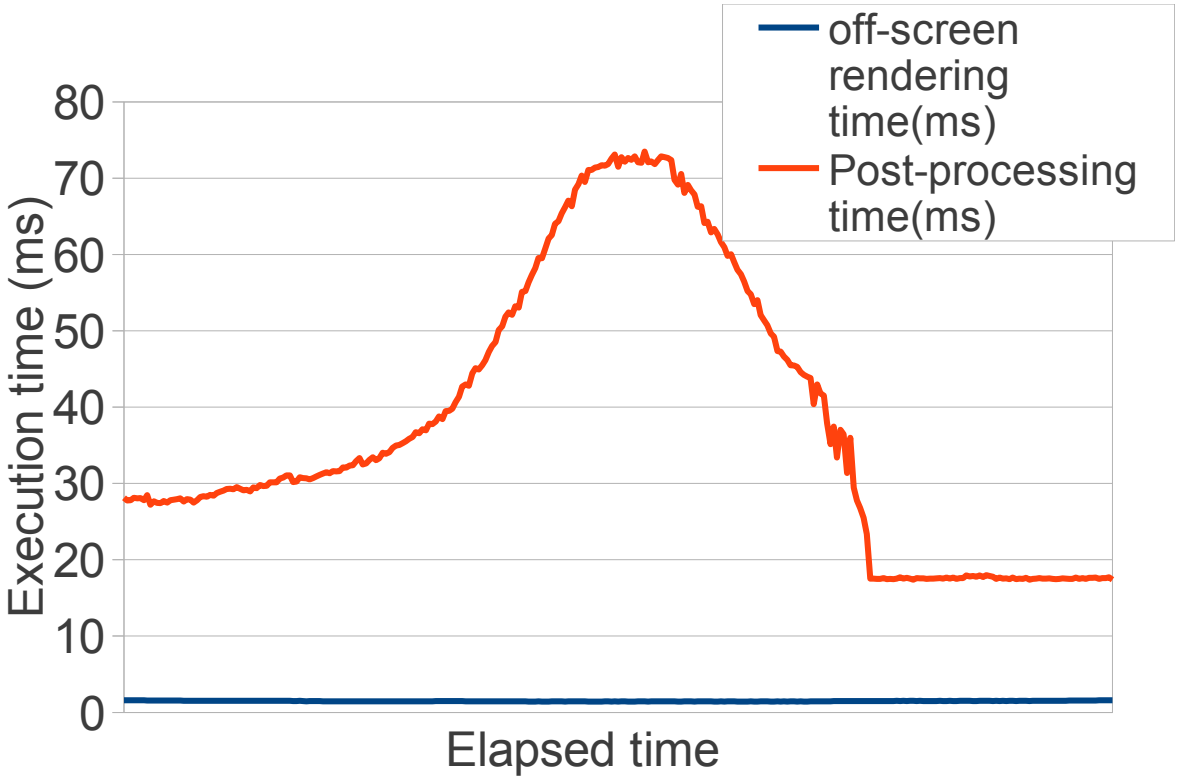

Figure 16: The execution time of both drawing commands

The post-processing shader starts at 28 ms and as we zoom in the execution time peaks at 72 ms. Moreover, when the object disappears from the screen because it is rotated, the post-processing shader can run much faster, at about 17 ms, because all the pixels are background and thus the post-processing steps do not run.

Our post-processing shader program consists of computation and some texture loads and stores. Basically, the computation has to be done and it is almost inevitable to make changes to get a good speed-up. However, the texture loads/stores need to be investigated.

4.3.3 The Fermi's architecture

According to Nvidia's hardware specification of Fermi architecture, the GPU consists of a number of Streaming multi-processors (SM), and every SM with a number of cores. Basically, the GPUs consist of a 2-level hierarchy, the SMs and the cores. The cores are placed within the SMs. Figure 17 shows the Fermi's architecture.

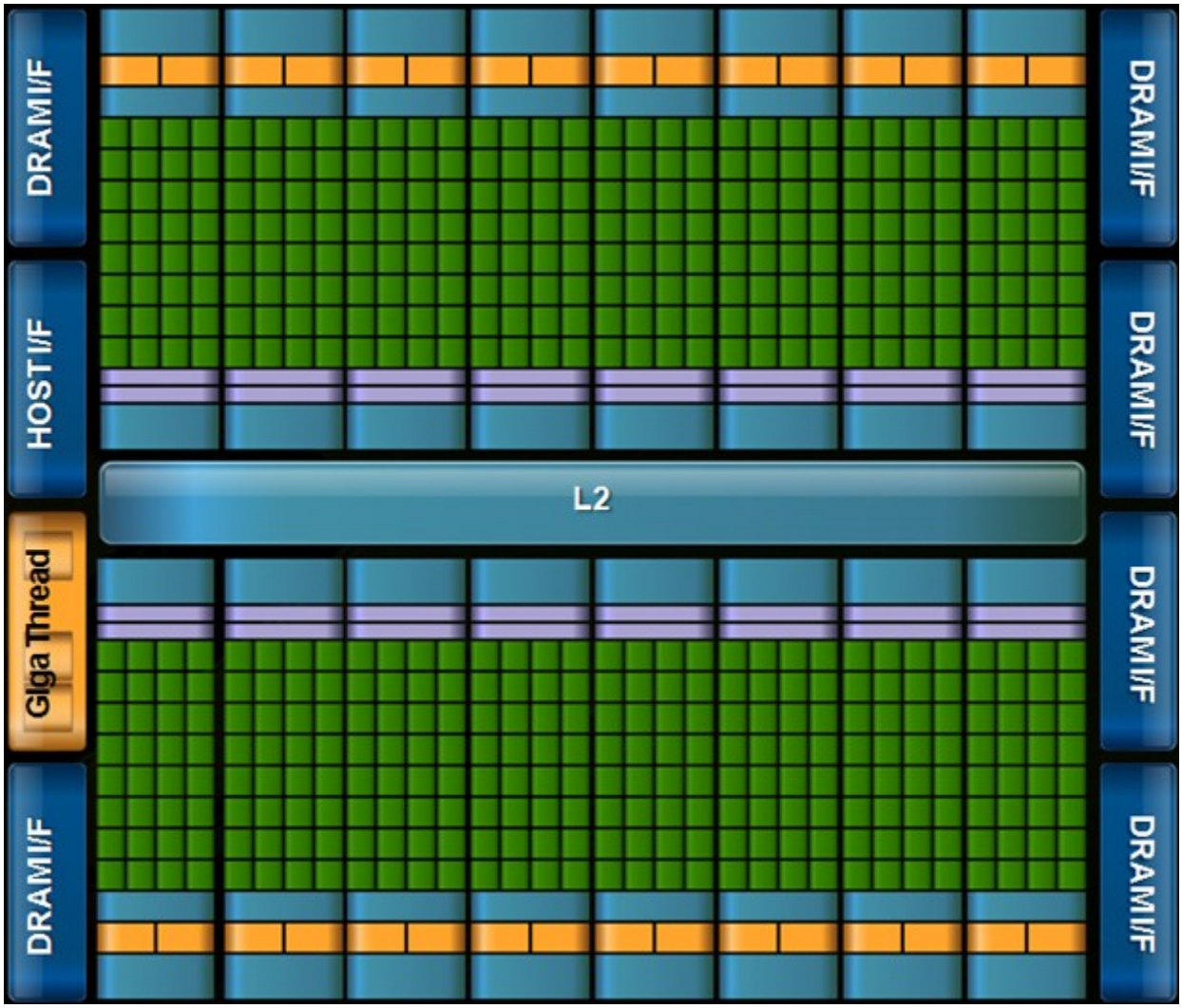

Figure 17: The Fermi's architecture

The Fermi GPU consists of 16 SMs, 6 64-bit Dynamic Random-Access Memory (DRAM) partitions and an L2 cache. Every SM, as shown in figure 15, consists of 32 cores, arithmetic units and 64 KB configurable shared memory and L1 cache. The shared memory in fact is a piece of fast Static Random-Access Memory (SRAM) memory that can be used both as shared memory and L1 cache and the sizes of these two memories can be configured. The shared memory is used for communication of the threads inside an SM and L1 cache is basically used for caching local and global data.

