Real-time Rendering of Large Surface-scanned Range Data Natively on a GPU

by

Sajid Farooq

A thesis submitted to the
School of Computing Science
at the University of Glasgow
for the degree of
Doctor of Philosophy

July 2013

Abstract: This thesis presents research carried out for the visualisation of surface anatomy data stored as large range images such as those produced by stereo-photogrammetric, and other triangulation-based capture devices. As part of this research, I explored the use of points as a rendering primitive as opposed to polygons, and the use of range images as the native data representation.

Using points as a display primitive as opposed to polygons required the creation of a pipeline that solved problems associated with point-based rendering. The problems investigated were scattered-data interpolation (a common problem with point-based rendering), multi-view rendering, multi-resolution representations, anti-aliasing, and hidden-point removal. In addition, an efficient real-time implementation on the GPU was carried out.

Acknowledgments: First praise is to Allah, the Almighty, for giving me the strength to complete this study, and the courage to spend years in a foreign country away from family.

I would like to thank my primary supervisor Paul Siebert for introducing me to medical visualization, for his patience, for always giving me his valuable time so that I could continue working, and for his guidance throughout this study. I would also like to thank my secondary supervisor, Paul Cockshott, for his sharp observations, for lending a keen eye whenever I needed a second opinion on my thesis, and providing a critical analysis when I needed it most.

I would like to thank my colleagues at CV&GL: Susanne and Maaha- for being great collegues in general, and guiding me around the department in particular, especially in the first years. Tom and Euan- for the countless intellectual and not-so-intellectual discussions. Paul Kier - for a never-ending supply of information about parrellization, compilers, optimization, and the coolest gadgets. My two Indian friends, Indra Deo and Anand, for allowing me to feel at home, culturally, and linguistically, and finally Gerardo - for being the guy in the lab who is always there when you need him.

I would like to thank my mother for always praying for me, my father for the computer gene I inherited (i'm sure there is such a thing...), my sisters who always told me to *hang in there*, and all of my friends in Glasgow who made living in Glasgow such a joy.

Finally, I would like to thank my wife for simply putting up with me (I know its hard...) and being the best wife one could wish for, my beautiful daughter Taqwah, who dissipated all my worries whenever I set my eyes upon her, and last but not the least, my son Mu'min for providing the much needed break from the thesis by intermittently threatening to unplug the computer (or internet) while I was working.

Contents

List of Figures

Chapter 1

Introduction

1.1 Aims

This thesis presents progress towards a real-time multi-resolution, multi-view point-based rendering architecture that permits display of large matrix-based 3D data sets for medical visualisation, particularly human anatomy data, at native imaging resolutions that is well adapted to the capabilities of modern GPUs. The algorithm we propose obviates the need for lengthy preprocessing of input data such as explicit surface reconstruction, and hence is suitable for real-time scanning and streaming applications. In addition, I show that the algorithm makes use of forward projection, avoiding backward projection, making it compatible with the capabilities of GPUs, and keeps *neighbourhood access* in image-space to a minimum to reap benefits of the highly parallel nature of modern GPUs.

1.2 Motivation

The rapid evolution of imaging and scanning technology has resulted in a proliferation of 3D scanning devices in the market [46]. 3D data has the potential to provide clinicians with an objective set of methods of assessing and measuring 3D surfaces . In the past clinicians were forced to resort to subjective measures such as relying on the naked eye, and carrying out surgical decisions based upon that data. Today, 3D scanned images of patients can provide unprecedented accuracy, and objective measurements of body surfaces to sub-millimetre resolution. Commercially available stereo-photogrammetric systems such the DI3D surface capture system are capable of capturing 3D scans up to 16 mega pixels in resolution. As these devices grow in number and sophistication, the resultant data they

output improves in quality, and in resolution.

Although improved data (and a higher resolution) is desirable for many reasons, it presents its own challenges. Firstly, large data sets are difficult to manage and process. Large scanned or photographed 2D pictures can usually be displayed rapidly at native imaging resolutions via standard image viewers built into the Operating System, since high resolution scanners and digital cameras have become a commodity consumer item. Home videos generated by high definition cameras can be played back in real-time. In stark contrast, no such tool exists for 3D data. Lengthy preprocessing steps are required to convert data into meshes, point-clouds, or other formats before display. Most of these data formats render at a lower scanned resolution since the data sets are too large to be displayed at full resolution. Planetary data is released by NASA on an almost daily basis, however, while the 2D photographs have gained in popularity and use, the 3D Digital Terrain Models (such as those generated by the HiRISE camera [116]) are still largely unexplored due to the difficulty in dealing with 3D data effectively. It is now possible to capture 3D/4D data in realtime, but the resultant 3D data must be processed offline order to reconstruct a displayable surface, usually a polygon mesh. To that end, we present an algorithm that will render any matrix-based point-sampled geometry (such as range-images, height-fields, digital terrain models), at native imaging resolution, and with negligible set-up time required. We believe our algorithm may find uses in domains where native-resolution rendering coupled with short setup/loading times and/or fast switching between data-sets is necessary such as online 3D catalogues, on-site virtual museums, real-time 3D video, 3DTV etc.

1.3 Historical Context

Cleft lip and/or palate accounts for one in every 600-700 live births in the UK. These children undergo surgical correction of the defects but are often left with significant post-operative surgical and psychological morbidity. To that end, the Computer Vision and Graphics Lab (CV&GL) of the Department of Computer Science (DCS) at the University of Glasgow has been involved in CLEFT 10, a project funded by the Scottish Executive Health Department, involving multidisciplinary assessment of residual deformities following surgical repair of cleft lip and palate. The Cleft 10 Project was a study that aims to assess residual soft tissue deformities and psychological adjustment in a group of 10-year-

old children following repair of cleft lip with or without cleft palate. The results of the study will be used to improve cleft services in Scotland with regard to planning for further surgical treatment and protocols of cleft management, including service requirements for psychological support [100].

This work was carried out in collaboration with Glasgow Dental School (GU, lead partner), Department of Statistics (GU) and the Department of Psychology (GU). The data for the project was obtained by imaging real patients and human models via a high-resolution static 3D capture system called DI3D [101]. DCS was responsible for the operational aspects of 3D face data collection of patient and control subjects and their processing and storage. A pipeline for batch data processing was constructed and the logistics of storage archival and distribution worked out as summarised in Figure 1.1 .

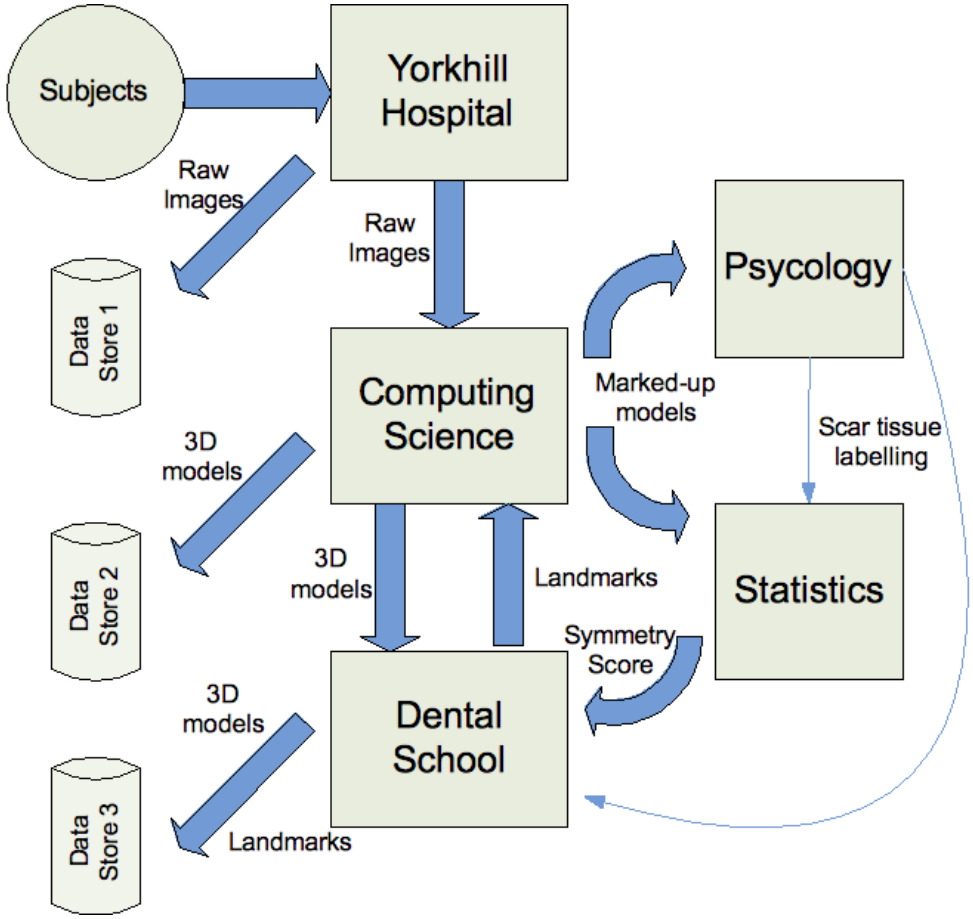


Figure 1.1: Pipeline for data processing, storage, and distribution during the Cleft-10 project

As depicted in the figure, the data was provided to DCS as RAW images, which then had to be converted into 3D models. While accuracy and detail were important, during

the project it very quickly became apparent that the high resolution data sets were too large to be manipulated and displayed at native resolution. In previous studies the pairs of range images representing the captured 3D face were merged into a single 3D point cloud which is then fused into a single polygonal mesh model using a variant of the well known Marching Cubes algorithm. Since polygon mesh representations are inefficient both in terms of storage space and also computations applied to this form of data, it is not possible (nor efficient) using current generations of computer to construct polygon meshes that represent the captured data the at full resolution. In order to make the data more manageable, a compression and conversion had to be carried out by the native (DI3D) software which resulted in a large portion of the data being discarded when converted to viewable formats such as VRML. In previous studies a 90% data reduction was not atypical, achieved by a combination of sub-sampling the range data and decimating the constructed polygon meshes.

1.4 Requirements

I will now detail these requirements:

Interactivity One of the primary advantages of using a computer-based visualisation system as opposed to static pictures is the element of *interactivity*. It was crucial that I design a system that permits clinicians to interact with the model to be visualised in a convenient manner, rotating, zooming and panning the model at will.

Real-time/Speed Traditional methods require preprocessing of data to reconstruct a *surface*. This creates a bottle-neck for applications such as 3DTV where it is now possible to capture live data, but not manipulate this data in real-time. Hence, this reconstruction phase must either be made fast enough to avoid becoming a bottle-neck for real-time applications, or be avoided altogether. In addition, as programmable GPUs become the predominant hardware for rendering, it is not enough to merely modify old CPU-based rendering architectures and make them work on current GPUs. New rendering methods must be proposed that are designed specifically to take advantage of modern GPUs. With the advent of OpenCL, we expect the gap between CPU and GPU rendering techniques to widen.

Visualisation Large 3D data sets are difficult to visualise. There are three overriding concerns that the clinicians presented in terms of visualisation:

- Conversion of the original data into a form suitable for display generally requires a downsampling (and hence degradation) of data. Therefore, *preservation of native samples of the data is a primary concern.*

- Display devices can only represent data at a finite resolution, therefore, if the model to be displayed has a higher resolution than the viewport, it is faster to replace the model with one that has a lower resolution. Consequently, *it must be possible to easily and efficiently represent the data at multiple-resolutions, or at multiple Levels-Of-Detail (LOD).*

- Finally, triangulation-based devices can only recover depth information from a particular *point-of-view.* Generally, multiple captures of different views around the object are necessary. This results in several 2.5D captures, each with a partial view of the object. Hence, *multi-view integration techniques are required to join these partial views into a single 3D representation in an automated manner.*

Upgradeability An important consideration was that although at the beginning of the project, the data was composed entirely of static 3D captures, since 4D capture systems were on the horizon, it was important to maintain a rendering pipeline that made it possible to upgrade a 4D capture system in the future.

1.5 The Proposed Method

In this thesis, we present two algorithms, one based on image pyramids (usually Gaussian) deals with real-time rendering of large multi-view, multi-resolution data sets at native imaging resolutions, whereas the second is based on the Laplacian pyramid that deals with effective multi-view blending.

Both algorithms begin by creating an image pyramid of the source data (range-images, textures, masks). This pyramid is used to produce a series of 3D objects in scale-space. Detailed treatment of both algorithms will be provided in later chapters, however, a diagram of the Laplacian Projection pipeline provides an overview of the algorithm (figure 1.2).

1.6 Hypothesis

Range images are a better native representation for visualising 3D Graphics, especially medical visualisation, as opposed to polygons, or point-clouds, since range-images are the native data format for most 3D capture systems, they are regular, compact, provide connectivity, and allow GPU optimisation due to their matrix-like nature.

1.7 Thesis statement

This is an investigation on real-time visualisation of high-resolution scanned data with demonstrations that preprocessing, and the GPU bandwidth consumption of lossless data, are two significant bottlenecks in state-of-the-art algorithms.

1.8 Contribution

The contribution of this research is listed as follows:

- *Novel scattered-data interpolation mechanism*: Naive point-based rendering will result in holes in the displayed image due to insufficient sampling. A novel hole-filling method is proposed that is designed to be executed in parallel via a shader on the GPU, does not require pre-computation (pyramid generation) to be done on the CPU, and does not rely on expensive backward projection algorithms such as ray-casting.

- *Extension of Burt and Adelson's multi-resolution Spline to 3D*: While the image mosaic as proposed by Burt and Adelson is an accepted method for seamlessly joining multiple images, I am the first to extend the algorithm to merge two 3D models in image-space, rendered in real-time. I call this Laplacian Projection.

- *Novel use of GPU memory to compactly store a multi-resolution 3D model*: Owing to the grid/matrix nature of range images, the points in a range image are offset by a fixed linear increment on the horizontal and vertical axis. Only the depth value (z-value) changes unpredictably. I propose to store depth information via a floating-point texture, and provide the x, y values as re-usable indices where the offset is calculated on the GPU in a shader. This saves GPU memory bandwidth considerably, the actual savings depending on the size of the indices array. In addition, this

floating-point texture is pyramidised (mipmapped) on the GPU with little additional processing time, which also saves GPU memory bandwidth considerably.

- *Real-time streaming of range images at native resolutions*: The proposed architecture makes it possible, for the first time, to render 4D data sets at native resolutions where frame-to-frame coherence is not necessarily available or predictable.

- *A working demonstration system*: I have provided an implementation of a complete system that may be used to visualise range images that practically illustrates the use (and performance) of the proposed algorithms.

1.9 Thesis Plan

This report consists of eight chapters:

1. This chapter introduces the aims and objectives of the thesis, and presents a road map for the rest of the chapters.

2. In this chapter, I present a literature review of state-of-the-art in point-based graphics techniques. I presents the traditional methods, their limitations, and how they have been met by the state-of-the-art point-based rendering techniques, if at all.

3. In this chapter, I talk about the data, its nature, and how it is obtained, i.e, the workflow.

4. This chapter revises some mathematical prerequisites, and then details Pyramidal Projection, the proposed method for real-time rendering of surface anatomy data. I also discuss the merits of the algorithm, and present a case for how it reduces memory usage compared to other algorithms.

5. This chapter explores anti-aliasing, and explains how anti-aliasing is carried out in our algorithm.

6. This chapter extends the proposed method to handle multiple views. In addition, an *offline* multi-view blending algorithm the author calls Laplacian Projection is introduced and discussed. This chapter also discusses hidden surface removal, and how it is relevant to our work. An experimental framework for hidden point removal is also presented.

7. In this chapter, I present the results achieved with the proposed algorithms, and validate them.

8. In the final chapter, a summary of the highlights of the proposed method is presented, and a discussion is presented on wether I met the objectives I set out to achieve at the start of the research.

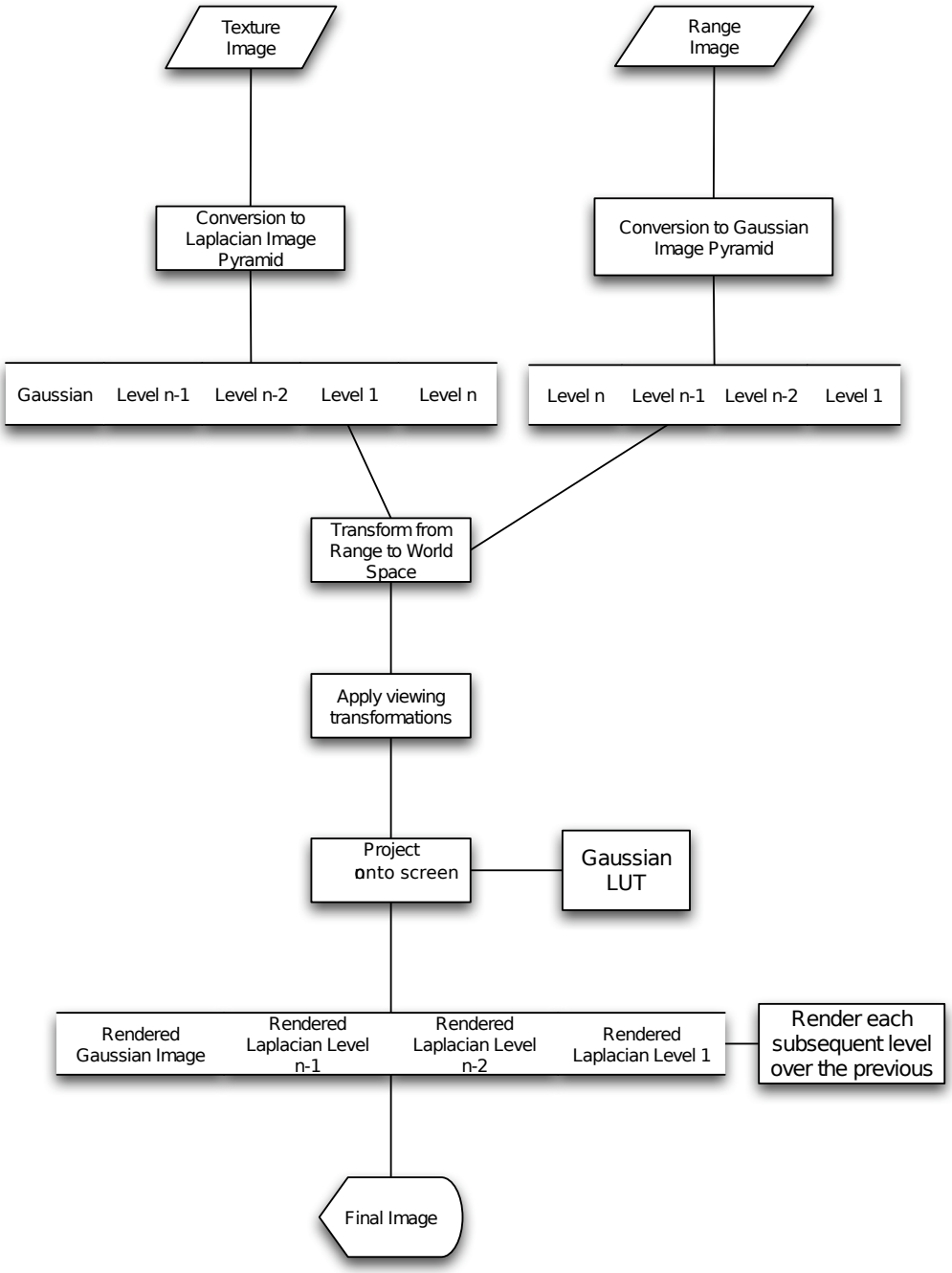


Figure 1.2: Overview of Laplacian Projection

Chapter 2

Literature Review

2.1 Historical Perspective

2.1.1 Motivation

Graphical representations may be broadly categorised into geometry-based, versus sample-based representations [70]. More fundamentally, in two dimensional graphics, this distinction gives rise to vector graphics (geometry-based) and raster graphics (sample-based). Geometry-based representations generally have the advantage of having smaller memory requirements, maintaining connectivity information, and being resolution-independent. Sample-based representations have the advantage of being faster to process, and easier to manipulate.

There is a basic set of assumptions, however, behind the underlying advantages of each, and a violation of these causes a role-reversal. The advantages of each is tied inextricably to the display device. Display devices are naturally limited in *resolution*, i.e, a finite number of samples may be displayed. The advantages of geometry-based graphics assume that the geometric primitives occupy an area larger than the size of a single output sample. When the size of a geometric primitive reduces to the size of an output sample (a pixel), it is more memory efficient to simply replace the entire geometric primitive with a single point [73]. Therefore, increasing sample density favours sample-based representations.

Graphical data may be generated in many ways, including procedural techniques, however, the use of scanning devices that capture real-world data present special challenges above and beyond those associated with traditional data sources [2, 46, 92, 152]. Data obtained from scanners is dense (and thereby difficult to process), and contains outliers.

In addition, there have been few successful techniques for accurately merging data from multiple views (multiple stereo cameras, for example). As the output from these devices grows in size and complexity, it becomes more practical and attractive to move away from geometry-based representations to sample-based representations to store these large data-sets. For 3D data, this has resulted in an increasing move towards Point-Based Graphics [148] [144] [103] [137] [4] [71] [11].

2.1.2 Particle Systems

Point-based graphics are essentially a sample-based representation for 3D graphics. Historically, it was realised early during the development of computer graphics as a discipline that while a geometric representation excels at representing solid objects, it is harder to depict objects with *soft* forms, or those without a clearly defined surface such as fire, clouds, or smoke. The earliest use of a sample-based representation for depicting 3D graphics, therefore, was to model such fuzzy objects. One of the earliest uses of points was by Csuri et al to model smoke [45]. Though the point based representation of smoke was only part of a larger body of work done by the authors, the animation of smoke arising from a chimney, consisting of 300,000 points, established points as a viable data-representation for soft surfaces. The paper lay the foundations for what would later become *Particle Systems*:

"..points comprising the object can be treated as a separate data entity, and will have associated with it properties of intensity and chromaticity, position, and orientation, as well as any properties necessary for animation purposes. The main motivation for this investigation is that non-solid objects can be more accurately represented, with more realistic visual cues, such as the billowing of clouds, the dancing of fire, or the flowing of water." [45]

James Blinn, while at the Jet Propulsion Laboratory (California, United States), successfully modelled other clouds for rendering planetary objects [13]. Reeves consolidated, systematised, and expanded upon previous literature on rendering of soft forms using points and named such algorithms *Particle Systems* in his seminal paper "*Particle Systems: A Technique for Modelling a Class of Fuzzy Objects*" [135].

2.1.3 Point-Based Graphics

While particle systems were adequate, in fact perhaps the *only* method, to appropriately model soft forms, it was the pioneering work by Levoy and Whitted in 1985 [96] that finally established points as a viable *general purpose* representation for the modelling and

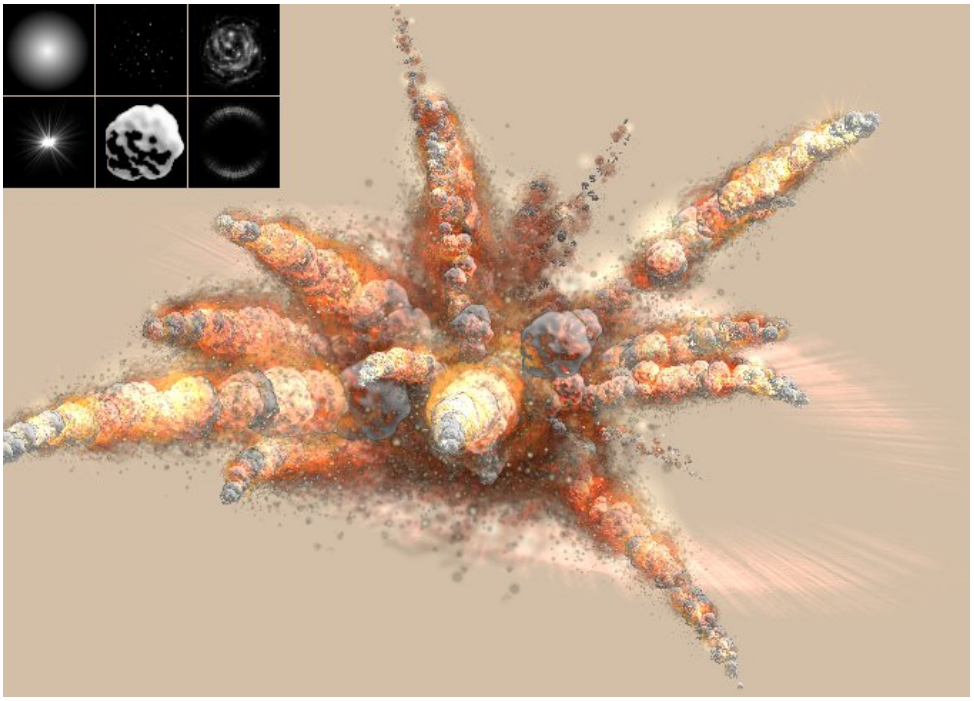


Figure 2.1: An example of a soft form (explosion) being modelled and rendered using a particle system. Each point is displayed as a small image. The images at the top left are used to model the individual points [117].

rendering, something that was hitherto a forte of the ubiquitous *polygon* (a geometry-based representation).

The original pioneering work done by Levoy and Whitted on point based graphics proposes a two step process: First convert a model into a point based representation, and then proceed to render those points. The rendering algorithm is similar to polygon rendering in as much as a point is a $[x, y, z, h]$ tuple much like a vertex in a polygon. Each point goes through the rendering pipeline much like a vertex would in polygon rendering: It is projected, clipped, and shaded. So far, the sample (or raster) based data has been treated as if it were vector data. The point spread function of a single point projection is approximated with a Gaussian kernel to provide a smooth roll-off at the edges. Since the pixels may overlap, depending on the viewing angle, the pixel densities would accumulate where pixels overlap, causing an undesirable pattern. The pixels are therefore spatially normalised.

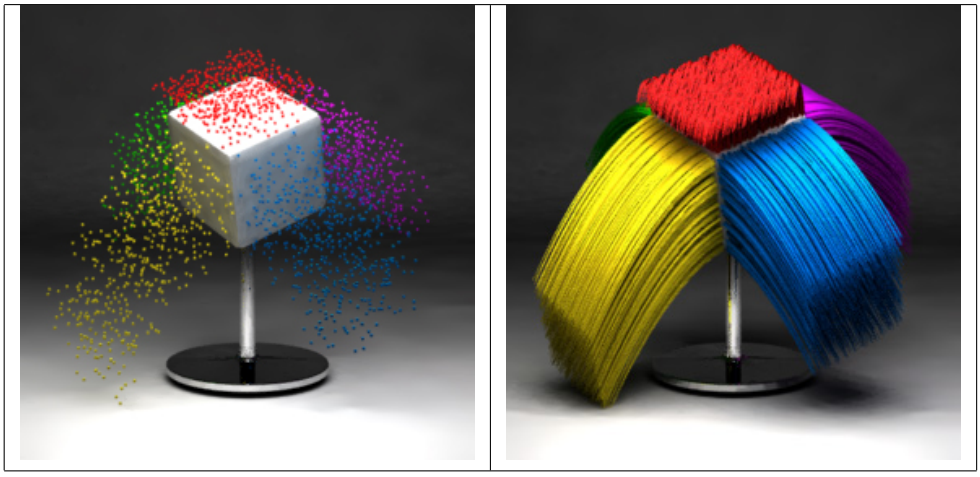


Figure 2.2: Particles systems, with a standard particle emitter (left) and a particle systems emitting "strands" (right) [117]

2.2 A review of Modern Point-Based Rendering Algorithms

The recent resurgence of point-based rendering is attributed to two popular point-based rendering systems proposed at around the same time: QSplat [144], and surface splatting [126]. The various categories of point-based rendering algorithms may be broadly classified as follows:

- Bounding-Sphere based algorithms [49, 123, 144, 158]
- Surface Splatting [70, 88, 103, 106, 164, 175, 176]
- Implicit Surfaces [35, 137]

It is worth noting that I have chosen to omit discussion on voxel/volumetric methods. This is for two reasons: Firstly, voxel-based methods are generally considered distinct from point-based rendering methods, and as such are not suitable for inclusion in a discussion exclusively on PBR methods. Secondly, voxel/volumetric rendering methods are generally not suited to rendering of surface data. They excel at rendering subsurface/volumetric data, whereas the data under discussion comprises surface scans, originating from range images.

2.2.1 Bounding-sphere Algorithms

In point-based rendering, bounding-sphere algorithms can be traced back to the pioneering algorithm QSplat [144]. It is necessary, therefore, to discuss QSplat in some detail, since

other methods derived from QSplat share the same underlying principles [49, 123, 144, 145, 158].

QSplat was originally designed during the course of the Digital Michelangelo Project to render the multi-million point-sampled models being scanned and produced as part of the project. QSplat was designed to display large geometric models obtained from the project, in real time. Since QSplat forms the basis for other inspired hierarchical point-based rendering algorithms, it is necessary to describe briefly its implementation. At its simplest, QSplat is a level-of-detail rendering algorithm. It uses a hierarchy of bounding spheres for visibility culling, level-of-detail control, and rendering. The tree is built in a preprocessing pass as described in algorithm 2.1.

Algorithm 2.1 The algorithm describing the contraction of a tree in a preprocessing pass in QSplat. Reproduced from [144]

```
BuildTree(vertices[begin..end])
{
if (begin == end)
        return Sphere(vertices[begin])
else
        midpoint = PartitionAlongLongestAxis(vertices[begin..end])
        leftsubtree = BuildTree(vertices[begin..midpoint])
        rightsubtree = BuildTree(vertices[midpoint+1..end])
        return BoundingSphere(leftsubtree, rightsubtree)
}
```

Each node of the tree contains the sphere centre and radius, a normal, the width of a normal cone , and optionally a colour. After the construction of the hierarchy, algorithm 2.2 is used for display.

Algorithm 2.2 The basic algorithm used for rendering in QSplat. Reproduced from [144]

```
TraverseHierarchy(node)
{
if (node not visible)
        skip this branch of the tree
else if (node is a leaf node)
        draw a splat
else if (advantage of recursing further is too low)
        draw a splat
else
        for each child in children(node)
                TraverseHierarchy(child)
}
```

QSplat is a recursive algorithm, and a heuristic is used to determine recursion depth. The recursion depth decides the level-of-detail rendered at any given instant. As the camera rotates, the recursion depth is reduced to provide faster updates, but a courser model. When the camera stops moving, QSplat proceeds to refine the model until it renders the finest level of detail. This dynamic modification of recursion allows QSplat to always remain interactive.

Streaming QSplat [145]adds view-dependent network streaming to the QSplat algorithm, thereby making it possible to stream data on-demand over the internet. Generally, transmission of large 3D datasets is made possible by a employing mesh-simplification methods and progressive transmission. Streaming QSplat, however, makes use of points as a display primitive. The immediate advantage is that points may be rendered without connectivity information (as point-clouds), thereby reducing memory requirements, and making it simpler to transmit data progressively via a hierarchical data structure such as one adopted by QSplat.

QSplat, though real-time, is inherently a CPU-based algorithm. The level-of-detail structure employed by QSplat generates the point-cloud to be rendered without batching, i.e, a point at a time [70]. Mapping such an algorithm to the GPU would require *immediate mode* rendering rather than permitting the use of more modern batching schemes such as vertex buffer objects. For large datasets, this is particularly problematic since the GPU spends most of its time waiting for new data rather than rendering existing data. **Sequential Point Trees** [49] further QSplat by providing a data structure that permits offloading adaptive rendering of point clouds to the GPU. Like their predecessor, QSplat, Sequential point trees are based on a hierarchical point representation, however, the nodes of a hierarchical point tree are rearranged to a sequential list, such that all points that are typically selected during a hierarchical rendering traversal are densely clustered. The cpu-based hierarchical rendering traversal is replaced by sequential processing on the GPU. In addition, SPTs make it possible to integrate a point-based rendering algorithm with that of triangle rendering.

Attempts by Dachsbacher et al [49] to render *fuzzy splats* either failed or resulted in large frame-rate drops. Therefore their implementation of Sequential Point Trees can only render opaque squares (which is suitable for small points only), and hence, lacks anti-aliasing. SPTs suffer from memory availability issues as the entire LOD point-hierarchy must fit in video memory. In addition, the approach undertaken by SPT makes it impossible for any

visibility culling to be performed before caching and GPU processing. Therefore, one of the most appealing aspects of QSplat, and one that makes it efficient - branch skipping - is not possible with SPTs [49, 70]. **XSplat** [123]remedies these problems by enhancing the sequential data arrangement by two different interleaved sequential orderings: one in space for individual points and one in the LOD-metric for blocks. It takes advantage of the fact that individual points in an SPT LOD hierarchy are based on a point-cloud without connectivity information. Thus, points may be processed out of order, and that culling and LOD information may be evaluated independently for each point. In addition, XSplat takes all memory levels of the system into account for caching data and allows for visibility culling. In another extension - **view dependent sequential point trees** [158]- two types of indices are constructed to permit rendering in an order mostly from near to far and from coarse to fine. As a result, occluded points are culled in a view-dependent manner efficiently on the GPU while preserving the advantages of sequential point trees. Wimmer and Scheiblauer take sequential point trees in a different direction with **instant points [163]** by compromising rendering quality for speed. Instant points focus on the need for rendering that foregoes preprocessing that is generally required for point-based data. A modification of SPT is proposed: Memory optimised sequential point trees (**MOSPT**), a version of SPTs improved for unprocessed point clouds. In addition, they propose nested octrees, a structure that makes it possible to perform out-of-core rendering, as well as containing MOSPTs as elements.

Richter and Döllner [140] present an out-of-core method for rendering massive point-clouds primarily obtained via LiDAR capturing methods. The method makes combined use of an oct-tree, similar to surface-splatting, while preprocessing data in a manner similar to QSplat variants. The data structure is calculated and serialised during a preprocessing step similar to QSplat. Where QSplat stores one point per leaf node, Richter and Döllner allow for more than one point to be stored in the leaf nodes, helping prevent deep trees. The preprocessing phase also takes into consideration available RAM and makes sure that the data-structure contains enough data per node to allow maximum throughput of data while preventing an overflow of GPU memory.

Problems with state-of-the-art in bounding-sphere algorithms

Bounding-sphere (or in fact any bounding-volume) algorithms require a partitioning of the 3D model into a spatial hierarchy. This is an expensive per-primitive computation that is

performed in a pre-processing pass. This limits *streaming* applications, where 3D data is encoded as multiple frames of the same object at different poses, and new data must be streamed dynamically. In such a case, each new frame would require the pre-processing to be performed afresh.

Another problem with bounding-sphere algorithms is their inherently CPU-based approach. The hierarchical data structure proposed by QSplat cannot be efficiently stored in GPU RAM because the GPU can only store data in either vertex memory (a linear array) or texture memory (a 2D array). Traversal of such a hierarchy requires constant CPU intervention, which leaves GPU performance in a less than optimal state. Converting this to a linear data structure such as Sequential Point Trees [49, 158] mitigates this problem to a degree, but still do not solve the aforementioned problem of multi-frame data where data is different every frame.

The proposed method address both of these problems. It performs preprocessing in real-time on the GPU making pre-processing delays negligible, stores data in a 2D array so it is optimally stored on the GPU, and foregoes CPU intervention for data traversal.

2.2.2 Splatting Algorithms

The most common point-based rendering implementation currently in use is splatting [52, 70, 87, 103, 148, 175, 176]. Splatting is a simple yet efficient method for rendering point-sampled geometry, both surfaces and volumes, however it is more commonly used for rendering surfaces [160, 175]. *Surface* splatting is a forward-projection algorithm that uses standard z-buffering to perform hidden-surface removal. In surface splatting a 3D object is represented as a collection of samples of its surface. These sample points are reconstructed, low-pass filtered and projected to the screen plane [133].

2.2.2.1 Core algorithm

Naive point rendering would perspectively project each point on an image plane, and assign the nearest pixel the colour of the projected point. Point clouds, however, lack the connectivity information otherwise associated with polygon meshes, and thus naive point projection inevitably leads to holes due to inadequate sampling. On the other hand, if multiple points occupy a single pixel, this makes the rendering order-dependent. Splatting solves these problems via interpolating the colour of each point over a finite region in image space. These finite regions are known as *splats* (disks or ellipses in object space). In

addition to properties common to all points (such as location, colour, normals), these splats are composed of two tangent axes (u,v) and corresponding radii that define their spatial extent. A preprocessing pass makes sure radii are chosen so the splats overlap in object space sufficiently to guarantee a watertight rendering without holes or gaps in between samples. The precise interpolation method used, and the support region is calculated via a footprint function. For anti aliasing, an elliptical Gaussian reconstruction kernel is assigned to each splat, which results in an elliptical projection, called a *footprint,* in image space. This is similar to the *footprint* projection as proposed by Westover [159,160]. The footprint function associated with a point, in effect, scatters the energy of a point to neighbouring points. As each splat is projected, the colour contributions of each of the overlapping splats are weighted, accumulated, and normalised. This results in a smooth surface reconstruction in image space, thus solving the scattered-data interpolation problem, and depending on the footprint function, also performs anti-aliasing.

Using the standard mathematical notation for splatting, as defined in [70], we will denote a grayscale (scalar-valued) image by a function $\phi(x, y)$. An output image produced by rendering with the point splatting algorithm may be described as equation 2.1 where ρ_i represents a footprint function associated with a point indexed by i, and c_i is a grayscale value associated with the aforementioned point.

$$\phi(x, y) = \sum_i c_i \rho_i(x, y) \tag{2.1}$$

Equation 2.1 , however, does not guarantee that c_i will be normalised, hence, the basic splatting algorithm is extended as in equation to include normalisation.

$$\phi(x, y) = \frac{\sum_i c_i \rho_i(x, y)}{\sum_i \rho(x, y)} \tag{2.2}$$

The algorithm described by equation 2.2 can be implemented as a two-pass algorithm. In the first pass, all the points are traversed and their footprint functions ρ_i and shaded values c_i are calculated. The footprints are rasterised and the contributions stored in a buffer. At this point, the buffer stores an image equivalent to that produced by equation 2.1, along with depth values, and weights $w(x, y) = \sum_i \rho_i(x, y)$. In the second pass, all the pixels in the buffer are traversed each pixel is normalised by the corresponding weight.

2.2.2.2 Splatting algorithms

Surface splatting is an effective point-based rendering technique. It is discernible from the earlier discussion that the quality of a footprint function plays an important role in the quality of the final image. Designing suitable footprint functions, therefore, is an important element of splatting algorithms. **EWA splatting** [174, 176]was designed to provide high-quality anti-aliasing of point-sampled surfaces. The footprint function consists of elliptical Gaussian kernels (along with effective low pass filtering) to provide anisotropic texture filtering, in a manner similar to Paul Heckbert's EWA filtering [79], from where it takes its name. **EWA *volume* splatting** [174]permits rendering of volumetric data (in a manner similar to Westover [159, 160]) in addition to point-based data.

The core splatting algorithm as proposed in Zwicker er al. [175] is purely a CPU-based rendering algorithm. This limits its performance to 2 million splats/sec as measured on a 3.0 GHz Pentium 4 CPU. Bosch et al. in their paper **Efficient high quality rendering of point sampled geometry [22]** propose a highly efficient hierarchical data structure based on an octree to optimise rendering performance. Bosch et al go on to show show that their representation is optimal with respect to the balance between quantisation error and sampling density. One of the primary methods by which rendering efficiency is gained, is the reduction of computation required for point projection by the clever use of the proposed hierarchical representation. While the perspective projection of a 3D point would generally cost 14 additions, 16 multiplications, and 3 divisions per point in homogeneous coordinates with a 4×4 matrix (and the divide by w), the proposed method reduces this to 4 additions, no multiplication and 2 divisions per point [22]. This algorithm is still CPU-based, however, performance is improved to 10 million splats/sec as measured on a 3.0 GHz Pentium 4 CPU [22, 70].