The L2 cache memory as well as the main memory can be seen by all the SMs and thus by all threads. All the data that is stored on the L1 cache of every SM has to be stored to the L2 cache as well. Thus, the L2 cache is used to cache the main memory and the L1 cache is used to cache the L2 cache memory.

The difference between the cache systems of the CPUs and the cache systems of the GPUs is that the L1 cache in GPU has to be coherent with the main memory and not with the L1 cache of every SM as it is in the multiple cores in CPU. Therefore, if an application wants its data to be coherent among the SMs, the data that is stored on the L1 cache has to be flushed to the main memory whenever it is modified. Furthermore, although all the SMs, share the L2 cache and they can communicate through this memory, the data is still flushed to the main memory. The cache protocols state that if a cache line is evicted from the L1 cache, then it has to be evicted from the L2 cache as well[14].

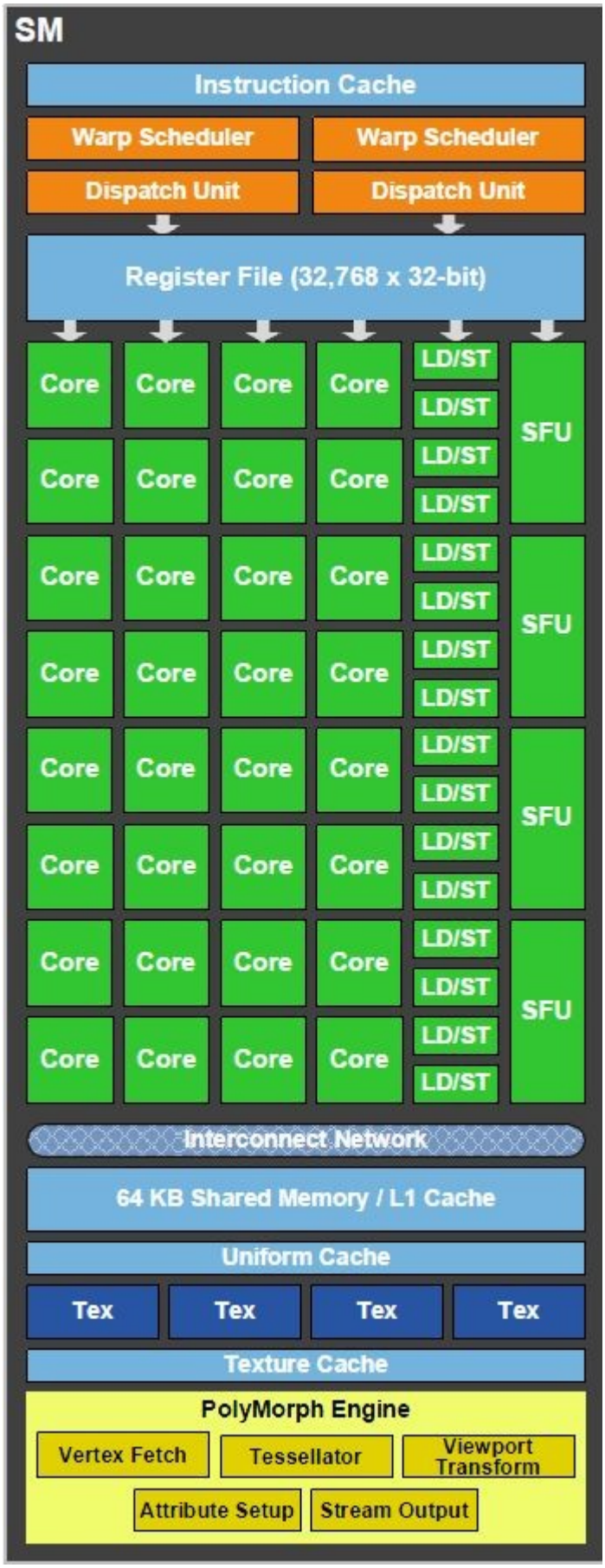

Figure 18: The SM architecture

4.3.4 Assessing the shader bottleneck

As mentioned in section 4.3.2, the post-processing fragment shader contains computation and texture loads/stores. The computation cannot be avoided but the texture loads/stores should be investigated more in depth.

Because the post-processing steps require the neighbour pixels of every pixel, the texture has to be stored in a coherent memory so that all the cores can see the changes. Therefore, the texture must exist at the main memory. At the start of the program, every shader invocation reads its current pixel as well as the neighbours. The texture is stored at the DRAM, it is moved to the L2 cache and then every pixel with its neighbours are moved to the L1 of each SM. The shader invocations then apply their computation to the pixel and store the pixel back to the main memory.

Furthermore, if two shader invocations read consecutive pixels that map on the same block line and one of the shaders modifies one pixel and stores it back to the main memory, then the cache line of the other shader invocation is invalidated. Thus, the shader has to go back to the main memory to pick up the pixels again even if it has not applied its computation yet.

Consequently, the post-processing fragment shader program suffers from cache misses and its performance is bandwidth-bound. In the next section a way of improving this performance penalty is described.

4.4 Optimising the GPU

We have seen in the previous section how texture loads/stores work. Because they have to be seen by all the cores, they are stored in main memory which is not as fast as the L1 cache memory. Moreover, we have seen that the GPU consists of multiple SMs with many cores in each, and all these cores in each SM have a shared memory which can be used for communication among the cores.

Our proposal involves a change in the program as well as a small change in the algorithm itself. Instead of producing the best result according to the algorithm but in a slow time, we would like our program to produce the results faster even if the results are not the same as they were for the original version but they could be acceptable.

Therefore, the texture is decomposed across the multiple SMs with a 2D regular domain decomposition, and in every SM, every block of the texture is decomposed across the multiple cores with a 2D regular domain decomposition again. The cores in each SM communicate through the L1 cache memory but the SMs do not communicate because otherwise, the program will be again limited by the main memory's bandwidth. Thus we expect holes in the result images at the edges of

the decomposition across the SMs. Ideally, we would like the edges of the decomposition to be as small as possible and the 2D decomposition achieves that and creates smaller edges than the 1D decomposition. What is more, by decomposing the image in the two dimensions, more threads can run. In particular, in the 1D decomposition, the number of threads that can run is limited to the size of the decomposed dimension whereas in the 2D decomposition, the number of threads is limited to the product of both dimensions.

The fragment shader does not give control on how to map the shader instances. In order to achieve this functionality, a kernel has to be written using a General-purpose Graphics Processing Unit library (GPGPU). However, the latest OpenGL version (4.3) offers a new shader, the compute shader. The compute shader can perform any kind of general purpose computation on data that is stored on the GPU. It is written in GLSL language and the process of compiling it is the same as that for compiling vertex and fragment shaders. Nevertheless, Compute Unified Device Architecture (CUDA) or Open Computing Language (OpenCL) could be used instead of compute shader. When a compute shader is invoked, the number of blocks (or workgroups) that will run in the x, y, z dimension is specified. Inside the compute shader, the number of threads that will run in every block is configured.

The code of the compute shader is very similar to the post-processing fragment shader but the texture is loaded on a shared variable and exists there until the end of the execution of the compute shader where it is stored back to the texture. The shared variable is shared to all the cores in each SM and exists on the L1 cache. Furthermore, the shared variable is a two dimensional array with halos. The declaration of it is shown below.

```
shared vec4 sharedImage[gl_WorkGroupSize.x + 2][gl_WorkGroupSize.y + 2];
```

The built-in variables `gl_WorkGroupSize.x` and `gl_WorkGroupSize.y` contain the number of threads running in the block (or workgroup) in the x and y dimensions respectively. The `getNeighbours()` function shown in listing 2 in section 3.1.2 that is used to read the current as well as the neighbour pixels has been replaced with the function `cacheImage()` which is called only once at the start of the compute shader. The thread with id zero in each block, sets the value of the halos to zero and then every thread using its id reads the corresponding pixel of the texture. The vector variable `gl_LocalInvocationID` contains the id of the thread running in the block position in the x and y dimensions and the vector variable `gl_GlobalInvocationID.xy` contains the id of the thread in the global position.

```
uint currentX = gl_LocalInvocationID.x + 1;

uint currentY = gl_LocalInvocationID.y + 1;

void cacheImage(){
        if (gl_LocalInvocationID.xy == vec2(0.0)){
                for (int i=0; i<gl_WorkGroupSize.x + 2; i++){
                        sharedImage[i][0] = vec4(1.0, 0.0, 0.0, 0.0);
                        sharedImage[i][gl_WorkGroupSize.y + 1] = vec4(1.0,
                0.0, 0.0, 0.0);
                }
                for (int j=0; j<gl_WorkGroupSize.y + 2; j++){
                        sharedImage[0][j] = vec4(1.0, 0.0, 0.0, 0.0);
                        sharedImage[gl_WorkGroupSize.x + 1][j] = vec4(1.0,
                0.0, 0.0, 0.0);
                }
        }
        sharedImage[currentX][currentY] = imageLoad (image,
        ivec2(gl_GlobalInvocationID.xy));
}
```

Listing 3: The cacheImage() function used to copy the image to the cache memory

The halos are set as background pixels. Thus, the post-processing step of filling background pixels has to be altered and say that even if one or two masks contain background pixels do not mark the pixel as background and change its value as described in section 3.2. The reason for that is that the pixels at the edges of a block will always have a mask containing background pixels. So, we have to say that even if one or two masks contain background pixels, continue with filling that pixel.

Also, in the post-processing fragment shader, the multiple threads used an atomic counter to decide when to stop the post-processing steps. Although this counter is implemented on the hardware and is fast enough, it causes the threads to run the post-processing steps more times. In particular, if a change was made to a pixel on the right bottom of the texture, then a shader invocation that executes a pixel on the left upper of the texture had to run again. However, it is almost inconceivable that the left upper pixel will have to make a change. In the compute shader, this mechanism was implemented with a shared variable instead of an atomic counter. Therefore, the threads run as long as changes are made in the same block and their execution is terminated faster.

Another optimisation that we have done is to reduce the memory requirement of the texture. Each pixel of the texture consists of four components, RGBA. For each component, 32 bits are required and thus 16 bytes per pixel. For every drawing command, although the texture is now stored on the shared memory and the communication takes place on it, the texture has to be loaded from the main memory to the shared memory. Because of the big size of the texture, a bottleneck might be caused at the interconnect and in the worst-case scenario the texture will not fit on the shared memory. Therefore, the size of every pixel component of the texture has been reduced to 16 bits. That could reduce the traffic on the interconnect and also make the GPU to schedule the block execution on the SMs better because of the reduced size of the texture. Also the memory footprint will be reduced but not significantly. However, by applying that optimisation, we lose accuracy in the computation but we believe that 16 bits for every component in the RGBA covers a wide range of numbers and offers enough accuracy.

After adding the compute shader, the post-processing fragment shader does not have to do any computation. Therefore, it becomes a pass-through shader that only flushes out the texture on the screen.

4.5 Results

The original version of the program has produced very good results of the point cloud data sets when the camera is not very close to the object. By optimising the program we expected holes in the image at the edges of the block decomposition. Figure 19 shows the output of the optimised version. The output is as good and accurate as the output of the original version. However, because of the small change in the algorithm a small impact can be noticed at the edges of the object.

Figure 19: The hand data set produced with the optimised version of the program on a 512x512 viewport

The drawback of the image-based point cloud visualisation algorithm is when the camera is very close to an object or in general when the points are sparse. Therefore, we expect the output of the optimised version in a close view of the object to be even worse than the original version.

Figures 20 and 21 show the difference in the output of the optimised and the original version respectively when the camera is very close to the object. Both programs cannot cope in such cases but the output seems to be very similar. However, in the optimised program (figure 20) we can see some holes projecting background pixels caused by the SMs not communicating each other. Although we can barely see the holes, their shapes are rectangles indicating the regular domain decomposition. Nevertheless, the output of the optimised program is acceptable.

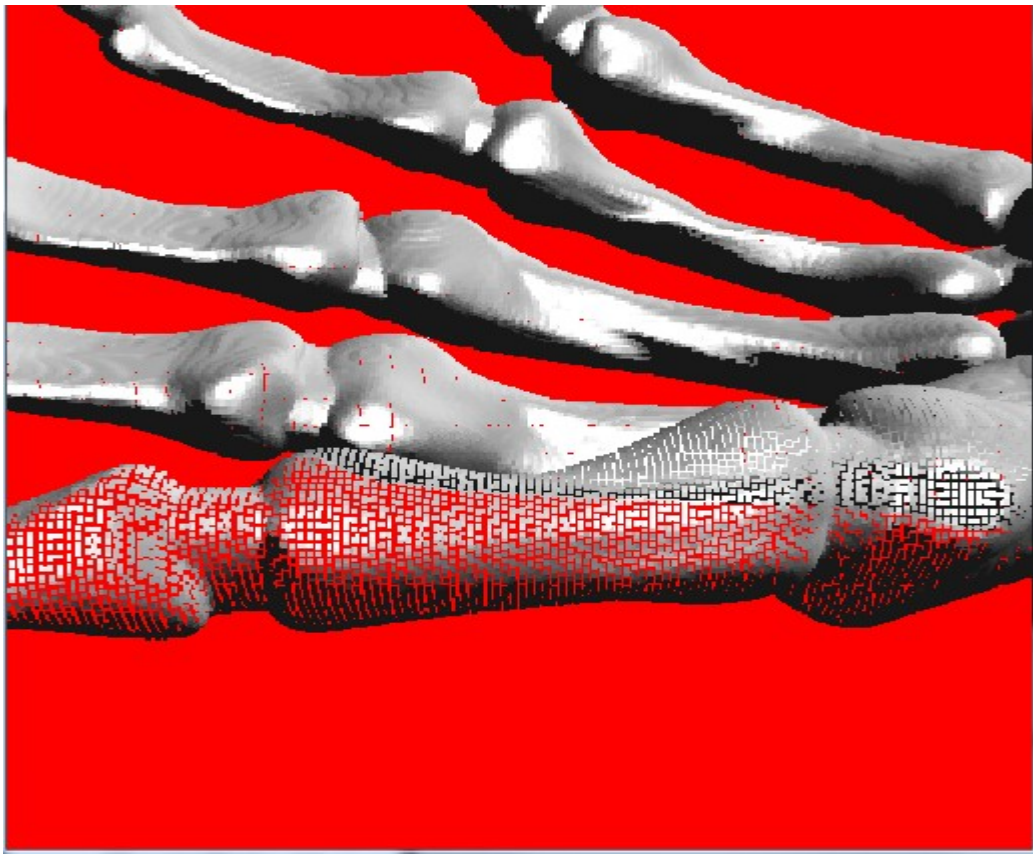

Figure 20: The hand data set produced with the optimised version of the program on a 512x512 viewport in a close view