One of the earliest attempts to use the GPU to accelerate splatting algorithms is published by Ren et al [136], known as **Object space EWA surface splatting**. An object-space formulation of the EWA filter is provided, making it amenable to hardware acceleration via traditional triangle-based rasterisation hardware. The technique is described as a two-pass algorithm, in effect an emulation of an A-Buffer. The first pass renders each surfel into the Z-buffer as an opaque polygon. This determines visibility (and z-ordering) of the surfels in the next pass. In the second pass, the object space EWA resampling filter is simulated via a polygon with a semi-transparent alpha texture, and then projected in screen space yielding the screen space EWA resampling filter, also known as

an EWA splat. During rasterisation, the Z-buffer generated in the first pass determines whether a splat is visible, ensuring only the splats closest to the viewer are accumulated. The algorithm renders between 1.6 to 3 million points per second with object level culling enabled, on an ATI Radeon 8500 graphics processor and a 1GHz AMD Athlon system with 512 MB memory. Another approach also based on an object-space interpolation scheme is **Confetti** [122]. Among other things, Confetti includes a hardware accelerated rendering algorithm based on texture mapping and α-blending as well as programmable vertex and pixel-shaders.

A problem with the object space EWA approach is that each point must be rendered as a polygon, increasing the number of points four fold [19]. Botsch et al propose the use of axis-aligned (image-space) squares for rendering points [19]. The *point size* is adjusted inside a vertex shader, and pixels outside the splat are discarded using either an alpha test, or a *KILL* (or discard in GLSL) shader command. This results in an elliptical splat, obtained from an image-space square that is described via a single point. On modern hardware, such an image-space square is known as a *point sprite* [15,132]. It should be noted, however, that the paper makes use of point sprites for an affine projection, rather than a true perspective projection. Thus, while the splat centre is correctly transformed, the outer contour is not, resulting in small holes in the final rendering [70]. **Perspective accurate splatting** [177]attempts to solve these problems by using an affine transformation that correctly transforms the outer contour. However, the method employed results in projection errors in the splats interior, as well as having high computational complexity, severely limiting rendering performance. An improved, and more efficient, method based on local raycasting to determine true perspective is presented by Botsch et al. in **Phong splatting** [21].

With improving hardware, features such as multiple render targets with true floating point precision and blending make it possible to implement all the computations required for surface splatting directly on the GPU. Botsch et al. [18], in their paper **High-quality surface splatting on today's GPUs**, present an entirely hardware accelerated deferred shading framework and a simple but effective approximation of the EWA pre-filter. Chen et al. present an adaptive algorithm that makes it possible to render hardware accelerated volume graphics based on EWA splatting by storing splat geometry and 3D volume data locally in GPU memory [34]. Neophytou and Mueller propose to accelerate the traditional image-aligned splatting scheme [160] that helps avoid blurring of zoomed-in views, by utilising the GPU [113]. Weyrich et al. continue the hardware acceleration approach, and

in fact, go so far as to design their own point-based rendering hardware in the form of FPGA and ASIC chips [161]. Their technique is also novel in that it provides seamless integration of the point-based graphics architecture into a conventional, OpenGL-like graphics pipeline, so as to complement triangle-based rendering.

Hübner et al extend the splatting algorithm to accommodate multiple views, a method based on deferred blending that does not resort to multiple passes [88]. Their method exploits the programmability of modern graphic processing units (GPUs) for rendering multiple stereo views in a single rendering pass. The views are calculated directly on the GPU including sub-pixel wavelength selective views.

Splatting has been integrated with shading algorithms traditionally designed for use with polygon based rendering systems, such as Phong shading [21]. In addition, Splatting has been used to simulate physical properties generally associated with cameras such as motion-blur that is caused by long shutter speeds [81].

Problems with state-of-the-art in Splatting algorithms

Splatting is arguably the most popular point-based rendering solution [87,88,136,161,174], however, splatting algorithms are strongly coupled with unorganised point-clouds as an input data-structure. Unorganised point clouds lack connectivity information, which makes it difficult to determine surface structure without lengthy pre-processing. For stereo-capture systems, connectivity information exists. Stereo-systems produce range images from photographs, which are matrix data-structures, and hence retain connectivity information. Discarding this connectivity information, I believe, is the wrong approach to take.

The proposed method makes use of the matrix data structure offered by stereo-capture systems to efficiently store data on the GPU, and avoid having to deduce surface structure in a separate pass. In addition, range images are easier to manipulate than point-clouds. Range images are natively a matrix data structure. A mere pointer is required to the original data structure during a CPU-to-GPU memory transfer, and a block transfer is performed in one operation via function calls like *glBufferData*(*). Point-clouds, on the other hand, must be converted to an appropriate data-structure on the CPU before being handed over to the GPU.

2.2.3 Implicit Surfaces

Points have been used as a fundamental data representation method for surfaces, defined both implicitly and parametrically.

Implicit surface methods revolve around the idea of a function $f()$ on all of 3D space that produces a continuous surface that may be sampled and rendered. The function $f()$ is always zero at the surface of the object, negative inside, and positive outside. The function solved for $f(x) \leq 0$ therefore represents a watertight *boundary* of the surface represented by $f()$. The most common categories of implicit surface methods with regards to point-based rendering and modelling are Point set surfaces [4], Radial basis algorithms [137] and surface evolution methods such as the Level Set algorithms [35]. Other methods for point based modelling include triangulation methods such as those based on Voronoi/Delauny triangulation methods [70].

Point Set Surfaces [4] by Alexa et al is the seminal paper in the area [35] [5]. In this algorithm, the generation of points on the surface of a shape is represented as a sampling process. Up-sampling and down-sampling the representation increases or decreases the density of points respectively. Point set surfaces are motivated by differential geometry. The goal is to reduce the geometric error by approximating the surface locally via polynomials using the moving least squares (MLS), an algorithm for the reconstruction of continuous functions from unconnected point samples. The reconstruction is computed via a weighted least squares measure, a measure that is biased towards the region around the point at which the reconstructed value is requested. The pioneering work also presents a novel point-based rendering algorithm in order to display point set surfaces.

Progressive Point Set Surfaces [59]were proposed in 2003 by Fleishman et al as an extension to PSS that provide progressive refinement capabilities to the basic PSF algorithm. A projection operator is defined that allows the progressive computation of displacements from smoother to more detailed levels. An algorithm is devised that, based on the properties of the projection operator, allows the construction of a base point set. This based point set becomes the starting point from which a PPSS can be constructed from any given manifold surface via a refinement rule (and the projection operator).

Guennebaud and Gross propose **Algebraic Point Set Surfaces** [75]as an improvement over standard point set surfaces by adapting the moving least squares algorithm to locally approximate the data using algebraic spheres. In a follow up paper [74], Guennebaud et al present a new, more generic solution is simpler and more efficient. One of the main

advantages of the new approach is that it provides enhanced control of the curvature of the fitted sphere and requires only a single intuitive parameter to do so.

In order to make solving PDEs easier over point set surfaces, Qin et al propose a novel *meshless* method for point set surface processing [131]. The method is based on anisotropic diffusion and is notable for introducing the Petrol–Galerkin (MLPG) method into the field of graphics.

Implicit surfaces involve computationally expensive operations, such as the moving least squares approximation, and often the k-nearest neighbours search. Heinzle et al propose a hardware architecture are processing unit for point sets in order to improve the performance of such operations [80]. An FPGA implementation, along with a detailed analysis of performance, is provided.

Moving least squares approaches are, by virtue of the least square approximation, sensitive to outliers. While uniform noise is handled well, and the resultant surfaces are smooth, an assumption is made that the data is sampled from a smooth manifold. In addition, the low-pass filtering process may result in extra smoothing. Öztireli et al attempt to overcome these limitations in their paper [119], via a novel MLS based surface definition, and with the aid of *robust statistics*.

In a scientific computing context, **Radial Basis Functions** are primarily applied in the reconstruction of unknown functions from known data [149]. The utility of radial basis functions in the reconstruction of incomplete 3D geometry was first recognised by Carr et al [31]. Radial basis functions are presented as a solution to the problem of interpolating incomplete surfaces, such as a 3D scan of the human skull. The proposed solution was tested for the design of cranial implants in order to repair holes in the skull. Depth-maps of the skull's surface were obtained from CT scans via ray-tracing, and radial basis functions were then used used to interpolate over the defect regions.

Carr et al further expand upon the existing work by [30] provide efficient methods for fitting and evaluating RBFs that make it possible for the first time to model data sets consisting of millions of via only a single RBF. The algorithm take hole-filling into account, and provides smooth extrapolation of surfaces.

Dinh et al proposed using radial basis functions to reconstruct surfaces specifically from range data, such as those produced by stereo-scanners [53]. The method relies on computing a sum of weighted radial basis functions in order to smooth the noisy range data by reconstructing an implicit surface that is locally detailed, yet globally smooth.

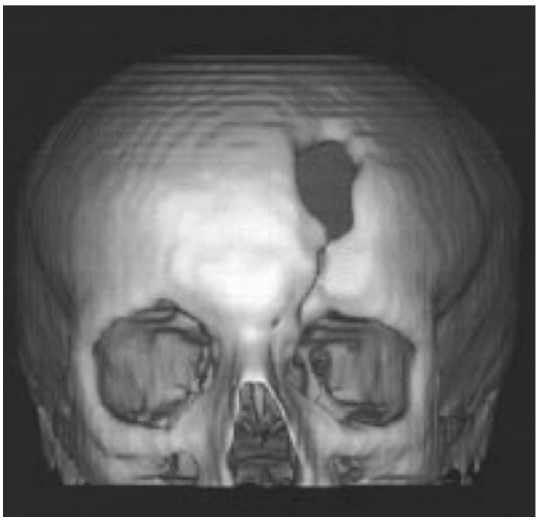


Figure 2.3: A human skull with a defect, rendered via the method described by Carr et al [31]

Since radial basis functions are a meshless method (connectivity information is not required) [149], they can be used to model surfaces where the data only exists as a point cloud. Reuter et al realised this, and proposed using radial basis functions for the modelling and rendering of point-based data in their seminal paper *Point-based Modelling and Rendering using Radial Basis Functions* [138]. The advancement of this paper over Carr et al is the fact that radial basis functions are used for both modelling as well as rendering. The rendering is performed via a bounding-sphere hierarchy similar to QSplat and variants [144].

Botsch and Kobbelt utilise the computational power of the GPU to compute deformation of meshless points in real-time using a special set of deformation basis functions [20]. They report performance of 13M splats per second on a nVIDIA GeForce 6800 Ultra GPU, on Linux. Corrigan and Dinh go further by providing a *GPU* solution for the computation *and* rendering of implicit surfaces [42]. The GPU is used to perform interpolation, weighting, and summation of RBFs.

Wang and Wu propose to deal with the problem of reconstructing a surface with a *smaller* number of RBFs, thereby reducing computation time, and using Orthogonal Least Squares to provide a local method that discards global reconstructions that are impractical. Special care is taken to ensure that the RBF technique is applicable to large point sets [167].

Radial basis functions, though primarily used for meshless surface reconstructions, have also been applied to traditional mesh-based surfaces [24], especially in hole-filling applications [130].

The **Level Set Method** (LSM), first introduced by Osher and Sethian in 1988 [118], is a numerical technique for tracking interfaces and shapes as the evolve. It can be understood as the implicit equation for a closed curve that represents the cross-section of a surface as it evolves along a direction.

In computer science, LSM are used to represent discretely sampled dynamic level sets functions. Bischoff and Kobbelt [12]combine the implicit representation provided by Level Sets with the topology preserving properties of parametric via voxels. *Cuts* are placed on the edges of a voxel grid, i.e, whenever the model nears a topological change, resulting in a sub-voxel accurate reconstruction of the contour.

The level set method was extended to accommodate points by Corbett in 2005 [40]. Instead of using a uniform sampling of the level set, the continuous level set function is reconstructed from a set of unorganised point samples via moving least squares.

Problems with state-of-the-art in implicit surface rendering algorithms

Implicit Surface rendering methods produce renders of very high visual fidelity, however, this quality comes at the expense of computational complexity and performance. Implicit surfaces involve computationally expensive operations, such as the moving least squares approximation, solutions to differential equations, numerical methods, Radial Basis functions, and often the k-nearest neighbours search [4,35,42,75,119,131,167]. These operations are not only expensive, they are not always amenable to hardware acceleration due to their non-parallel nature. Hence, implicit surface rendering methods are suitable where a complete model must be reconstructed from incomplete data, or where achieving the highest quality rendering is of more importance than real-time performance. Additionally, during this research, I have not come across an implicit surfaces rendering algorithm that allows rendering of streaming or 4D data.

2.3 Processing Large Datasets

While modern 3D digital photography and 3D scanning systems have made it possible to capture complex real-world objects with ease, dealing with the resultant data in its entirety is still an active area of research. The scanning techniques generate a large number of point samples, and require efficient strategies to deal with the storage, management, and display

of these point samples.

2.3.1 Acquisition Methods

The most common camera-based acquisition methods use **stereo-photogrammetry** [48]to reconstruct a 3D scene from a pair of images taken from a stereo pair of cameras. In general, and especially for sensitive applications with live subjects such as medical imaging, the benefits of using stereo-photogrammetry over contact based scanners are many. For one, they are comfortable to the subject, since they do not require physical contact, and on the other hand, the scanning procedure is much faster than contact based scanners: The actual capture takes a fraction of a second while most of the processing (recovery of 3d information) can take place off-line [152].

The basic algorithm revolves around using the parallax, and the correspondence, between pixels in a pair of images to recover depth information. Given parametric information about the cameras used, such as focal length, and the distance between the cameras, it is possible to construct vectors (rays) from each camera centre through each pixel on the sensor. The intersection from the rays from corresponding pixels in two pictures makes it possible to *triangulate* the position of a point on an object, and thereby recover its depth.

Stereo-based acquisition systems may be classified into **passive stereo** [77] [48] and **active stereo** [58] [70]systems. The inclusion of a controlled textured light source differentiates active stereo systems from passive stereo systems. The textured light source is not aligned to the cameras and its purpose is to provide additional detail into the scene, making it easier to solve the *correspondence* problem.

It is possible to take active stereo a step further by eliminating the additional camera if the pattern of projected light is known in advance. A system with such a configuration is known as a **structured-light** [152] [90, 120]system. While less costly than general active-stereo due to the elimination of a camera, the disadvantages of such a system are that the projected light pattern must be of sufficient resolution, and that the light must now be aligned to the camera. Such as system is very difficult to calibrate in practice [70] .

If the projected light used in structured-light systems is replaced with a laser, we are led to **light stripe** systems [90,120]. A laser *stripe* is swept across the scene, and captured. The deformation of this slice provides clues to the geometric structure of the scene upon which the laser is shone, assisting triangulation, and thereby, depth recovery. Light stripe systems are naturally slow, since only a stripe is captured at a time. However, lasers are

brighter and more focused than traditional lights, and the physical setup of the scanner in general permits greater accuracy.

Another type of acquisition system that uses lasers is the **Time of Flight** system [151]. Time of Flight refers to the time it takes light to travel from the source to the object being scanned, and back. Much like RADAR or SONAR, given that the speed of light is constant, the distance of an object may be measured via its *time of flight*.

***Pulsed* Time of Flight** [70]relies finding the depth of a single point at a time. A pulse is fired, timed, a mirror then rotates the direction of the pulse, and the process repeats, until the entire scene is capture. Naturally, this is slow, however, these systems are far more practical for much larger scenes where stereo-based systems (or triangulation-based systems in general) would require a difficult calibration.

Where Pulsed Time of Flight sends a brief pulse of light, ***Modulated* Time of Flight** [70]relies on a continuous beam of light from a laser. The intensity of this light, however, is modulated at a high frequency. The phase difference between this outgoing light, and its reflection is dependent upon the distance between the scanner and the scanned object, and is used to recover depth information of the scene.

In addition to the aforementioned methods, numerous methods exist to recover 3D information from a single image, a pair of images, or an entire video sequence such as **Shape from focus** [112], **Shape from Shading** [170], and **Structure from Motion** [10].

For the purposes of this thesis, the primary advantage of using stereo-based systems is that they produce data in a matrix format. Matrices of intensity (colour texture), depth, and masks are returned so that they may later be reprocessed to create either point-clouds or polygon meshes. Its important to note here that the matrix-based data sets returned from stereo-based systems are the *native* data-sets i.e the triangulation process mentioned earlier generates a range image, while polygon meshes and point clouds have to be generated via further processing of this matrix-based data.

2.3.2 Surface Reconstruction

The work described in thesis makes use of stereo-based 3D scanners, i.e, DI3D and C3D. The output from these scanners is in the form of range images. In order to display range images, the data from multiple range images is combined, any holes left by the scanning procedure are filled, and a single *surface* is obtained. This process is known as *surface reconstruction* [44] [70, 120, 148]. Scanned data suffers from various problems such as noise,

outliers, or even missing samples that appear as holes in the data set. The reconstruction process attempts to overcome these faults and produce a *water-tight* model. This may entail removing some points, or adding some, in order to provide sufficient sampling to ensure surface continuity in line with the original surface.

The surface reconstruction phase is also affected by the data representation that will be used to render the data. Polygon meshes contain inherent connectivity information, and water-tight models remain water-tight during rendering. Point-clouds however, will produce holes even for adequately sampled and water-tight models if viewed at a sampling rate higher than the native data sampling rate. The lack of connectivity becomes obvious, and the individual points disintegrate, leaving holes in between.

The most popular surface reconstruction method, marching cubes [102], was proposed by Lorenson and Cline in 1987, as a method for extracting a polygonal surface from a voxel data set. Surface reconstruction is a wide, and well-researched area. I will limit the discussion to methods that involve range images.

Chien et al present a robust method for the reconstruction of a volume/voxel model from concave objects stored as range images [36]. Rutishauser et al provide a method of dealing with occlusion in range images by merging several range images from various views [146]. The technique works with objects of arbitrary shape, and may be updated with additional views, reducing noise in areas of overlap. Turk and Levoy present **zippered polygon meshes**, an *incremental* method for extracting a polygon mesh from range images [155]. The incremental approach permits scans to be acquired one at a time, resulting in minimal overhead, as all the data need not be present at once.

Hilton et al present **Marching Triangles**, an implicit surface polygonisation technique that creates a triangulated model of a manifold implicit surface from range images [82, 83]. Curless and Levoy demonstrate the integration of a large number of range images (up to 70) yielding seamless, high-detail models of up to 2.6 million triangles [47].

Pulli et al focus on the issue of *robustness* during reconstruction [128, 129]. A method is presented that provides robustness via interval analysis techniques, while relying on a hierarchical data structure in the form of octrees for computational efficiency. Reed and Allen propose an incremental reconstruction method that can automatically re-orient the sensor for the next acquisition so that each additional sensing operation recovers surfaces that are occluded, or not modelled [134].

Wyngaerd and Van Gool present a method for the automatic pre-alignment of surfaces,

a task that was previously possible only manually [141, 166]. Wu et al use stereo capture methods to extract the 3D shape of live pigs in order to track their health over 14 weeks [165]. The stereo imaging system does not require structure lighting techniques, and the resultant 3D models of the pigs were reportedly qualitatively good in appearance, and locally smooth, with an RMS deviation of ± 0.6 mm.

Sagawa et al approach the problem of range data processing and merging via parallel computing [91]. They propose a method that speeds up the computation, and reduced memory requirements of, computing signed distances, and discuss a method of parallel computing on a PC cluster. Wand et al present algorithms for processing, and interactively editing large point-clouds that are derived from 3D scanners [157].

Ju et al present a novel range image merging and reconstruction method based on self-correction [93]. They demonstrate that the self-correction approach is capable of repairing a reconstructed 3D surface that has been damaged by depth discontinuities. Li and Wee propose a novel approach to eliminating overlaps that are found in registered data sets [97]. A noteworthy aspect of the algorithm is that it deals with the registered range images natively, i.e, without converting them to polygon meshes first.

Mu et al extend the current literature by augmenting a regular camera with two types of depth sensors in order to recover a 3D surface. [111].

2.3.3 Visualising Large Datasets

Several methods have been proposed to incorporate multi-resolution support into point-based rendering systems. Rusinkiewicz and Levoy describe a system for representing and progressively displaying meshes that combines a multi-resolution hierarchy based on bounding spheres with a rendering system based on points. A single data structure is used for view frustum culling, back-face culling, level-of-detail selection, and rendering [144].

Rubin and Whitted describe exploit a hierarchical bounding volume representation to efficiently solve visibility, and rendering of complex objects [143]. Westover [159] describes a technique that extends Levoy and Whitted's [96] basic point-based rendering algorithm to allow multi-resolution point-based rendering of volumetric data.

Laur and Hanrahan improve the basic hierarchical representation for volume rendering by augmenting it with a pyramidal volume representation, thereby adding an element of *progressive refinement* [94]. Funkhouser et al look at managing the complexity of handling large data sets involved in creating interactive walkthroughs of buildings [63]. The

algorithm uses spatial subdivision based on kd-trees, cell-to-cell visibility analysis, and a LOD database. This work is then furthered by incorporating an *adaptive* strategy that dynamically adjusts the level of detail in a scene to always achieve a consistently interactive frame-rate [62]. A further enhancement, related to the LOD database, is to precompute cell visibility information, and thereby attempt to predict which models would be visible next as the viewpoint changes [61].

While Z-Buffers have become a standard component of modern rendering, Greene et al were the first to employ a hierarchical representation for the Z-Buffer [69]. Rossignac and Borrel use vertex clustering to approximate complex polygonal models at multiple resolutions for fast rendering [142]. Turk and Levoy present **Zippered polygon meshes**, pioneering work on the fusion, and subsequent display, of large data sets (in the form of range images) produced by 3D scanners [155]. Curless and Levoy look at handling large and complex objects via a voxel-based volumetric reconstruction algorithm [47].

Duchaineau present an algorithm to handle large terrains in real-time [56]. The algorithm is adaptive, produces guaranteed error bounds, takes advantage of frame-to-frame coherence typically possible in terrains to render thousands of triangles per frame. Hoppe proposes a method to render large terrains based on locally adjusting the complexity of the approximating mesh to satisfy a screen-space pixel tolerance [86]. In addition, it makes sure that the resultant rendered surface is both spatially and temporally continuous.

Aliaga et al present **MMR**, an algorithm for interactively rendering large data sets [6]. The work is notable in that it employs both geometric and image based techniques to accelerate rendering.

Levoy et al undertake the first large scale digitisation (archival) project, with some models having two billion polygons, known as the **Digital Michaelangelo** project [95]. Levoy et al then extend the basic QSplat algorithm to allow *streaming* of large data sets in real time, an extension known as **Streaming Qsplat** [145]. While mesh simplification and LOD techniques are a well researched area, Cignoni et al focus on the hitherto relatively unexplored aspect of dealing with meshes that consume large amounts of RAM. The solution presented is known as **Octree-based External Memory Mesh (OEMM) [37].** Renato Pajarola attacks the problem of LOD partition in the context of large-scale point based rendering. A spatial partitioning hierarchy is generated via a point-octree LOD generation algorithm [121].

Nuber et al [114]present a method based on out-of-core point-based rendering that

allows visualisation of higher-resolution datasets, providing images similar to texture-based volume-rendering techniques at interactive frame-rates and full resolution. Data is pre-processed by grouping points in the given dataset according to their value on disk and read back, when needed, from disk to immediately stream data to the rendering hardware.

Gobetti and Martin propose **Layered point clouds**, a multi-resolution structure for rendering very large point-based models. The algorithm relies heavily on pre-computation to create a LOD hierarchy, however, it provides network streaming, out-of-core rendering, and is simple to implement [67].

Correa and Rusinkiewicz present out-of-core algorithms to visualise large datasets on consumer PCs. A sort-first parallel extension of the system is presented that uses a cluster to drive a high-resolution, multi-tile screen [41].

Boubekeur et al propose to deal with large datasets via texturing techniques [23]. The scanned model is triangulated at a low resolution, and high-frequency detail is superimposed via a *normal* map that contains the normals obtained from the high-resolution data set. This ensures the model consumes low bandwidth while containing high frequency detail.

Wu et al observe progressive (or continuous) level of detail techniques in polygon based approaches, and transfer this concept to splat-based geometry representations. Their progressive splat decimation procedure uses the standard greedy approach but unlike previous work, it uses the full splat geometry in the decimation criteria and error estimates, not just the splat centres [164]. Yoon et al take advantage of view-dependence in order perform occlusion culling, simplification and out-of-core rendering in **QuickVDR** [169]. The model as represented as a clustered hierarchy of progressive meshes (CHPM).

Borgeat et al present **GoLD** (**G**eomorphing **o**f **L**evels of **D**etail), a view-dependent technique that uses geomorphing to smoothly interpolate between geometric patches, thereby providing continuous level-of-detail, and avoiding popping artifacts associated with other techniques [17]. Wimmer and Scheiblauer attack the problem of long preprocessing times associated with point-based rendering algorithms by proposing **instant points** [163], an extension of Sequential Point Trees [49], that uses nested octrees, and provides out-of-core rendering.

Virtual Inspector is a system that is aimed at permitting non-experts to view dense 3d models at interactive rates on consumer PCs, without sacrificing quality [29]. The GUI is XML-based, and hence easily extendable. It uses **Batched Multi Triangulation**

in order to create a continuous level-of-detail representation [38]. In their paper *Technical strategies for massive model visualization*, Gobetti et al report the state-of-the art regarding rendering large datasets, describing the situation in 2008 [66].

Bettio et al describe a method dealing with rendering large datasets over a network (via a client-server framework), such that dense models may be explored locally and remotely [11]. Du and Li present a method based on image pyramids stored in the GPU to render massive point clouds. It can automatically adjust the output size of the point cloud image according to available memory, while utilising the LOD characteristics of an image pyramid for dynamic resolution switching [55]. Huang et al propose an improved multi-pass GPU-based rendering algorithm that can acquire all visible splats after rasterisation and depth test in the first pass [87]. These splats are then rasterised and computed per-pixel in the second pass. This, similar to deferred shading [73], avoids unnecessary shading computations. **Gigavoxels** is a GPU based voxel rendering algorithm that can display datasets of theoretically infinite resolution [43]. The technique is adaptive, based on an oct-tree for data representation, and raycasting for rendering.

Goswami et al introduce a novel hierarchical LOD structure for rendering large point-based datasets. The LOD structure is based on multi-way kd-trees. The LOD tree is fully balanced and its depth can be controlled. Its notable that the LOD tree contains uniformly sized nodes, which makes memory management simple [68].

Naveen Kumar presents a compact representation for point sampled data using non-linear surface elements, and a method for efficient ray casting of a dynamic surface defined by Metaballs [16]. Richter and Döllner [140] propose to render massive point-clouds obtained with LiDAR capturing methods, using an out-of-core algorithm based on preprocessed (serialised, similar to QSplat variants [49, 145, 158]) data stored in oct-trees.

Schulz et al use point based representation techniques to solve the problem of representing space-filling trees for large hierarchies that occur commonly in life sciences and engineering [150]. Noteworthy is that the algorithm modifies the $\sqrt{5}$-sampling method [153]to effectively solve the problem of hole-filling in a point-based rendering context.

2.3.4 3D-TV and its Challenges

The problems in achieving 3D-TV may be broken down into four distinct components [154]:

1. Capture and representation of 3D scene information

2. Complete definition of digital 3DTV signal

3. Storage and transmission

4. Displaying the reproduced 3D scene

The functional components of 3D-TV make it a research area that overlaps with, and in fact is a superset of many other research problems, such as video-based 3D capture, storage of large data-sets, and novel visualisation techniques for the display of large data-sets associated with 3D capture. In addition to these problems, 3D-TV presents new challenges. Video sequences are bandwidth hungry, and 3D video sequences even more so. Efficient compression algorithms are therefore required to deal with the problems related to bandwidth overload [124, 125]. Often 3DTV systems require real-time 3D capture in order to provide *live* transmission [107]. 3DTV also presents novel visualisation problems, such as the ability to arbitrarily change the viewpoint mid-stream, a problem associated with free-viewpoint television [51, 168], and solutions to view-synthesis [9, 26, 104].

Currently, 3DTV systems may be divided into two systems [51]. One system broadcasts a single view 3D video that is comprised of a video that contains the colour (texture) information, and another signal that contains the depth. This kind of system simulates true binocular perspective, however, provides a limited viewing angle. To counteract this limitation, another system, comprised of multiple cameras, is used. Multiple cameras capture the scene and the user may freely alter his vantage point. This is known as **free-view** or **multi-view** video [51, 124, 168]. Multi-view video requires several views to be transmitted in order to enable the receiver to compute and render intermediate views [26, 104].

The history of modern 3D-TV algorithms for view-synthesis can be traced back to the **Depth Image Based Rendering** (DIBR) algorithms of McMillan et al [108]. The algorithms are based on the premise that the goal of all image based rendering algorithms is to generate a continuous representation of the **plenoptic function** from a discrete set of samples. The plenoptic function is defined as a full spherical map for a given viewpoint and time value, and an incomplete sample as some solid angle subset of this spherical map. Solving the plenoptic function makes it possible to generate novel views from existing views based on a *warping* of one camera view on to another view [33, 108].

Novel views are generally generated by a blending of two images, the left image IL and the right image IR to render a new synthetic view $Inew$ [51]. Zitnick et al extend the basic

algorithm by separating the various elements of IL and IR into separate layers by using image based modelling techniques (such as colour segmentation-based stereo algorithms) in order to improve the generation of $Inew$ [173]. **View-dependent depth estimation** is generally computationally expensive. To achieve real-time performance, Mori et al propose to skip the process entirely [109, 110]. They propose to precompute a depth map for each new image ($Inew$) [109]. This improves the quality of the warped images [51, 109]. This approach leads to new problems that do no arise in algorithms that use view dependent depth estimation. Mori et al address such issues in their work.

Do et al provide present a more rigorous analysis of the quality of warped images for 3DTV in addition to providing a new algorithm that is shown to have superior results to previous work [54]. The key feature of the approach is warp both the texture and the depth in the first pass simultaneously, and to leave blending of the final image to a later pass. This avoid errors that usually manifest themselves in the virtual depth map. Rendering quality is assessed in two ways. First, by varying the distance between the two nearest cameras and comparing the resulting PSNR. Secondly, by running a series of tests that measure the rendering quality using compressed video or images from surrounding cameras.

Abd Manap and Saroghan present a different layer-based algorithm for novel-view synthesis [104]. In this case, the depth map is separated into several layers of depth based on the disparity distance of the corresponding points. Based on masks, each layer of depth can be interpolated independently. The final novel view synthesis obtained by flattening all the layers into one layer. A multilayered approach has the advantage that the extracted new virtual object can be superimposed onto different 3D scene.

Yang et al present a view synthesis scheme where the depth map is not pre-processed [168]. Instead, two actual viewpoints are utilised, one being the main viewpoint while the other being an auxiliary viewpoint. These are used to generate the virtual viewpoint image. The auxiliary viewpoint is used in this case to help fill disocclussions. Any remaining holes are classified and filled with the help of the depth map and asymmetric dilation.

Marton et al present a complete real-time capturing and display system for 3D video that runs on a cluster-driven multi-projector light-field display [107]. As opposed to conventional light-field displays (which can produce blurry images where camera spacing is insufficient to sample the ray space), however, their method provides for all-in-focus rendering. The view-dependent depth is estimated on a GPU in CUDA via a customised multi-view space-sweeping approach.

Ateş and Alatan contribute to the GPU-based 3DTV algorithms by proposing a method that renders arbitrary views by using two high resolution colour cameras along with one low-resolution time-of-flight depth camera. GPUs are used to achieve real-time rendering. The presented ideas, however, are experimental and so are the results [9].

Petřík and Váša identify large data sets as the major bottleneck in rendering for 3D animations [125]. They thoroughly analyse the FAMC algorithm (Frame-based Animated Mesh Compression - an recent extension to MPEG4 for compression of dynamic triangle meshes) and propose to modify it by optimising and resolving the weaknesses of the algorithm.

2.3.5 Multi-view Techniques for Point Based Graphics

Multi-view integration is another vital aspect of point-based rendering techniques. Traditionally, this has been seen as an extension to the surface reconstruction problem, i.e, the problem was solved during a preprocessing pass where the data from multiple views was merged into one data set. This is the approach taken by Pulli et al in order to display data from multiple range images [128, 129]. Other techniques that rely on preprocessing, such as marching cubes, were elaborated on in the discussion on surface reconstruction [102] [36, 47, 82, 83].

Some modern multi-view techniques attempt to perform this operation in real-time [26, 32, 51, 107]. As Hübner et al [88] suggest, the fundamental drawback of current stereo and multi-view visualisation is the necessity to perform multi pass rendering (one pass for each view) and subsequent image composition + masking for generating multiple stereo views. Thus the rendering time increases in general linearly with the number of views. Hübner et al introduce a new method for multi-view splatting based on deferred blending [89]. Their method exploits the programmability of modern graphic processing units (GPUs) for rendering multiple stereo views in a single rendering pass. The views are calculated directly on the GPU including sub-pixel wavelength selective views. Marbach et al use tackle the problem in a single rendering pass via the use of modern GPU features such as geometry shaders, and layered rendering [105]. They found, however, that geometry shaders failed to perform as well as traditional vertex shaders on anything but the latest hardware. Moreover, performance of layered rendering is scene and application dependent so that even on modern hardware, not all scenes would see a performance gain.

Hilton [84], on the other hand, takes the traditional *polygonisation* approach by pro-

posing a continuous surface function that merges the connectivity information inherent in the individual sampled range images and constructs a single triangulated model. Ju et al [92] propose an algorithm for integrating range images by decomposing them into subset patches and running a *confidence competition* to identify and remove overlapping patches, merging the remaining patches into a single mesh. Hsin-Jung approach the problem by creating their own hardware (Altera FPGA) architecture [32]. A hybrid-parallel hardware architecture for depth-image based rendering (DIBR) system is proposed to generate multi-view images.

2.4 Discussion and Conclusion

As stated earlier, for range images to be useful in a clinical setting, the resolution must be preserved. In order for the visualisation to be effective, it must be real-time, i.e, interactive. If the clinical data is 4D, then each frame is different from the previous, and hence the rendering problem is compounded owing to the fact that now each full resolution range image must be read from disk (or cache) and passed for display in realtime. Under such circumstances, any pre-processing computations will significantly reduce real-time performance. The delay between one image being displayed, and the other being in the pipeline must be minimised to prevent lag associated with low frame rates. Problems with the state-of-the-art methods mentioned above with regards to native display of point-sampled data (especially range images) in real-time become evident when restrictions are imposed, such as minimal preprocessing, live (4D) streaming where there is no frame-to-frame coherence (i.e, content may be different from frame to frame), and no loss of data.

Bounding-sphere algorithms such as Qsplat [144], though fast and efficient, are suitable for static data. Data must be stored into a hierarchical representation in order to facilitate fast rendering. QSplat also doesn't support antialiasing on a GPU. Extensions to QSplat (such as SPTs and XSplat) have made it possible to *stream* the data, yet they rely on an even more compute intensive process, that of unrolling the hierarchical data structure into a serial one for dynamic streaming [49, 123, 158]. While this lowers per-frame memory requirements, and makes streaming large data sets possible, the method is still limited to static geometry. In essence, if the geometry changes per frame rather than viewpoint (i.e, frame-to-frame coherence information is not available), the preprocessing must be repeated. In addition, native QSplat is not GPU compliant [49].

Voxel methods [12, 43, 114] , on the other hand, are not well-suited for an architecture meant for visualisation of surface anatomy data. Voxels are best for volumetric models, but don't translate well into surface anatomy models, which would by necessity contain points describing a surface rather than a volumetric grid containing 3D voxels.

Implicit surfaces involve computationally expensive operations, such as the moving least squares approximation. They provide fine rendering quality, however, are difficult to optimise. Solutions to realtime rendering have included that of creating custom hardware, such as proposed by Heinzle et al [80].

Surface splatting, on the other hand, does not retain connectivity between points, information that is vital to perform measurements [73, 81, 87, 113, 122, 161, 174]. Wu et al's [164] progressive level-of-detail method, for example, is based on surface splatting. Though an elegant technique for visualisation purposes, surface splatting assumes a lack of connectivity information between the points. This is a significant disadvantage since although a Splat of unorganised points is acceptable for real-time display, it presents difficulties for measurement/assessment of the surface data. Measurement of surface data can only take place across a consistent regular model that contains connectivity from point to point, so that a measurement of any distance from any point to any other point can be made. My technique makes use of range images which preserve the connectivity between the points, and hence, once all the different views have been integrated, it is possible to perform measurements, and to assess the model.

In addition, surface splatting techniques rely on extensive preprocessing in *object space* that requires a significant setup time [106]. Surface splatting relies on a preprocessing pass that computes the basis functions and coefficients (called rk and wk respectively in the paper) that determine the properties of the splat [175]. Obviously, this preprocessing becomes a bottleneck for realtime streaming applications. The efficiency of splatting is not optimal for its reliance on two *object-order* passes: the visibility pass and the attribute pass, both have to process all displayed points [106]. This has implications for applications that may require streaming data, such as 3DTV or 3D Cinema or more relevant to our work, 4D captures required for clinical assessment. Assuming a frame rate of only 15 frames per second is adequate for real-time interaction, and only two pods, a single range image must be must be loaded and displayed within 0.03 seconds. This does not take into account time taken for other processing, such as measurement. In reality, the time per range image to be loaded and displayed is typically even less. Techniques such as surface splatting that rely

on pre-processing are clearly inadequate. They are most suited to offline preprocessing, rather than online streaming. Even improvements such as Progressive splatting rely on object-order preprocessing, and hence lack the capacity to stream multiple point clouds, especially where frame-to-frame coherency is not available [164] . Finally, surface splatting techniques do not work natively with range images.

Polygons are an inefficient data-structure for rendering. They have large memory requirements, are wasteful of computational power when polygon size is reduced to pixel sizes, and are entirely inadequate for the integration of multiple-views of large models. In addition, polygons are difficult to manipulate and process, which is evident when trying to merge multiple models [46, 102, 152, 155] .

It is evident that the methods outlined above are inadequate for the purpose of rendering real-time surface scanned range data where frame-to-frame coherence information may not be available. Both surface splatting and polygons are inadequate data-structures for the purposes of real-time visualisation of surface anatomy data natively from range images.

It is based on these observations that I have proposed an algorithm that furthers the state-of-the-art in point-based rendering, in context of medical visualisation. At its core lies the use of range images to store 3D data, and points as a display primitive. Range images are compact, easy to manipulate, and amenable to hardware acceleration due to their matrix-based nature. In the next chapter, I will explain the proposed method in detail.

Chapter 3

Data and workflow

3.1 Introduction

The proposed visualisation algorithms were tested two different kinds of data: Range images obtained during the Cleft10 project, captured via DI3D, a professional stereo-capture system, and digital terrain models (DTM) of planetary data obtained from the HiRISE website, captured by the HiRISE mission to Mars [116].