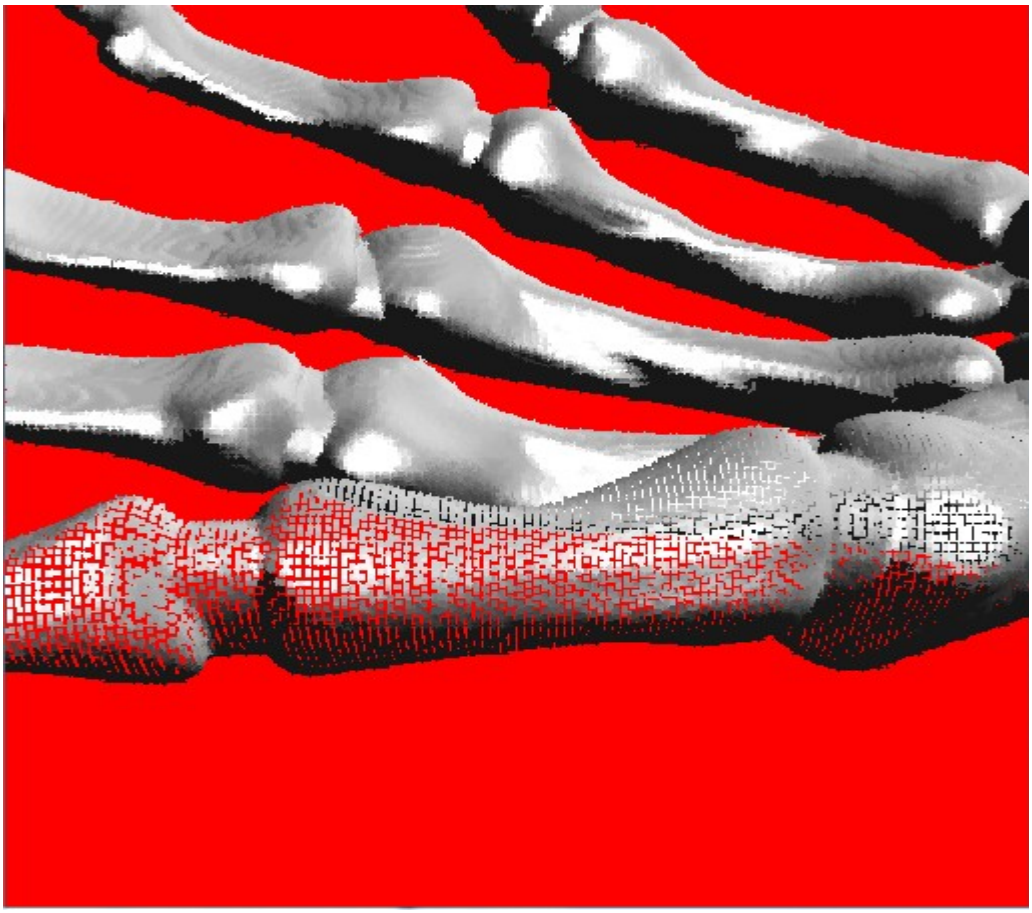

Figure 21: The hand data set produced with the original version of the program on a 512x512 viewport in a close view

4.6 Performance Improvement

As we have seen in the previous section the output of the optimised version of the program is very similar to the output of the original one. However, the performance has been increased as we can see on figure 22. For the hand data set with 327K points, the performance achieved is three times faster than the original version. What is more, as we zoom in and zoom out the performance remains stable, therefore, although the points become sparse as we zoom in, it doesn't affect the performance because the synchronisation occurs in each and every SM and not in all the SMs.

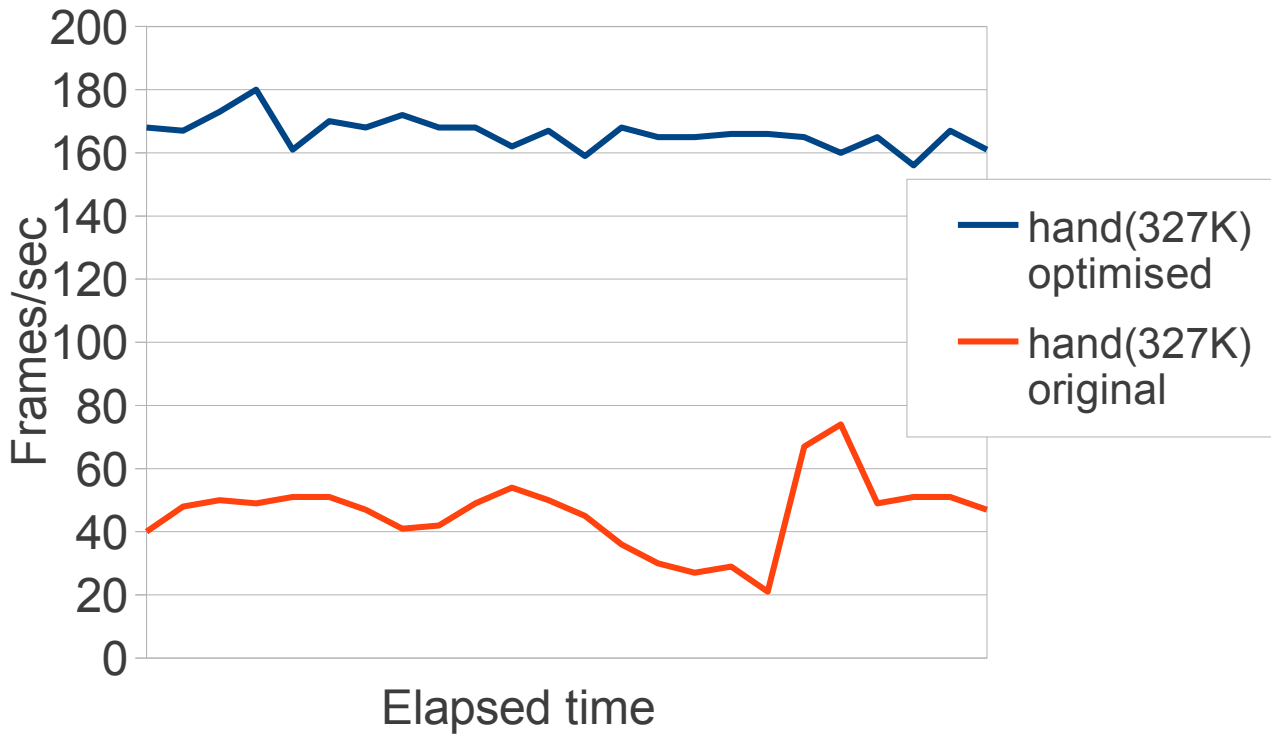

Figure 22: Performance comparison of the original and optimised program for the hand data set (327K points)

Figure 23 shows the performance improvement of the statuette data set consisting of 5 million points. The optimised version runs two times faster than the original one. Obviously, the performance improvement varies for different data sets. The performance improvement depends totally on the main memory accesses or more specifically on the number of iterations in the image processing steps. If the points are dense and the image-processing steps run for a small number of iterations, the performance will not benefit from the optimisation. However, if the points are sparse

enough so that the image processing steps run many times, then the performance will be sufficiently improved by the optimisation. Therefore, the performance of the statuette data set is not improved as much as the one of the hand because the points are dense and we can see that from the actual number of points (5 million).

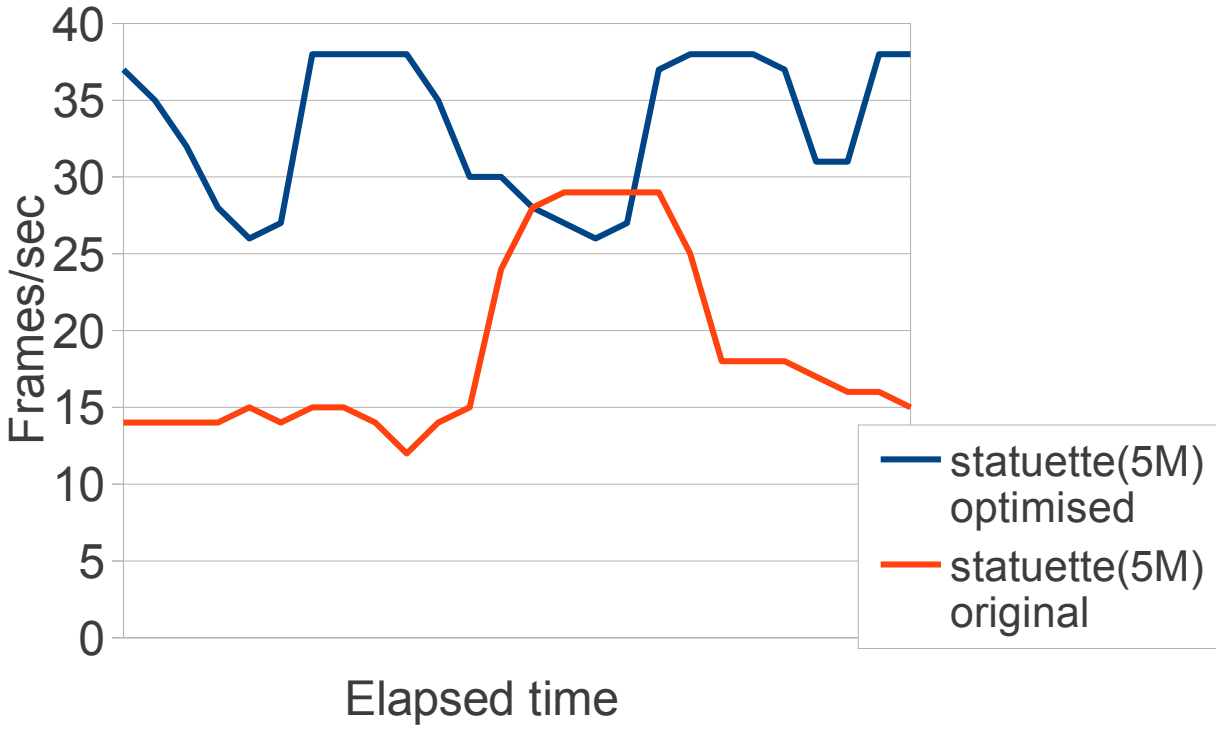

Figure 23: Performance comparison between the original and the optimised program with the statuette data set (5M points)

4.7 Advantages and disadvantages

In chapters 3 and 4 comprehensive explanation of the image-based point cloud visualisation algorithm was presented as well as the performance of its implementation and the optimisation applied.

The image-based point cloud visualisation algorithm basically projects the points on a texture and then applies some image-processing steps. Implementing the algorithm seems to be easy enough and if it is implemented with the latest OpenGL version then the development becomes much easier. The most important image-processing steps are the ones that fill in the holes. After the holes are filled, many image filters and effects can be applied to that image and there is no constraint. For example the image can be blurred or smoothed as proposed by the authors of the algorithm or the edges of the object can be detected. Concerning the performance, this algorithm runs incredibly fast

even when it runs with data sets containing millions of points because there is no direct dependency with the point magnitude. Although the application was not actually compared with others implementing a point cloud visualisation that perform the computation on object space, the image-based one is expected to be much faster especially when big data sets are visualised. Also, the optimisation applied was easy to implement and the performance improvement was more than sufficient.

On the other hand, if the data set contains sparse points or if the camera is close to the object the image-based algorithm does not cope very well. Basically, the problem is spotted in the filling background pixels step. Even when this particular step is changed as proposed on section 3.2, most holes are filled but the edges are damaged. Moreover, because it applies all its computation on image space, the number of pixels constrain the details of the object. In particular, if the object is too small but consisting of many points or if it is narrow and expanded in the z direction which can happen while rotating it, the image-processing steps might not produce the desired results. For example two parts of the object (they could be two fingers from the hand data set) are distinguished by background pixels. While zooming out and the object becomes even smaller, those background pixels will probably map on one pixel. Therefore, the background pixels filling processing step will probably fill in those background pixels and the two fingers will no longer be distinguished.

All in all, the image-based algorithm performs all its computation on image space and is very fast. The most appropriate use of it could be in visualising dense data sets that do not place the camera too close to the objects. The optimisation seems to produce acceptable results and can be used in performance critical applications.

5 The WebGL implementation

WebGL is a low-level 3D graphics API based on OpenGL for Embedded Systems (OpenGL ES). OpenGL ES is a stripped down version of OpenGL. WebGL provides similar functionality with OpenGL but in an HyperText Markup Language (HTML) context. It is exhibited through the HTML5 canvas element as Document Object Model interfaces. The HTML5 canvas provides rendering in web pages with different rendering APIs.

WebGL is supported on different platforms under different GPU architectures and thus it does not provide all the functionality of OpenGL because portability has to be ensured. However, transition from OpenGL to WebGL is straightforward although some OpenGL functionality is absent in WebGL.[15]

5.1 OpenGL and WebGL textures

Textures are the key component of the image-based point cloud visualisation algorithm and thus a brief explanation of the differences between the OpenGL and WebGL textures is given. As explained in section 2.3.4, textures are images that are glued on objects to give high fidelity to them. Both in OpenGL and WebGL the `glTexImage2D` is used to create 2D textures. This function takes many arguments such as the type of the texture and what components to store for colours. Additionally, the type of the data to be stored is specified. OpenGL textures support many types including floats and this is the type used in the OpenGL implementation. However, WebGL, only supports unsigned bytes and unsigned shorts because portability among multiple GPUs must be ensured. However, when a texture is read inside a shader program, it is converted to float. Floats support a wide range of numbers in comparison to short integers.

In WebGL the support of non-power of two (NPOT) is limited. Both the width and the height of the textures has to be a power of two. There are many restrictions in NPOT textures and the most important one concerning this dissertation is that sampling a NPOT texture will return (0, 0, 0, 1) RGBA colour if the texture has not been set up properly.[16]

5.2 Similarities and differences with the OpenGL implementation

Moving from OpenGL to WebGL is straightforward. The overall structure of the WebGL implementation is very similar to that of OpenGL one but with many significant differences caused by the absence of some functionality in WebGL.

The WebGL implementation follows the same flow with the OpenGL one. First an off-screen render is done; creating a texture projecting the points and then another render is done which process the

texture created by the previous render. Also two shader programs are used as with for the OpenGL implementation, one to create the texture and one for post-processing. In WebGL, the shaders' source code is passed to the compilation function as a DOMString data type.

The buffers for storing the point coordinates, surface normals and colours are the same as in the OpenGL implementation. The point cloud files are stored on a web server. The three.js library is used to perform an Asynchronous Javascript and XML (AJAX) request to the server[17]. The server sends the point cloud file to the client and then a function in the three.js library is used to interpret the data and create arrays with the points, their properties, if any, and the faces. However, the function for interpreting the data has to be modified for data sets that contain point properties such as surface normals. The three.js library is a high-level API that abstracts all the low-level features of WebGL and it is very popular[18]. After the arrays have been created, they are passed to the GPU and exist there until the end of the execution as in the OpenGL. The WebGL buffers are exactly the same as in the OpenGL. For the point coordinates, four components are used, the first three are the coordinates of the points and the last one is used for normalisation. The surface normals consist of three coordinates because they indicate a direction and the colours consist of four components (RGBA).

One difference exists in the first render where a texture is created with the points projected onto it. The image-based algorithm states that the points that are not seen by the camera respecting their surface normals must be discarded. In OpenGL, the gl_ClipDistance built-in array is provided which is used for user clipping. However, this array is not available in WebGL. Nevertheless, the points that are not seen by the camera are assigned with background colour and their depth is set to zero.

The most significant difference exists in the post-processing part of the implementation. WebGL does not support image variables. Recap from section 2.3.5, image variables provide the functionality of both reading and writing to textures where plain textures can only be read. Therefore, two textures are used and a ping-pong trick is done as explained in section 3.3, alternative implementation, where two textures are used and each time the data is read from one texture and written to the other one. Furthermore, atomic counters are not supported in WebGL because they are implemented in hardware and WebGL must ensure portability across multiple GPU architectures. Thus, the number of iterations that the post-processing algorithms run is fixed.

5.3 The user interface

As in OpenGL, WebGL does not provide functions for capturing the user's input. Therefore, an external library has to be used to capture user's input or to run a specific function that produces an animation.

For the construction of the user interface of the point cloud visualisation web page, the JQuery

library has been used[19]. The web page contains three sliders that are used to transform the object. One applies zooming in and out and the other two apply the rotation in the x and y axes. In particular the values of the sliders are utilised by a Javascript function and according to their values the projection matrices are transformed. Also, when the AJAX request to the server is performed, a progress bar is appeared on the middle of the web page showing the progress made in downloading the point cloud file. Finally the canvas used for WebGL rendering is placed among the other features of the web page. All those features can be included in an ordinary web page which might not be necessarily dedicated to point cloud visualisation.