My testing setup consisted of a system with a set of 3 NVidia GTX 8800 GPUs in an SLI configuration, another with an ATI Radeon 4870x2, and a rig with an ATI Radeon HD 5870 GPU. Rendering performance was tested at multiple screen resolutions. My datasets were *Steve*, *Nairn, Melas Chasma* and *Mawrth Vallis*. *Steve* and *Nairn* are two-pod DI3D captured human heads with 3000x4500 pixels generated by each pod. In addition, alignment and masking information is provided in the DI3D file. *Melas Chasma* and *Mawrth Vallis* are Mars surface data captured by the High-Resolution Science Experiment (HiRISE) camera in orbit around mars. Two sequential crops of 3000x4500 each were taken from each data-set (two from Melas Chasma and two from Mawrth Vallis) to simulate two views. Automatically generated mipmaps were used to create the image pyramid for each of these data-sets. For Laplacian Projection, the *steve* dataset was used, and the pyramids (Gaussian and Laplacian) were generated using Matlab for this dataset.

Since this thesis is primarily concerned with surface scans in a clinical context, I will now explain how the DI3D datasets such as *Steve* were obtained for the Cleft10 project, and their motivation. Consequently, in this chapter, I will elaborate on the nature of the data that the proposed renderer must deal with. I will first provide an overview of the experiment (the Cleft10 project) run by the department of Psychology and their aims. I

will then discuss the nature of the collected data (range images, image pyramids). Finally, I will explain the actual data-collection process. This will include a discussion on the actual equipment used (DI3D), and the process/setup used by the clinicians in order to collect data for their experiment.

3.2 Background

The aim of the investigation (of the Cleft10 project) was to characterise, at 10 years of age, residual soft tissue deformities following repair of cleft lip with or without cleft palate; and relate these to psychological adjustment. Also, the development of a preliminary objective grading system of the residual facial deformities was to be explored and its usefulness as a tool to assist in the clinical decision making to be assessed.

3.3 Nature of the Data

3.3.1 Range images

An image sensor converts an optical image (or light) to an electric signal. Each pixel of a digital image represents the light (colour) information arriving at the corresponding location on the sensor. In figure 3.2, the sphere represent a real-world object that the camera is pointing at. Light hits points a,b and c and is reflected towards the camera, and arrives at the sensor at points d,e, and f respectively. Points d,e,f encode the *colour* of the object as seen by the camera. An image containing such colour information is known as a *texture* or an *intensity image*.

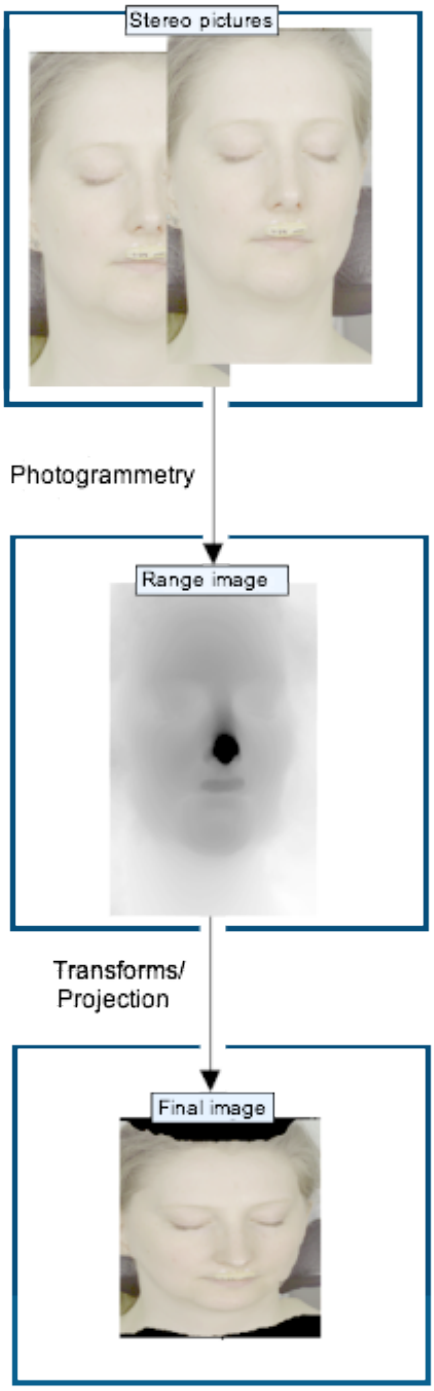


Figure 3.1: A simplified pipeline depicting how a 3D projection is produced from acquired data.

As discussed earlier, stereo-photogrammetry makes it possible to recover the *depth* associated with a pixel via triangulation. In the case of figure 3.2, the distance from a to d, b to e, and c to f may be recovered, and stored in another image. An image that contains depth information from the camera (or more appropriately, the centre of projection) to

the object is known as a *range image*. Since stereo-photogrammetry makes use of two or more cameras for capture, the range image is generally aligned to one of the cameras. This has the benefit of aligning colour information from the texture image with depth from the range image.

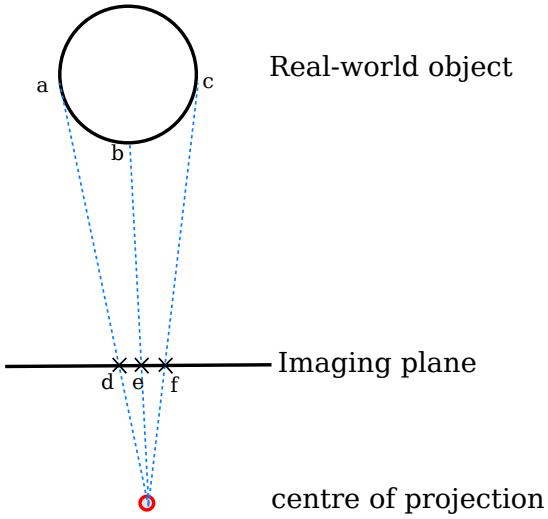


Figure 3.2: Light arriving from points a,b, and c on the object are recorded on the sensor at d,e, and f.

Range images are the fundamental data structure for holding 3D information from triangulation-based 3D capture device. Range images capture data from a particular point of view, and hence occlusions are not captured. For this reason, the data stored in range images is often called 2.5D as opposed to 3D. Despite this drawback, an advantage to the way range images are constructed is that they hold implicitly all connectivity information required for the reconstruction of a 3D model. In addition, being essentially a 2D matrix, manipulating range images is as convenient as manipulating 2D images. By virtue of their regular structure, range images may be stored on special-purpose GPU texture memory, and even be compressed *on the GPU* if required.

Range images store depth information relative to the centre of projection, i.e, in *camera-space*. The parameters for the original camera are required in order to transform the range data into *world-space*: a global coordinate system containing 3D models after their local transformations have been applied. The world-space permits multiple range images from different cameras to be aligned and placed together. In order to simplify computation, it is at times desirable to convert range images into *height-fields* (also known as *Digital Elevation Maps*).

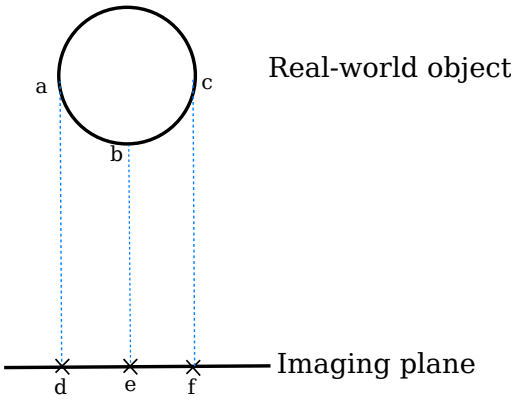


Figure 3.3: In a height-field, the depth is measured from the surface of the object to the imaging plane.

Height-fields (figure 3.3) store depth information in a 2D array similar to range images, however, the encoded depth is not computed from the camera centre, but rather perpendicularly to the imaging plane. In simpler terms, range images may be seen as a depth encoding of a perspectively projected scene, whereas a height-field may be seen as a depth encoding of a parallel-projected scene.

3.3.2 Gaussian Image Pyramid

The Gaussian pyramid is a Level-of-Detail representation for images, i.e, a hierarchical data structure the defines images at various *levels* (of detail). This generally results in an additional memory overhead, however, processing can be carried out on a less detailed image dynamically when needed, resulting in faster processing.

The Gaussian pyramid construction begins with the application of a low-pass (generally Gaussian) filter to the source image. The filtered image is then subsampled to remove redundancy. For this research images were subsampled by a factor of 2 to obtain an octave pyramid. The subsampled image is treated as the source image, and the process repeated until a hierarchy of n low-pass (generally Gaussian) filtered versions of the original image is obtained. Successive *levels* in this hierarchy comprise smaller images containing correspondingly lower frequency information. The original Gaussian image pyramid (as proposed by Burt and Adelson [27]) was constructed via a filter that approximated the true Gaussian kernel. In order to get a more accurate representation, I use a true discrete Gaussian kernel. Hence, each level is smoothed by a symmetric Gaussian kernel and resampled to compute the next level.

Using a Gaussian image pyramid has the benefit of noise suppression (due to the

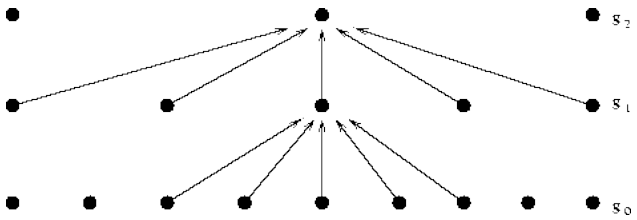


Figure 3.4: A Gaussian pyramid. Image courtesy of Jean-Michel Jolion.

smoothing applied at each level), and that of providing a convenient basis for a multi-scale representation. For the purpose of this work, a half-octave Gaussian pyramid is used as a multi-scale representation for intensity images, masks, and range images.

The Gaussian kernels for this research were created in Matlab using the **fspecial** command.

A 3x3 pixel kernel was used, with a spread of 0.5 to maintain the highest fidelity. A subsampling rate of 2 was used(each subsequent image in the pyramid as we traverse downwards is smaller by a factor of 2 in each dimension).

On a GPU, the pyramid is generated automatically, entirely on the GPU. While it is possible to create a true Gaussian pyramid on the GPU, it requires a multi-pass rendering algorithm. For practical reasons (maximum rendering speed), this research recommends the use of nearest-neighbour interpolation since it is done automatically via a single function call on all current GPUs in one rendering pass. For example, in OpenGL, the function call **glGenerateMipmap(GL_TEXTURE_2D)** performs the pyramid generation. The type of filtering cannot be specified, however, it is possible to request the GPU to perform the most accurate filtering algorithm available through the function **glHint (GL_GENERATE_ MIPMAP_HINT, Hint)** where **Hint** is set to **GL_NICEST**. We do not recommend the usage of glHint, however, since the particular algorithm chosen by the GPU cannot be ascertained in advance. We have chosen to always (consistently) use nearest neighbour interpolation rather than rely on an indeterminate GPU criteria.

3.3.3 Laplacian Image Pyramid

The Laplacian pyramid is a more compact version of the Gaussian pyramid [28]. While the Laplacian pyramid may be generated in a manner similar to the Gaussian pyramid, i.e, by performing convolutions with true Laplacian filters, the Laplacian pyramid is generally generated through a process known as the Difference-of-Gaussians (DoG) instead.

The Difference-of-Gaussians pyramid is constructed thus: Take the source image S,

and filter it with a Gaussian filter (with the same parameters as those used in a Gaussian pyramid). The image contains low-frequency information. We call this image L. The image L is subtracted from S. This image, that we shall call image H, contains only the high-frequency detail of image S. We can now subsample L with negligible loss of information. The images H and L are sufficient to reconstruct the original image S. If we treat L as if it were S, and repeat the process, we arrive at a hierarchy of high-pass images $H_1 ... H_n$ and a single low-pass image L at the top of the pyramid. L contains the low-frequency information, while progressive high-frequency images are added to incrementally add more detail until we arrive at the source image S.

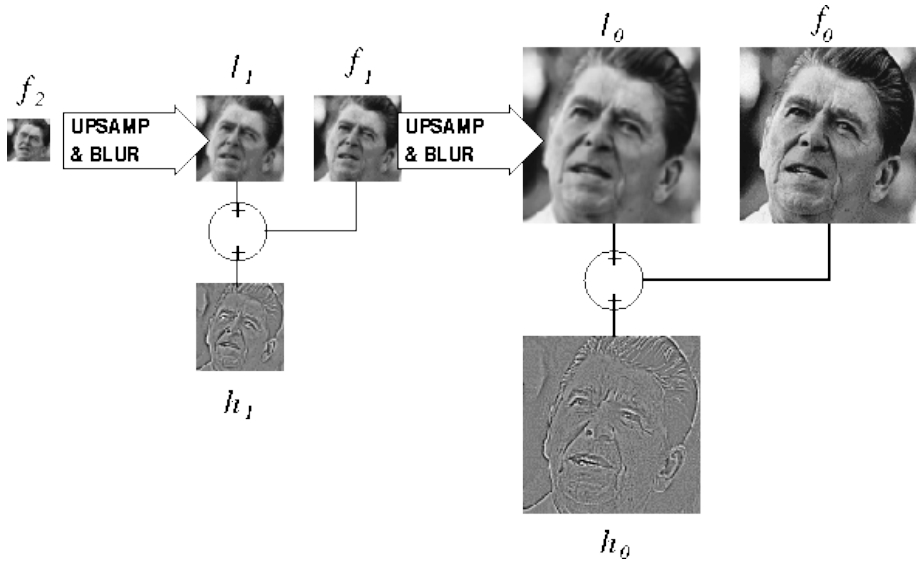


Figure 3.5: The construction of a Laplacian image pyramid. l is a low-pass image, h is a high-pass image, and f is the *final* image at each level [25].

In many applications, Laplacian pyramids are preferred because of their compactness. The two images, high-frequency information H, and low frequency information L, are more amenable to compression separately than the original image S [28]. For the purpose of this work, Laplacian pyramids are used primarily as a blending mechanism, as explained by Burt and Adelson in their work on image mosaics [27]. Once again, a 3x3 pixel kernel was used, with a spread of 0.5 to maintain the highest fidelity. A subsampling rate of 2 was applied (each subsequent image in the pyramid as we traverse downwards is smaller by a factor of 2 in each dimension).

3.3.4 Pyramidising Range Data

Image pyramids are a standard mechanism for handling multi-resolution intensity images [27, 28, 162]. Burt and Adelson show that the pyramidisation process, and in fact even the multi-resolution spline, does not introduce errors of its own to the image [27]. It is less obvious, however, that the pyramidisation process also works for *3D* data, such as range images [98, 147]. In fact, Burt and Adelson's seminal paper on multi-resolution splines [27] begins by asserting that a pair of images may be represented as a pair of surfaces above the (x, y) plane (figure 3.6), and then proceeds to show how the multi-resolution spline joins the two surfaces such that the edge is not visible. Technically, the images that create such a surface are referred to as height fields.

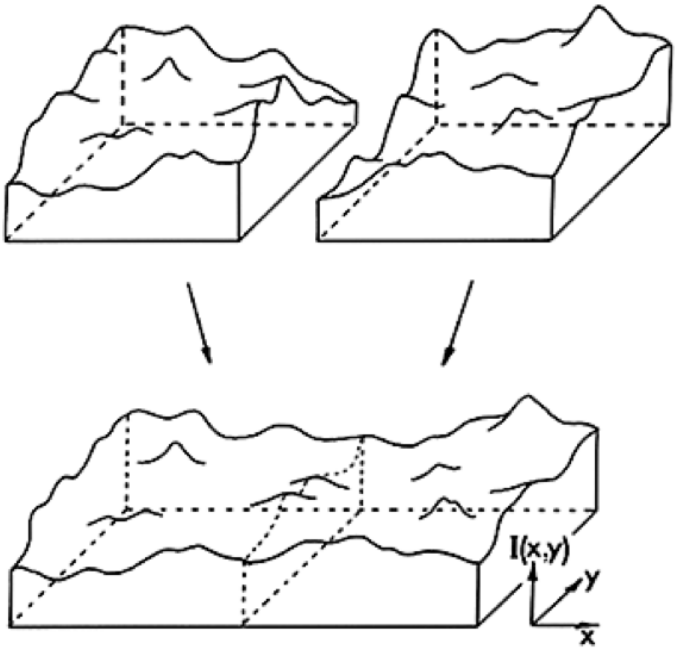


Figure 3.6: Burt and Adelson explain that a pair of images may be represented as a pair of surfaces above the (x, y) plane [27]. The problem of image splining is to join these surfaces with a smooth seam, with as little distortion of each surface as possible.

Range images contain depth data, similar to height fields, however, where height fields store the depth normal to the imaging plane, range images store depth from a surface to the centre of the imaging plane. Range images preserve surface continuity information, i.e, neighbouring samples in a range image correspond to neighbouring images on the captured surface at a given sampling rate (resolution). Therefore, a low-pass filter, such as a Gaussian function, may be used to obtain range images at lower frequencies. In short, it is possible to construct a Gaussian pyramid from a range image to obtain a multi-resolution

3D surface at various levels in scale-space. A Laplacian pyramid is a linear transformation of the Gaussian pyramid (a difference-of-Gaussians), and does not introduce any "error" into the image. Therefore, it is possible to construct a Laplacian pyramid from a set of range images, and arrive at the Gaussian pyramid via a lossless linear transformation.

For Pyramidal Projection (real-time rendering), as noted, the Gaussian pyramid is approximated as a box-filtered mipmap generated entirely on the GPU. It is important to note that this pyramid is a data structure that preserves the original data faithfully: The data inside a Gaussian pyramid (and mipmap) is range data, and has not been *converted* or *encoded* in a different format. Each of the pixels at the base of the Gaussian pyramid faithfully represents the original range values that may be fetched without additional processing. The additional layers of the Gaussian pyramid provide redundant data that make multi-resolution fetching of data faster. This *preprocessing*, however, is done entirely on the GPU and does not contribute to processing time during rendering.

3.4 The Data Collection Process

I will now explain the various components used during data collection for the Cleft10 project: The technology used, the workflow, and the data produced.

3.4.1 Data Collection

The investigation was carried out on two groups of 10 year old Scottish children. Group 1 consisted of 50 children with unilateral cleft lip (UCL) and group 2 consisted of 50 children with unilateral cleft lip and palate (UCLP). All the cleft cases have been treated following the same surgical protocol which has been adopted by the managed clinical network for cleft services in Scotland, CLEFTSiS. The Cleft patient groups were recruited from the CLEFTSiS database. Children aged from 9.5 to 10.5 years were considered for recruitment.

The data set used in this research project comprises 2.5D range images and their corresponding 2D stereo-pair images obtained from a DI3D system. The images were captured at a fixed pose similar to a standard passport photograph.

3.4.2 DI3D

The Cleft10 data-sets were computed from stereo-pair images captured by using the DI3D™ FCS-100 stereo-capture system. The system is comprised of two pods (each consisting

two 12 megapixel digital cameras) and bundled software: DI3Dcapture™ and DI3Dview™ that allows capture, management and presentation of 3D facial surface data. The stereo-capture process from DI3D results in two primary outputs: The raw range image (along with intensity images), and (after processing) a 3D polygon model in VRML format. The VRMLs can be loaded into the Facial Analysis Tool [76] where anatomical landmarks were placed on the models by a professional clinician. It is worth noting that these VRML files do not contain the full resolution data available in the raw range image files as it would not be possible to interact with the raw 3D data in real-time due to the volume of available data. While it would be sensible to use the 2D images, which are in line with the range maps, for the placement of these landmarks, the clinicians do require a full 3D interpretation and interaction on the 3D models in order to place landmarks accurately [101].

Figure 3.7: The DI3D stereo-capture system

3.4.3 Image Capture

The configuration of the stereo cameras (a single pod) is illustrated in Figure 3.8. This depicts a single pod system mounted on a camera rig, attached with two portable flash units and connected to a personal computer. The cameras are of the following specifications: DCS 14N Pro Kodak Digital Cameras and each photograph has a resolution of 4500×3000 pixels. The capturing process begins with manual initiation via a user interface that ensures

simultaneous capture of the stereo-pair. The stereo-pair images are then transferred from the cameras to the computer where models are ready to be built.

The stereo pair of digital cameras is placed in front of a dental chair with adjustable height and headrest. Subjects are asked to sit on the dental chair, and the height of the seat is adjusted to ensure the subject's face is in line with the cameras. Subjects are asked to perform a sequence of facial exercises in order to help them relax their facial expression to ensure the *fixed pose* criteria is met. The resulting images have the appearance of a standard passport photograph.

The software used to perform image capture is known as DI3Dcapture™, software that is bundled with the DI3D package. After image capture, the images are checked visually via live-preview and on the camera display before being transferred to the computer. Several images are captured in order to accommodate pose and acquisition errors and the most suitable set of images are selected. Upon capture, checking, and selection, the images are finally exported as TIFF format images, ready for the corresponding 3D models to be built. Intensity images are saved as standard TIFF files while the range images are saved as floating-point TIFF files.

Figure 3.8: A single pod stereo-pair stereo-capture system, complete with portable flash units and mounting gear

3.4.4 Models Building

DI3Dcapture™, a software packaged bundled with the DI3D system, is used for the computation of the calibration error of the system. In addition, it is used for the construction of range images and 3D polygon models from the stereo-pair of 2D images. There are three steps in the building of 3D models: 1) stereo matching, i.e correspondence, 2) surface reconstruction (photogrammetry) and 3) polygonisation. The models are built using the direct range mesh method.

3.4.4.1 Calibration

Before the metric range values can be recovered from captured images, it is necessary to calibrate the cameras so that the mathematical model used by the system conforms to the reality of the actual placement of the cameras. For this purpose, a calibration target (an object with predefined patterns) is captured in 13 different orientations within the field of view of the cameras and the resultant images are used to calculate the geometry of the cameras and their relative orientations [156]. This information can then be used to recover the range values from the *disparity* imaged produced from the stereo-pairs, thus enabling a 3D model to be built.

3.4.4.2 Range Surface

As mentioned in the previous section, the starting point for building a 3D model is the stereo-pair of photographs captured from the high-resolution digital cameras. A scale-space based matching algorithm computes the dense disparity map from the stereo-pair. This map can be split into vertical and horizontal disparity maps. In addition, a confidence map is produced. The confidence map indicates the probability that each matched value is indeed correct. This is grayscale coded, i.e, the lighter the shading, the greater the confidence. The calibration data is used in conjunction with the matched data to produce depth values for each pixel [100]. This process results in a range image. From the range image, as explained earlier in the chapter, the 3D world coordinates for the model are constructed and a polygonal model is built. The model is a triangulated mesh with the accompanying intensity images (the 2D photographic textures) superimposed onto the mesh. Since the raw range image is too dense to convert into a displayable polygon mesh, the resultant polygon mesh is generally of a lower resolution than the original range image. This compressed mesh can then be exported as a VRML file which is a commonly used

format for 3D files and can be viewed with a 3D viewer, for example, GLView.

3.5 GPU

The primary focus of this research has been the development of a native GPU-based point-cloud rendering algorithm. Due to the nature of the GPU, some trade-offs had to be made regarding the quality of the rendering in order to provide faster (native) rendering on the GPU. For this reason, I have given preference to native GPU functions over better algorithms where the GPU functions were significantly faster. Since the GPU functions are intrinsically linked to an API (such as OpenGL) that is provided to access those functions, care has been taken to mention both the API function as well as the underlying implementation that it uses (as of the writing of this thesis). This ensures that if the implementation of the API changes, it still remains possible to implement the proposed rendering aglorithms.

While this thesis assumes that a GPU-based rendering solution is sought, where real-time rendering is not a goal, or where concerns over the quality of rendering override those of interactivity, slower but better quality approaches have been mentioned for reference.

3.6 Conclusion

This chapter explained the nature of the data used in this research, and the workflow followed in order to obtain the various parameters and results. This completes the pre-requisites for understanding the proposed methods, as detailed in the next chapter.

The next chapter examines Pyramidal projection, a method that uses the properties of image pyramids in order to allow hole-filling of point-based data in real-time on a GPU. It is important to understand the properties of image pyramds in order to understand pyramidal projection. Additionally, it is crucial to understand the properties of Burt and Adelson's image mosaics in order to appreciate the 3D blending performed during Laplacian Projection, as explained later in the next chapter.

Chapter 4

The Proposed Method

4.1 Introduction

A naive method for rendering point-samples would be to simply forward project each point individually. While this would require very little computation, and hence render very quickly, the lack of connectivity between the individual points soon becomes apparent in the form of *holes.* This is a sampling problem, and manifests itself as the camera moves in beyond the native resolution of the captured data, or when the data is viewed from an angle where data is not captured in sufficient detail.

There are two possible ways to deal with holes in 3D data. The problem may be solved in either *object-space*, or *image-space*. An object-space approach would require iteration over all the vertices in a preprocessing pass in order to determine the maximum hole size in order to determine an appropriate method for hole-filling in a later pass. An image-space approach would treat the problem as a scattered-data interpolation problem, i.e, that of identifying where samples exist in the rendered image, and how to colour the portions where samples do not exist. The important point to note is that while the object-space approach would require preprocessing, this would be done once every time a new model is loaded, while the image-space approach would require hole-filling per frame. However, for 4D data, this distinction makes little difference if frame-to-frame coherence is not available since a new model is loaded every frame. In such a case, the image-space approach is more efficient since the problem is reduced to two dimensions, versus three dimensions in object-space computations.

4.1.1 Viewport Size and Holes

A viewport represents the area on the screen that will be filled by the rendered image. The viewport is described either in pixels, i.e, screen-coordinates, or is normalised from 0 to 1. For the purposes of this discussion, a viewport will be represented in pixels, unless otherwise noted.

I propose that the size of the holes in a *rendered* (projected) image (in pixels) is directly proportional to the size of the viewport in which the image is to be rendered. A 3D model rendered into a smaller viewport will have fewer holes. This is intuitively visible in figure 4.1. *A hole is a pixel in the rendered image for which a sample does not exist.* It should be noted here that our definition of holes refers to holes that occur in image-space *after* projection. We are not concerned with holes that arise due to a lack of information in the *source* data. For our purposes, we assume the original data is perfect, or that hole-filling has been performed on the source data and that the surface is fairly continuous.

The image in figure 4.1 depicts a hole. This hole is not simply equivalent to a background-coloured pixel, but rather, it refers to a location between valid samples that remains unoccupied. The term *rendering* here refers to the process of re-projecting the samples from the source 3D model to the imaging plane. Intuitively, if the same source 3D model is reprojected into a smaller viewport, the distance between the source samples will decrease. Since a hole is the unoccupied distance between samples, holes will tend to shrink. Another way to look at this is via a rule that is followed implicitly during standard rendering: Suppose that if a model were to be rendered at a certain resolution, it would project to an area that is composed of 50% hole and 50% sample (non-hole) adjacent to each other. After re-projecting into a 50% smaller (in each dimension) viewport, the hole and sample project to the same pixel during rendering due to the finite resolution of the destination viewport. If the destination pixel is empty, the sample will be written to the empty pixel location (and the 50% hole will be effectively discarded, thereby providing hole-filling), while if the destination pixel is non-empty, the either the new sample replaces the existing pixel, or the existing pixel remains unmodified, depending on the z-values of both source and destination sample and pixel.

We can state the relationship between holes and viewport size mathematically. Generally, a mapping occurs when bringing a world-coordinate system into screen-coordinates, known as window-to-viewport mapping. Since we would like to compare the size of a hole in pixels for two different viewports, both coordinate systems are in the same units, hence

we are essentially performing a viewport-to-viewport mapping.

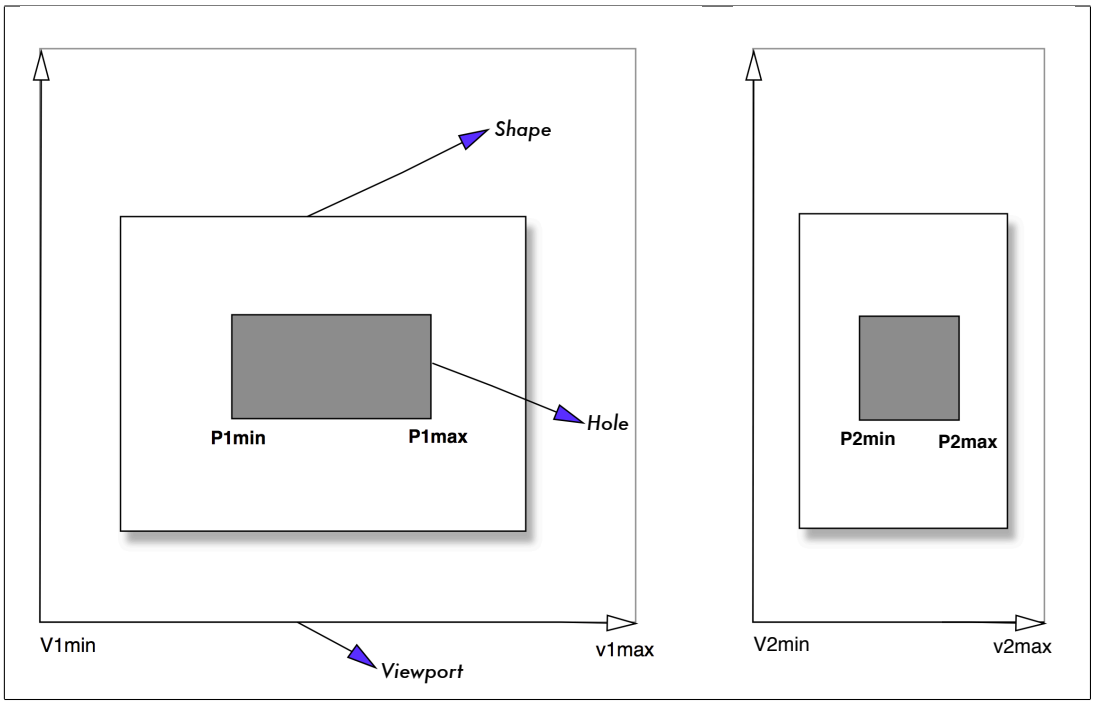


Figure 4.1: A viewport to viewport transformation where the width of the two viewports is different

If we define $P1$ as any point in viewport 1, then Equation 4.1 defines its mapping onto viewport 2 where $V1min$ and $V1max$ define the extents of viewport 1, while $V2min$ and $V2max$ define the extents of viewport 2.

$$P2 = (P1 - V1min)\left(\frac{V2max - V2min}{V1max - V1min}\right) + V2min \tag{4.1}$$

In our case, we can assume both viewports to be rendered in the same location, hence, the origin of both can be set to 0. $V1min$ and $V2min$ both equal 0. Equation 4.2 shows the simplified equation.

$$P2 = P1\left(\frac{V2max}{V1max}\right) \tag{4.2}$$

The simplified equation shows that the viewport-to-viewport mapping is equivalent to a single *scaling* transformation, where the amount of scale is proportional to the ratio between the two viewport sizes. If the scaling is predetermined, then the scaling can be substituted with a constant k that defines the scale. For an octave image pyramid, each subsequent image is half the resolution of the previous image, as the pyramid is traversed bottom-up. In such a case, $k = 0.5$ can always be assumed.

$$P2 = k(P1) \tag{4.3}$$

If we constrain our discussion to one dimension, then we can define $P1min$ and $P1max$ to be the horizontal extents of a hole that exists in viewport 1, and $P2min$ and $P2max$ can similarly be defined as the horizontal extents of a hole in viewport 2. If H1 is the length of the hole in viewport 1, then $H1 = P1max - P1min$. Similarly, if $H2$ is the length of the hole in viewport 2 then $H2 = P2max - P2min$. Equation 4.4 takes this simplification to its conclusion, that proves that under a viewport-to-viewport mapping, the size of a transformed hole $H2$ can be determined as a affine scaling transformation of $H1$.

$$\begin{aligned} H2 &= P2max - P2min \\ H2 &= k(P1max) - k(P1min) \\ H2 &= k(P1max - P1min) \\ H2 &= k(H1) \end{aligned} \tag{4.4}$$

In an octave image pyramid, $k = 0.5$, therefore, $H2 = 0.5(H1)$. Thus, it is shown that in an image pyramid, the size of the hole will decrease by 50% in each dimension as the pyramid is traversed bottom-up.

4.1.2 Image Pyramids for Hole Filling

Hole-filling in image-space is computationally less expensive than in object-space since image-space computations are performed only in two dimensions versus three for object-space computations. Rendering an image in a smaller viewport reduces the hole size (in pixels) and therefore further reduces the workload of an image-space interpolator. Upsizing the hole-filled image back to its native resolution can then be left to efficient image interpolation algorithms such bicubic interpolation. If the size of the hole reduces to sub-pixel sizes, then the rasteriser will (depending on the discretisation algorithm), eliminate the hole completely. According to the OpenGL specification, any GPU adhering to the OpenGL standard must implement rasterisation via the *trunc* method (which is a truncation of the values after the decimal), thereby ensuring that sub-pixels are ignored. This

does not take into account built-in GPU anti-aliasing, however, in this thesis an alternate antialiasing method is presented. An adequate hole-filling method, therefore, would be to reduce the size of the holes until they reach sub-pixel sizes. This would obviate the need for an additional hole-filling pass.

This method, however, does not preserve high-frequency detail present in the original model, and will also result in severe aliasing. A standard method of keeping aliasing in check during minification (scaling down) is to perform a low-pass filtering operation to reduce high-frequency detail before the reduction. If the process is repeated over a number of iterations, this is algorithmically similar to the way an image pyramid is constructed, as shown in algorithm 4.1.

Algorithm 4.1 Constructing a Gaussian image pyramid

```
 //The first image in the image pyramid is at full-resolution
Pyramid[0] = originalimage;
//A variable that will hold the modified image
image = originalimage;

for (int i=1; i< numlevels; i++)
        {
        //Apply a low-pass filter
        blurredimage = lowpass(image);

        //Reduce the image in size (50% in each dimension)
        reducedimage = reduce(blurredimage);

        //Save the image in a pyramid
        Pyramid[i] = reducedimage;
        }
```

In an octave image pyramid, as mentioned earlier, $k = 0.5$, therefore, $H2 = 0.5(H1)$. By that account, it would require only a 4-level pyramid to reduce a hole of 8 pixels down to 1 pixel, and 5 levels to eliminate it.

An image pyramid is therefore an ideal data structure for storing high-resolution point-sampled data: It allows rendering into smaller viewports so that hole-filling may be performed, and preserves higher frequencies for full-resolution rendering applications, such as medical imaging.

4.1.3 Opting for Matrix Data on the GPU

There are many methods to store the point-sampled geometry, however, on a GPU, data may be stored either in vertex memory, or video RAM. Any data-structure that is not in the form of a Matrix (texture memory/ video RAM) or a linear contiguous Array (Vertex memory) , will need to be stored *outside* the GPU. This is an undesirable outcome since data must then be transferred back and forth between CPU and GPU, causing a drop in real-time performance. To circumvent such a situation, we propose the use of a matrix data structure such as range images or height-fields as the source data for 3D point-based rendering.

Apart from hole-filling, the choice of a matrix-based data source (such as range images) was influenced by the fact that range-images are the native data-structure for most 3D scanners, and rendering range images natively makes it possible to remain close to the original source, thereby avoiding unnecessary data loss through conversions. They are also naturally amenable to being converted to image pyramids so that we can make use of hole-filling as mentioned earlier. In addition, matrix-based data structures are compact and ideal for GPU acceleration. Although the techniques related here are applicable to both range-images and height-fields, I will mention range images for brevity and assume the same principles apply for height-fields, Digital Elevation Maps, or other matrix-based data sources, under the appropriate transformations, unless otherwise noted.

A range image, and a texture image are enough to display a 3D image from arbitrary views. Additionally, a mask may be used to eliminate the background from the object of interest. For dynamic lighting effects, or simple back-point culling, a matrix containing the normals corresponding to the image (in other words, a normal-map) may also be required.

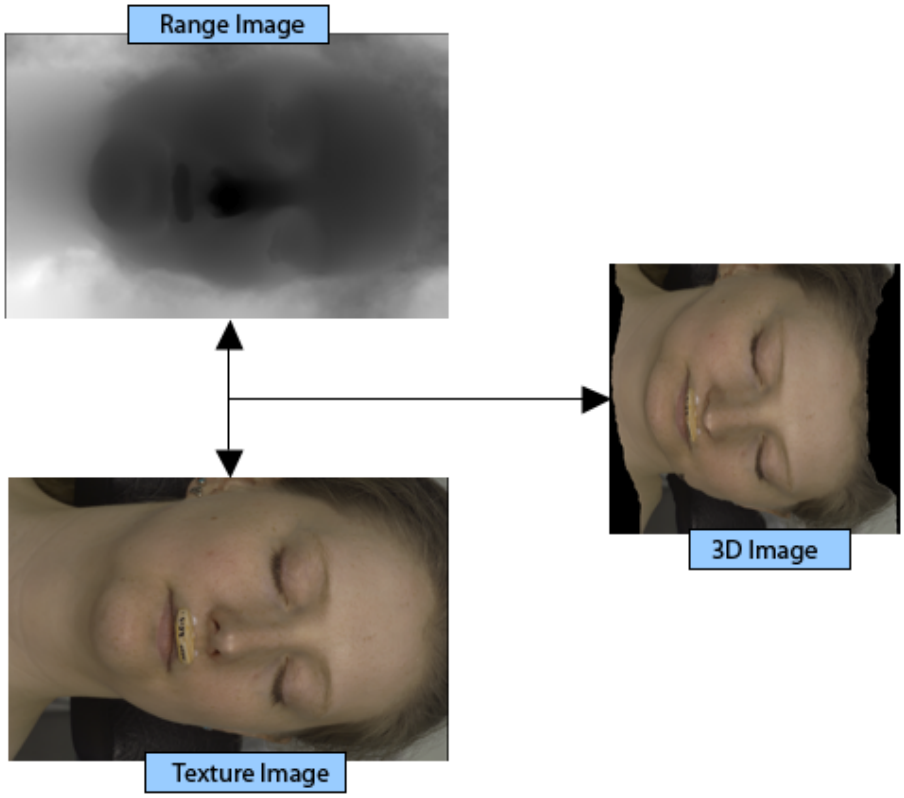


Figure 4.2: A range image **R** and a texture image **T** combined to display an arbitrary 3D view

4.1.4 Opting for Floating Point Textures on the GPU

GPUs provide two types of memory access, one for storing vertices via Vertex Buffers, and the other for storing textures. Range images, being a Matrix structure, can be efficiently stored in Video RAM as a floating point texture. This has several advantages over storing data in a Vertex Buffer. Data is not limited to the number of vertices a 16-bit or 32-bit index can store. Image processing operations are made possible on the GPU. The images can be used as destinations for *Render-toTexture* operations. Most importantly, texture memory provides random access to (neighbouring pixels in) memory, while Vertex Buffers can only operate on the current vertex [60].

While vertices are stored uncompressed in GPU memory, textures may be optionally compressed, and even pyramidised (called mipmapping) on the GPU without CPU intervention beyond the initial setting of parameters. Creating an image pyramid in GPU memory makes it possible to store not only vertex data in texture memory, but an entire multi-resolution 3D model in texture memory. Such a data structure is a novel contribution of this thesis.

Currently, the filtering operation performed during mipmapping is left to the GPU, which by default applies a box filter rather than a true Gaussian filtering, however, OpenGL allows providing the GPU with a quality preference as a *hint*, though the specification does not enforce the GPU to follow the provided hint. In any event, the GPU pyramidisation scheme is suitable where accuracy may be traded off for speed, such as streaming 4D images. Otherwise, the source data may be stored as pre-filtered textures in GPU memory before rendering.