5.4 Performance

Although the implementation has been written in Javascript and runs on an interpreter within the browser, the core parts run asynchronously on the GPU. Therefore, if the Javascript code runs fast enough, the performance is expected to be the same as in the OpenGL implementation. The parts of the program that run on the interpreter and are written in Javascript are the creation and manipulation of the transformation matrices and the functions for capturing the user input through the sliders.

As stated before, the WebGL version has to run with a fixed number of iterations. The number that was chosen is 17 iterations for each of background pixels and occluded pixels filling processing steps. It is a sufficiently large number and ensures that the holes will be filled. If the holes are filled in the early iterations, the next iterations will run but will be very light and will not cause a performance loss. Figure 24 shows the performance of the WebGL implementation compared with the OpenGL one. We can see that the two implementations run in the same speed which is about 50 frames per second. The OpenGL implementation uses the atomic counter mechanism explained in section 3.1.2 and thus it does not run redundant iterations and the performance remains stable as long as the properties of the texture are similar. On the other hand, the WebGL runs many iterations and depending on the properties of the textures sometimes might be redundant and sometimes not. In Figure 24 we can see that the performance of the WebGL implementation has many deviations. The deviations in the WebGL performance might not depend on the implementation but on external factors such as the web browser.

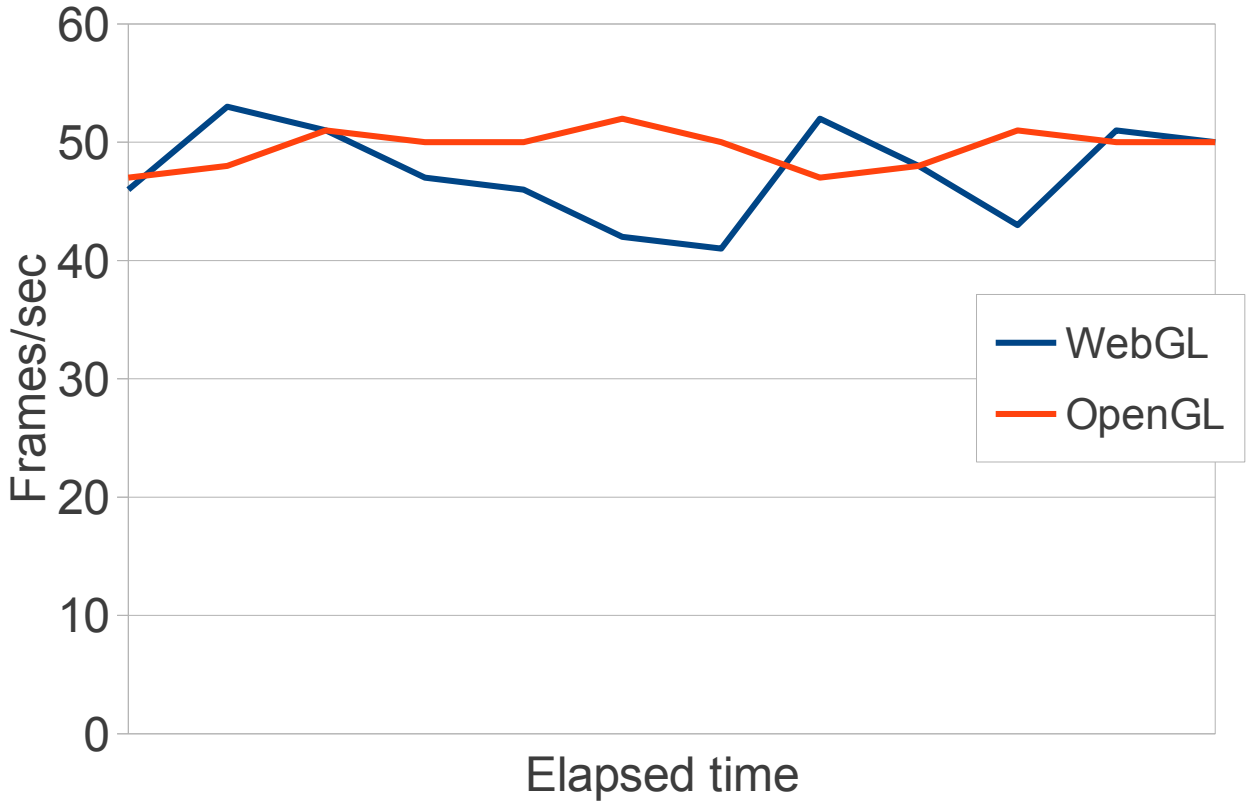

Figure 24: Comparison between the original OpenGL and the WebGL implementations

6 Future work

The primary goal was the implementation of the image-based point cloud visualisation algorithm, optimisation and finally a WebGL implementation of the algorithm. All those goals have been achieved but even more work can be done on both the OpenGL and WebGL implementations.

6.1 Octrees

The performance of the image-based point cloud visualisation algorithm depends mainly on the size of the image or the object size. However, the point magnitude affects the performance of the implementation because all of the points have to be transformed and particularly be multiplied inside the vertex shader. A proportion of them are discarded later in the OpenGL pipeline and therefore, the matrix-vector multiplication of those points is redundant.

Octrees are tree-based hierarchical data structure that provide the benefit of indexing the points based on their position. Points are not stored on the Octrees based on their coordinates but rather on their spatial position in respect to the other points. Therefore, points that are not seen by the camera can be discarded before the render and no redundant vector-matrix multiplications will be issued in the vertex shader.

A proposal for future work would be the utilisation of Octrees. In particular, the points will be read from the file and stored on an Octree. Then, depending on the transformation applied to the object, a particular algorithm can be used to determine which sub-trees have to be rendered. Many algorithms exist for traversing trees and one could be the depth-first search algorithm. The Octree can be stored on the CPU main memory and in every frame particular sub-trees can be sent to the GPU for rendering. Alternatively, another investigation could be to store the Octree on the GPU main memory and eliminate the data transfer from the CPU to the GPU memory. The performance is expected to increase because of the elimination of the redundant vector-matrix multiplications but an overhead will exist in the tree-traversal algorithm.

6.2 Multiple GPUs in a cluster

For all graphics applications, the level of detail of the objects is limited to the number of pixels offered by the monitor. In the image-based point cloud visualisation algorithm one of the disadvantages is that if the camera is far away from the object, fewer pixels are used to project it and thus not much detail can be given to it. Also, the number of points included in the data set can not only make the GPU run out of memory but slow down the performance because of the matrix-vector multiplications that have to be done for each point.

One proposal for future work would be to create a big monitor made up of smaller ones, a multi-tile monitor. Every monitor will be connected to a computer node with a GPU. All the nodes are connected each other creating a computer cluster[20]. One approach would be to make the first render that creates the texture on the front-end and then distribute the texture to the cluster nodes where each node will run the image-processing steps and output its one piece of texture on its monitor. However, the initial texture must be big enough to be distributed and projected by all the monitors in such a fashion and might not be possible to be done.