Owing to the grid/matrix nature of range images, the points in a range image are offset by a fixed linear increment on the horizontal and vertical axis, while the depth value (z-value) changes unpredictably. I propose to store depth information via a floating-point texture, and provide the x, y values as re-usable indices in a Vertex Buffer where the offset is calculated on the GPU in a shader. This saves GPU memory bandwidth considerably, the actual savings depending on the size of the indices array.

4.2 Pyramidal Projection

I will now explain Pyramidal Projection. The following section will present the conventions that are followed for the symbols that appear throughout the text.

4.2.1 Conventions used

T : Matrix-based data structures are represented by a boldfaced capital letter such as **T** or **M**. The symbol **T** in particular represents a texture.

$\widehat{T}$: A wide hat symbol over a capital letter denotes an image pyramid. $\widehat{T}$ represents an image pyramid of **T**.

$\widehat{T}_i$: A subscript along with the wide hat symbol is used to indicate a particular a image in an image pyramid. $\widehat{T}_i$ represents the image indexed by i in the image pyramid $\widehat{T}$ obtained from the texture image **T**.

$\widehat{T}_L$: A subscript L with an image pyramid denotes a Laplacian (Difference Of Gaussians) pyramid.

4.2.2 Overview of Pyramidal Projection

Pyramidal projection is a point-based-rendering algorithm primarily concerned with scattered-data-interpolation (hole-filling) via a multi-resolution image pyramid. Anti-aliasing, multi-resolution rendering, hidden-point-removal, and multi-view rendering can all be added to the basic pyramidal projection framework naturally. In this section, however, we will focus on the basic pyramidal projection process. In later chapters, we will add the aforementioned added functionality to the basic renderer.

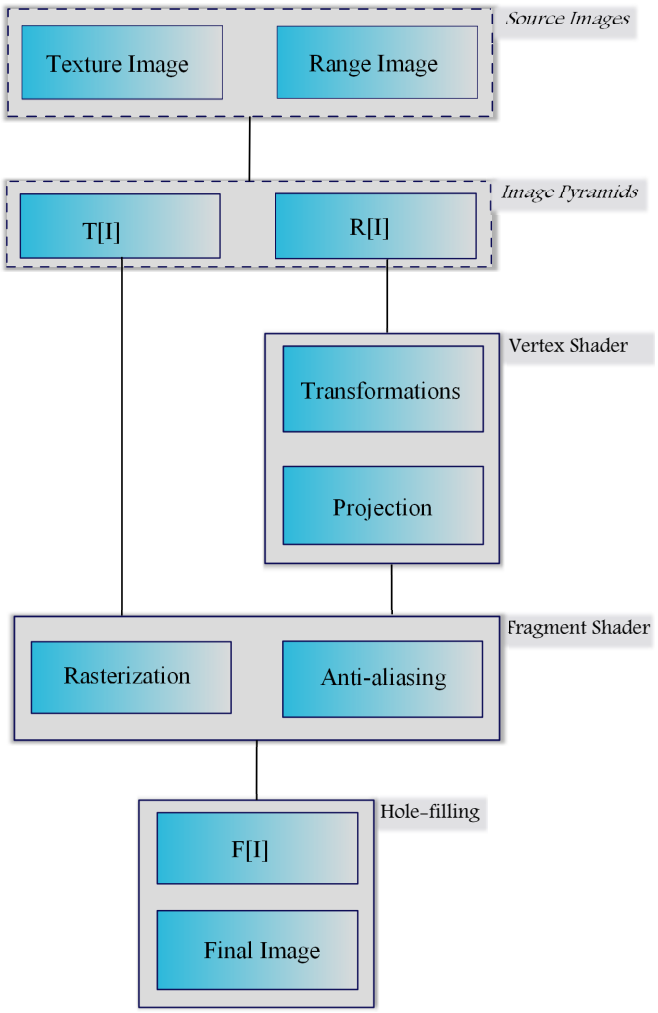


Figure 4.3: Overview of the proposed method. Dotted lines represent pre-processed elements.

4.2.3 Pyramidal Projection: An Intuitive Explanation

Pyramidal projection is based on the fact that a lower-frequency point-sampled image, rendered in a smaller viewport, will have fewer holes than a high-frequency point-based rendered image. This low-frequency image may be coupled with a high-frequency image

in order to fill holes in a full-resolution image. In short, a full-resolution image is rendered at its native resolution, with the gaps filled-in by the lower-frequency image. This is technically a more natural method of interpolation since high frequency detail, when not available, is replaced with a filtered sample, similar to how the human eye filters objects that are beyond their natural viewing distance.

In short, the process is as follows: Generate image pyramids from a source range image, colour texture, and optionally a mask. Render each range-image in the image pyramid, starting with the lowest-resolution (and with fewest holes), and render progressively higher-resolution images *over* the previous images. With the appropriate transparency settings, this ensures that the high-resolution image samples will overwrite low-resolution samples where they exist, and the low-frequency image will *show through* the holes left by the high-resolution image.

Algorithmically, the process is described in a simplified form in algorithm 4.2.

Algorithm 4.2 Intuitive explanation of the steps involved in Pyramidal Projection

```
a) Generate image pyramids from range, texture, and mask images
b) Render a filtered version of the 3D model into a viewport
half the size in each dimension.
c) Expand the resulting image via a 2D interpolation method.
d) Render the high-frequency image over the low-pass filtered image.
e) Repeat with the next image in the pyramid.
```

4.2.4 Pyramidal Projection: The algorithm

I have implemented the proposed algorithms both in *software*, using Matlab, and on the *hardware* (on a programmable GPU) using OpenGL/GLSL. The CPU based algorithm involves tight loops, and large memory-to-memory transfers to simulate data being copied to various buffers. The GPU implementation on the other hand is highly parallel, and GLSL generally provides *shaders* that represent the computations that will be performed on a single pixel, point, or vertex. In addition, the GPU provides features (and consequently introduces new terminology) that are not present on the CPU.

The algorithm provided in the previous section does not take into account how the algorithm may be mapped onto existing hardware. In practice, the memory layout of the GPU, and available API (OpenGL for example) features dictate how the algorithm will be implemented. Since images are first projected (rendered) from 3D to 2D, and then another operation is performed to enlarge the 2D image, on a GPU this operation must be carried

out in 2 rendering passes.

Since this thesis presents a real-time GPU-based algorithm as a novel contribution, the GPU implementation (one which includes the two passes) is described as algorithm 4.3.

Algorithm 4.3 Pseudocode describing the algorithm

1. Setup
   (a) Load Range, Texture, and Masks called R,T, M
   (b) Generate Gaussian pyramids $\hat{R}$,$\hat{T}$,$\hat{M}$ from R,T,M
   (c) Allocate texture memory for each of the rendered images in $\hat{F}$
2. Pass 1
   (a) Render each model in $\hat{R}$,$\hat{T}$,$\hat{M}$ into $\hat{F}$ like the following:

```
for every image at index i, in ^R
{
Clear ^F (i.e, set all values, including alpha, to zero)
Set size of ^F based on i (smaller viewport as i increases)
Render each ^R ,^T, ^M into ^F
}
```

3. Pass 2
   (a) Display rendered images on the screen in order of resolution, smallest first, like the following:

```
for each item in ^F (in reverse order)
{
Expand ^F to screen size;
Render Image in ^F, with transparency (i.e, overlay mode)
}
```

I have presented the CPU version of the algorithm in psuedocode in algorithm 4.4, and the GPU based algorithm in more detail in algorithm 4.5.

4.3 Details of the Algorithm

The algorithm is divided into 3 distinct phases: Setup, Pass 1, and Pass 2. I will now describe in more detail each of the steps of the proposed method. As an oversimplification, the basic idea of the algorithm is to render the model multiple times, in multiple viewports of varying resolution, and use the data from viewports with no holes fill in data for viewports with holes. There are three salient points to bear in mind:

1. Multiple versions of the model must be created, at various resolutions (i.e, levels-of-detail). This is done via an image pyramid, as discussed in the next section.

2. The multiple models must be rendered in multiple viewports of varying resolutions. Hence, separate areas of display memory must be dedicated to rendering each of the images before the final image can be generated. In terms of OpenGL, these areas are called Frame Buffers. In figure 4.3, these are represented by $\widehat{F}$.

3. Each of the images in $\widehat{F}$ are overlaid on top of each other, allowing the lower resolution image to *show-through* from the holes in the higher-resolution images, effectively providing hole-filling.

I will now explain each phase, with a focus on the GPU implementation, as the focus of Pyramidal Projection is real-time interactivity.

4.3.1 The Setup Phase

The setup phase takes care of data management: Loading the data from files, and setting up a pyramidal data structure. The setup phase is concerned with the creation of image pyramids. On the GPU, mipmaps provide for a fast and built-in mechanism to implement image pyramids.

The setup phase begins by assuming that our data-acquisition method has provided us with a range-image **R** for 3D information, which is a matrix of 32-bit floating point values containg range values. In addition, provided is a texture-map **T** that contains a triplet of R,G,B values, a byte each, for colour information, and optionally a mask **M** of a floating value that separates background information from valid 3D points. Omitting the surface reconstruction phase, we proceed to the setup phase by creating an image pyramid from the **R**,**T**, and **M** to create arrays $\widehat{R}$ (figure 4.4) ,$\widehat{T}$, and $\widehat{M}$. The image pyramid is generated by convolving **R**, **T**, and **M** by a low-pass filter, such as one generated by the standard Gaussian equation, and then subsampling by a factor of two. In Pyramidal Projection, we use mipmapping to generate the image pyramids as it is a capability built-in to all current GPUs, and hence does not require a round-trip to the CPU, or require additional passes. Also, if done on the GPU, the operation incurs negligible additional cost, and hence is suitable for 4D image sequences where the operation may be performed every time the frame changes. Image-space pyramidisation, or dynamic LOD generation is one of the attractive features of the proposed method, and is only possible since our source

data - **R**,**T** and **M** - are 2D matrices. Currently, it is not possible to specify a Gaussian mipmapping on the GPU, therefore a box-filter, being the fastest convolution method available, is used. This subsampling ensures that each level $\widehat{R_i}$ of the image pyramid is an octave apart from the subsequent layer in the pyramid. This means that as the pyramid is traversed, each subsequent image higher up in the pyramid ($\widehat{R_{i+1}}$) will be a quarter the resolution (half in each dimension) of the previous image ($\widehat{R_i}$).

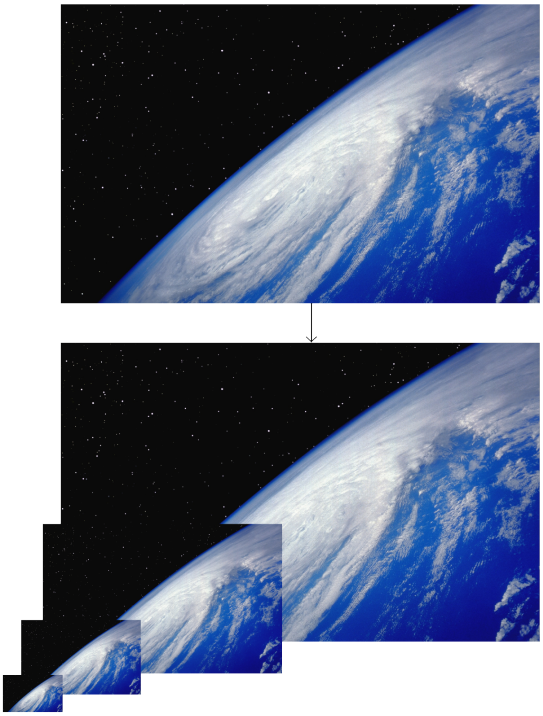


Figure 4.4: A texture **T** (above) and its corresponding image pyramid $\widehat{T}$ (below)

As mentioned earlier, a single image, and its corresponding texture are enough to create a 3D model. $\widehat{R}$, $\widehat{T}$, and $\widehat{M}$ effectively constitute a series of 3D models. In fact, the image pyramid represents a series of models in object-space at various levels-of-detail (see figures 4.2 and 4.5).

The next step in the Setup phase, after pyramidization, is to setup the vertex arrays to store vertex data that must be passed to the GPU. Since there are two passes, two types of vertex data are passed to the GPU.

First, the vertex data consists of the x,y indices (32-bit floats) that are passed in a grid format corresponding to the pixel values on the imaging plane for each range value that must be reconstructed to recover depth at each pixel value. The x,y index values indicate

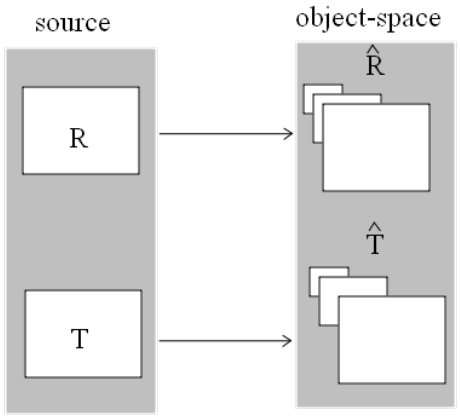


Figure 4.5: A texture **T** (above) and its corresponding range image **R**, when pyramidised, are enough to define a 3D model at various levels of detail.

the pixel on the imaging plane for which depth is to be recovered. Since the imaging plane is always a rectangular matrix, the x,y index values are always integers in the range 0 to resloution-1.

Secondly, in the next pass, vertex data must be passed for each screen-space quad that is used to attach each of the "layers" as a texture from the frame-buffers that makes it possible to "see-through" to the next layer behind it. Since the camera is normalized and orthographic in the next pass, and the quads cover the entire screen, the vertex values for the quads are normalized from 0 to 1, with 0 and 1 representing either extremes of the screen (See algorithm 4.6).

The final step is to create the frame-buffer objects. The frame-buffer objects (FBOs) are storage areas used by the GPUs to perform off-screen renders. The results of the first pass are rendered into an FBO - **F** -, which will subsequently be attached as textures in the second pass. We create n FBOs, where n refers to the number of levels in the image pyramid, so as to create $\widehat{F}$. In practice, we have found n=5 to be a practical value both in terms of memory consumption, and the amount of hole-filling performed. Each FBO is composed of RGBA 32-bit floating points (generally known in OpenGL as GL_RGBA32F). The dimensions of each FBO correspond to the pyramid level that it is meant to store. All values of $\widehat{F}$ are cleared to zero, including the alpha value. Depth-sorting is set so that new values overwrite older values only if they are nearer to the camera. In OpenGL this is known as glDepthFunc(GL_LESS).

At this stage, the one and only difference between range-images and height-fields be-

comes apparent. If our source data were composed of height-fields, it would be possible to render the data at this point: no further processing would be required. However, for range-images, an additional transformation from range-space (camera centred coordinates) to world space is required. This transformation is carried out in the vertex buffer in the first pass.

4.3.2 Rendering - Pass 1

Rendering pass 1 begins with the vertex shader. **R**, **T**, and **M** are attached to the vertex shader as textures. **R** is a matrix of 32-bit floating point values. **T** contains a triplet of R,G,B values, a byte each, for colour information, and **M** of floating point values. In addition, the Model-View and the Projection matrices are provided, based upon parameters provided at the time of image capture of the real-world camera used to perform the capture. Finally, the frame-buffer is attached so that the rendering destination is set to the FBO rather than the screen.

The primary problem with a point-based projection is the appearance of holes due to the discrepancy between the viewport resolution, and the sampling of the object to be displayed. These holes can be overcome by adequately sampling the object. This has been described by Grossman and Dally [71] as follows:

"We can...in principle, choose a set of surface point samples which are dense enough so that when the object is viewed there will be no holes, independent of the viewing angle. We say that an object or surface is adequately sampled (at the given resolution and magnification) if this condition is met"

To provide this sampling, based on our earlier observation about the relationship between sampling and viewport size, we will render each of the models obtained from $\widehat{R}$, $\widehat{T}$, and $\widehat{M}$ to a different viewport size in video memory. To define such areas of memory, we construct an array of Frame Buffer Objects (FBOs - See appendix) in GPU memory, which we will refer to as $\widehat{F}$. The rendering into each of the buffers in $\widehat{F}$ is carried out as depicted in figure 4.6.

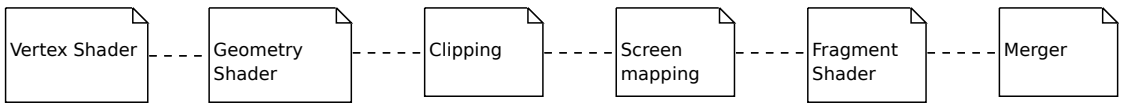


Figure 4.6: The GPU pipeline revisited in context of the proposed algorithm.

Each of the buffers contained in $\widehat{F}$ must be an octave apart. Each of the buffers is

initially cleared, and assigned an alpha of zero to indicate it doesn't contain any valid points so far (line 19). The alpha will be used to indicate whether a pixel is occupied or not. In addition, the value of **M** is checked, and a binary comparison is performed. If the value of **M** is anything greater than 0.01 (or another suitably small threshold), then the point is processed further, otherwise it is flagged for removal. At this stage, if the source images are range rather than height-fields, then they can be transformed into world-space dynamically in the vertex shader. We proceed to render all the models $\widehat{R_i}$, $\widehat{T_i}$, and $\widehat{M_i}$ into $\widehat{F_i}$ (as per figure 4.6 and line 21 in algorithm 4.5) where $0 \leq i < N$. N is the number of images in the array.

The images in $\hat{F}$ will be merged during the second rendering pass in order to construct the final image. This makes it necessary to store the images in $\widehat{F}$ in video memory until the second rendering pass is over. Since $\widehat{F}$ is a *frame buffer object*, it can be made to store its data to textures in video memory by defining textures as the *render target* for $\widehat{F}$ rather than the screen. The output from the rendering is then routed directly to textures in video memory rather than being displayed on the screen. After rendering, another pass can recombine the images (residing now in separate textures) to form one final image (figure 4.7).

4.3.3 Rendering - Pass 2

Via *render targets* (an explanation of render targets is given in the appendix), in essence, $\widehat{F}$ is a collection of textures that represents an image pyramid in screen space which can be used to perform fast and efficient hole-filling. These textures hold projected images of the 3D model (since they have passed through the entire 3D pipeline including projection) and therefore are 2D in nature. These 2D pictures can be easily drawn as screen-aligned polygons.

It is important to note that on the GPU, a texture may only be displayed on the screen when bound to a polygon. This polygon, however, may be entirely two-dimensional, and a single polygon will suffice for an entire texture (resulting in one texture per level of the pyramid). Therefore, in order to display the rendered images (now saved as textures), we attach each of the textures in $\widehat{F}$ to a screen-aligned textured polygon (line 34 and 35 in algorithm 4.5), and use a normalized orthographic camera. Since the viewport now ranges from 0 to 1 in each dimension, and the vertices of the polygons range from 0 to 1, all the polygons align to the screen (refer to figure 4.8 and line 36 in 4.5). The highest

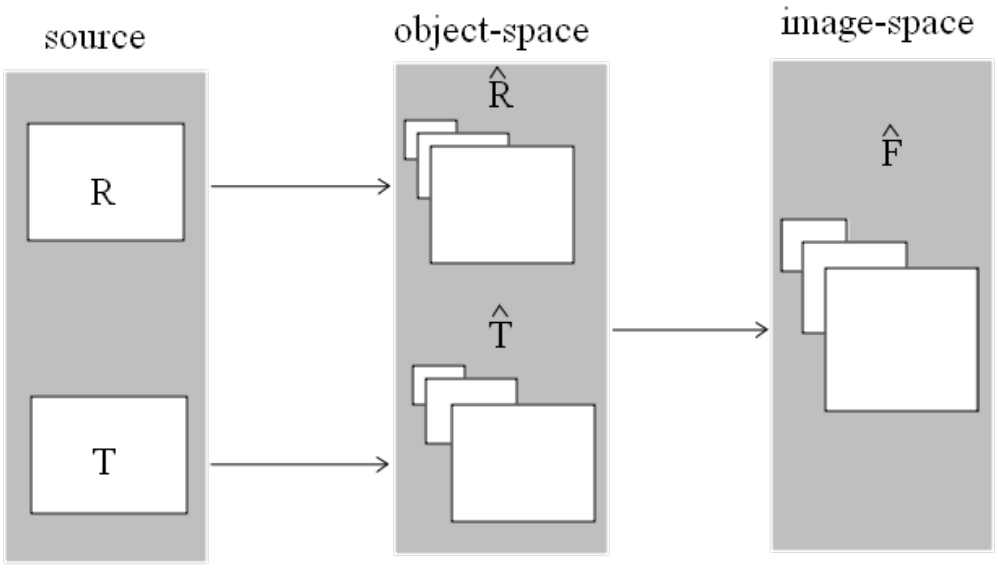


Figure 4.7: The FBO (Frame Buffer Object) is a collection of textures representing the screen-space image pyramid of the rendered object.

resolution image in the pyramid will contain most detail but also potentially the most holes. These screen-aligned polygons are rendered to the screen in descending order, i.e, with $\widehat{F_{i-1}}$ being rendered first, and $\widehat{F_0}$ being rendered last (line 37 in algorithm 4.5). This has the effect of drawing lower-resolution models first, with subsequent higher resolution models replacing data where it exists at a higher frequency. Where there are holes in the higher-resolution models, having an alpha of zero in the texture, they will allow the lower resolution models to show through. In this way, we solve the scattered-data interpolation problem in GPU-accelerated manner.

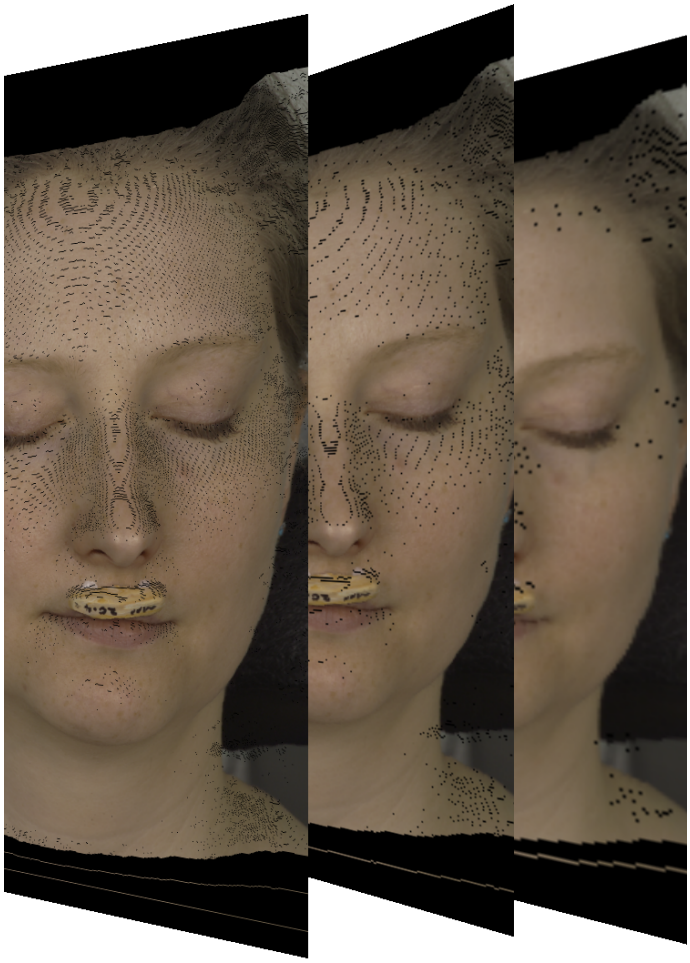


Figure 4.8: Individual images in the FBO shown. The lower resolution (subsampled) images have been rescaled to match the highest resolution image. Note how there are fewer holes in progressively lower resolution images.

4.4 Discussion and Conclusion

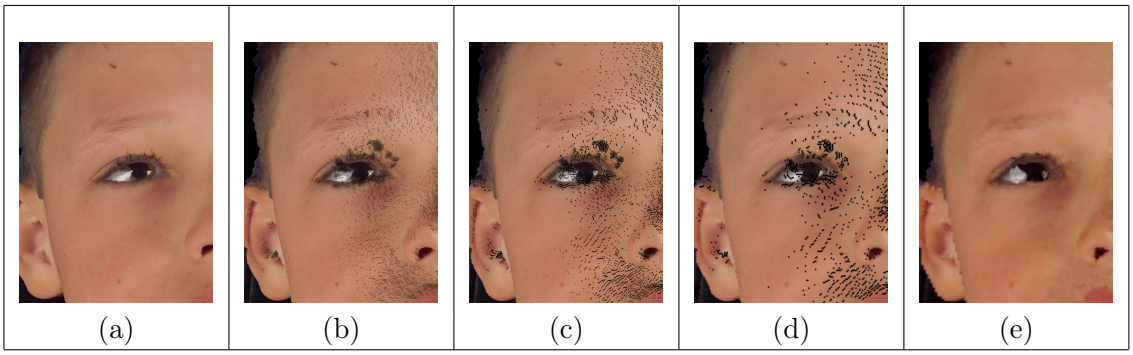


Figure 4.9: Figures (b) to (e): Individual textures in an image pyramid from highest resolution to lowest. Figure (a) The final hole-filled image. These are shown in higher resolution in figures 7.11 to 7.15 in chapter 8.

This chapter presented a novel GPU-based scattered-data interpolation algorithm for rendering point-based data. There are two important conclusions to be drawn. First, due to the reliance of Gaussian Projection on image pyramids for rendering, regardless of the size of I, an octave image pyramid will incur a constant additional expense of up to 33%. With an image pyramid of only 5 levels, it is possible to remove a hole 8 pixels wide. Second, $\widehat{F}$ contains an ordered LOD version of the scene. Depending on the distance of the camera from the scene, it is possible to reduce or increase the number of buffers that must be rendered, permitting only the lower-resolution versions to be displayed if the camera is far, and subsequently higher resolution versions to be displayed as the camera moves nearer. Further, since $\widehat{F}$ is a screen-space image pyramid, it allows seamless transitions from one level to the next via a simple linear blending function. This provides a more pleasing effect compared to the pop-in and pop-out effect noticed in other LOD rendering methods.

Figure 4.10: The final hole-filled image of Steve in more detail.

4.4.1 Memory Consumption in the Proposed Method

A mesh-based system will generally store data as raw vertices and an index-list iterating over shared vertices. For a single view of 3000x4500, the raw number of vertices

is 13,500,000. From the same matrix-based source image, a simple triangle count is $(length-1) * (width-1) * 2 = 26985002$ triangles. Note that the aforementioned vertex count takes into consideration vertex-sharing, and the connectivity provided by the matrix structure. Without connectivity information, a raw vertex count would be 3 vertices per triangle, i.e, $26985002 * 3 = 80955006$.

Assuming each vertex has x,y,z data stored as floats, and R,G,B data stored as single bytes, per-vertex memory required is 15 bytes. Each triangle requires at-least 3 16-bit index entries. However, for a large dataset such as a single-view of 3000x4500 samples, the 16-bit index runs out of precision, and we must resort to either splitting up the data set into multiple geometries (each with its own indexing and vertex information) or using 32-bit indices. Either way, memory consumption increases. Therefore, assuming 32-bit indices, each triangle requires at-least 3 32-bit index entries. The total memory required for a single view of 3000x4500 is the sum of raw vertex data and indexing information per triangle:$(15 * 3000 * 4500) + (26985002 * 12) = 202500000 + 323820024 = 526320024$. In other words, over **500 MB**. Clearly, it would be difficult to fit more than a single view of 3000x4500 onto the GTX 8800 GPU as it is limited to 768 MB.

A much more compact method is to use a triangle-strip. The triangle-strip is composed of $n + 2$ vertices, for n triangles. Hence, it requires $26985002 + 2 = 26985004$ indices, as oppossed to 3 indices per vertex. In this case, the total memory required for a single view of 3000x4500 samples is $(15*3000*4500) + (26985004*4) = 202500000 + 107940016 = 310440016$. That is nearly **300 MB**. This is a more manageable dataset, i.e, one that will fit in the GTX 8800 GPU. However, multiple views would still over-burden GPUs with 512 MB RAM, such as those common in current laptops.

A pyramidal point based system stores data as raw vertices, with pyramidal overhead, which is 1/3rd of the original data. Pyramidal projection does not require an additional buffer to store indexing data (and requires less memory than polygons) since 16-bit U,V indexing information replaces the x,y floats in vertex data, and the z value is stored as a float in a depth-mapped texture attached to the shader. The per-vertex memory consumption, including the 8-bit RGB triplets attached as a texture to the shader is 11 bytes. Total memory consumption for a single view is hence (length∗width∗11)+1/3(length∗width∗11) = 198000000. In other words, nearly **190 MB**. This is over **2.6 times** a reduction in memory consumption over raw vertex storage, and an over **1.5 times** reduction over triangle-strips. This makes it possible to render large datasets with a pyramidal approach

on commodity or portable devices such as laptops, where previously native imaging would be memory-limited.

4.4.2 Further Reducing Memory Consumption: Index compression

As alluded to earlier, since matrix-based data-sets have regular spacing in the XY direction, with multiple views of the same size, it is possible to store the *uv* indexing information only once, and share it between multiple views (as we have done for Melas Chasma and Mawrth Vallis) reducing memory usage even further. This is depicted visually in figure 4.11.

Owing to the grid/matrix nature of range images, the points in a range image are offset by a fixed linear increment on the horizontal and vertical axis, while the depth value (z-value) changes unpredictably.The depth information is stored as a floating-point texture, and the x, y values as provided *uv* indices in a standard Vertex Buffer. Since the *uv* offset between each sample is fixed, the *uv* vertex buffer is well suited to *tiling*.

The image is split into NxM sized tiles. Memory in the *uv* vertex buffer only need be allocated for exactly one tile. This tile can be re-used for rendering every other tile, and the *uv* values be dynamically calculated based on the dimensions of the tile, the index of the tile to be rendered, and the *uv* values of the current sample being rendered. The exact relationship is defined as shown in equation 4.5 where $currentUV$ is a vector that refers to the computed u and v values of the tile being calculated, $TileIndex$ refers to an integer index of the current tile being rendered, $TileDimension$ refers to the width and height of the tile in question, *uv* refer to the u and v values of the current sample, and finally $*$ is a component-wise multiplication.

$$currentUV = TileIndex * TileDimension + uv \tag{4.5}$$

This computation is done on the GPU in a shader, and due to the highly parallel nature of the GPU, the addition and multiplication has a negligible effect on rendering speed while it saves GPU memory bandwidth considerably, the actual savings depending on the size of the indices array and the number of tiles.

If we suppose a single view consists of 3000x4500 samples, and 5x5 tiles, then the dimensions of a single tile are 600x900 samples. The total memory consumption is composed of *uv* indices plus the floating point depth texture and colour texture. The floating point texture takes 4 bytes per sample, and the RGB values a byte each, for a total

of 7 bytes per sample. The uv indices are both 16-bit, consuming 4 bytes per index sample. According to equation 4.6, this results in a total memory consumption of approximately 92 MB. With an additional pyramidal overhead, the total memory consumption is $96660000 + 1/3(96660000) = 128879999$, or **123 MB**.

$$memory = (600 * 900 * 4) + (7 * 3000 * 4500) = 2160000 + 94500000 = 96660000 \quad (4.6)$$

From the raw 500 MB, the compressed 123 MB represents more than **4 times** a reduction in memory consumption with negligible performance cost. Compared to the non-tiled version at 192 MB, a tiling of 5x5 at 123 MB represents at-least a **1.5** times reduction in GPU RAM consumption. Compared to the 300 MB of triangle-strips, this represents an over **2.4 times** reduction in GPU RAM usage.

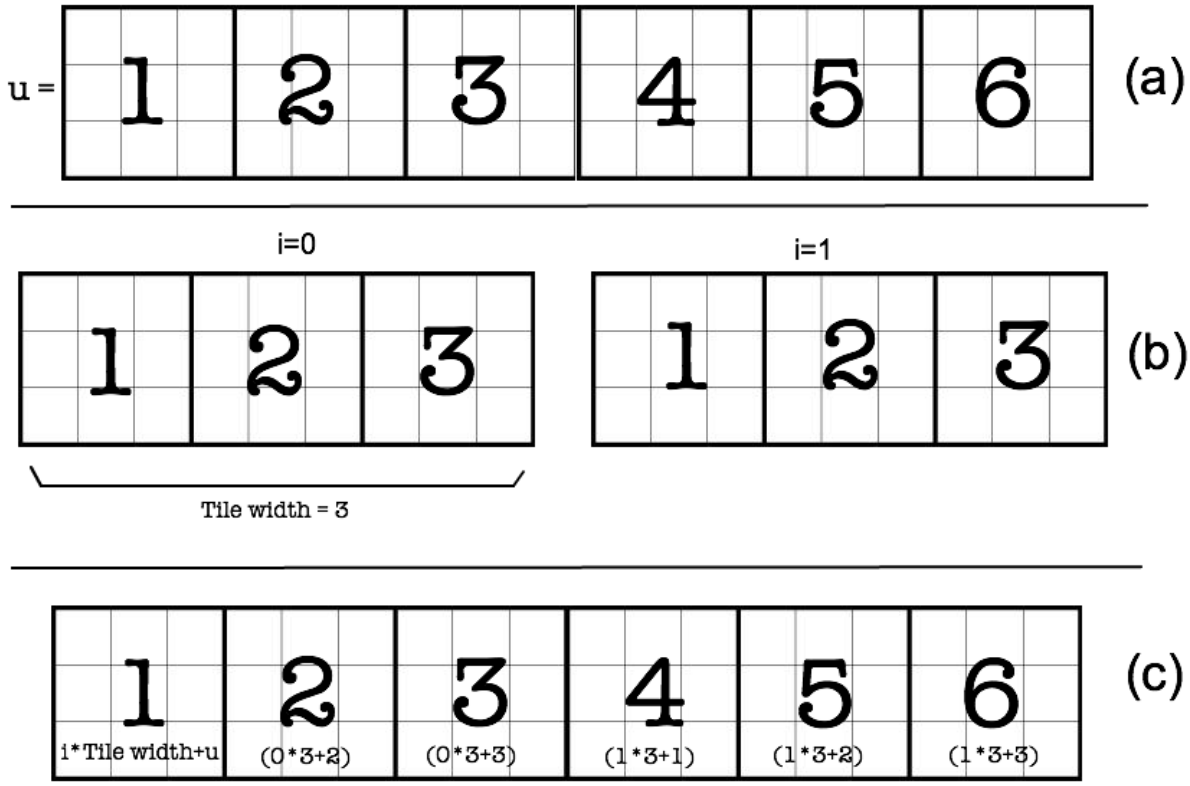


Figure 4.11: The range image in one dimension, as it is split into tiles, so indices may be reused. u indicates the horizontal index. (a) u increases as the samples of the range image are traversed. (b) The range image is split into two tiles. Since the information in both tiles is the same, the same tile may be shared. (c) The original u value is recovered using a simple equation involving a single addition and multiplication.

4.4.3 Multi-resolution Techniques in the Proposed Algorithm

The primary method used in pyramidal projection to store a multi-resolution representation of the scanned object belongs to a family of algorithms known as LOD (Level-of-Detail) algorithms [50]. The basic premise of LOD techniques is that the perceived detail of an object increases in proportion to its displayed size. For general perspective cameras, this means an object will have less perceived detail as it moves further away from the camera. In such a case, a complete model may be replaced with another simpler version of the object, without any perceived loss of detail. Therefore, it is possible to store an object at multiple resolutions in scale-space, and switch between models of varying resolution depending on the distance from the camera.

LOD techniques can be classified into the following:

Discrete This is the original LOD as proposed by Clark as early as 1976 [39]. During a preprocessing phase, multiple versions of an object are created at various resolutions and then replaced based on distance at run-time.

Continuous/progressive During a preprocessing phase, a mesh-simplification system algorithm constructs a data structure that encodes the model in a way that at run-time, the model at a desired LOD can be extracted. The important difference between continuous and discrete LOD is that in the continuous version, the encoding method preserves a continuous spectrum of detail rather than discrete models at specific resolutions. In addition, a particular model at a particular resolution is extracted at run-time rather than stored [85].

Though the primary motivation for the development of LOD techniques has historically been the improvement of performance, multi-resolution algorithms such as LOD provide an effective way to control aliasing. Sampling theory dictates that aliasing is introduced when there is a disparity between a signal and the sampling rate. LOD techniques attempt to maintain a consistent sampling rate by reducing the number of samples as the projected size of the object decreases. In the proposed method, we are primarily concerned with using LOD techniques as an antialiasing measure. Using image pyramids as a core data representation method provides the proposed algorithm with a natural multi-resolution capability which can be exploited for LOD and antialiasing purposes.

4.4.3.1 Dynamic Level-Of-Detail via pyramids

An image pyramid contains a scale-space representation of an image. The proposed rendering algorithm takes this representation a step beyond, adding an extra dimension. By virtue of the fact that depth information is present, image pyramids represent a 3D model at various levels in scale-space. The respective convolutions avoid aliasing. Therefore, the Pyramidal Projection algorithm is inherently an anti-aliased multi-resolution rendering algorithm. It is possible to dynamically switch between various versions of the 3D model in scale-space.

The overall rendering pipeline for Pyramidal Projection permits selective rendering (figure 4.3). By selective rendering, I refer to the fact that it is not necessary to render *all* the models represented by $\widehat{T}$. A particular level in the pyramid (representing a model at a particular resolution) can be chosen in isolation for rendering with adequate hole-filling. This can be done for a model indexed by `selection`, if all coarser levels in the pyramid (`selection =< i <= N`) are rendered as well. The extremely simple algorithm is as follows:

Algorithm 4.7 An algorithm describing how to render a particular model in the image pyramid

```
//selection is the index of the selected model that
//we wish to render
//N is the number of levels in an image pyramid
for (int i=N-1;i>=selection;i--)
        {
        render(  )
        }
```

In addition, since the image pyramid also exists in screen space in the form of textured screen-aligned quads as $\widehat{F}$, it is possible to dynamically turn on or off a particular level in $\widehat{F}$ by simply rendering or discarding the relevant polygon to which the texture is attached, and observe its effect on the final rendered picture. This is an effective way to judge how much contribution a particular level in the pyramid makes to the hole-filling.

Apart from the ability to dynamically switch between various levels-of-detail, it is even possible to automate switching. In speed-critical applications, if a camera moves further away beyond a particular distance, it may be faster to switch to rendering a lower resolution version of the image. This reduces the GPU load. Additionally, it has the benefit of reducing aliasing since switching to a lower resolution version when the projected image is smaller prevents overs sampling. The details of automatic level-of-detail switching are

discussed in more detail in the chapter on Anti-Aliasing.

4.4.4 Limitations of The Proposed Algorithm

I would like to now discuss some of the limitations of the current work. The proposed renderer works well for instantaneously viewing large data-sets at native imaging resolutions. However, there are obvious limitations to the use of Pyramidal Projection, most of which arise from the initial assumptions made about the usage of the rendering algorithm in context of the Cleft 10 project.

The proposed algorithm makes no effort to pre-align the images, and it is assumed that this information is available, and stored as the camera's extrinsic parameters. In practice, alignment information is usually available (such as in systems such as C3D, or DI3D), or not necessary (such as multiple height-fields obtained from one large data-set). In context of Cleft 10, the data was pre-aligned by the DI3D software.

Pyramidal Projection does not provide infinite detail. As mentioned earlier, clinicians are primarily concerned about the preservation of original information present in an image, and being able to interact with this information in a real-time manner. Therefore, by design, Pyramidal Projection was developed to permit viewing large data-sets at native imaging resolutions with adequate hole-filling. Moving far beyond the native imaging resolution is not guaranteed to provide adequate interpolation, hence, like QSplat, camera constraints may be required for pleasant interaction with the models [144]. This is a natural consequence of avoiding surface reconstruction and is a trade-off in favour of faster set-up times.

As pointed out in chapter 6, the hidden point removal algorithm assumes a blending mode where each new pixel completely overwrites the old one (equivalent to glBlend Func (GL_ONE, GL_ZERO) in OpenGL). That is, $newPix = (1.0 * incommingPix) + (0.0 * existingPix)$ where $incommingPix$ is the pixel which is to be rendered and $existingPix$ is the pixel that is already rendered in the location at which $incommingPix$ is to be rendered. Since our anti-aliasing algorithm relies on a different blending mode, at present, the hidden point removal algorithm does not work with anti-aliased points, and has trouble with z-fighting. These are areas that I would like to investigate further in the future. In practice, however, the z-buffer is adequate for hidden point removal in most cases.

Finally, as common to all point based rendering algorithms that rely on splats larger than a pixel, silhouette edges can be less well-defined. Pyramidal Projection can exaggerate

the problem since the interpolation is a 2D process and lower-frequency information will produce larger and fuzzier silhouettes.

4.4.5 Contributions of Pyramidal Projection

I will now list the contributions that Pyramidal projection makes to the literature.

Novel scattered-data interpolation mechanism

Pyramidal projection is a novel hole-filling method that is designed to be executed in parallel via a shader on the GPU, does not require pre-computation, and does not rely on expensive backward projection algorithms such as raycasting.

In this chapter I presented a novel scattered-data interpolation (hole-filling) algorithm for real-time rendering of point-sampled geometry natively from range images. I provided the algorithms (both for the CPU and the GPU), and I presented a discussion on how the proposed method makes it possible to fit large data sets onto the GPU where this was previously not possible.

Novel use of GPU memory to compactly store a multi-resolution 3D model

While it is common to store height-maps in texture memory on the GPU (commercial game engines such as Unity and UDK do so), LOD generation (creating the multi-resolution representation) is done outside the GPU, in a preprocessing pass. Pyramidal Projection is able to *generate* (and store) a *multi-resolution* 3D object entirely on the GPU. This is made possible via built-in GPU support for image pyramids in the form of mipmaps.

Algorithm 4.4 Pseudo-code explaining the Matlab (CPU) version of the proposed algorithm

```
1  //CPU VERSION (Software)
2  //SETUP
3  int N;          //Levels in a multi-resolution pyramid
4  //Load Range, Texture, and Masks
5  Load(R,T,M)
6
7  ^R = GaussianPyramid(R, N)
8  ^T = GaussianPyramid(T, N)
9  ^M = GaussianPyramid(M, N)
10 //^R,^T,^M together constitute a series of anti-aliased 3d Models
11
12 //Allocate memory for rendered (projected) images
13 CreateBuffer(^F);
14
15 //PASS 1
16 //Render each model in ^R,^T,^M into ^F
17 for(i=0; i<N; i++)
18         {
19         //set all values, including alpha, to zero
20         ClearBuffer(^F[i]);
21         RenderIntoBuffer(^R[i], ^T[i], ^M[i], ^F[i]);
22         }
23
24 //Now ^F is a collection of projected
25 //images (with transparency) at various resolutions
26 // display on the screen in order of resolution, smallest first
27
28 //PASS 2
29 //Allocate memory for final image
30 CreateBuffer(FinalImage);
31
32 for(i=N-1;i>=0;i--)
33         {
34         currentImage = ExpandImageToScreenSize(^F[i]);
35
36         //Copy all pixels of ^F[i] to FinalImage, replacing existing pixels
37         CopyPixels(FinalImage, ^F[i]);
38         }
39
40 DisplayImage(FinalImage);
```

Algorithm 4.5 Psuedocode explaining the GPU (GLSL) version of the proposed algorithm

```
1  //GPU version (Hardware)
2  //SETUP
3  int N;          //Levels in a multi-resolution pyramid
4  //Load Range, Texture, and Masks
5  Load(R,T,M)
6
7  ^R = GaussianPyramid(R, N)
8  ^T = GaussianPyramid(T, N)
9  ^M = GaussianPyramid(M, N)
10 //^R,^T,^M together constitute a series of anti-aliased 3d Models
11
12 //Allocate video-memory
13 CreateFBO(^F);
14
15 //PASS 1
16 //Render each model in ^R,^T,^M into ^F
17 for(i=0;i <N;i++)
18         {
19         //set all values, including alpha, to zero
20         ClearFBO(^F[i]);
21         RenderIntoFBO(^R[i], ^T[i], ^M[i], ^F[i]);
22         }
23
24 //Now ^F is a collection of projected
25 //images (with transparency) at various resolutions
26
27 //Bind each of these images to a polygon,
28 //and display on the screen
29 // in order of resolution, smallest first
30
31 //PASS 2
32 for(i=N-1;i>=0;i--)
33         {
34         currentPolygon = CreateRectangularPolygon;
35         AttachTexture(currentPolygon, ^F[i]);
36         ScalePolygonToScreenSize(currentPolygon);
37         RenderPolygon(ScreenPoly);
38         }
```

Algorithm 4.6 Setting up vertex data

```
void SetupVertexData(int inWidth, int inHeight)
{   //Setup Vertex Indices
//Iterate through each pyramid level
for (int level=0; level < NUM_LEVELS; level++)
        {
        int step = pow(2.0, level);
        int width = inWidth / step;
        int height = inHeight / step;
        int total = width * height;
        IndexArray[level] = new Vertex[total];
        //Set-up the indices for the entire matrix
        for(int cols=0;cols<width;cols++)
        {
        for(int rows=0;rows<height;rows++)
        {
        IndexArray[level][(rows * width) + cols].position[0] = cols;
        IndexArray[level][(rows * width) + cols].position[1] = rows;
        }
        }
        }
//2 triangles. This is to store the screen-aligned quads
//3 vertices each. 0,1,2  and 2,3,0
//Vert 0
ScreenQuadVertices[0].position[0] = 0.0;
ScreenQuadVertices[0].position[1] = 0.0;
ScreenQuadVertices[0].texcoord[0] = 0.0;
ScreenQuadVertices[0].texcoord[1] = 0.0;
//Vert 1
ScreenQuadVertices[1].position[0] = 0.0;
ScreenQuadVertices[1].position[1] = float(WINDOW_HEIGHT);
ScreenQuadVertices[1].texcoord[0] = 0.0;
ScreenQuadVertices[1].texcoord[1] = 1.0;
//Vert 2
ScreenQuadVertices[2].position[0] = float(WINDOW_WIDTH) ;
ScreenQuadVertices[2].position[1] = float(WINDOW_HEIGHT);
ScreenQuadVertices[2].texcoord[0] = 1.0;
ScreenQuadVertices[2].texcoord[1] = 1.0;
//Vert 2
ScreenQuadVertices[3].position[0] = ScreenQuadVertices[2].position[0];
ScreenQuadVertices[3].position[1] = ScreenQuadVertices[2].position[1];
ScreenQuadVertices[3].texcoord[0] = ScreenQuadVertices[2].texcoord[0];
ScreenQuadVertices[3].texcoord[1] = ScreenQuadVertices[2].texcoord[1];
//Vert 3
ScreenQuadVertices[4].position[0] =  float(WINDOW_WIDTH);
ScreenQuadVertices[4].position[1] = 0.0;
ScreenQuadVertices[4].texcoord[0] = 1.0;
ScreenQuadVertices[4].texcoord[1] = 0.0;
//Vert 0
ScreenQuadVertices[5].position[0] = ScreenQuadVertices[0].position[0];
ScreenQuadVertices[5].position[1] = ScreenQuadVertices[0].position[1];
ScreenQuadVertices[5].texcoord[0] = ScreenQuadVertices[0].texcoord[0];
ScreenQuadVertices[5].texcoord[1] = ScreenQuadVertices[0].texcoord[1];
}
```

Chapter 5

Anti-Aliasing

In this work, antialiasing has been achieved by approximating each sample's point spread function via a Gaussian function. Since each sub-pixel value will require a re-computation of the Gaussian function (an expensive operation), a lookup table has been used to rapidly retrieve various precomputed Gaussian functions. I will now explain the antialiasing method used in this work in more detail.

5.1 Introduction

Rendering techniques, by default, are prone to producing unseemly aliasing artifacts. Aliasing is caused by the disparity in resolution of the original signal and the resolution of the output mechanism. When the original sample has a larger number of samples than the output device can represent, the reconstructed signal on the output device is different from the original signal. Several samples in the original signal may correspond to the same output sample in the reconstructed signal (known as *nearest-neighbour* sampling), becoming in effect *aliases* of each other. The reconstructed signal is therefore sufficiently different from the original to cause artifacts. This is generally the case when a continuous signal must be represented on a physical device, which by its nature, can only represent the signal discretely. The aforementioned artifacts may not always be obvious in static images, however, the moire pattern and flicker is immediately apparent in an animation.

Most anti-aliasing algorithms rely on various forms of interpolation in order to reconstruct a signal that attempts to be faithful to the original. For point sampled geometry, connectivity information may not be present during rasterisation. For such a reason, interpolation between samples is not possible, and other ways must be sought to provide

Figure 5.1: Magnified, and aliased letters on the left. The same letters magnified, and anti-aliased on the right.

appropriate anti-aliasing.

5.2 The Point Spread Function (PSF)

A single point on an imaging system is seen by virtue of light being reflected from that point arriving at the sensor of the imaging system. This point of light is in reality spread out over a finite area on the sensor. This *spread* can be approximated mathematically via a **point spread function** (**PSF**). The final image, a collection of point samples, is computed as a sum of the PSF of each point. For a perfect lens with an aperture that is perfectly circular, a single point of light would produce a pattern known as an *airy disc*.

$$I(\theta) = I_0(\frac{2J_1(kasin\theta)}{kasin\theta})^2 \tag{5.1}$$

where I_0 is the maximum intensity of the pattern at the airy disc centre, J_1 is the Bessel function of the first kind of order one, $k = 2\pi/\lambda$ is the wavenumber, a is the radius of the aperture, and θ is the angle of observation, i.e. the angle between the axis of the circular aperture and the line between aperture centre and observation point.

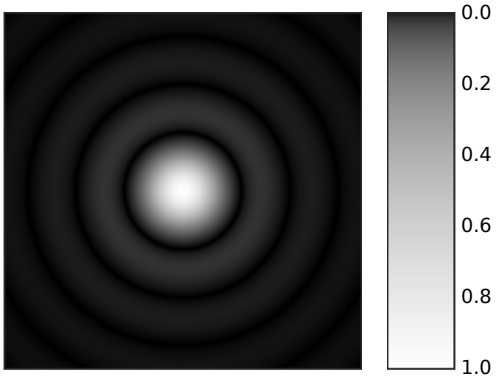


Figure 5.2: A computer generated image of the airy disc [117].

The intensity of the airy disc pattern tends to zero radially outwards. It is therefore more cost effective to ignore the relatively smaller information contained in the outermost

rings, and approximate the airy disc via a Gaussian function:

$$I(x) \approx I'_0 exp(\frac{-x^2}{2w^2}) \tag{5.2}$$

where I'_0 is the irradiance at the centre of the pattern, and w is the Gaussian width.

5.3 Point Sprites and the Gaussian approximation of the PSF

A realistic rendering of a point requires that rather than be defined as a single pixel, a point be approximated via its PSF. As discussed above, it is cost effective to approximate the airy disc as a Gaussian function, therefore, I have used a discrete 3x3 pixel Gaussian kernel, displayed as a *point sprite* (see appendix). The kernel itself is nothing more than a discretised version of the normalised Gaussian distribution. Though typically, the Gaussian kernel would have the highest intensity at the centre, and progressively fall off we move radially outwards, this is not necessarily true if the point does not fall on a pixel centre. It is possible, however, to precompute these sub-pixel offsets and store them in a look-up table for rapid access.

To correctly approximate the *spread* of a point, on-screen points are replaced with small finite screen-aligned patches that approximate the PSF of each point. The small screen-aligned patches can best be represented by *point sprites* (see appendix), a feature available on modern GPUs, whereas the Gaussian kernel is used to approximate the PSF. The kernel can be thought of as the *alpha Map* for the point sprite, resulting in a circle that is opaque at the centre, and progressively fuzzier (transparent) as we move radially outwards.

5.4 The Gaussian Kernel Look-up Table

By default, for a pixel at an integer location, the Gaussian distribution would be centred. However, for every sub-pixel projection, a recalculation of the kernel would be required to discretise the Gaussian function for that particular sub-pixel location. Though faster than a complete calculation of the airy function per point, it is still a costly operation, given that there may be millions of points that are projected to non-integer locations.

For a finite number of sub-pixel offsets, it is possible to generate a table of precomputed Gaussian kernels. Since this need be done only once, I performed this pre-computation in

Matlab.

5.4.1 Offset Normalisation of Pixels in the LUT

In order to generate the appropriate offset, we are concerned only with the fractional offset. Therefore, given the x and y locations of a point sprite, the offset may be stored as shown in algorithm 5.1.

Algorithm 5.1 The algorithm for computing the pixel offset from the pixel centre.

```
shiftx=x-floor(x);
shifty=y-floor(y);
```

It seems natural to place the pixel centre at 0. This is an arbitrary restriction, however, and may be different depending on the rendering API used. For the present discussion, we will assume that the pixel centre is at 0, though we will later see that OpenGL assumes pixel centres are offset by half a pixel, and we will revise our assumptions accordingly.

For a normalised pixel (one with a total width of unit length, i.e, 1), a pixel centred at 0 would mean the maximum extents of a pixel are at 0.5. Supposing for a pixel with $x = 3.5$, the point sprite would have exactly half of its intensity at pixel 3 horizontally (because its exactly in the middle of two pixels). In essence, the point sprite would be most opaque at its right, and the left boundaries would be mostly transparent (figure 5.3).

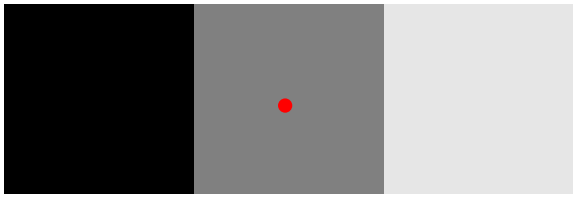


Figure 5.3: The discrete Gaussian kernel for a point at offset of 0.5. The red dot represents the centre of the point sprite.

Consequently, any value beyond half a pixel (0.5) would leave the pixel with an overall intensity less than half a pixel. In fact, since a pixel is assumed to be of unit length, an offset of more than 0.5 would move the point sprite over to the next integer pixel location and *wrap-around* its intensity. This means that rather than treating a pixel at $x = 3.6$ as being centred at $x = 3$ with an offset of 0.6, we can treat it as a pixel with $x = 4.0$ and an offset of -0.4. This is akin to the pixel slowly moving from one pixel over to the next. As it moves, its contribution decreases from one, and increases in the next. The wrap-around ensure that the offset is normalised, and is between -0.5 and $+0.5$.

The wrap-around effect is implemented by a simple conditional check before we generate the offset for the Gaussian kernel, like so:

Algorithm 5.2 The algorithm that implements the wrap-around

```
if shiftx > 0.5
        shiftx = 1-shiftx; % wrap it around
end
if shifty > 0.5
        shifty = 1-shifty; % wrap it around
end

//generate the kernel for this offset
Offsetkernel = GenGaussian(shiftx, shifty)
```

5.4.2 Programming the Kernel in Matlab

Since the Gaussian kernel is such an important tool in Computer Graphics and Vision, it is provided as one of the built-in *filters* inside Matlab. Given the filter type (Gaussian), and kernel size (in pixels) and the sigma, the following function can be used to generate a standard, centred Gaussian kernel:

```
fspecial('gaussian', [kernelsize kernelsize], sigma)
```

At this juncture, it's important to note the distinction between the point sprite size, and the kernel size. The point sprite size is the size of the final circular *splat* that will be displayed on screen. We use a GPU feature called point sprites to represent this splat. This is usually small (3x3 or 5x5). The kernel size, on the other hand, signifies the number of subdivisions within the point sprite. The kernel size in effect determines how fine sub-pixel shifts can be. The values in the kernel are summed up to determine the intensity of the relevant pixel in the point sprite.

The kernel size and the point sprite size have a linear relationship. The kernel is an expanded (subdivided) version of the point sprite. Hence, given an expansion factor, we can determine one from the other.

One caveat to keep in mind is that the kernel size must be odd, since the point sprite must have an odd number of pixels. A conditional check at the time of the kernel generation is implemented as outlined in algorithm 5.3.

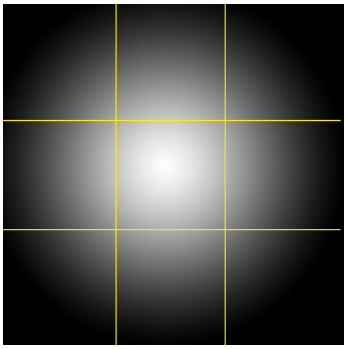


Figure 5.4: A centred point sprite, and the underlying Gaussian function it approximates

Algorithm 5.3 Determining the size of the kernel

```
test = ceil(sigma*4.0);
if test is even
        kernelSize = test+1;
else
        kernelSize = test;
end
```

Here, `sigma*4.0` is the size of the kernel after it has been expanded. If it is even, it is made odd by the addition of 1, otherwise it retain its original value.

The Matlab filter, by default, generates a centred Gaussian kernel. However, in order to implement sub-pixel shifts, shifted Gaussian kernels must be generated. Apart from the solution of mathematically regenerating the kernel for each possible shift, we can do so by using a 'sliding-window' (to borrow a familiar term).

5.4.3 Generating the offsets

Assuming the kernel is normalised, each of the pixels in the point sprite is a sum of the relevant pixels in the kernel. Instead of regenerating the kernel for each shift, the kernel can be copied over into a larger window and slid around in the window. The shifted pixels can then be summed as they would have been had the kernel actually been regenerated in a shifted position. The window is an empty kernel with a one-pixel extra border to allow for the shift. Mathematically only a single-pixel border is required since the shift can be a maximum of half-a-pixel in any direction before we wrap-around to the next pixel as discussed earlier.

Generating the kernels is a time consuming process. For each pixel, it involves a sum of all the pixels in a kernel, as well as some processing to calculate the shifts. Given

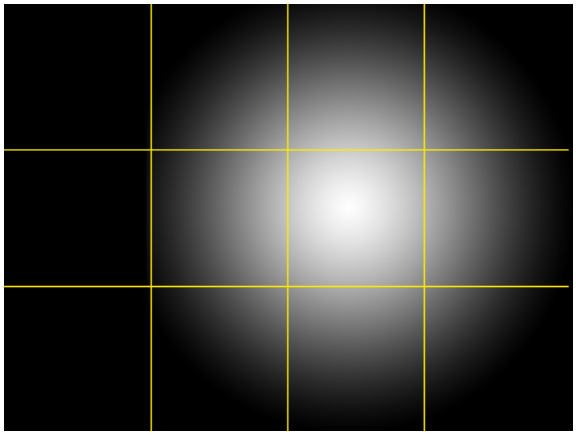


Figure 5.5: kernel offset within a larger window

that the kernel size is known in advance, all possible shifts can be generated in advance, and re-loaded just before rendering in a pre-processing step. Storing the kernels in a two-dimensional array allows the usage of the shifted values (multiplied by the expansion factor) as an index into the array to retrieve the appropriate kernel in a very rapid manner.

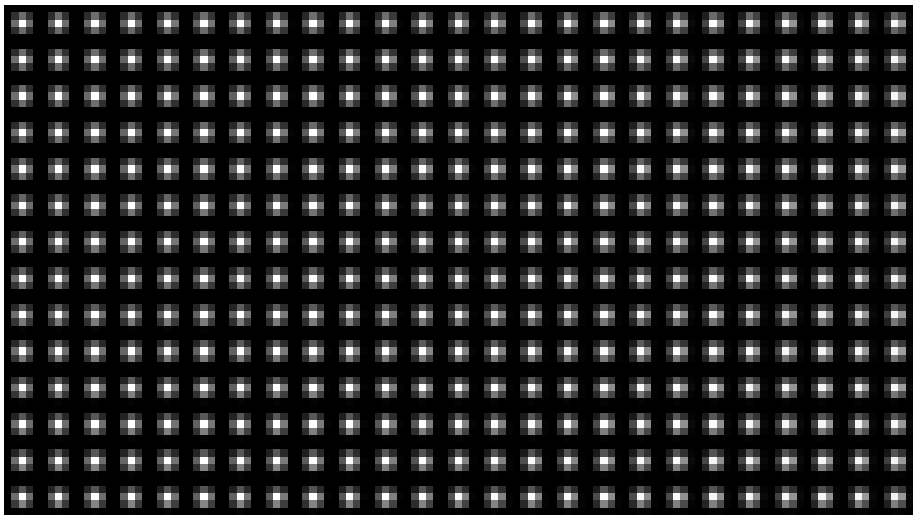


Figure 5.6: A crop from the final LUT

5.5 Displaying the point sprites via GLSL

A *point sprite* is a screen-aligned polygon that always faces the camera, and its position is defined by a single point, its centre. Other properties of a point sprite include its size, whether its smoothed, and possibly a texture. Point sprites have proved popular for implementing particle systems as each particle may be drawn as a hardware accelerated point sprite. Since point sprites only require a single point to define their location, they

have lower memory requirements than regular screen-aligned polygons, and are often faster to render than polygons of the same size because they do not have to perform calculations that manually align them to the screen, and gave a constant depth over the entire area. In essence, a point sprite may be treated as a scaleable hardware accelerated point.

We may use a point sprite to represent the 3x3 kernel we generated earlier. After having generated LUT, accessing the correct kernel for a particular point sprite is a matter of indexing. The indexing will depend on the sub-pixel value of the point. On a GPU, after rasterisation, pixel values are returned at integer coordinates. In general, it is not possible to directly access sub-pixel values. It is therefore necessary to generate the sub-pixel manually in the vertex shader, before rasterisation by the fragment shader. Algorithm 5.4 shows how sub-pixel values may be retrieved in a vertex shader.

Algorithm 5.4 The vertex shader in GLSL that provides us with sub-pixel coordinates after projection

```
1  //VERTEX Shader
2
3  out vec2 FinalPos;
4  in vec4 UVIndex;
5  uniform float Width;
6  uniform float Height;
7  void main()
8          {
9          gl_FrontColor = gl_Color;
10         gl_Position = ftransform();
11         vec3 n=gl_Position.xyz/gl_Position.w;
12         vec2 lowerleft =vec2(0.0,0.0);
13         vec2 origin;
14         origin.x=lowerleft.x+Width/2.0;
15         origin.y=lowerleft.y+Height/2.0;
16         FinalPos.x=(Width/2.0)*n.x+origin.x;
17         FinalPos.y=(Height/2.0)*n.y+origin.y;
18
19         //in the fragment shader, we will obtain
20         //the fractional values by simply this:
21         //FinalPos - trunc(FinalPos)
22         }
```

5.6 Normalisation

A monitor has a fixed gamut of luminance values that can be displayed physically. Generally, colour is represented in 24-bits and split into red, green, and blue *channels*, a byte

each. This means that a pixel can have a maximum *energy* of 255 per channel. In Gaussian projection, each point *over-wrote* the pixels it occluded, therefore ensuring that the maximum energy for a pixel would remain within limits (a byte per channel). During Laplacian projection, however, projected pixels are *summed* with the values already in the frame buffer. As points accumulate on the frame buffer, the energy for pixels on the frame buffer often exceeds the threshold, and the pixel is displayed as a bright white spot.

In order to keep the energy of each pixel in the frame buffer within limits, the pixels must be normalised. An additional buffer, which we shall call a normalisation buffer, can be used to perform normalisation with little additional effort. The psuedocode in algorithm 5.5 shows how the normalisation buffer is used.

Algorithm 5.5 Psuedocode for normalisation

```
1  //clear normalization buffer to zeroes
2  NormalizationBuffer = CreateBuffer(0);
3
4  //Draw the point and update the buffer
5  for each pixel in range projected at i,j
6          {
7          FrameBuffer(i,j) = RenderPoint();
8          NormalizationBuffer(i,j) = NormalizationBuffer(i,j) + 1;
9          }
10
11 //Now normalize the frame buffer
12 for each pixel in FrameBuffer indexed by x,y
13         {
14         FrameBuffer(x,y) = FrameBuffer(x,y) / NormalizationBuffer(x,y);
15         }
```

The normalisation buffer is first cleared (line 1-2). The normalisation buffer will keep a count of the points falling into the frame buffer at this particular location. All the points are then rendered to their respective final locations in the frame buffer (lines 4-9), here referred to by indices i,j. The count of the pixel indexed by i,j in the frame buffer is increased to indicate it has received energy (a point has been drawn there). Finally (lines 11-15), the frame buffer is *normalised*, i.e, the energy accumulated into each pixel of the frame buffer is divided by the number of points falling into it.

5.7 Discussion and conclusion

In this chapter I demonstrated how antialiasing works with the proposed renderer. I presented an algorithm that is amenable to hardware acceleration, and naturally works with points as a native rendering primitive. It takes advantage of point sprites as hardware accelerated textured 2D points in order to perform sub-pixel anti-aliasing on modern GPUs. An important point is that the proposed anti-aliasing technique relies on the GPU to follow exactly the OpenGL specification. This is so that when the algorithm computes sub-pixel values manually, the computed values are the same as those generated by the GPU for rasterisation of those points. Unfortunately, this is not always so, as I have noticed that a driver update on May 2010 caused the NVidia GTX 8800 to fail to generate pixels at locations determined by the algorithm. The ATI machines, however, still reported correct anti-aliasing.

In the next chapter, I will discuss the multi-view merits of this algorithm, and discuss some problems with multi-view rendering. In addition, I will present Laplacian projection, an offline rendering algorithm suitable for high-resolution merging of 3D models in image space.

Chapter 6

Multi-view Intergration and Rendering Algorithms

Having described Pyramidal projection, a multi-resolution algorithm for point-based rendering of large range images on the GPU, I will now describe the hitherto unsolved problem of multi-view rendering of multi-pod data.

This thesis presents two multi-view algorithms, the first is a simple extension to the Pyramidal projection method proposed earlier. The purpose of this extension is to permit *real-time* multi-view blending, keeping in mind the need to forego preprocessing. The second method, which I call **Laplacian projection**, is a high quality offline multi-view algorithm (published in [57]), that is based on Laplacian pyramids and the Burt and Adelson multi-resolution splining algorithm [27].

6.1 Real-time Multi-view Rendering

A significant number of 3D scanners are view dependent. Stereo-photogrammetry techniques in particular rely on a stereo camera setup (each called a pod) that, by virtue of its limited field-of-view, only has a partial view of the object to be scanned. In order to overcome this limitation, it is common to have multiple pods set up around the object. A rendering of the complete view of the object therefore requires the patches from different views to be merged together. This merging process is usually part of the surface reconstruction phase. Pyramidal projection obviates the need for a separate preprocessing pass, therefore merging is done as part of the rendering operation.

6.1.1 Extending Pyramidal Projection

The proposed method benefits from its reliance on points as a rendering primitive rather than polygons, and its two pass rendering nature. Since pyramidal projection relies on points, complex and error prone algorithms such as *Marching Cubes* that rely on polygon merging can be avoided, and overlapping points from multiple views in the first pass of the algorithm can simply be blended together to form the final multi-view image.

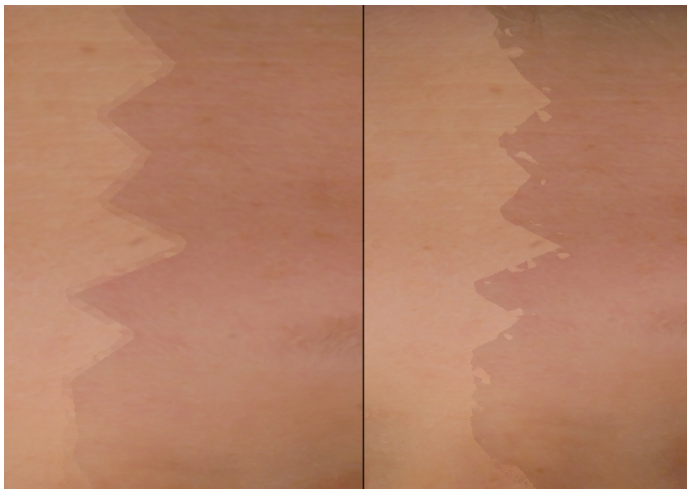
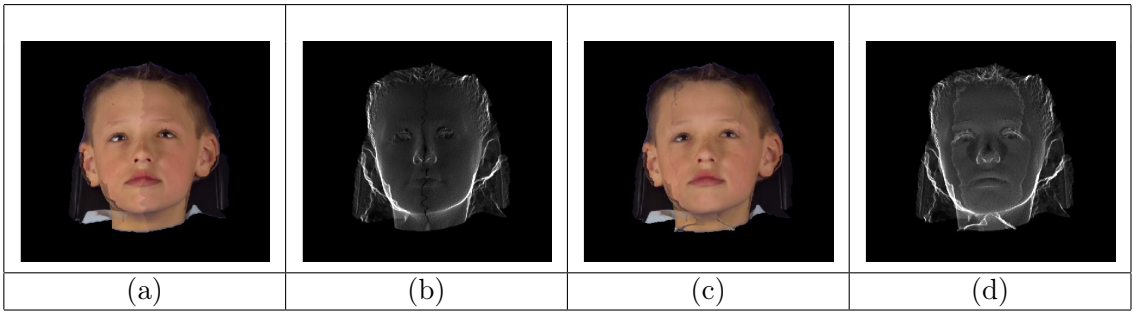


Figure 6.1: With (left) and without (right) a simple linear blending between overlapping pixels in the two views.

We can recap the rendering algorithm, with multi-view rendering included, as follows:

1. From **T1**, **R1**, **M1**, and **T2**, **R2**, **M2** create $\widehat{T1}$, $\widehat{R1}$, $\widehat{M1}$ and $\widehat{T2}$, $\widehat{R2}$, $\widehat{M2}$.
2. Create $\widehat{F}$ to store the final rendered image pyramid.
3. For each level of $\widehat{T1}$, $\widehat{R1}$, $\widehat{M1}$ and $\widehat{T2}$, $\widehat{R2}$, $\widehat{M2}$.
   (a) Render the first view into $\widehat{F_i}$
   (b) Turn on appropriate blending mode.
   (c) Render the second view into $\widehat{F_i}$
4. Overlay all the layers in $\widehat{F}$ for appropriate hole-filling.

The effect of this blending, with a linear interpolation, can be seen in figure 6.1. It should be noted that blending will only work if masks have appropriate overlap at the boundaries so that they can be blended via our blending mechanism. Also, the larger the overlap, the smoother the discontinuity between the two views.



(a) (b) (c) (d)

Figure 6.2: An example of the multi-view integration with (left) less overlap and (right) more overlap. These images were rendered on a CPU via Matlab.

6.2 Laplacian Projection

Image pyramids have the potential to provide more convincing blending than linear blends as outlined earlier. Burt and Adelson have expanded their original Laplacian Pyramids to permit natural blending of multiple images seamlessly to create one mosaic, an algorithm known as the multi-resolution spline [27]. I have extended the multi-resolution spline algorithm to 3D imagery. In order to explain the extension, I will first explain the multi-resolution spline.

6.2.1 Introduction to the Multi-resolution Spline

The working of the multi-resolution spline is best summarised in the original paper proposing the technique, as follows [27]:

"We define a multi-resolution spline technique for combining two or more images into a larger image mosaic. In this procedure, the images to be splined are first decomposed into a set of band-pass filtered component images. Next, the component images in each spatial frequency band are assembled into a corresponding bandpass mosaic. In this step, component images are joined using a weighted average within a transition zone which is proportional in size to the wave lengths represented in the band. Finally, these band-pass mosaic images are summed to obtain the desired image mosaic. In this way, the spline is matched to the scale of features within the images themselves. When coarse features occur near borders, these are blended gradually over a relatively large distance without blurring or otherwise degrading finer image details in the neighbourhood of the border."

It is important to note here that the definition of the word **spline** here is taken differently from as it is generally understood in a mathematical or graphics context. Rather than referring to a piece-wise curve, it refers to techniques that deal with seam removal in

mosaics, as explained later in the paper:

"The problem...may be stated as follows: How can the two surfaces be gently distorted so that they can be joined together with a smooth seam? We will use the term image spline to refer to digital techniques for making these adjustments."

6.2.2 Properties of the Multi-resolution Spline

The goal of the multi-resolution spline is to eliminate the seam that appears at the boundary where two images have been merged. At the simplest, the boundary may be made less prominent by adding a linear interpolation (a linear ramp) to n pixel values on either side of the boundary to arrive at equal values on the boundary itself. This method produces a smooth blending between the two images, however, depending on the size of n, the seam may not be entirely invisible. If n is too small, the seam will appear as a blurred edge between the images. If n is too large, a blend of features from both images will be apparent at the boundary, resulting in ghosting, similar to a double exposure of a negative in photography.

The choice of n is therefore related to the size of the features in the image. If n is any larger than the smallest prominent features in the image, then a double exposure effect will be visible. On the other hand, to prevent a seam from being visible, the transition n should be at least comparable in size to the largest prominent features in the image. For a majority of common images, a suitable n cannot be computed that will satisfy both requirements. Stating the requirements more formally in terms of spatial frequency, and taking the Nyquist limit into account, the paper proposes the requirements as follows [27]:

"T should be comparable in size to the wave-length of the lowest prominent frequency in the image. If T is smaller than this the spline will introduce a noticeable edge. On the other hand, to avoid a double exposure effect, T should not be much larger than two wave lengths of the highest prominent frequency component in the images."

T refers to the transition zone we have defined as n. It is this requirement that leads to the conclusion that the band width of images to be splined should be roughly one octave. Since an appropriate T cannot be found for an image that occupies more than an octave, the image is decomposed into a set of band-pass component images, and a separate spline with an appropriately selected T is then performed in each band.

The entire procedure is thus: First, the images that are to be blended are converted to Laplacian pyramids. A Laplacian pyramid decomposes an image into a set of band-pass

filtered component images, each an octave apart from the subsequent image in the pyramid. The layers (representing each spatial frequency) are assembled into a bandpass mosaic. During this phase, the individual layers in the pyramid are blended with a transition zone that is proportional in size to the wave lengths represented in the band. Lastly, the blended Laplacian pyramid is *reconstructed*, i.e, the individual layers in the Laplacian pyramid are expanded and summed to obtain the final blended image.

The novelty of using a multi-resolution spline is that the blend is sensitive to the frequency component of the images, i.e, larger features can be blended over a larger transition zone, whereas smaller-scale features receive blending over a smaller transition zone to preserve detail.

6.2.3 The Proposed Method: Laplacian Projection

Laplacian Projection is an extension of the original multi-resolution spline to handle merging of 3D views. The multi-resolution spline is an image space algorithm, i.e, it operates on images rather than 3D objects. The typical images provided as input to the multi-resolution spline are photographs, which implies that the input images have gone through the perspective/viewing projection/transformation process. Laplacian Projection is based on the idea that each of the *renders* from multiple views can be treated as separately rendered *virtual photographs* that can be submitted to the multi-resolution spline algorithm as inputs. This obviates the need to stitch together the multi-view 3D objects in object-space before rendering, and treats it as a post-processing problem that may be solved after rendering.

6.2.3.1 Rendering in 3D from Laplacian image pyramids

Rendering a 3D image from a Gaussian image pyramid is intuitively and programmatically simple. Each of the images in the texture Gaussian image pyramid $\hat{T}$ provide the intensity information, while the range Gaussian pyramid $\hat{R}$ provides the spatial information. In Laplacian Projection, the intensity information (the texture pyramid) $\hat{T}$ is obtained via a Laplacian pyramidal process. Hence, each, the base image is a Gaussian filtered image that contains the lowest frequencies, while the remaining images are obtained via the Difference-of-Gaussians approach mentioned in chapter 3. Each of the images in $\hat{T}$ in Laplacian Projection represents a different frequency spectrum. The range image pyramid $\hat{R}$ is still derived via a Gaussian process, therefore the spatial information remains unmodified. Combined, the intensity information $\hat{T}$ and range information $\hat{R}$ are enough to

recreate an image pyramid containing renders of a 3D model. The resultant pyramid is a standard Laplacian pyramid that when reconstructed (when all levels are upsized and added together), will faithfully recreate the original 3D model with all the frequency content of the original.

It is worth noting that using a Laplacian Pyramid to store the intensity images for a 3D rendering does not introduce errors or distortions. This is evident from the fact that a Laplacian pyramid is simply an affine transformation (subtraction to be precise) of the Gaussian pyramid, so that if given that the Gaussian pyramid does not introduce errors during rendering where $\hat{T}$ and $\hat{R}$ are both Gaussian filtered, then it is mathematically valid to assume that a Laplacian pyramid will not introduce errors either where $\hat{T}$ is a Laplacian pyramid and the range pyramid $\hat{R}$ is still Gaussian.

In this manner, we extend the multi-resolution spline to handle 3D objects, where the viewpoint may be dynamically altered. The primary difference being that the source images are 3D renders generated from range images. Consequently, Laplacian projection performs 3D multi-view integration in image space via a multi-resolution spline. Laplacian Projection relies on a Difference-of-Gaussians (DoG) pyramid to simulate a Laplacian filter.

Algorithm 6.1 provides an overview of the Laplacian Projection process. Since Laplacian Projection is an offline rendering process, and does not require special allocation of hardware buffers, it may be completed in a single pass. The algorithm is similar to Gaussian Projection. It is important to note, however, that each level in the pyramid represents different band-pass frequencies. Consequently, each time a new level is rendered, it adds more detail to the base image. The rendering *adds* samples during rendering rather than replacing them (as is common in rendering).

Algorithm 6.1 An overview of the algorithm for Laplacian Projection

1. Setup
   (a) Load Range, Texture, and Masks called R,T, M
   (b) Generate Gaussian pyramids $\hat{R}$ and$\hat{M}$ from R and M. Generate a Laplacian pyramid $\hat{T}$ from T
   (c) Allocate buffer to hold intermediate output in F. Set F to hold smallest level (Gaussian) image in $\hat{T}$.
2. Pass 1
   (a) Render each model in $\hat{R}$,$\hat{T}$,$\hat{M}$ into F like the following:

```
for every image at index i, in ^R–1
{
Expand F by one octave
Set rendering mode to "addition"
Render each ^R ,^T, ^M into F
}
```

3. Display F

The complete algorithm, including multiple views, is presented in algorithm 6.2. The resulting images have a smooth blend between two different models.

6.3 Handling Occlusion

6.3.1 Back-face Culling

Laplacian projection is similar to Pyramidal Projection, with the introduction of a Laplacian pyramid to replace a Gaussian pyramid for the textures, and a pyramidal reconstruction phase instead of the *overlay* in the last step. The difference in blending mode means that the rendered views to be splined must already have hidden surfaces (points) removed, otherwise the otherwise hidden points will contribute to the splining operation, causing unintentional blending.