Another approach would be to distribute the points to the multiple nodes in the cluster and each node will perform both render commands, one to create a texture and one to process it. This solution seems ideal because the front-end will not run out of memory by having all the points stored on its memory. Furthermore, the performance is expected to increase linearly and also the quality of the images would be much higher. While the object is rotated, some points will have to move to another node. Therefore, the points on each node have to be indexed and probably stored on an efficient data structure which could be an Octree. Also, for sending the points between the cluster nodes, two different methods could be investigated. One is to transfer the points from the GPU memory to the CPU and send them to the other nodes, and another way is to use the CUDA GPUDirect function which supports Remote Direct Memory access (RDMA) transfers across an infiniband network between GPUs.[21]

A framework has been developed by the University of California, the CGLX framework, which supports high performance visualisation within a multi-tile environment. CGLX provides an API that gives access to the functionality for managing distributed displays as well as tools for managing clusters. It is closed source and it is supported on Linux and Mac OS systems and it is free for non-commercial purposes. Therefore, this particular framework could be investigated and used for the image-based point cloud visualisation algorithm.[22]

6.3 WebCL

Web Computing Language (WebCL) is a Javascript binding to the Khronos OpenCL standard for heterogeneous parallel programming. It enables web applications to take advantage of the multicore CPU systems and the GPU within the web browser. Currently, there are three implementations of OpenCL. The first is provided as an extension of Mozilla Firefox web browser by Nokia. The second one is implemented by Samsung and it is provided in the WebKit web browser engine and finally the third one is implemented by Motorola and is accessible through the NodeJS Javascript library. All those implementations are not stable yet and perhaps they do not support all the standard functionality.[23]

Therefore, WebCL could give more control to the WebGL implementation by enabling the optimisation done in the OpenGL one as well as more features. In particular, instead of running the second render command that applies the image-processing steps in a loop, the loop could moved

inside the shader program as is done within the OpenGL implementation. In order for this feature to be accomplished, WebCL must provide the same functionality of both reading and writing to textures as the OpenGL image variables do and also a mechanism to to determine when to stop the processing steps.

Furthermore, if both reading and writing to textures is supported by WebCL then the same optimisation applied to the OpenGL implementation can be applied to the WebGL one as well. The texture could me decomposed over the different SMs and allow the cores inside each SM to communicate through the L1 cache instead of the GPU main memory.

6.4 PCL library

Another thing that we had considered was to compare the optimised image-based point cloud algorithm with a conventional implementation of the PCL library. However, we decided not to do this due to time constrains and prioritising and investigating the performance of the OpenGL implementation.

The PCL library abstracts all the low level details of OpenGL and provides functions for visualising point clouds. It uses a conventional approach of visualising point clouds and therefore it would be interesting for future work to make a comparison between the image-based and the PCL implementations.[24]

Conclusion

Throughout this dissertation a comprehensive explanation of point cloud visualisation as well as the image-based algorithm was presented. Point cloud visualisation is used for visualising large data sets containing a large number of points coming primarily from 3D scanners and it is used in many fields such as medicine and geology.

The image-based point cloud visualisation algorithm performs all its computation on image space. In particular it creates an image with the points projected on it and then applies some image-processing steps. Therefore, the performance of this algorithm depends mainly on the image size or window size but the point magnitude can also cause a performance penalty. Although the image-based algorithm performs very well and the quality of the visualisation is high enough, it cannot cope with sparse data sets or when the camera is too close to the object and it becomes sparse.

Furthermore, by accessing the low level features of the GPU, a significant performance improvement was achieved. Because the image-processing steps are iterative the texture had to be stored on a coherent memory which is the main memory of the GPU. For every iteration, the multiple shader instances read a pixel with its neighbours from the texture, applied some particular image processing an then stored the pixel on the main memory. However, the GPU consists of multiple SMs and each SM with multiple cores. All those cores inside an SM share an L1 cache which can be configured to be used as shared memory. Therefore, by using a compute shader, which is a new feature of OpenGL and which provides the functionality of performing general purpose computations, the shared memory of each SMs was used and the multiple cores in each SM communicated through this memory whereby the communication between the SMs was cut off. Also, although the quality of the output of the optimised implementation was not as good as the original one, it is acceptable.

The OpenGL implementation needs to linked with specific libraries and run on a configured environment. On the other hand, WebGL is a stripped down version of OpenGL, can be exposed through a web browser and run on many platforms without having to configure them. The WebGL implementation is similar to the OpenGL one but with some major differences because of the absence of some of the OpenGL functionality. Not surprisingly, the performance of the WebGL implementation is very similar to the OpenGL one because all the computations take place on the GPU and the only part of the program that runs on the CPU are the matrix multiplications.

Finally, future works were proposed. The utilisation of Octrees can be investigated in the OpenGL implementation as well as run it on a cluster. Also, the same optimisation applied to the OpenGL implementation could be applied to the WebGL one by using WeCL. What is more, a comparison with a conventional implementation using the PCL library could be investigated.

Appendix A

In this section, screen-shots of the implementation are shown when running with different data sets. All the data sets were downloaded from the Stanford University's repository (http://graphics.stanford.edu/data/3Dscanrep/) and rights for using for research purposes are given.