Occlusion is an important depth cue in natural vision. Opaque objects block light from objects between the light source or another object and the observer. If the opaque object blocks a light source, this causes the observer to be in shadow. If the opaque object blocks reflected light arriving from another object, it causes occlusion. Occlusions cause objects closer to the eye to block objects further away. These occluded surfaces must be removed

Algorithm 6.2 The algorithm explaining the basic procedure for Laplacian Projection

```
1  //PREPROCESSING
2  int N;          //Levels in a multi-resolution pyramid
3  //Load Range, Texture, and Masks
4  Load(R1,T1,M1)
5
6  //Load second view
7  Load(R2,T2,M2)
8
9  ^R1 = GaussianPyramid(R, N)
10 ^T1 = LaplacianPyramid(T, N)
11 ^M1 = GaussianPyramid(M, N)
12
13 ^R2 = GaussianPyramid(R, N)
14 ^T2 = LaplacianPyramid(T, N)
15 ^M2 = GaussianPyramid(M, N)
16
17 //^R1,^T1,^M1 together constitute a series of anti-aliased 3d Models
18 // The original models must overlap (through masks) by atleast a pixel.
19
20 CreateBuffer(^F) //to store the final rendered image pyramid.
21
22 For each level of T1, R1, M1 and T2, R2, M2
23         {
24         Render the first view into ^F[i]
25         SetBlendingMode(ADD)
26         Render the second view into ^F[i]
27         }
28
29 FinalImage = ReconstructPyramid(^F)
```

in order to obtain a more realistic image, as well as improve efficiency by culling away objects that cannot be seen, thereby reducing computational load. Various hidden surface removal algorithms exist, and the choice of which one to use depends on the parameters of the application.

In Pyramidal Projection, occlusions were handled automatically by the GPU through the use of a Z-Buffer. If the composite blending mode is "add" rather than replace, then the depth buffer will not allow points nearer to the camera to overwrite points behind, but rather, sum them together. Hence, these occluding points must be culled *before* blending takes effect. Back-face culling has been used as a first-cut easy-to-implement algorithm to cull a large number of polygons based on their orientation alone. Back-face culling assumes a polygonal data source, however, we have provided an implementation that works for

point-based-rendering as well with minor modifications.

Algorithm 6.3 Back-face culling

```
1  //v1,v2,v3 are three neighbouring vertices on a polygon
2  //camera = camera vector
3  e1 = v3 - v1
4  e2 = v3 - v2
5  n = cross(e1,e2)
6
7  if dot(camera, n) < 0
8          Render();
9  else
10         handle_backface();
```

Back-face culling refers to the process of eliminating polygons that are not facing the camera. The idea is that polygons that have a normal that points away from the camera are most likely the back-facing polygons of a closed convex shape, and hence are occluded by the front-facing polygons of the same shame. Since back-face culling is by definition view-dependent, it must be carried out at render-time in order to accommodate changing viewpoints. Back-face culling assumes that every polygon has an orientation, and that that every polygon facing away from the camera is hidden. The last assumption may not always hold true, however, it is a heuristic that works well for most objects, especially convex objects such as a sphere, or cube. The normal to a polygon is representative of its orientation. If the normal of a polygon faces away from the camera vector, then we may consider the polygon as being a back-facing polygon. Algorithm 6.3 suggests a general polygon-based back-face culling algorithm.

Given three neighbouring vertices $v1$,$v2$ and $v3$, it is possible to calculate the normal n to the face f by first constructing two vectors $e1$ and $e2$ that represent the edges of the polygon. The normal n is simply a cross-product of the two edges:

$$n = e1 \times e2 \tag{6.1}$$

The next step is to determine if the normal is in the direction of the camera. Generally, the dot product relates two vectors by the angle between them:

$$a \cdot b = |a|\,|b|\,cos\theta \tag{6.2}$$

If the camera vector c and normal n are normalised, then the equation simplifies to

$$c \cdot n = cos\theta \tag{6.3}$$

Hence, c and n are related by the cosine of the angle between them. The cosine conveniently normalises the angle. This signifies that if the dot product is less than 0, the surface is facing the camera, and hence, can be seen. On the other hand, if the dot product is greater than or equal to 0, the surface can be culled (i.e, it is completely facing the other direction).

Polygons retain connectivity information, making it trivial to determine edges $e1$ and $e2$. Generally, point clouds do not contain such information. Range images, however, despite being point data, maintain connectivity information. For a point-based rendering mechanism, for data sources where the connectivity information is retained (such as range images), back-face culling works with only a minor modification. The neighbourhood information in the range image depicts surface continuity. Hence each group of 3 neighbouring vertices in a range image can be treated mathematically as a polygon, and a *normal* can be constructed by treating each point as if it were a vertex of a polygon. This method, however, presents us with a per *face* normal.

A per *point* normal may be computed in various ways, including differential techniques. While this may be done in real-time, since Laplacian Projection is an offline algorithm, this is performed as a preprocessing step to save computational load during rendering. I have used *bicubic spline surface-fitting* to produce a precomputed normal map for rendering purposes. Once a per-point normal has been obtained, algorithm 6.3 may be repeated with the new found normal, and each point may be culled independently of other points. While this technique is expensive in terms of computation time, it is acceptable for an offline algorithm such as Laplacian projection.

6.3.2 Enhancing Hidden Point Removal

Back point culling is an efficient method for culling a majority of the hidden points, however, it assumes a convex surface. In the future we would like to be able to scan various other body parts such as the human hand, which may pose more complex occlusion problems. Currently, hidden surface removal is handled primarily via a z-buffer.

Z-Buffering is a depth-ordering algorithm used almost universally by GPUs today (as evidenced by their inclusion as a standard feature in standard graphics APIs such as DirectX and OpenGL [14]). Although other methods exist, z-buffers are simple to implement,

efficient, and amenable to hardware acceleration. As opposed to back-face culling (which operates in object or world-space), z-buffering is performed during the last phase of the rendering pipeline, i.e, during rasterisation. During rendering, for every rasterised point, the z-value of the incoming (currently rasterised) point is compared with the z-value of a pixel already in the buffer. If the z-value is greater, the incoming point is occluded and is discarded. If the z-value is smaller, the incoming point is closer to the camera than the pixel existing in the z-buffer, and hence, is overwritten with the newer point. In this manner, the z-buffer makes sure the scene is depth-ordered correctly, with pixels closer to the camera overwriting pixels further away.

A hierarchical z-buffer may be thought of as an extension of the z-buffer in scale-space. The *proposed* hidden point removal method is a variation of the *hierarchical z-buffer [69].* The basic premise of our hidden-point removal algorithm is similar to that for our hole-filling algorithm: *The rendering of the lowest-resolution image must contain no holes, and hence, be correctly rendered via a hardware accelerated z-buffer.*

Since OpenGL provides z-buffering by default [7, 14], manual z-buffer management is generally not necessary. The proposed algorithm, however, poses problems during depth-sorting due to the existence of holes. Holes create depth discontinuities, and as points are shown that should otherwise have been occluded, the illusion of a continuous surface is broken. Repeating the basic assumption that the lowest-resolution image must contain no holes, and therefore be correctly depth-sorted, it can be deduced that similar to hole-filling described in chapter 4, information from the correctly sorted image at a higher level in the pyramid (one with less information) can be used to fill in information as we travel down the pyramid. The idea is similar to the variation of hierarchical z-buffers as proposed by Grossman and Dally [71], however, our HPR algorithm is well-integrated with our rendering algorithm requiring little additional effort, and it makes extensive use of the GPU. Since currently Laplacian projection is an offline algorithm, the hidden point removal algorithm has been integrated into Pyramidal Projection.

Algorithm

Algorithm 6.4 Psuedocode describing preprocessing and pass 1 of the proposed hidden-point removal algorithm

```
1  //PREPROCESSING
2  int N;          //Levels in a multi-resolution pyramid
3  //Load Range and Texture
4  Load(R,T)
5
6  ^R = GaussianPyramid(R, N)  //Range images
7  ^T = GaussianPyramid(T, N)  //Textures
8  //^R,^T and together constitute a series of anti-aliased 3d Models
9
10 //Allocate memory for rendered (projected) images
11 CreateBuffer(^F);
12 CreateBuffer(^Z);
13
14 //PASS 1
15 //Render each model in ^R,^T into ^F
16 for(i=0; i<N; i++)
17 {
18         //set all values, including alpha, to zero
19         ClearBuffer(^F[i]);
20
21         for (every pixel indexed by u,v in ^R[i])
22         {
23         //compute position after projection
24         newposition = RenderPoint(^R[i][u][v]);
25
26         //save colour of projected poiint
27         ^F[i][newposition.x][newposition.y] = ^T[i][u][v];
28
29         //save u,v of current point into current Z
30         ^Z[i][newposition.x][newposition.y] = uv;
31         }
32 }
```

Algorithm 6.5 Psuedocode describing pass 2 of the proposed hidden-point removal algorithm

```
1  //Now ^Z is a collection of projected
2  //images at various resolutions where each projected pixel
3  //contains u,v values indexing the source range image
4
5  //PASS 2
6  //Allocate memory for final image
7  CreateBuffer(FinalImage);
8  CreateBuffer(currentImage);
9
10 //Render lowest-resolution image first
11 RenderIntoBuffer(^R[N-1], ^T[N-1], CurrentImage);
12 //Up-scale to screen-size
13 FinalImage = ExpandImageToScreenSize(CurrentImage);
14
15 for(i=N-2;i>=0;i--)
16         {
17         createBuffer(oldImage);
18         //Expand the image using linear interpolation
19         oldImage = ExpandImageByFactorOfTwo(^F[i-1]);
20         currentImage = ^F[i];
21
22         for every pixel in oldImage
23                 if difference(oldImage.uv, currentimage.uv/2.0) < epsilon
24                         RenderPointIntoBuffer(FinalImage);
25         }
26
27 DisplayImage(FinalImage);
```

Range images are represented as a 2D matrix in which each element may be uniquely identified in *(u,v)* coordinate space. Each point at each level in scale-space in the range image pyramid $\widehat{R}$ is assigned a unique identifier (the *u,v* coordinate) that is stored as an additional attribute of each point (see appendix). Then, an additional buffer, we will call this $\widehat{Z}$, is created. Every time an image is rendered from $\widehat{R}$ into $\widehat{F}$, its corresponding buffer in $\widehat{Z}$ is filled with the unique identifiers representing the points being projected. This allows us to identify uniquely each point in screen-space *after* it has been projected. Recalling the basic rendering pipeline from chapter 4, each image is stored in the GPU as a texture in a pyramid called $\widehat{F}$. While rendering to $\widehat{F}$, we can simultaneously render the *(u,v)* attributes to $\widehat{Z}$ via MRT (see appendix). Since rendering is done with a z-buffer, $\widehat{Z}$ will contain *(u,v)* values of only the *visible* pixels. This means that $\widehat{Z}$ effectively contains a set of textures that identify for each corresponding texture in $\widehat{F}$ which particular point is visible, by providing us with its *(u,v)* value.

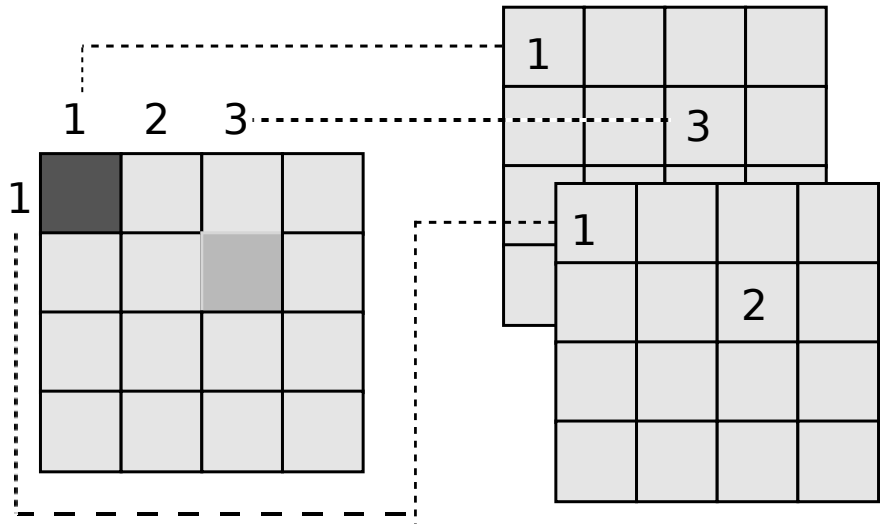


Figure 6.6: One way of computing unique identifiers is to store the index values of a range image (left) as extra attributes for a vertex. If vertex attributes are not present, these unique indices may be stored in separate arrays aligned with the range image (right)

For a rendering of $\widehat{R_{N-1}}$, where N is the number of levels in a pyramid, $\widehat{Z_{N-1}}$ effectively contains a list of identifiers for each visible point in the original range image at the apex of the pyramid. Recall that we assume that the apex of the pyramid is small enough to be hole-free. Since each pixel in a pyramid has a predictable relationship to a pixel across other pyramidal levels due to scale-space continuity, it is possible to predict visibility of an entire image from the correctly ordered image and its visibility information stored in $\widehat{Z_{N-1}}$. When an image is rendered from the apex of the pyramid downwards, the points in $\widehat{R_i}$ can be deemed visible if the corresponding points in $\widehat{R_{i-1}}$ are visible.

Starting with $\widehat{R_{i-2}}$, the image just below the apex of the pyramid, in the vertex shader, we can check its *(u,v)* value to see if it is the same, or near (say delta) the *(u,v)* value of the corresponding pixel in the image $\widehat{R_{i-1}}$. If the test passes, we allow the pixel to be drawn since it means it was passed by the z-buffer at a lower resolution as being the fore-most pixel, and if it fails, we reject it as being occluded. From here onwards, we can continue in this manner all the way up the pyramid until the $\widehat{R_0}$ has been rendered.

Discussion

It is important to note here that even though we have determined the visibility of a point at the vertex shader, modern hardware does not permit it to be discarded other than in the fragment shader, even for point-based rendering. A simple method to discard the vertex is to move it outside the viewing frustum so it will be culled, however, this entails additional

computation and we hope that in the future, a vertex-discard will be permitted in the vertex shader where the rendering primitive is a point.

Another point to note about the hidden point removal algorithm is that it assumes a blending mode where each new pixel completely overwrites the old one. That is, $newPix = (1.0 * incommingPix) + (0.0 * existingPix)$ where $incommingPix$ is the pixel which is to be rendered and$existingPix$ is the pixel that is already rendered in the location at which $incommingPix$ is to be rendered (`glBlendFunc ( GL_ONE, GL_ZERO )` in OpenGL). This makes it difficult to perform anti-aliasing as the proposed anti-aliasing algorithm relies on an accumulative blending mode. That is, $newPix = (0.5 * incommingPix) + (0.5 * existingPix)$ where $incommingPix$ is the pixel which is to be rendered and$existingPix$ is the pixel that is already rendered in the location at which $incommingPix$ is to be rendered (`glBlendFunc ( GL_ONE, GL_ONE )` in OpenGL).

Finally, it is important to take z-fighting into account. Z-fighting occurs during 3D rendering when two or more primitives have similar values in the z-buffer. Affected pixels are rendered from one point or the other arbitrarily, in a manner determined by the precision of the z-buffer. In an image pyramid, image have less information as the pyramid is traversed upwards and hence z-fighting is more common. A low-resolution z-buffer can only select select a single pixel where many pixels would otherwise have been (even those with very nearly the same depth) rendered in the high-resolution version from various locations. By choosing only a single point, and therefore a single z-value as the visible z-value, the low-resolution z-buffer selects a certain set of points and rejects all others. While this is intended, this causes problems when a surface is continuous and many points share similar, but not the same z-value. Several points on a surface are discarded as not having the correct z-value, despite those points being on the same surface. This causes neighbouring points in the subsequent layers of the pyramid to be discarded, causing holes in progressively higher-resolution images in the pyramid. A solution is to take depth into account, and only reject a pixel if it has a depth difference greater than an epsilon value. Problems like these are part of my ongoing research into efficient hidden point removal.

6.4 Problems With Laplacian Projection

Since the last phase in generic pyramidal projection was responsible for scattered data interpolation, Laplacian Projection does not, by default, perform hole-filling. This also

implies that Laplacian Projection, as it is, requires the display resolution to match the source resolution in order to avoid introducing under sampling (that may cause holes). Oversampling, on the other hand, produces extensive blur since points that would otherwise be culled via a z-buffer are blended.

For an image that is natively captured at a resolution of 3000x4500 pixels, Laplacian projection at any viewport resolution lower than its native resolution would blur the image, depending on the amount of oversampling. The effect of this is visible in figure 6.7.

A Laplacian pyramid separates an image by frequency. Blurry images are visually associated with low-pass filters. Therefore, the additional blur due to oversampling and a lack of hidden point removal may be mitigated to an extent by increasing the contribution of the high frequency layers in a Laplacian pyramid. The effect of increasing the weights of high-frequency layers is visible in figure 6.8. Note, however, that this is a visual effect, and a lack of detail caused by the unwanted blurring will still be visible in areas where sufficient high frequency content is not available to cover the loss of detail, or where unwanted blending has damaged the image beyond repair.

6.5 Contributions

In this chapter, I explained the multi-resolution capabilities provided by the proposed algorithm, demonstrated how the proposed method performs multi-view rendering in real-time via screen-space blending, and showed that Laplacian Projection may be used for offline high-quality multi-view rendering if more precise multi-view blending is required.

Laplacian Projection is a novel algorithm that extends the multi-resolution spline to accommodate 3D images. It has been published in GRAPP 2009 [57] and is reprinted in Appendix B.

I also described back point culling, and how it is integrated into the proposed rendering algorithms. In addition, I presented an enhanced hidden point removal algorithm. The final details of the enhanced algorithm are part of ongoing research.

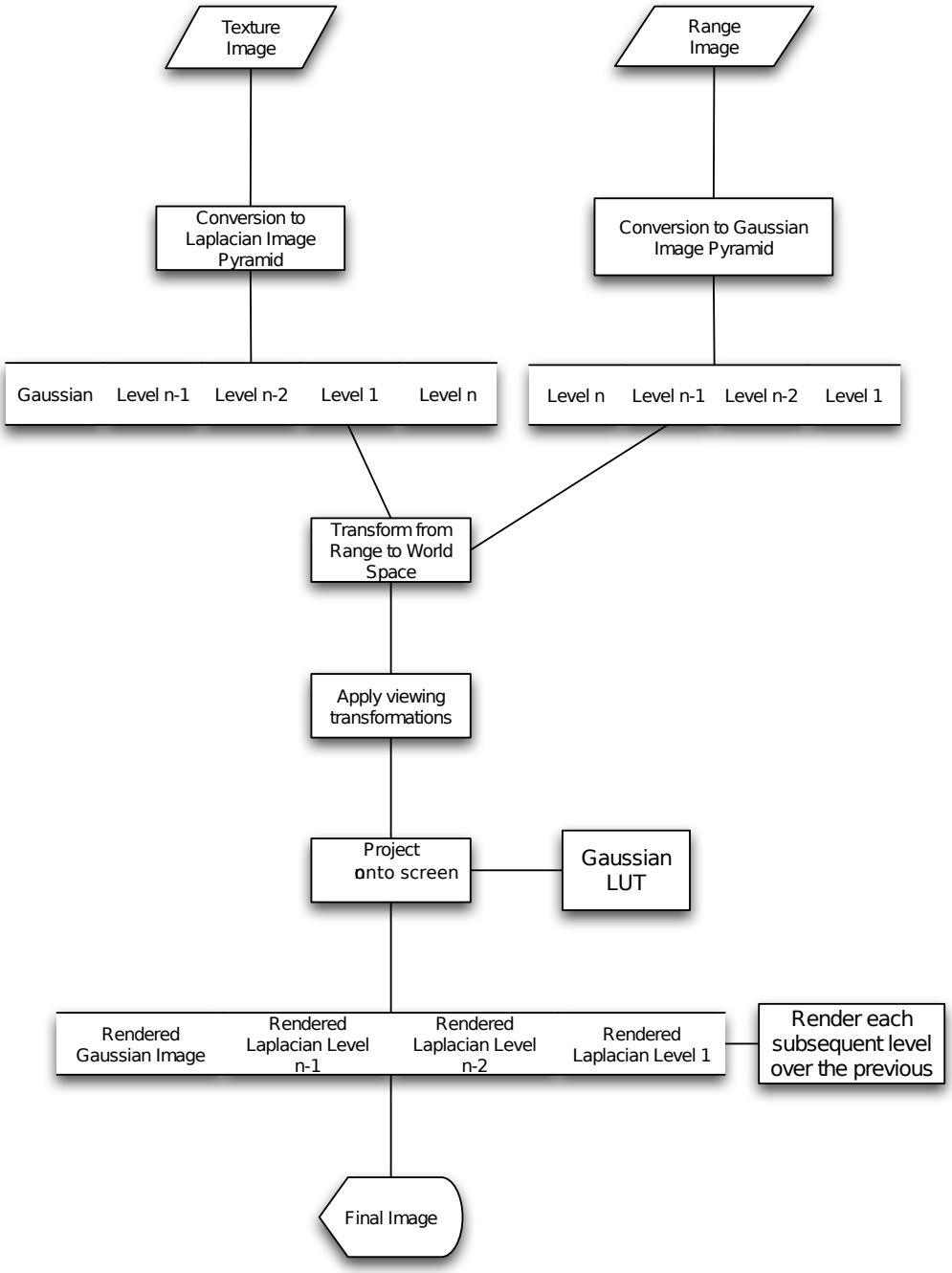


Figure 6.3: An overview of Laplacian Projection

Figure 6.4: Result of Laplacian Projection. Masking of the two views was omitted to make the blending process visible. Note the blending between the blue background and the face where the background should have been masked.

Figure 6.5: The back-face culled image (left) and the culled back-faces (right)

Figure 6.7: The effects of oversampling on two images rendered in differing viewports. The original image contains 3000x4500 pixels. (left) The image is rendered at a resolution of 800x800 pixels. (right) The image is rendered at a resolution of 2048x2048 pixels. Note that the closer the image is rendered to its native resolution, the fewer the effects of oversampling, and hence the sharper the image.

Algorithm 6.6 The vertex shader for hidden point removal

```
1  //VERTEX SHADER
2  uniform sampler2D PrevTexture;
3  out vec2 Texcoord;
4  out vec2 FinalPos;
5  in vec4 UVIndex;
6  uniform float Level;
7  uniform float Width;
8  uniform float Height;
9  const int MAXLEVELS = 4;
10 out int discard_point;
11 out vec2 currentUVIndex;
12 out float zvalue;
13 out float oldzvalue;
14
15 void main()
16 {
17 Texcoord    = gl_MultiTexCoord0.xy;
18 gl_FrontColor = gl_Color;
19 gl_Position = ftransform();
20 zvalue = gl_Position.z;
21 currentUVIndex = vec2(UVIndex);
22 discard_point = 0;
23 if(Level < MAXLEVELS-1)
24         {
25         vec3 n=gl_Position.xyz/gl_Position.w;
26         vec2 lowerleft =vec2(0.0,0.0);
27         vec2 origin;
28         origin.x=lowerleft.x+Width/2.0;
29         origin.y=lowerleft.y+Height/2.0;
30         FinalPos.x=trunc( (Width/2.0)*n.x+origin.x );
31         FinalPos.y=trunc( (Height/2.0)*n.y+origin.y );
32         vec2 normalizedFinalPos;
33         normalizedFinalPos.x = ((FinalPos.x+0.5) / Width);
34         normalizedFinalPos.y = ((FinalPos.y+0.5) / Height);
35         vec4 prevUVIndex = texture2D(PrevTexture, normalizedFinalPos);
36         oldzvalue = prevUVIndex.z;
37         vec4 compUVIndex = ceil(UVIndex/2.0);
38
39         if(compUVIndex == ceil(prevUVIndex) || compUVIndex == 0.0)
40                 discard_point =0;
41         else
42                 discard_point = 1;
43         }
44 }
```

Algorithm 6.7 The fragment shader for hidden point removal

```
1  //FRAGMENT SHADER
2  uniform sampler2D Kernels;
3  in vec2 FinalPos;
4  in int discard_point;
5  uniform float Level;
6  const int MAXLEVELS = 4;
7  in vec2 currentUVIndex;
8  in float zvalue;
9  in float oldzvalue;
10 uniform int Width;
11 uniform int Height;
12
13 void main()
14 {
15 if(Level < MAXLEVELS -1)
16         {
17         if (discard_point ==1)
18                 {
19                 if(abs(zvalue-oldzvalue) > 0.1)
20                         {
21                         discard;
22                         }
23                 }
24         }
25 gl_FragData[0] = gl_Color;
26 gl_FragData[1] =vec4(currentUVIndex.x,currentUVIndex.y,zvalue,1.0);
27 }
```

Figure 6.8: (left) The original Laplacian projection without any weight adjustment. Laplacian projection with the high-frequency layers given a higher weight during reconstruction of the pyramid. Note the marked improvement in visual fidelity. Also note the lack of detail in areas such as under the eyes.

Chapter 7

Experiment and Results

7.1 Performance Comparison : Empirical Data

I will now present a performance comparison of the proposed rendering system aginst other popular point-cloud rendering systems.

Name/publication	GPU details	Resolution	samples/second
XSplat [123]	GeForce 5900	512x512	6-12 Million
GPU Surface Splatting [18]	GeForce 6800 Ultra	512x512	23-25 Million
Deferred Blending [172]	GeForce 7800 GTX		up to 25 Million
Image Reconstruction [106]	GeForce 7800 GTX	1024x1024	50-60 Million
High Efficiency...on GPU [87]	GeForce 7300 GT	512x512	28-70 Million
Pyramidal Projection	ATI Radeon HD 5870	1680x1050	121.5-675 Million

Table 7.1: A comparison of Pyramidal Projection with other point-based rendering algorithms.

My testing setup consisted of a system with a single ATI Radeon 5870 GPU. Rendering performance was tested at a resolution of 1680x1050 pixels . As a sample, we took *Steve* and *Nairn* as our data-sets. These are two-pod DI3D captured heads with 3000x4500 pixels generated by each pod. In addition, alignment and masking information was provided in the DI3D file.

Each source image has 3000x4500 samples per view. For two views, the data consists of 27 million samples. I will now explain the results of some of the scenarios under which the data was tested on the ATI Radeon HD 5870 GPU.

The worst case scenario in Pyramidal Projection is when the entire object is visible (so no view-frustum culling can be performed) and with all the levels available for hole-filling

(in our case, 5 levels). This is an unrealistic scenario, however, since when fully zoomed out, the dataset is larger than the resolution of the viewing device (our monitor), and therefore hole-filling is not necessary. It still serves as a good *stress-test*, and may become the norm when retina displays become more common. In such a scenario, on an ATI Radeon HD 5870, and at a resolution of 1680x1050 pixels, frame-rates of between 4 and 5 frames-per-second were observed. This equates to an average of 121.5 million samples per second.

The best case for the renderer, in which the entire object is in view, is when hole-filling is not necessary at all, i.e, when hole-filling has been turned off. Under such circumstances, a consistent frame-rate of 25 frames-per-second was observed. This equates to 675 million samples per second.

A typical scenario is one in which a clinician zooms in to the dataset to the maximum possible extents, beyond which hole-filling fails. This provides the maximum achievable interpolation, and the most detail that the system permits. Under such conditions, most of the rendering is concentrated in a small area, and extraneous samples are culled by the viewport-culling mechanism. All of the levels are turned on, so that hole-filling is performed to its maximum potential. In such a case, a frame-rate of 8 frames-per-second was observed. This equates to 216 million samples per second.

An average scenario is one where the user zooms is fairly close, but not to the maximum available detail, or extents. Under such conditions, nearly half of the model is outside the view-frustum, and generally 3 levels of detail are required to perform adequate hole-filling. Under such a scenario, a frame-rate of 15 frames-per-second was observed (405 million samples per second) for a 2-level render, while 10 frames (270 million samples) per second for a 3-level render.

Overall, from the worst case to the best case, the average performance of the system is 400 million samples per second, which amounts to nearly 15 frames per second, an acceptable level of performance for full-resolution native rendering in real-time.

In addition to the above, I experimented with other hardware setups. I ran the experiment on the *steve* dataset on a computer with a set of 3 NVidia GTX 8800 GPUs in an SLI configuration, and another with an ATI Radeon 4870x2. Rendering performance was tested at a resolution of 1280x1024. For a single view of *Steve* with masking information, the Nvidia GPUs achieve 69 FPS and the ATI GPU 45 FPS (frames per second), while for two views the Nvidia GPUs achieve 31 FPS and the ATI GPU 21 FPS respectively.

Scenario #	Scenario	level count	samples / sec	frames / sec
1	Full-view, no hole-filling	1	675 million	25
2	Medium close-up, one layer hole-filling	2	405 million	15
3	Medium close-up, 2 layer hole-filling	3	270 million	10
4	Medium close-up, 3 layer hole-filling	4	216 million	8
5	Extreme close-up	5	162 million	6
6	Full-view, complete hole-filling	5	121.5 million	4.5
	Average (best to worst)		400 million	14.75

Table 7.2: The performance of Pyramidal Projection on an ATI Radeon HD 5870, with the contribution of each additional level.

7.1.1 GPU Scores

In the aforementioned results I have cited the frame-rates for the proposed method compared to other methods. However, frame-rate alone is not a sufficient indicator of algorithmic performance, since the GPU performance also plays a part in increasing frame-rates. A truly standardised test would require access to the same hardware for obtaining results for each of the algorithms. Since this is not feasible, an alternative is to use GPU *benchmarks* to get an idea of the relative performance of various GPUs and factor out the influence of the GPU from our results.

A benchmarking software must be run on each of the GPUs in question, with identical settings, to obtain a valid *score*. I chose to use the **3DMark06 [64]** benchmarking software to validate my results. On the one hand, this software is compatible with the older cards such as the GeForce 7800 GTX, and on the other, this benchmarking software has been used by reputable benchmarking websites such as **Tom's Hardware [139]** so that a large database of scores is maintained by the site for various GPUs.

Computing Scores

3DMark06 breaks the scoring into three parts [65]:

- SM2.0 Score
- HDR/SM3.0 Score
- CPU Score

A Shader Model is a set of features that a GPU supports. 3DMark06 tests features defined in both the Shader Model 2.0 (SM2.0) and Shader Model 3.0 (SM3.0) specification. In addition, 3DMark06 bases its scoring on the CPU being used. If the CPU is constant

during testing, with the only variable being the GPU, the score reflects the difference in performance between two GPUs. The final 3dMark Score is derived from a combination of all three of the above.

The actual scoring is performed as follows:

- SM2.0 Score = 120 x 0.5 x (SM2 GT1 fps + SM2 GT2 fps)
- HDR/SM3.0 Score = 100 x 0.5 x (SM3 GT1 fps + SM3 GT2 fps)
- CPU Score = 2500 x Sqrt (CPU1 fps x CPU2 fps)

Where GT1 fps is the average frame rate measured in SM2.0 graphics test 1 and CPU fps refers to the frame rate measured in the CPU tests.

The final 3DMark06 score is computed as follows:

- Final 3DMark Score = 2.5 x 1.0/ ((1.7/GS + 0.3/CPU Score)/2)

where GS for hardware capable of running all graphics tests = 0.5 x (SM2S + HDRSM3S) and GS for hardware capable of running only SM2.0 graphics tests = 0.75 x SM2S.

Scores and interpretation

For the performance comparison to be more credible, it is logical to compare GPUs for algorithms that have been tested settings that are close (such as resolution and CPU type), and a GPU that is close in performance to the GPU used in the proposed method. The GPU used in the proposed method is an ATI Radeon HD5870 at a resolution of 1680x1050 pixels. From table 7.1, the next most powerful GPU is the GeForce 7800 GTX, and the performance of the algorithms using the GPU ([106] [172]) were tested at resolutions of 1280x1024. The rest of the algorithms were tested at resolutions of 512x512, which is too low for a fair comparison to be made.

GPU Type/Brand	Resolution	Quality Settings	3DMark06 Score
NVidia GeForce 7300	1280x1024	Default	1612
NVidia GeForce 7800 GTX	1280x1024	Default	5686
ATI Radeon HD 5870	1280x1024	Default	23640

Table 7.3: The final 3DMark06 scores for three GPUs as obtained from Tom's Hardware [139]. The relative difference in the scores points to a relative difference in performance between the GPUs. A higher score means better performance.

The GPU closest to the NVidia GeForce 7800 GTX is the NVidia GeForce 7300. The former scores 5686 points on 3DMark06, while the latter scores 1612. From the scores

alone, the former GPU is estimated to be 5686/1612 = 3.5 times better than the latter (table 7.4). By that account, resolution differences notwithstanding, we would expect Image Reconstruction [106] (which runs at on average 55 million samples per second) to perform at slightly over 15 million samples per second on a NVidia GeForce 7300 GT. By that comparison, it performs worse than the algorithm proposed in *High Efficiency Real Time Rendering for Point-Based Model on GPU* [87], which performs at at least 29 million samples per second on the same GPU (table 7.1.

The NVidia GeForce 7800 GTX is the GPU used in both Deferred Blending [172], and Image Reconstruction [106]. The GPU scores 5686 points on 3DMark06. The proposed method uses an ATI Radeon HD 5870, which scores 23640. From the scores alone, the latter GPU is estimated to be 23640/5686 = 4.1 times better than the former.

GPU 1	GPU 2	GPU1 / GPU 2
NVidia GeForce 7800 GTX	NVidia GeForce 7300 GT	3.5
ATI Radeon HD 5870	NVidia GeForce 7800 GTX	4.1

Table 7.4: A normalised comparison between the performance of two pairs of GPUs.

By that account, Deferred Blending would run at 102.2 million samples per second, and Image Reconstruction would run at 225.5 million samples per second on the ATI Radeon HD 5870 (table 7.5. The proposed method runs, on average, at 400 samples per second. This is a **177% increase** in performance over Image Reconstruction and a **390%** increase over Deferred Blending.

Algorithm	Original (million samples/sec)	Performance ratio	HD 5870
Deferred Blending [172]	25	4.1	102.5
Image Reconstruction [106]	55	4.1	225.5
Pyramidal Projection	400	1	400

Table 7.5: A comparison of how fast algorithms tested on a GPU closest in performance would run on the ATI Radeon HD 5870

7.1.2 Analysis and Conclusion

According to the benchmarks, Pyramidal Projection performs several times better than state-of-the-art point-based rendering algorithms. Although using the same type of GPU on each algorithm was not possible, I attempted to *normalise* the effects of the GPU on the scores by using a standard benchmarking tool (3DMark06), and a source of information (Tom's Hardware) where standardised results were available for various GPUs. This made

it possible to compare the *relative* performance of the GPUs. From this relative score, it was possible to get an approximate idea of the performance of other algorithms if run on the same GPU. Cancelling out the effect of the GPU difference, a 177% increase in performance was observed for Pyramidal Projection over Deferred Blending.

While this level of a performance difference is expected, given the GPU-native nature of the algorithm, it is worth noting that the figures for other algorithms do not include pre-processing times, whereas Pyramidal Projection performs all of the computations in real-time, hence, the performance increase in real-world scenarios is even greater than that reflected by the numbers. In addition, the relative difference in performance between GPUs is generally exaggerated by the benchmarking scores. So while the benchmarking scores between two GPUs would indicate that GPU A performs thrice as fast as GPU B, in reality one would not expect GPU to play a game on GPU B at 30 fps, while the same game on GPU A at 90fps [139]. This is obvious when observing frame-rates of leading games on competing GPUs on Tom's Hardware. In conclusion, Pyramidal Projection would likely perform even better than a **177%** performance increase over Image Reconstruction, if both were to be tested on the same GPU.

The average frame-rate of Pyramidal Projection for uncompressed range data is 15 frame-per-second. Coincidentally, the term *real-time rendering* refers to rendering at a frame-rate of 15 frames per second or higher [3]. By that account, Pyramidal Projection achieves the goal of performing real-time rendering on large native surface scanned medical data. On the other hand, it is observed that any content that renders at a frame-rate of at least 4 frames per second is perceived as being *interactive*, if not realtime [1]. By that account, Pyramidal Projection remains interactive, even under the worst-case scenarios presented in this research (scenario 6, table 7.2).

7.2 A Survey of the Proposed Rendering System

The primary testable parameters of the rendering system, as presented in the thesis are:

1. Rendering Speed
2. Interactivity
3. Quality of Visualization

An objective (numerical) evaluation of rendering speed was carried out via a comparison of various benchmarks of several competing algorithms on modern GPUs. Ultimately, however, a tool intended for use by a human operator necessitates qualitative testing of usability and quality, (interactivity and visualisation) that are otherwise difficult to test objectively.

The experiment is designed to assess an operator's perception of responsiveness of the system, its rendering quality, the utility of the hole-filling algorithm, and ease of use. The study followed "hall-way" testing, whereby 10 random participants took part in the study. Their backgrounds were sufficiently random, from not being familiar with 3d systems at all, to being avid gamers, or graphic designers, using real-time 3d systems routinely. As a sample, we took *Steve* and *DMF_1003102_AU9_100* as our data-sets. Steve, as mentioned earlier is a 2-pod data-set consisting of 2 views of 3000x4500 samples each, while *DMF_1003102_AU9_100* is a single-pod data-set comprising of 3000x4500 (13.5 million) samples.

7.2.1 The Experiment

The participant is presented with several tasks to perform, and then presented with a scale upon which to rate those tasks. This rating will provide a basis on which to judge the effectiveness of the parameters of the experiment.

The experiment is divided into 2 sets, Set **A** and Set **B**. Set **A** aims to measure the qualitative effectiveness of the *quality of visualisation*. Set **B** aims to measure both the *interactivity* and *rendering speed* of the system.

7.2.1.1 Level of experience

Participants were first asked what their level of familiarity was with computers in general, and then 3D systems (including 3D game) in particular:

1. *What is your level of experience with computers and their operation (using the mouse, keyboard, nad basic operations including surfing the web)? Please rate from 0 t0 5, with 0 being complete inexperienced (never used a computer before), and 5 being an expert on computer usage (i.e, use it every day, for several hours a day).*

2. *What is your level of experience with 3D rendering systems (including 3d games, visualisation systems, and/or simulations)? Please rate from 0 to 5, with 0 being*

completely inexperienced, and 5 being an expert on the usage of 3D rendering systems.

This question was intended to gauge the randomness (of backgrounds, i.e, familiarity with 3d systems) of the sample.

7.2.1.2 Procedure for Set A

SET A

The DI3D viewer software presents you with a list of 3D scanned faces, in the form of files ending in the .di3D extension. Select "Steven01.di3D" and click "Open" to begin visualising the face.

Zoom-in on the face by performing a right-mouse click and then dragging. Keep dragging until the face begins to visibly break apart, with "holes" appearing. The system allows you to "fill" these holes to various degrees, or "levels".

Press 1 to perform basic hole-filling. Press 1 again to undo the effect of level 1 hole-filling.

Press 2 to perform additional hole-filling. press 2 again to undo the effect of level 2 hole-filling.

Repeat the same process for all levels up to 4.

Zoom out of the data (again by performing a right-mouse click and dragging) until holes disappear again.

Press "q" to exit visualising the current face. You will now be returned to the list of faces, ready to be opened.

Repeat with the procedure with the face "DMF_1003102_AU9_100.di3D".

7.2.1.3 Questions for Set A

Unless otherwise noted, rate on a Likert scale from 0-5, 0 being totally disagree, and 5 being completely agree.

1. *Does the new visualisation system succeed in presenting the 3D scanned data at a high enough resolution to be called "photo-realistic"?* (0-5)

2. *Does the hole-filling/interpolation methods used by the viewer improve the visual quality of the visualisation of the 3D scanned data?* (0-5)

3. *Are you satisfied with the overall quality of results obtained from the visualisation?* (rate from 0-5, 0 being not satisfied at all, and 5 being extremely satisfied)

7.2.1.4 Procedure for Set B

SET B

Select "Steven01.di3D", by selecting it from the given list as before, and then clicking the "open" button.

Try rotating the face at a 90 degree angle in any direction by performing a left-Click on the mouse and dragging until the desired angle is reached.

Now perform a "pan" (i.e, translate on an axis) on the face. To do so, you must first press space-bar to switch to "pan" mode (rather than rotate), and then left-click and drag.

Now using the "pan" and "rotate" mechanisms you learned, try to orient the view so that the tip of the nose is visible.

Now zoom-in, and pan, so that the "left" eye is at the centre of the screen.

Now rotate the model 90 degrees so that the camera is facing the right ear front-on.

Press "q" to exit visualising the current face. You will now be returned to the list of faces, ready to be opened.

Repeat with the procedure with the face "DMF_1003102_AU9_100.di3D".

7.2.1.5 Questions for Set B

1. *Was it easy to perform the given tasks?* (0-5)
2. *Was the system interactive/responsive at all times during the tasks?* (0-5)
3. *Was the performance of the system fast enough to be called "real-time"?* (0-5)
4. *Are you satisfied with the overall level of interaction of the visualisation?* (rate from 0-5, 0 being not satisfied at all, and 5 being extremely satisfied)

7.2.2 Results

I will now present the results of the experiment.

7.2.2.1 Result: Level of experience

The sample group consisted of a people from fairly diverse backgrounds. They were mostly familiar with computers and their usage (see figure 7.1), with most (60%) using the computer regularly, for several hours a day.

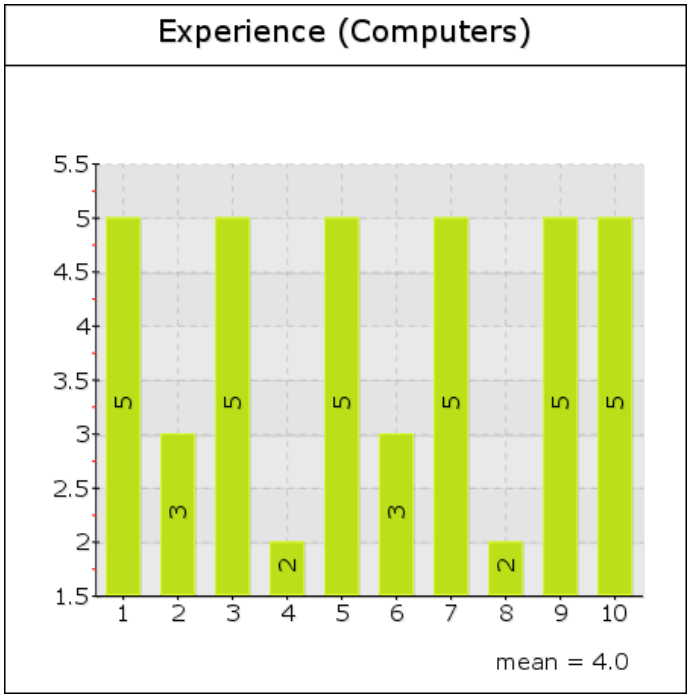


Figure 7.1: Answers to the question: *How familiar are you with computers and their operation?*

The second question asked was: *What is your level of experience with 3D rendering systems (including 3d games, visualisation systems, and simulations)? Please rate from 0 to 5, with 0 being completely inexperienced, and 5 being an expert on the usage of 3D rendering systems.*

The purpose of the question was to make sure that (apart from familiarity with computers) the sample we chose included people with a wide variety of backgrounds and levels of expertise using 3D systems in particular. Since the system may find uses in games, medical applictions, or even simulations, rather than relying on a single definition of "expertise", I chose to be inclusive. Participants that had prior experience using software that required navigation in 3D space, such as computer games, simulations, or other visualisation systems, were deemed "experts" owing to the fact that they would be good judges of the parameters: quality, speed, and interactivity/useability.

As can be inferred from figure 7.2, the mean experience, on a level from 0 to 5, was **2.2**. This is nearly 50%, meaning that our chosen sample consists of a fairly representative mix, with a range of people from those with no experience using 3D systems at all, to those that use it routinely.

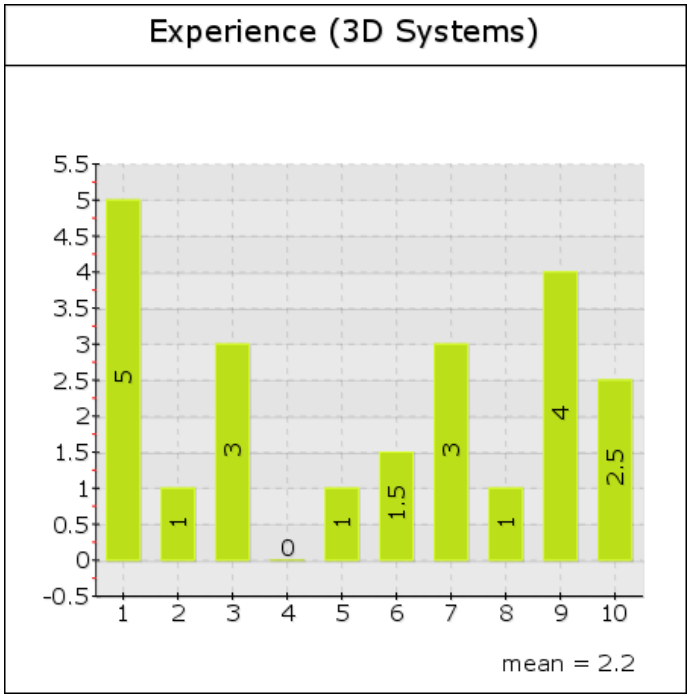


Figure 7.2: Answers to the question: *What is your level of experience with 3D rendering systems?*

7.2.2.2 Set A: Question 1

A rendering is a virtual *photograph* of a 3D model. Judging the *quality* of a visualisation system is a subjective measure, and influenced by previous experience with real-time 3D systems, such as 3D games. It is safe, however, to assume that all participants would be familiar with photographs. Our goal, in rendering, is to match as faithfully as possible the likeness of the original face. The face of the human subject, therefore, is the ground truth. Since the human face is 3D, while a rendering is 2D, a more suitable goal is to hold a *photograph* of the human subject as the ground truth. Therefore *photo-realism* - the similarity of a rendering of 3D model to a photograph - is deemed a suitable parameter to guage the quality of the rendering.

The mean result (see figure 7.3) of the quality was **4.45**, from 0 being completely non-photorealistic, to 5 being a replica of the human face (like a photograph). That is a result of nearly 90%. We can safely conclude that the rendering is very faithful to the original face in likeness.

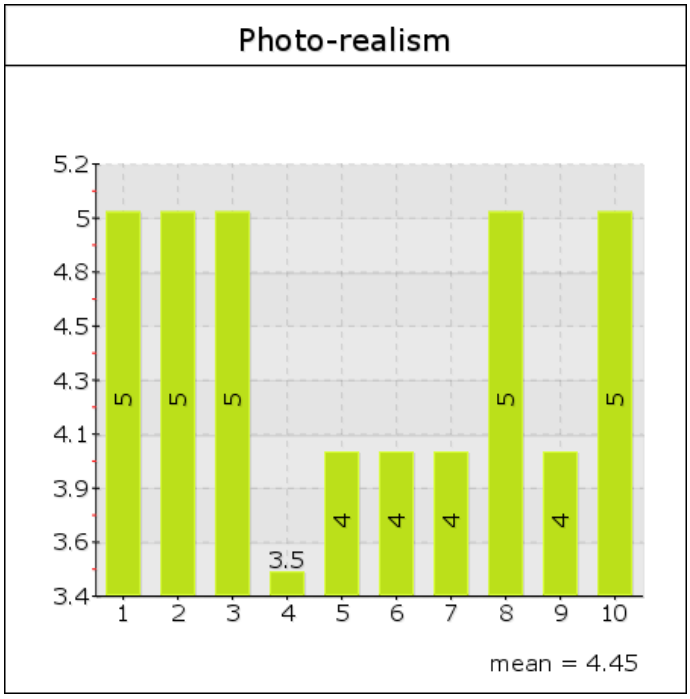


Figure 7.3: Answers to question 1 from Set A

7.2.2.3 Set A: Question 2

The participants were walked through the software, shown the holes that appear when the model is zoomed-in beyond its limits, and then presented with the hole-filling functionality. The participants were then asked whether they felt the hole-filling improved the visual appearance of the render.

The mean result (see figure 7.4) was **4.35**, from 0 being no improvement atll, to 5 being a significant improvement. It is clear from the results that the participants agreed that the interpolation added a significant improvement to the visualisation quality.

7.2.2.4 Set A: Question 3

The purpose of this question was to present the participants with the chance to provide their overall impressions of quality of the rendering software.

The mean quality rating was **4.54** (see figure 7.5), a rating of over 90%. This is indicative that the participants rated the quality of the rendering very highly.

7.2.2.5 Set B: Question 1

In Set B, the participants were again walked through the software, this time focusing on performing specific tasks. The purpose of the exercise was to understand how easy the

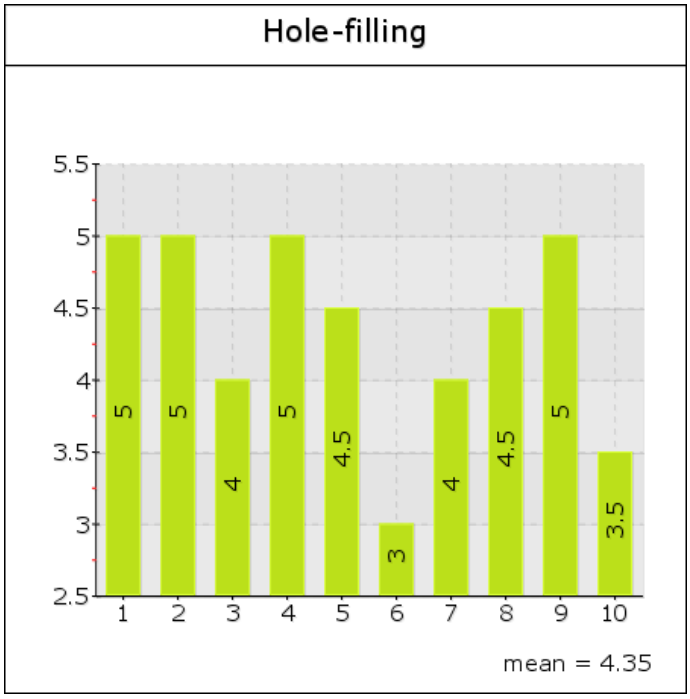


Figure 7.4: Answers to question 2 from Set A

software was to use. The given tasks made use of navigation features provided by the viewing tool, which had a button-mapping that closely matched existing 3D authoring tools (such as ZBrush and Maya).

Surprisingly, there was little difference in usability for expert users versus novice users. This is indicated by a mean score of **4.7** and median of **5** (see figure 7.6). Novice users found the system just as easy to use as the expert users.

7.2.2.6 Set B: Question 2

An important aspect of useability is the responsiveness of a system. Lag, or delays while trying to interact with a system reduce its *interactivity*. This question was asked in order to guage user satisfaction the with the interactive nature of the system.

The mean result was a score of **4.55** indicating that participants were satisfied with the interactivity provided by the rendering system.

7.2.2.7 Set B: Question 3

Participants were satisfied with the real-time performance of the rendering software provided. The mean score was **4.2**, 84% satisfaction.

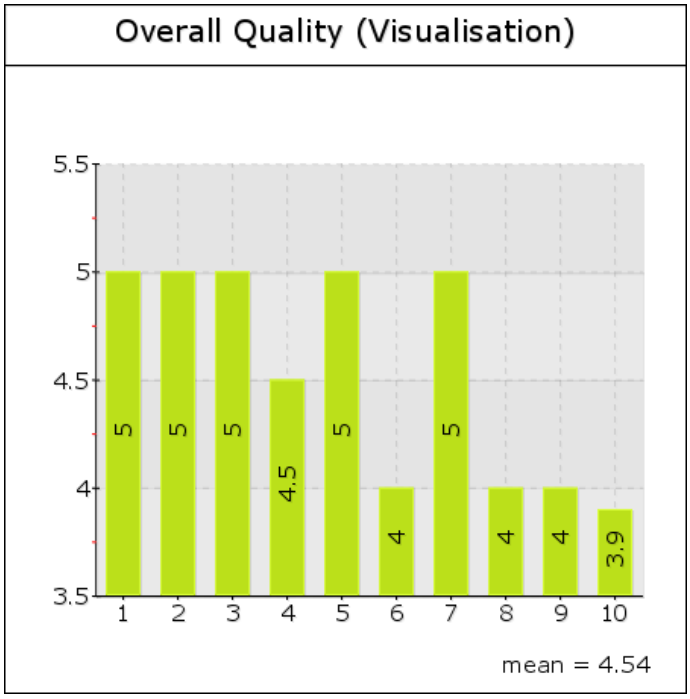


Figure 7.5: Answers to question 3 from Set A

7.2.2.8 Set B: Question 4

The sample group was satisfied, overall, with the interaction that the rendering software provided. The mean score was **4.48**, nearly 90% satisfaction.

7.2.3 Conclusion

Results of the experiment validate the claims made in this thesis. Questions were asked relating to the rendering quality, interactivity, useability, and performance of our system. The mean scores of the responses from participans, both novice and skilled, were consistanly above **4.0**, indicating a high level of satisfaction with all the parameters.

7.3 Results of the Proposed Methods

I will now present some images that visually depict the results of the methods proposed in this work, such as hole-filling, antialiasing, and GPU rendering.

7.3.1 Hole-filling

The results of scattered data interpolation in Pyramidal Projection are shown in 7.10. The background is coloured red to make the missing data more obvious. In 7.10, a single layer

projected as points leads to holes due to a lack of explicit connectivity between points. As the camera zooms in, the points spread further and further apart, and the background begins to show through. Adding more levels of the pyramid begins to perform the intended interpolation. Lower frequencies gradually replace holes at the higher frequencies.

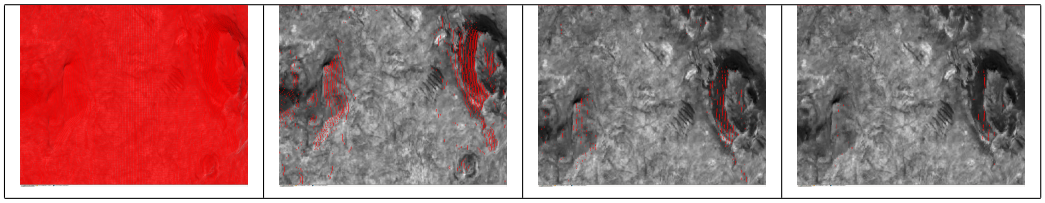


Figure 7.10: Results of hole-filling by progressively adding more levels of the pyramid. The red pixels are holes (i.e the background showing through).

Figures 7.12 to 7.15 provide high-resolution snapshots of individual levels of the Gaussian pyramid during a rendering of the Steve model. Figure 7.15 shows the lowest resolution layer in the pyramid rendered while figure 7.11 is the final image after the hole-filling has been completed.

Figure 7.11: A final hole-filled render of the Steve model

Figure 7.12: Level 0: Highest resolution level in isolation in the Steve render

Figure 7.13: Level 1 of the Steve render during Gaussian Projection

Figure 7.14: Level 2 of the Steve model rendered during Gaussian Projection

Figure 7.15: Level 3 of the Steve model rendered during Gaussian Projection

7.3.2 Anti-aliasing

The anti-aliasing algorithm approximates the point spread function, thereby distributing the energy of a single point to multiple pixels. Apart from the intended benefit of smoothing jagged edges (aliasing), using larger than a single pixel to approximate a point results in single-pixel hole-filling. The result is shown in 7.16. Note how the anti-aliasing smooths otherwise aliased high-frequency detail such as the curves that define the nostrils.

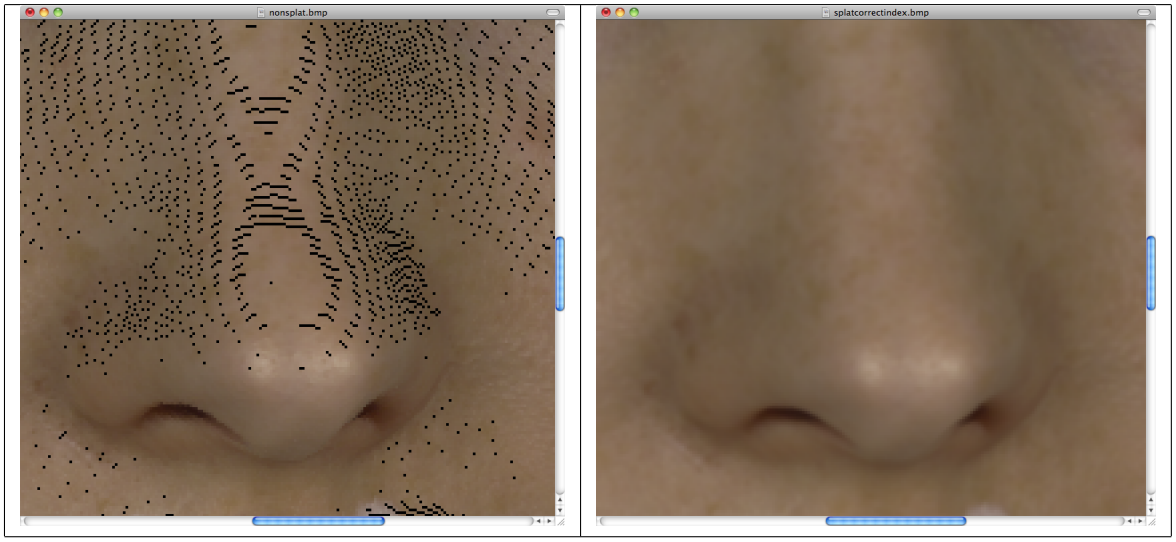


Figure 7.16: (a) Non anti-aliased pixels (orthographic camera) (b) Antialiased pixels

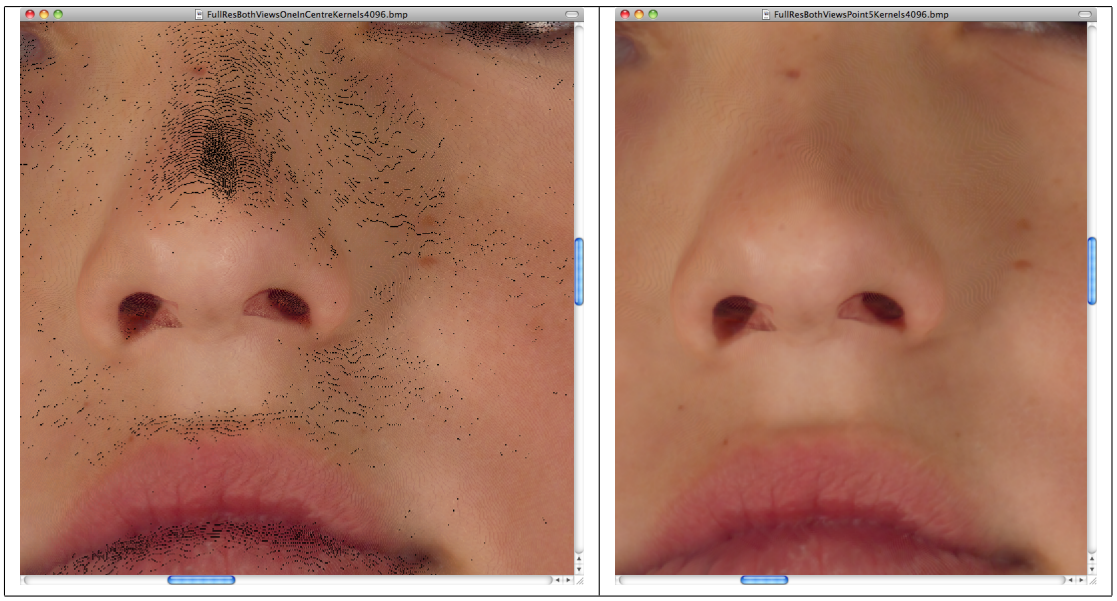


Figure 7.17: (a) Full resolution render without Anti Aliasing and HSR (b) Full resolution render with Anti-aliasing

Another form of anti-aliasing was employed in the proposed algorithm while zooming out. As the camera moves away, the higher frequency detail is indistinguishable from the lower frequency detail, and hence can be dropped entirely, leaving the anti-aliased low-frequency detail. As the camera moves out, the opacity of the higher frequency level is reduced until it becomes completely transparent, in effect, disappearing. The standard fog equation was used to control the opacity non-linearly. The result is shown in 7.18.

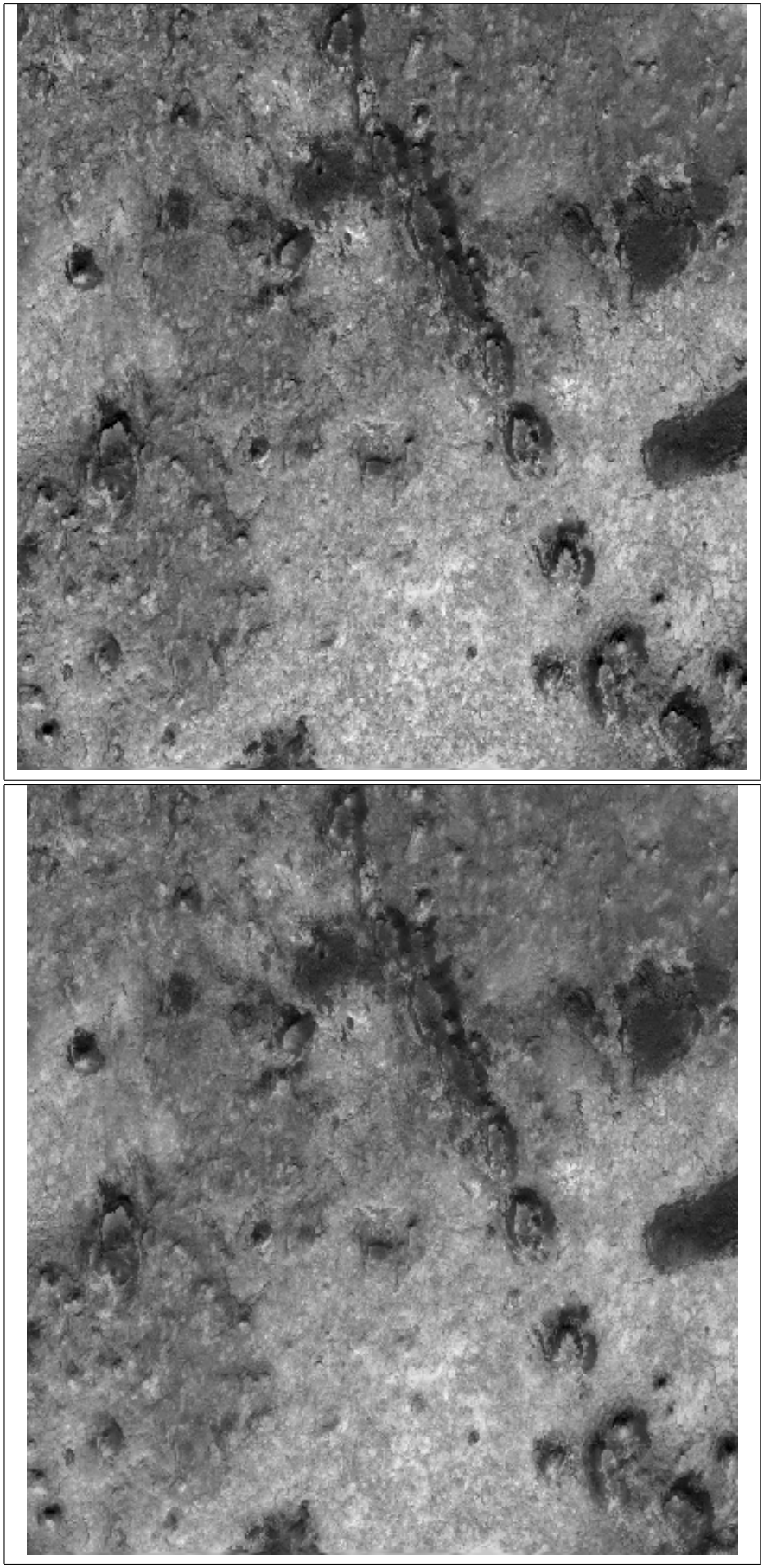


Figure 7.18: A comparison between renders *with* LOD anti-aliasing (bottom) and *without* LOD anti-aliasing (top)

7.3.3 Comparing the CPU and GPU results

The algorithm was first implemented in Matlab. Later, a GPU-based version was implemented in order to provide real-time interactivity. The results are visually identical, as shown in 7.19 , however, the GPU version runs an order of a magnitude faster.

Figure 7.19: The Matlab implementation with sub-pixel splatting (left) and the raw GPU implementation with sub-pixel splatting (right). Note that the results are visually indistinguishable while there is an order of a magnitude of a difference in rendering speed. Also note that the GPU version in this case is single-view only.

Each source image has 3000x4500 samples per view. For two views, the number of samples exceeds 25 million points. For a single view of *Steve*, the Nvidia GPUs achieve 69 FPS and the ATI GPU 45 FPS (frames per second), while for two views the Nvidia GPUs achieve 31 FPS and the ATI GPU 21 FPS respectively.

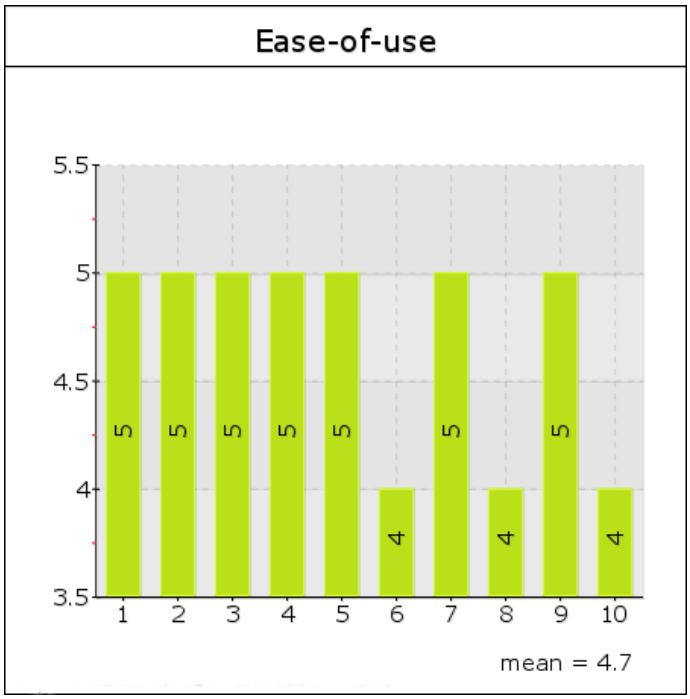


Figure 7.6: Answers to question 1 from Set B

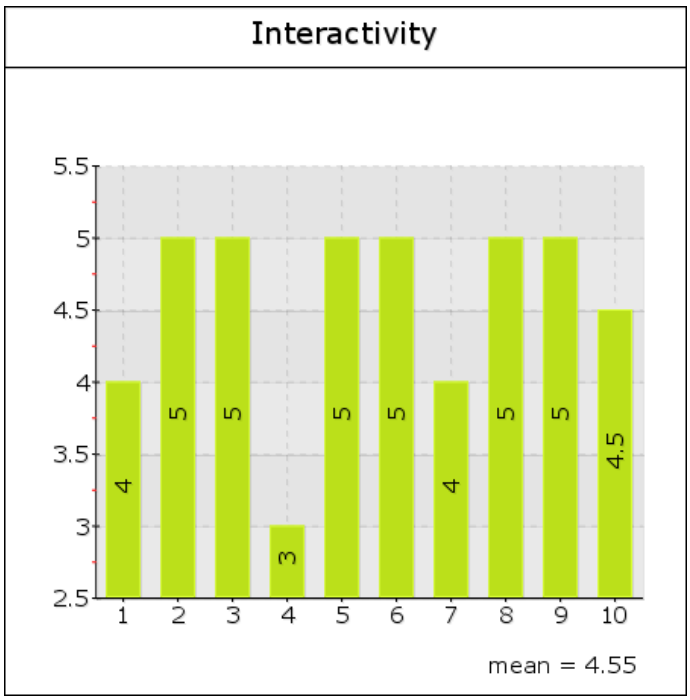


Figure 7.7: Answers to question 2 from Set B

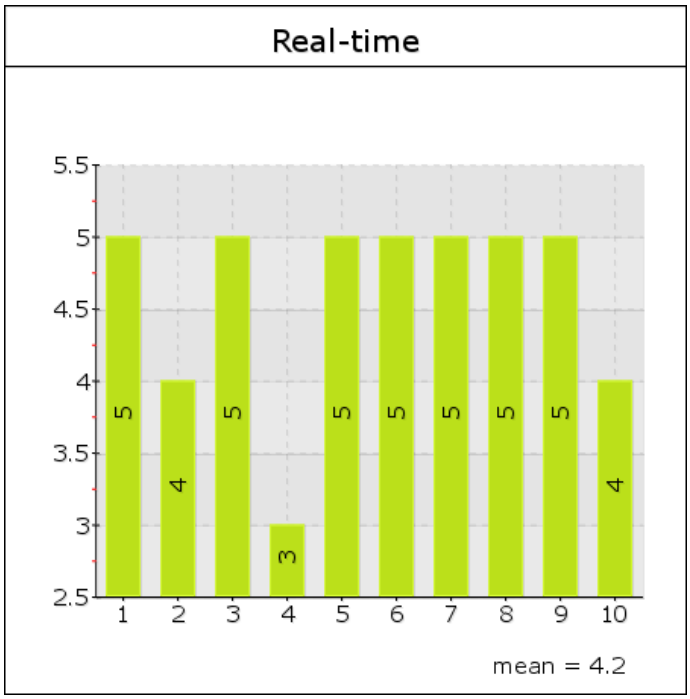


Figure 7.8: Answers to question 3 from Set B

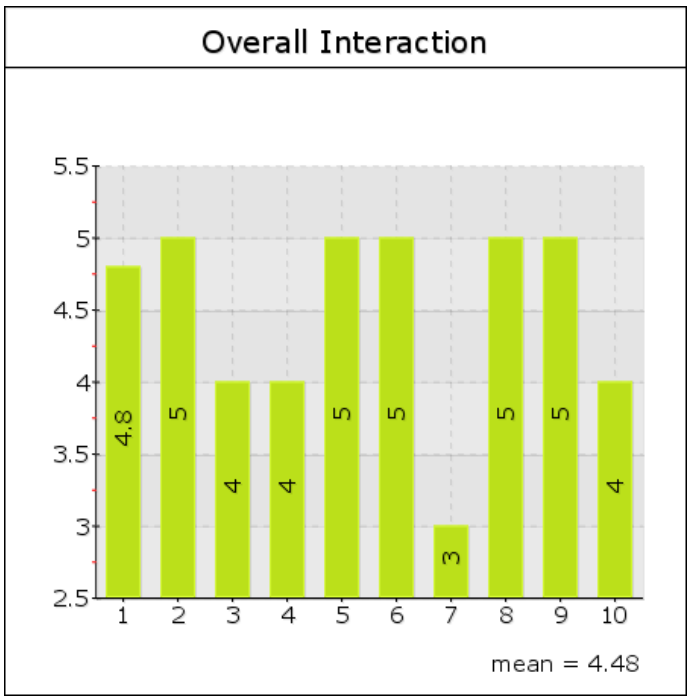


Figure 7.9: Answers to question 4 from Set B

Chapter 8

Discussion

8.1 Hypothesis and Thesis Statement Revisited

At the beginning of the this thesis, the following hypothesis was made:

"Range images are a better native representation for visualising 3D Graphics, especially medical visualisation, as opposed to polygons, or point-clouds, since range-images are the native data format for most 3D capture systems, they are regular, compact, provide connectivity, and allow GPU optimisation due to their matrix-like nature."

The hypothesis has been verified in the course of this thesis. The aforementioned properties of range images have been utilised to provide a rendering system that has clear advantages over traditional point-cloud based systems. The proposed rendering system makes use of the regularity and matrix nature of range images to store 3D models natively on the GPU, and the inherent connectivity obviates the need for offline preprocessing, making it possible to stream data in real-time where frame-to-frame coherence information is not available.

In chapter 1, I posed the following thesis statement:

"This is an investigation on real-time visualisation of high-resolution scanned data with demonstrations that preprocessing, and the GPU bandwidth consumption of lossless data, are two significant bottlenecks in state-of-the-art algorithms."

A review of the literature in state-of-the-art algorithms reveals that all major point-based-rendering methods suffer either from long preprocessing computations, or require out-of-core methods to display large datasets, or often both. The proposed algorithm obviates the need for preprocessing, and provides a mechanism for saturating the GPU so that CPU-to-GPU transfers are kept to a minimum by performing all relevant computations

such as LOD management natively on the GPU.

8.2 Revisiting Objectives

At the conclusion of this work, it is important to revisit the original requirements that I set out to fulfil, and whether the present work succeeds in doing so. As mentioned in the very first chapter, the aim of my research was to explore real-time rendering of large data sets, particularly human anatomy data obtained by stereo capture devices , at native imaging resolutions. I will now revisit the requirements and discuss wether I fulfilled these requirements.

8.3 Requirements

Interactivity First and foremost, I aimed to develop a tool that was interactive. Users perceive rendering performance of at least 4 frames per second as interactive [1]. According to the provided benchmarks, the worst-case performance of Pyramidal Projection for the tested data was 4.5 frames per second, while the average frame rate was 15 frames per second. This satisfies the requirement of interactivity. While I developed the initial versions of the algorithm in Matlab, it became obvious that Matlab's software implementation would not provide the interactivity needed to visualise data at this scale. When later ported to C++, I experimented with setups where a large scene would be displayed at a lower resolution during interaction (such as rotation with the mouse), and rendered at full resolution when interaction would cease (upon a mouse-up for example). In addition, for the sake of a smooth interactive experience, I experimented with several GUI setups and even a full-screen GUI-less setup. In the end, I chose the full-screen GUI-less route as it provided the most screen real-estate (something that was desirable for very high resolution data), and was the closest experience to examining a live patient.

Real-time/Speed Rendering is said to be *real-time* if it achieves at least 15 frames per second [3]. Evidence was presented that Pyramidal Projection renders, on average (i.e under a typical scenario), at 15 frames per second. This satisfies the requirement of providing real-time rendering for large surface scanned data. As mentioned earlier, the early prototypes of the algorithm were tested in Matlab. Matlab proved to be too

slow for visualising data in real-time. The code was ported to C++, and a completely GPU-based implementation was carried out in GLSL. This allowed us to render the complete two-pod data at real-time frame-rates. The real-time implementation was tested on both NVidia and ATI GPUs, the only two major brands in desktop GPUs currently.

Visualisation

- For visualisation, it was very important that we *preserve the native samples of the data*. To that end, I succeeded in displaying the entire two-pod data without any data loss or compression. The proposed algorithm always prioritises the native full resolution samples, and only resorts to lower frequency data in the case the high resolution data needs interpolation. In order to carry out such a feat, however, I had to rely on points rather than polygons as a native data representation method. This created its own visualisation problems, the most obvious being the existence of holes due to a lack of explicit connectivity. The hole-filling method had to be fast on the one hand, so the real-time interactive component was not diminished, and realistic on the other so that it would give a good idea of how the original samples would have looked like. In addition, since the original samples were far more important in some contexts, such as medical imaging for example, than the interpolated samples, it was important to provide the option of removing hole-filling entirely to make the original data obvious. Such a feature is provided, and the hole-filling recovers a reasonable degree of detail while maintaining an interactive rate.

- The proposed rendering methods rely on a pyramidal structure: a convenient and memory-efficient structure that allows us to *render our 3D data at multiple levels of detail.*

- Finally, *I have presented two algorithms for the multi-view integration of 2.5D range images into one complete 3D model.* Laplacian projection was presented as a non-realtime method based on image mosaics and multi-resolution splining for smooth integration of multiple views in image space. Pyramidal Projection, on the other hand, was presented as a less visually pleasing, but real-time algorithm for merging multi-view data dynamically.

Upgradeability An important consideration was that although at the beginning of the

project, the data was composed entirely of static 3D captures, since 4D capture systems were on the horizon, it was important to maintain a rendering pipeline that made it possible to upgrade a 4D capture system in the future. At the end of the research, 4D capture systems are indeed available. The proposed system obviates the need for surface reconstruction, and performs any setup natively on the GPU at runtime, making the preprocessing delays negligible. By virtue of this fact, the proposed algorithm can be easily integrated with a 4D rendering pipeline. This makes it possible to try setups where time-based data is buffered in advance, and fed sequentially to the rendering algorithm presented in this research, to be rendered in real-time.

8.4 Summary of Salient Features of Pyramidal Projection

The advantages of the proposed method over traditional methods are as follows:

The proposed method is **fully GPU-based:**

- The source data (range pyramids) is contained entirely on the GPU natively.
- Pyramidisation/preprocessing is done entirely on the GPU, so setup time is negligible
- Can be architecturally turned into an out-of-core mechanism (via tiles) without additional overhead.

The proposed method is **memory-efficient:**

- Indices can be reused/shared in order to preserve memory.
- Pyramidisation always adds a constant 1/3rd extra overhead.
- Pyramidisation can be accomplished on the GPU via mipmapping, therefore obviating the need for allocation of an additional 1/3rd of main memory.

The proposed method is **lossless**:

- The proposed method renders at full-resolution in real-time. It does not require data compression.
- Performs adequate hole-filling to provide good visual representation for visual-feedback
- Current 3D scanned data can be displayed natively, i.e in the native format (range) and at native resolutions.

The proposed method is **streamable**:

- The proposed method performs minimal preprocessing, and entirely on the GPU, making it possible to stream data without requiring frame-to-frame coherence.
- The speed of the rendering is only limited by memory-to-memory transfer speeds, making it possible to achieve faster streaming when GPU RAM speeds increase.

In the next section, I will describe the contributions I made in an attempt to fulfil the requirements outlined above.

8.5 Contributions

While solving many problems associate with polygons, point-based rendering presents its own challenges, especially when implemented on GPUs that are designed to render polygon data. I will now present the primary contributions made by this research while attempting to solve these problems.

8.5.1 Novel scattered-data interpolation mechanism

Polygons are a vector-based representation, and therefore resolution independent: Given a set of vertices, it is possible to interpolate between them at any scale. Points, on the other hand, are a sample-based representation. The samples inherently lack connectivity and as transformations are applied to the individual samples, visual connectivity is broken. If appropriate interpolation is not performed, this would cause *holes* to appear between samples. In essence, this is a scattered-data interpolation problem. In standard texture mapping, a similar problem is faced, however, by virtue of the pixels being on a single plane, interpolation problems are easily solved by *backward projection*, i.e, projecting from pixel-space to texture-space [78]. During 3D point-based rendering, in contrast, backward projection is difficult since the 3D points do not represent a surface that can be projected onto. Therefore, a standard procedure is to construct a surface out of point-clouds, or other point-based representation before visualisation [16, 36, 59, 75, 83, 93, 126, 175]. Modern point-based rendering algorithms delegate this reconstruction phase to a preprocessing pass [103, 126, 144, 175]. This makes such algorithms unsuitable for time-based 3D display such as streaming 3DTV [26, 51, 54, 125, 154].

This thesis presents Pyramidal Projection, a novel point-based rendering algorithm that performs scattered-data interpolation via a scale-space approach. It intelligently uses

information from the lower-frequencies where high-frequency information is not available. In addition, Pyramidal Projection uses forward projection only. Modern GPUs are primarily rasterisation engines, geared towards forward projecting large numbers of polygons. By relying on this optimisation of forward projection, the proposed method manages to extract a high frame-rate from the GPU during rendering.