A.1 Dragon

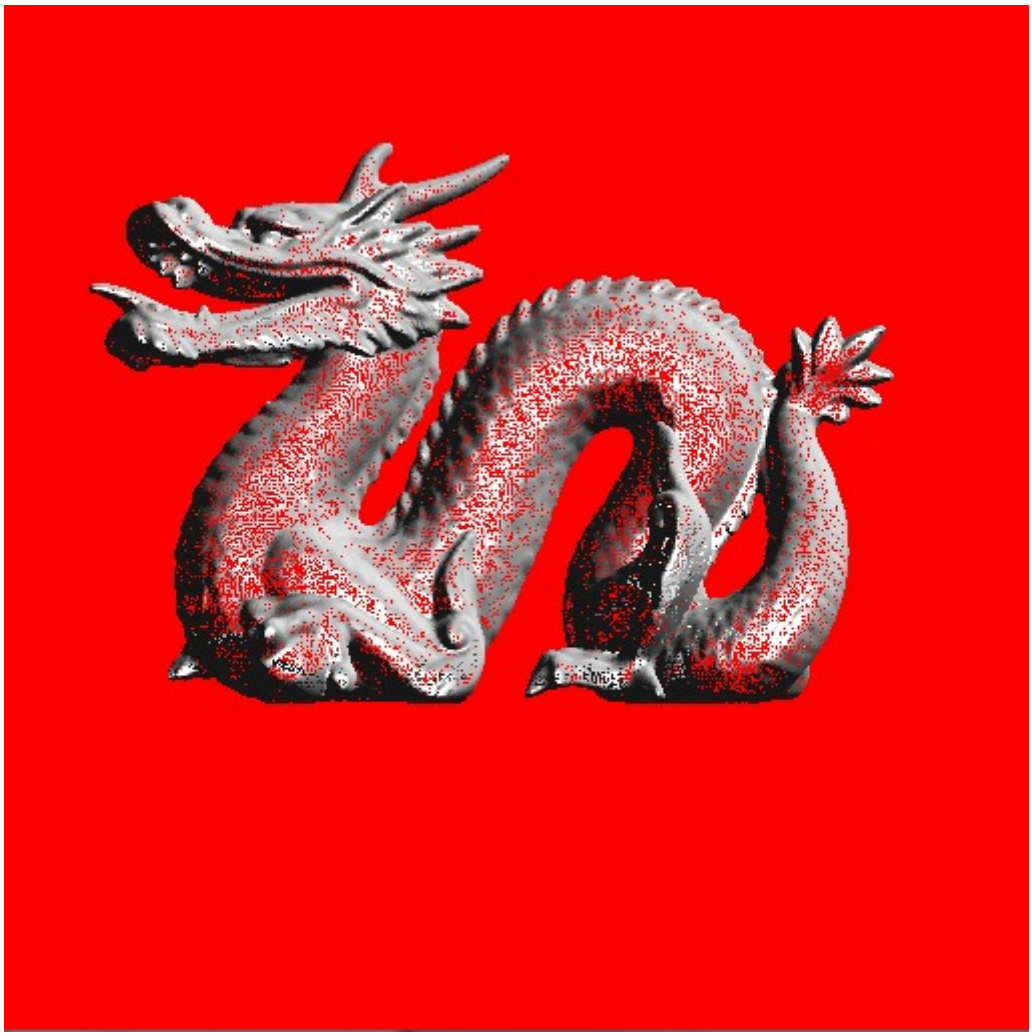

Figure 25: The dragon data set consisting of 566K points without any processing

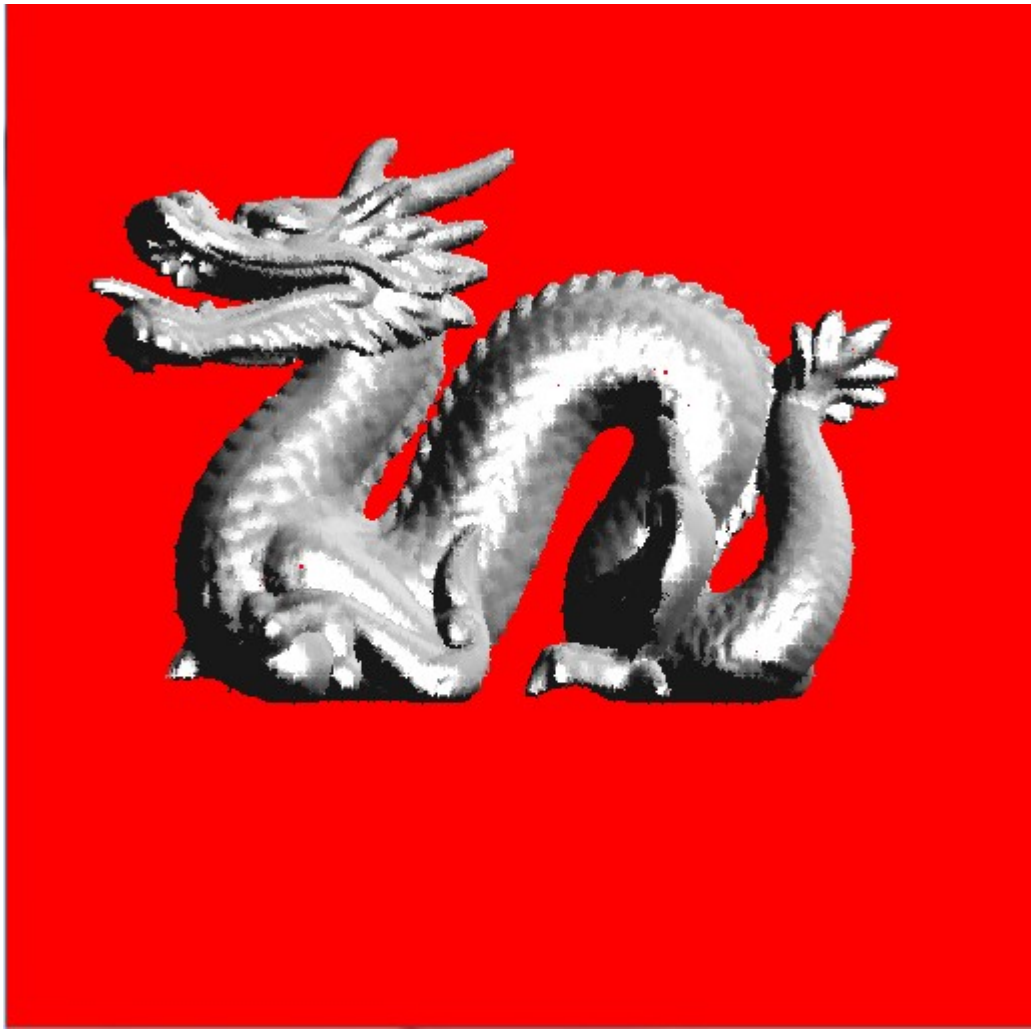

Figure 26: The dragon data set consisting of 566K points after holes were filled

A.2 Happy Buddha

Figure 27: The happy biddha data set consisting of 543K points without any processing

Figure 28: The happy biddha data set consisting of 543K points after holes were filled

A.3 Asian Dragon

Figure 29: The Asian dragon data set consisting of 3.6M points without any processing

Figure 30: The Asian dragon data set consisting of 3.6M points after holes were filled

A.4 Thai Statue

Figure 31: The Thai statue data set consisting of 5M points without any processing

Figure 32: The Thai statue data set consisting of 5M points after holes were filled

References

1. 3D scanner - Wikipedia, the free encyclopedia [WWW Document], n.d. URL http://en.wikipedia.org/wiki/3D_scanner (accessed 17.8.13).
2. Rosenthal, P., Linsen, L., n.d. Image-based Point Cloud Rendering
3. From point cloud to surface: The modeling and visualization problem [WWW Document], n.d. URL http://www.isprs.org/proceedings/XXXIV/5-W10/papers/remondin.pdf (accessed 17.8.13).
4. Hierarchical data structures and algorithms for computer graphics. I. Fundamentals - IEEE Computer Graphics and Applications [WWW Document], n.d. URL http://www.cs.umd.edu/~hjs/pubs/SameCGA88b.pdf (accessed 4.4.13).
5. Documentation - Point Cloud Library (PCL) [WWW Document], n.d. URL http://pointclouds.org/documentation/tutorials/pcd_file_format.php (accessed 8.17.13).
6. PLY (file format) - Wikipedia, the free encyclopedia [WWW Document], n.d. URL http://en.wikipedia.org/wiki/PLY_(file_format) (accessed 8.17.13).
7. Jinbo Li; Xiuli Ma; Yangyang Jia; Xueli Zhou, "3D visualization for heart model from point clouds," Audio, Language and Image Processing (ICALIP), 2012 International Conference on , vol., no., pp.1077,1081, 16-18 July 2012 doi: 10.1109/ICALIP.2012.6376776 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6376776&isnumber=6376576
8. Zhiyong Li; Zuoxun Zeng; Bruhn, R.L., "Using LiDAR Data Visualization to Investigate Origin of Uphill-Facing Scarps in Mountains, Alaska," Environmental Science and Information Application Technology, 2009. ESIAT 2009. International Conference on , vol.1, no., pp.84,87, 4-5 July 2009 doi: 10.1109/ESIAT.2009.93 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5200070&isnumber=5200042
9. LIDAR shows where earthquake risks are highest - seattlepi.com [WWW Document], n.d. URL http://www.seattlepi.com/local/article/LIDAR-shows-where-earthquake-risks-are-highest-1052381.php (accessed 8.17.13).
10. Shreiner, D., Sellers, G., Kessenich, J. and Licea-Kane, B. n.d.. OpenGL programming guide, 8th edition.
11. OpenGL Mathematics [WWW Document], n.d. URL http://glm.g-truc.net/ (accessed 8.18.13).
12. GLUT - The OpenGL Utility Toolkit [WWW Document], n.d. URL http://www.opengl.org/resources/libraries/glut/ (accessed 8.18.13).
13. NVIDIA GPU Programming Guide [WWW Document], n.d. URL http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf (accessed 8.17.13).
14. NVIDIA Fermi Architecture Whitepaper [WWW Document], n.d. URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (accessed 8.17.13).

15. WebGL - OpenGL ES 2.0 for the Web [WWW Document], n.d. URL http://www.khronos.org/webgl/ (accessed 8.17.13).
16. WebGL and OpenGL Differences - WebGL Public Wiki [WWW Document], n.d. URL http://www.khronos.org/webgl/wiki/WebGL_and_OpenGL_Differences (accessed 8.17.13).
17. AJAX Tutorial [WWW Document], n.d. URL http://www.w3schools.com/ajax/ (accessed 8.17.13).
18. three.js - JavaScript 3D library [WWW Document], n.d. URL http://threejs.org/ (accessed 8.17.13).
19. jQuery [WWW Document], n.d. URL http://jquery.com/ (accessed 8.17.13).
20. Hamano, T., Fujihara, R., Yamashita, A., 2009. for SPring-8 Central Control Room.
21. NVIDIA GPUDirect | NVIDIA Developer Zone [WWW Document], n.d. URL https://developer.nvidia.com/gpudirect (accessed 8.17.13).
22. CGLX Project [WWW Document], n.d. URL http://vis.ucsd.edu/~cglx/ (accessed 8.17.13).
23. WebCL - Heterogeneous parallel computing in HTML5 web browsers [WWW Document], n.d. URL http://www.khronos.org/webcl/ (accessed 8.17.13).
24. PCL - Point Cloud Library (PCL) [WWW Document], n.d. URL http://pointclouds.org/ (accessed 8.17.13).