8.5.2 Extension of Burt and Adelson's Multi-resolution Spline to 3D

Triangulation-based devices can only recover depth information from a particular *point-of-view*. In order to capture an object in its entirety, multiple captures around the object are necessary. For display, multi-view integration techniques are required to join these partial views into a single 3D representation in an automated manner. Merging data composed of polygons has been a difficult problem due to issues regarding connectivity, among others [46, 102, 155].

I proposed and implemented a multi-view visualisation algorithm that made use of the flexibility afforded by points (stored in range images) for data representation. Learning from the advantages of the multi-resolution spline as proposed by Burt and Adelson, Laplacian projection creates a seamless mosaic of multiple views of *3D* data in *image space* by rendering at multiple levels of detail. Previously, image pyramids were used as a 2D scale-space representation [27, 28, 50, 162]. The novelty of this work is the implementation of an image pyramid for storing *3D* data, and first use of the multi-resolution spline to blend 3D data rather than 2D images.

8.5.3 Novel Use of GPU Texture Memory to Store a Multi-resolution 3D Model

As part of this work, I presented a mechanism of storing an entire multi-resolution 3D model directly on GPU memory. I presented evidence of how the presented data structure makes it possible to store large datasets on the GPU natively where previously this was not possible, and out-of-core methods were required [41, 140, 169]. I presented evidence that the proposed algorithm makes it possible to store multi-view 3D data on a consumer GPU such as the NVidia GTX 8800 (768MB). Where a single view of range data consisting of 3000x4500 samples would generally require 500 MB, and it would be possible to store only a single view natively on the GPU, our method reduces the memory requirements to 190 MB for a single view, making it possible to store multi-view data natively on the GPU.

8.5.4 Realtime Streaming of Range Images at Native Resolutions

As part of this research, I proposed an implemented a realtime algorithm that works natively with range images, performing camera/model transformations dynamically via shaders, without any loss of data or compression compared to polygon-based methods. As pointed out in chapter 2, current methods either perform a conversion to polygon meshes, or require surface reconstruction [4,70,71,73,99,126,137,155,164,171,174,176]. A novelty of the proposed algorithm is that it obviates the need for a reconstruction phase associated with traditional point-based rendering algorithms, and thereby provides real-time rendering rates with minimal setup time. The minimisation of a setup phase makes it possible to stream data without the explicit need to have frame-to-frame coherency. Therefore, applications such as 3DTV become possible with this approach.

8.6 Future Work

This work provides an initial investigation into the development of multi-view rendering algorithms for surface scanned data. In the thesis, several areas have been identified where improvements could be made. I will now discuss a possible avenue for further exploration that may overcome these.

8.6.1 Overcoming Current Limitations

The proposed algorithm made use of points as a primitive to display range data natively at interactive frame-rates. However, the usage of points introduced their own limitations: Moving far beyond the native imaging resolution is not guaranteed to provide adequate interpolation, silhouettes are blurred, and robust hidden-point removal remains difficult. Points are a light (in terms of memory) and fast (for rendering) primitive, while polygons are scalable due to their vector nature. Ideally, we would like to maintain a light and fast rendering primitive that is scalable: A combination of points and polygons.

Current GPUs offer similar features in *geometry-shaders*, i.e, shaders that generate geometry per sample. An idea worth exploring is to use points for storage, and projection calculations, treating them much like vertices, and then creating polygons in *screen-space* from the projected points to fill any holes. The creation of polygons in screen-space restricts the polygonal data-structure to purely 2D processing, limiting its impact on performance. This can be accomplished easily with *geometry-shaders*.

Intuitively, this may be performed as such: The range image and texture image comprising the 3D model are loaded. A buffer (`bufferXYZ`) is created (on the GPU, this would be a set of Frame Buffer Objects) to store the `x,y, z` value ,of each point after projection. In the first pass, the algorithm goes through each pixel, performing computations that ultimately result in a projected pixel. This point, however, instead of being displayed on the screen, is saved in the aforementioned buffer. The point is saved at the same index as the original range image, so that now each point in the buffer stores the projected values of the corresponding point in the range image.

A range image, by its nature, preserves connectivity information. The range image may be treated as a mesh where four neighbouring pixels are vertices of the same polygon. Since `BufferXYZ` maintains the same indices as the range image, the connectivity information is preserved. Each set of four neighbouring points creates a polygon. However, since `BufferXYZ` contains points *after* projection, the resulting mesh is in image space.

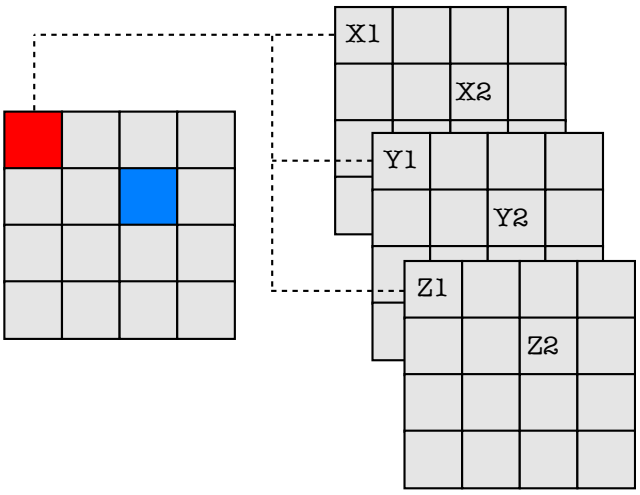


Figure 8.1: During *Pass 1*, the range/texture image is mapped to a set of buffers containing the X, Y, and Z values after projection, in the same index locations as the original range image.

Proceeding to the second pass, `area` is a function that computes the area of a polygon. It takes a buffer containing `x,y,z` information, and given a point `x,y`, it computes the area of the polygon between `(x,y),(x+1,y),(x+1,y+1),(x,y+1)`. If this area is more than epsilon (a value larger than a pixel), then the polygon is drawn in its entirety. Otherwise, the polygon is too small to be drawn in its entirety and may be reduced to the four individual samples that define it.

The two primary advantages to such a research would be:

- The proposed future work will provide infinite zoom capability, since screen-space polygons are scaleable.
- The proposed future work will solve the blurry silhouette problem of Pyramidal Projection.
- Hidden-point removal will be easier since surface continuity will be preserved due to connectivity provided by the samples in the form of screen-space polygons. These screen-space polygons do not *break,* and hence do not allow points behind to show through.
- Adequate hidden-point removal will consequently solve the over/under sampling effects observed in Laplacian Projection, lifting the restriction on viewport size.

Currently, geometry shader performance is lacking when dealing with large datasets, however, we hope this will improve in the future, and open up new avenues for the advancement of the current research.

8.7 Applications of the Proposed Algorithm

Based on the existing work, I see several applications for my research. Apart from the existing application domain of medical visualisation, I believe a rendering framework that minimises setup time is ideal for domains that require streaming or time-based 3D rendering. Some obvious choices are Real-time 3DTV where dynamic viewpoint changes are possible, such as those being investigated by Technicolor (personal communication, May, 2010), or even interactive 3D cinema. Other choices are the visualisation of real-world data from very high-resolution sources, such as planetary data produced by NASA [116].

Appendix A

Definitions

A.1 Display List

A display list is a group of OpenGL commands that have been stored for later execution. When a display list is invoked, the commands in it are executed in the order in which they were issued. Most OpenGL commands can be either stored in a display list or issued in immediate mode, which causes them to be executed immediately. Display lists may improve performance since you can use them to store OpenGL commands for later execution. It is often a good idea to cache commands in a display list if you plan to redraw the same geometry multiple times, or if you have a set of state changes that need to be applied multiple times. Using display lists, you can define the geometry and/or state changes once and execute them multiple times. [15]

A.2 Vertex Array

Vertex data may be placed into arrays that are stored in the client's address space. Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to six arrays: one each to store edge flags, texture coordinates, colours, colour indices, normals, and vertices.

(OpenGL 1.1 spec)

A.3 Vertex Buffer Object (VBO)

A vertex buffer object (VBO) is a powerful feature that allows us to store certain data in high-performance memory on the server side. This feature proposes a mechanism—encapsulating

data within "buffer objects"— for handling these data without having to take them out from the server side, thereby increasing the rate of data transfers. VBOs help with: ⎵ Any data that would be pointed to by a client/state function. Typically we're talking about glVertexPointer(), glColorPointer(), glNormalPointer(), and so on. ⎵ Arrays of indices for drawing a set of elements (glDraw[Range]Elements()). The basic idea of this mechanism is to provide some chunks of memory (buffers) that will be available through identifiers. As with any display list or texture, we can bind such a buffer so that it becomes active. [115]

A.4 Frame Buffer Object (FBO)

Frame buffer Objects are a mechanism for rendering to images other than the default OpenGL Default Frame buffer. They are OpenGL Objects that allow you to render directly to textures, as well as blitting from one frame buffer to another. [72]

A.5 Pixel Buffer Object (PBO)

This extension expands on the interface provided by the ARB_vertex_buffer_object extension (and later integrated into OpenGL 1.5) in order to permit buffer objects to be used not only with vertex array data, but also with pixel data. The intent is to provide more acceleration opportunities for OpenGL pixel commands.

While a single buffer object can be bound for both vertex arrays and pixel commands, we use the designations vertex buffer object (VBO) and pixel buffer object (PBO) to indicate their particular usage in a given situation.

Recall that buffer objects conceptually are nothing more than arrays of bytes, just like any chunk of memory. ARB_vertex_buffer_object allows GL commands to source data from a buffer object by binding the buffer object to a given target and then overloading a certain set of GL commands' pointer arguments to refer to offsets inside the buffer, rather than pointers to user memory. An offset is encoded in a pointer by adding the offset to a null pointer. [8]

A.6 Shader

In a programmable pipeline (as opposed to a fixed-functionality pipeline), the code that runs on one of the programmable processors is known as a *Shader*. Shaders written in

GLSL are known as OpenGL Shaders, to differentiate them from shaders written in other languages such as RenderMan [132].

A.7 Vertex Shader

Vertex Shaders replace all the fixed functionality that was intended to process raw vertices such as modelling, viewing, and projection transformations and vertex shading. Vertex shaders are applied *per vertex* [3].

A.8 Fragment Shader

Fragment Shaders replace the fixed functionality that was intended to process rasterisation, such as interpolation, texture access, polygon-filling and shading. Fragment shaders are applied *per fragment*, where a fragment is generally equivalent to a pixel.

A.9 Multiple-Render-Targets (MRT)

MRT refers to the ability of modern GPUs to render into multiple buffers simultaneously. With MRT, fragment shaders may me used to write multiple values for each fragment and store them in off-screen buffers. These values may then be retrieved in another pass. This makes it possible to implement complex multi-pass rendering algorithms [132].

A.10 Point-Sprites

Point Sprites are texture mapped square polygons created dynamically in hardware from a single point using its size as the side length of the square. Point sprites have the advantage of being memory-efficient since only a single point is required to completely represent them as opposed to four that would otherwise be required. In addition, point sprites may be texture-mapped as well. Point sprites are most commonly used to implement particle systems.

A.11 Image Pyramids

An image pyramid is a type of multi-scale signal representation. In a Gaussian image pyramid, an image is convolved repeatedly with a smoothing kernel and subsampled, creating

a *pyramid* of images, each with successively higher frequencies while travelling down the pyramid.

A.12 Vertex Attributes

By default, OpenGL vertices have predefined properties such as colour, and position. However, under some circumstances it may be necessary to store additional information *per vertex*. OpenGL allows the definition of such custom properties via the `glVertexAttribPointer` keyword.

Appendix B

The GPU Pipeline

B.1 Programmable workflow

Though the modern GPU rendering pipeline is similar to the old fixed functionality rendering pipeline in that they both perform the same functions, the GPU pipeline varies in terms of the amount of *programmability* exposed to the programmer at each stage. Figure B.1 illustrates the programmable pipeline.

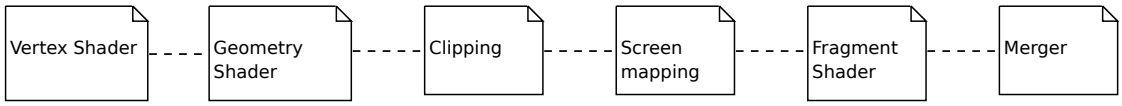


Figure B.1: The programmable pipeline as available on modern GPUs

It should be immediately obvious that the pipeline shares some similarities to the fixed-functionality pipeline. Figure B.2 makes the relationship between the two pipelines explicit.

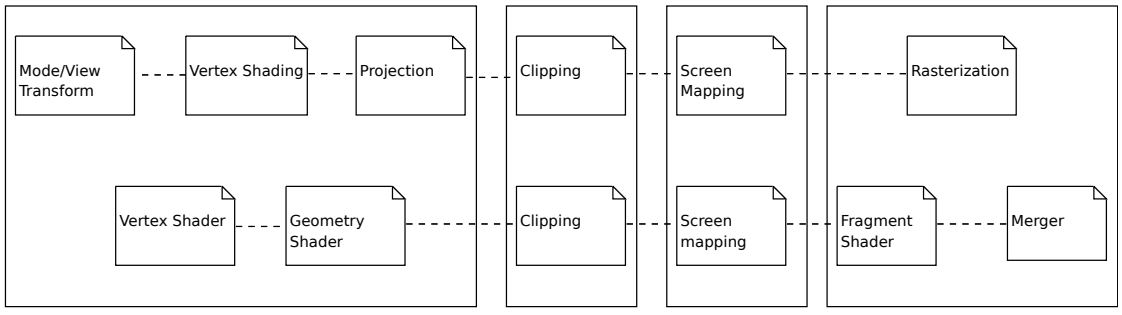


Figure B.2: A comparison of the fixed functionality pipeline (top) and the programmable pipeline (bottom).

The programmable pipeline revolves around the use of *shaders*, units of code that run

entirely on the GPU. The shaders are composed of a vertex shader, that operates per vertex, and a *fragment* shader, that operates per pixel. Optionally, it is possible to define a *geometry shader*, a shader that allows new geometry to be created based on data received from the vertex shader, entirely on the GPU. We will not discuss the geometry shader in this thesis as it is optional, and not relevant to our discussion.

In a programmable pipeline, the CPU-GPU boundary is depicted in B.3.

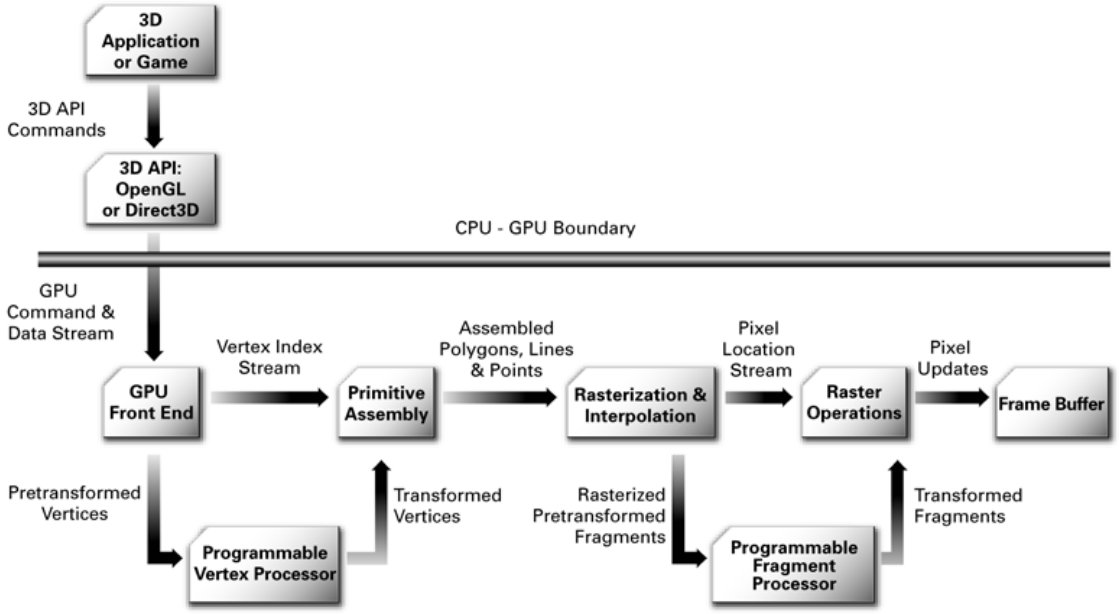


Figure B.3: The CPU-GPU boundary depicts the separation between cpu controlled and GPU controlled elements in the pipeline

B.1.1 The vertex shader

The first three stages of the fixed functionality pipeline are rolled into a *vertex shader* and optionally a *geometry shader*. The vertex shader receives vertex data (or points for point based rendering), and performs transformations upon them. Due to the parallel nature of the GPU, the vertex shader is applied to several vertices simultaneously. The programmable aspect of the vertex shader means that various vertex transformations can now be done on the GPU that were previously done on the CPU such as displacement mapping [127]. In addition, projection is also fully programmable and implemented on the vertex shader. This permits arbitrary projections to be applied directly on the GPU. Finally, the vertex shader may be provided with addition data (as variables passed from the CPU) such as lighting information, such that each vertex may have a different form of

shading equation applied.

B.1.2 The fragment shader

During rasterisation, a fragment shader is applied to each pixel that is associated with a primitive to be rendered to calculate its final colour. A fragment shader *interpolates* between the vertices (or points) computed in the vertex shader. The programmability of the fragment shader permits custom interpolation methods to be defined in addition to defaults such as bilinear or bicubic. Since a fragment shader performs interpolation per pixel based on vertex data obtained from the vertex shader (including lighting information), it is possible to apply per-pixel lighting effects that greatly improve the rendering quality as opposed to the strictly per-vertex lighting calculations that were provided previously by the fixed functionality pipeline.

Figure B.4 presents an overview of the interaction between shaders as defined in OpenGL.

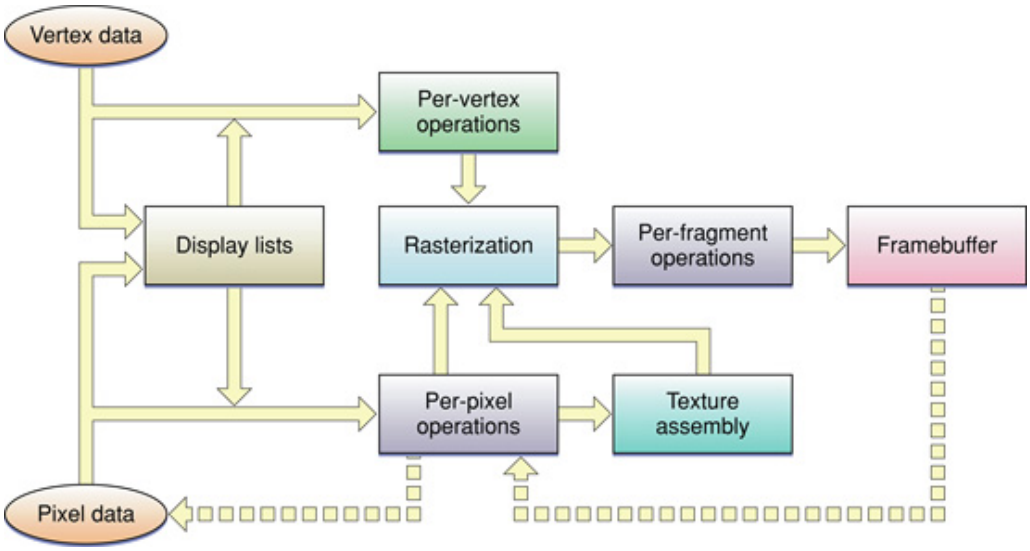


Figure B.4: The OpenGL Pipeline [7]

Appendix C

Publication

The work presented in this thesis has led to the following publications/presentations:

2009 "Gaussian Projection: A Novel PBR Algorithm for Real-Time Rendering," *SIGGRAPH 2009 poster.*

2009 "PYRAMIDAL MULTI-VIEW PBR A Point-Based Algorithm for Multi-view Multi-resolution Rendering of Large Data Sets from Range Images," *GRAPP 2009.*

2009 "Multi-view and Real-time Point-Based Rendering of Photo-Realistic Faces," *BMVA Symposium on Facial Analysis and Animation.*

2009 "Real-time Rendering of Scanned Data," *CVMP 2009, poster.*

PYRAMIDAL MULTI-VIEW PBR

A Point-Based Algorithm for Multi-view Multi-resolution Rendering of Large Data Sets from Range Images

Sajid Farooq, J. Paul Siebert
Computer Vision and Graphics Lab, Dept of Computer Science, University of Glasgow, Glasgow, UK
sajid@dcs.gla.ac.uk, pseibert@dcs.gla.ac.uk

Keywords: Multi-view integration, Point-Based Rendering, Display Algorithms, Viewing Algorithms, Stereo-photogrammetry, Stereo-capture.

Abstract: This paper describes a new Point-Based-Rendering technique that is parsimonious with the typically large data-sets captured by stereo-based, multi-view, 3D imaging devices for clinical purposes. Our approach is based on image pyramids and exploits the implicit topology relations found in range images, but not in unstructured 3D point-could representations. An overview of our proposed PBR-based system for visualisation, manipulation, integration and analysis of sets of range images at native resolution is presented along with initial multi-view rendering results.

1 INTRODUCTION

3D images have the potential to provide clinicians with an objective basis for assessing and measuring 3D surface anatomy, such as the face, foot or breast. Clinicians often resort to subjective measures that rely on naked eye observations, and carry out surgical decisions based upon that data. Today, 3D scanned images of patients can provide objective metric measurements of body surfaces to sub-millimetre resolution. Commercially available stereo-photogrammetry capture systems such as C3D (Siebert & Marshall, 2000) are capable of capturing 3D scans up to 16 megapixels in resolution. Although stereo-photogrammetry systems are desirable for many reasons, they present their own challenges. Large sets of data are difficult to manage, process, and visualize. In addition, stereo systems capture data from multiple 'pods' (each pod consisting of a pair of cameras) around the object, resulting in several 2.5D captures, each with a partial view of the object. Hence, multi-view integration techniques are usually required to join these partial views into a single 3D representation. The goal of this paper is to present progress towards a multi-view, multi-resolution method that permits clinicians to visualise, manipulate, measure and analyse large 3D datasets at native imaging resolution depicting 3D surface anatomy.

Traditionally, the most popular data representation method for displaying 3D data has been the 3D polygon. Large data sets, such as captured by stereo imaging devices, however, are so dense that polygon numbers must be reduced by means of mesh decimation, increasing the size of the remaining polygons and thereby losing resolution. In order to achieve 3D visualisation at native imaging resolution, it is more efficient to treat each 3D (2.5D) measurement as a Point rendering primitive (Levoy and Whitted, 1985) than attempt to render polygons. Large data sets converted to polygons also claim more memory than storing each individual point (as regular range images for example). Polygons are a notoriously difficult representation when it comes to multi-view integration. Marching-cubes (Lorenson and Cline, 1987) is a popular algorithm, however, it rarely works seamlessly with very high-resolution models. The standard techniques, Marching-cubes (Lorenson and Cline, 1987), Zippered Polygon Meshes (Turk, Levoy, 1994), all suffer a loss of resolution at the seams, and provide unpredictable results when polygonal resolution approaches pixel size. In light of the problems with polygon rendering methods, point-

based rendering (PBR) techniques have steadily been gaining interest.

2 PREVIOUS WORK

The idea of using Points as a rendering primitive was reported by Levoy and Whitted as far back as 1985 (Levoy and Whitted, 1985). The most common Point-Based Rendering implementation currently in use is Surface Splatting (Zwicker et al. 2001), where a 3D object is represented as a collection of surface samples. These sample points are reconstructed, low-pass filtered and projected onto the screen plane (Räsänen, 2002). Many extensions have been proposed for Surface Splatting since their introduction. Among others, Splatting has been extended to handle multiple views (Hübner et al. 2006).

Rusinkiewicz and Levoy describe QSplat, a system for representing and progressively displaying meshes that combines a multi-resolution hierarchy based on bounding spheres with a rendering system based on points. A single data structure is used for view-frustum culling, back-face culling, level-of-detail selection, and rendering (Rusinkiewicz and Levoy, 2000).

Both QSplat and Splatting techniques, however, have their limitations. QSplat, while efficient, relies on triangulated mesh data as input rather than native Point data, and lacks anti-aliasing features. Splatting, on the other hand, discards connectivity information that is vital in a clinical context for measurement and analysis of the underlying data.

Several multi-view integration approaches have been proposed. Hubner et al (2006) introduce a new method for multi-view Splatting based on deferred blending. Hilton et al (2006), on the other hand, take the traditional 'polygonization' approach by proposing a continuous surface function that merges the connectivity information inherent in the individual sampled range images and constructs a single triangulated model. Problems with both Splatting techniques, and polygon approaches have been mentioned earlier, making either multi-view technique less than ideal for clinical purposes.

Image pyramids were introduced by Burt and Adelson (1983a) as an efficient and simple multi-resolution scale-space mage representation. Image pyramids, in addition to providing a multi-resolution algorithmic framework, have found use in down-sampling images smoothly across scale-space. Image pyramids, although 2D in nature, were extended by Gortler et al (1996) in the landmark *Lumigraph* paper where they discuss the 'pull-push' algorithm. The latest use of the image pyramid in PBR techniques, and one that is closest to our work, is that of Marroqium et al (2008). They implement the image pyramid on the GPU to provide an accelerated, multi-resolution, Point Based Rendering algorithm based on scattered one-pixel projections, rather than Splats as proposed by Zwicker et al (2001).

Existing techniques, despite making use of range images, and/or image pyramids, have not made the combined use of the connectivity information provided by the former, and the multi-resolution capabilities provided by the latter, to provide a multi-resolution, multi-view PBR algorithm that could be used in a clinical setting for measurement and analysis. We propose a method that takes range images as its input, uses an image pyramid for down-sampling, and smoothly joining multiple views in image space via a multi-resolution Spline as proposed by Burt et al (1983b), and finally, projects the image using 3x3 pixel Gaussian kernels for sub-pixel accurate, anti-aliased rendering.

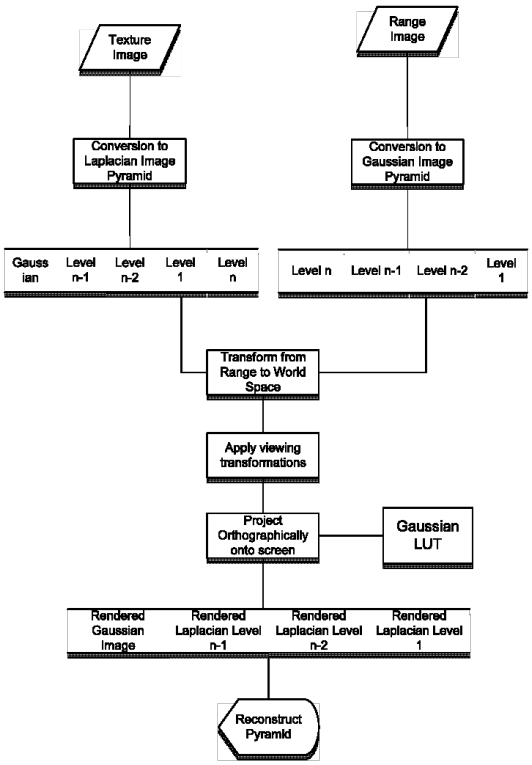


Figure 1: Overview of the rendering process for a single view.

The advantage of using range images, coupled with a PBR approach, is that our method renders

data at its native resolution, retains connectivity information for measurement purposes, and provides a matrix-like data-structure that is compact and ideal for GPU acceleration.

3 THE PROPOSED METHOD

The proposed method uses image pyramids, range images and the Gaussian kernels to provide anti-aliased, hole-free, multi-resolution 3D images. A high-level overview of the algorithm, for a single view, is as follows.

The input range image, provided in our case by a stereo-photogrammetry capture system, is first converted into a Gaussian Pyramid to provide several range images, at subsequently smaller resolutions. Since the range images together comprise 3D data, this effectively provides anati-aliased models at several resolutions. The corresponding texture image is converted into a Laplacian Pyramid, providing a texture image for each of the corresponding models to be derived from the range images. Starting from the apex, i.e the lowest resolution image in the pyramid, each pixel from the range image is transformed from range space to World Coordinates. The colour for this point is derived from the corresponding Texture image pyramid. Once in World Coordinates, the point goes through any pending viewing transformations. Finally, the pixel is projected onto the screen as a 3x3 Guassian kernel. This results in a series of images, of varying sizes, depending upon the level of the Pyramid they are generated from. The images form an image pyramid, in screen-space, with a Gaussian Image at the apex, followed by Laplacian Images containing successively higher-frequency detail.

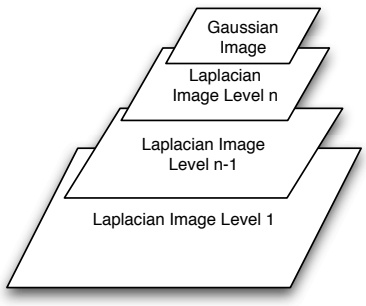


Figure 2: Single-view Output Pyramid.

The resultant images can now be recombined to form a Pyramid in viewport-space again. Though the method outlined above renders a single view, it is extendable to multiple views without any additional effort. A multi-view image can be obtained by repeating the process with another view (another input range image and texture image), and projecting each corresponding level into the same output space. The resulting images represent an image pyramid as before. The result of the reconstruction of this pyramid, however, is a blending of the two views together via a multi-resolution spline as proposed by Burt and Adelson (1983b).

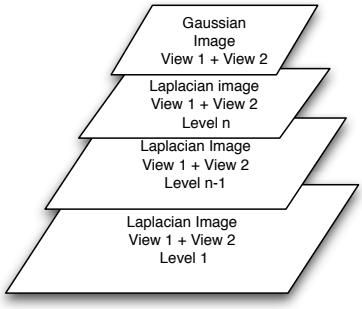


Figure 3: Multi-view Output Pyramid.

3.1 Details of the Rendering Algorithm

The proposed method makes extensive use of image pyramids as defined by Burt (1983a) for seamless splining of the two views, and of Gaussian kernels for sub-pixel anti-aliased display of the points. An explanation of the multi-resolution spline can be found in (Burt and Adelson, 1983b). An explanation of how the Gaussian kernel is used for rendering follows.

3.2 The Gaussian Kernel

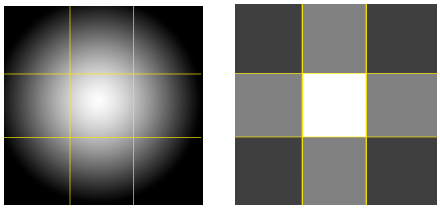


Figure 4: A continuous Gaussian function (left) and its approximation by a 3x3 pixel kernel (right). Shifted versions in x,y allow sub-pixel Gaussian splat placement.

A single point can be approximated by a continuous Gaussian function. For display, it needs to be transformed into discrete values. For every fractional

pixel value, a new Gaussian is generated, offset from the centre. In order to speed up the process, a Look-Up Table was generated for 10,000 kernels thereby providing 0.01pixel shift resolution in *x,y*.

If the image is rendered using the Gaussian kernels as-is, several bright patches appear on the final image where the Gaussian kernels overlap. The image is therefore *normalized* by dividing it by a *Splat map*.

Figure 6: The *Splat map* combining the two overlapping input range map views of Figure 5.

The *Splat map* is generated by first rendering the Gaussian kernels *without colour from the texture map* into a separate buffer to keep a count of the contribution from each Gaussian kernels that falls into each pixel. This defines each pixels weight. The *un-normalized* image is then divided pixel-wise by this *Splat map* to obtain the final, normalized, image.

4 ONGOING WORK

From the current results, it is obvious that Hidden-Surface Removal is required. Hidden-surface Removal may be implemented by treating a group of three connected points as an implicit polygon, and performing Back-Face Culling, and ordering the points using any of the well-known polygon-ordering techniques such as the Z-Buffer.

The existing method combines two views in image space via a multi-resolution spline, however, for the purposes of measurement, it is necessary to employ a multi-view algorithm that merges the underlying data. Ju et al (2004) describes view-integration based on polygons. We propose to extend their algorithm to work with range images and image pyramids, and make improvements to the basic algorithm in the process. The algorithm proposed by Ju et al begins with a blue-screen stereo capture of an object. The blue-screen permits masking of the background, selectively isolating the object. The range images are then decomposed into subset patches, categorising elements into *visible*, *invisible*, *overlapping*, and *unprocessed* patches when compared with a second range image. To resolve ambiguities in a range image, a *confidence competition* is conducted, whereby overlapping patches are culled, and the remaining *winning* patches are merged into a single mesh. It should be noted that this process needs to be carried out only once, as a pre-processing step.

Since our data representation uses groups of points (as opposed to polygons), it will work on individual pixels rather than breaking down the range image into patches. The following algorithm summarizes the process:

```
N = Num of Range Images
Masks of All range Images = 0

loop from 1 to N
      Compare every Range image i
      With every other Range Image j
      if i != j
            {
            project range-map j onto i
            find overlapping pixels

            for each overlapping pixel
            {
            For both views j and j:
            use confidence,
            normal_map, chroma_map to
            find competition_weight_i
            and competition_weight_j
            for current pixel

            if comptetion_weight_i -
            competition weight of j <
            threshold:
            mask[currentpixel] = 1.0
            }
```

Since multi-view stereo-photogrammetry relies on range images being generated from cameras in close vicinity, there will be considerable overlap between various range images that are produced from multiple views, especially those that are close. Before we integrate the models, it is necessary to take care of this redundant data. As proposed by Ju et al, it is necessary to carry out a 'competition' in which the best data from each range image is selected.

First, it is necessary to find precisely the redundant data, i.e., where range images overlap. Hence, we traverse through each range image, and scan every other range image from this point-of-view (by projecting them into range image space) to find the overlapping pixels.

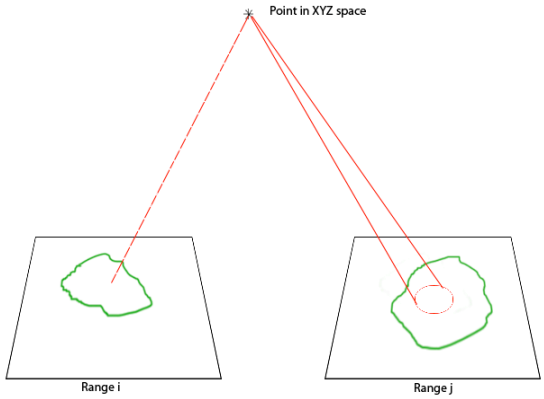


Figure 7: Scanning range image j from the point-of-view of range image i.

For each overlapping pixel from both views (view j and View i), we can isolate relevant data from the background with the help of a blue-screen/chroma mask we call **S**. If the pixel is deemed to be part of the model (and not the background), we can proceed to calculate the *confidence* that a pixel is visible from this view with the help of a "normal map" as well as a "confidence map" of the same view, depicting how confident the 3D scanner was about the regeneration of each individual point in 3D. We call the *Confidence* value **C**. In addition, for both views, for every overlapping pixel (in range space), we can consider how visible a point is to a particular view by checking how closely the normal points towards the view. We can represent this as **V** (for Viewing-Angle). The three maps together, then provide a selection mask with values [0..1], with 1 being completely visible, 0 being completely invisible, and a value in-between depicting a semi-visible pixel. This can be written as:

$$\textbf{Competition Weight} = \mathbf{S\ C\ V} \qquad (1)$$

The entire process is summarized in Figure 8 as follows:

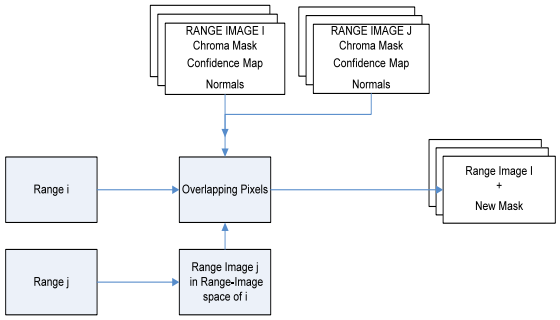


Figure 8: Confidence Competition overview

At this stage, we can determine which of the two views won the competition for this particular pixel. If it was view j, then we mask the current pixel in view i to that during projection, we will not choose this pixel from view i again.

A peculiar case arises when for a certain point, two views tie in the competition, i.e, when there is a 'draw'. In such a case, there are several paths that can be taken. An assortment of fusion/blending techniques is available. Which one of these techniques is most effective is a question that must be further investigated.

Once data-integration has been accomplished, measurement operations can be carried out natively over the range images. Traversing over the range images is decidedly straightforward due to the range image's matrix-like nature.

5 RESULTS AND CONCLUSIONS

Though the work is still ongoing, initial results of our system can be seen in the images that follow. An initial test result, based on a *shallow* blend, reveals the sources of the two input views, Figure 8. By creating a 6 layer deep pyramid, the blend better conceals the join between views, Figure 9.

Figure 9: Result of the proposed method with a Pyramid 3 levels deep

Figure 10: (Left) The result of the proposed method with a Pyramid 3 levels deep (Right) The result with a pyramid 6 levels deep

Without hidden point removal, self occluded regions of the model blend together in areas such as the chin and the ear towards the left of the image. While, the rendering is currently not carried out in real-time, the proposed method lends itself to GPU optimization. The above issues will be addressed in during our ongoing research work to implement the complete system for clinical visualisation, manipulation, measurement and analysis of multi-view range images of surface anatomy.

REFERENCES

A. Hilton, A.J. Toddart, J. Illingworth, and T. Windeatt. 1996. *Reliable surface reconstruction from multiple range images.* In Fourth European Conference on Com- (a) (b) (e) (f) (g) (c) (d)

Burt, P. J. and Adelson, E. H., April 1983a. *The Laplacian Pyramid as a Compact Image Code*, IEEE Trans. on Communications, pp. 532--540.

Burt, P. J. and Adelson, E. H. 1983b. A *multiresolution spline with application to image mosaics*. ACM Trans. Graph. 2, 4 (Oct. 1983), 217-236.

Dimensional Imaging, www.DI3D.com [28 Oct 2008]

Gortler, S. J., Grzeszczuk, R., Szeliski, R., and Cohen, M. F. 1996. The lumigraph. In *Proceedings of the 23rd Annual Conference on Computer Graphics and interactive Techniques* SIGGRAPH '96. ACM, New York, NY, 43-54.

Hübner, T., Zhang, Y., and Pajarola, R. 2006. *Multi-view point splatting*. In Proceedings of the 4th international Conference on Computer Graphics and interactive Techniques in Australasia and Southeast Asia (Kuala Lumpur, Malaysia, November 29 - December 02, 2006). GRAPHITE '06. ACM, New York, NY, 285-294.

Ju, X., Boyling, T., Siebert, J. P., McFarlane, N., Wu, J., and Tillett, R. 2004. Integration of range images in a Multi-View Stereo System. In *Proceedings of the Pattern Recognition, 17th international Conference on (Icpr'04) Volume 4 - Volume 04* (August 23 - 26, 2004). ICPR. IEEE Computer Society, Washington, DC, 280-283.

Levoy, Whitted, January, 1985. Technical Report 85-022. *The Use of Points as a Display Primitive,* Computer Science Department, University of North Carolina at Chapel Hill.

Lorenson, Cline, 1987. *Marching Cubes: A High Resolution 3D Surface Construction Algorithm.* In *SIGGRAPH '87 Proc.,* vol. 21, pp. 163-169.

Marroquim, R., Kraus, M., and Cavalcanti, P. R. 2008. Special Section: Point-Based Graphics: Efficient image reconstruction for point-based and line-based rendering. *Comput. Graph.* 32, 2 (Apr. 2008), 189-203.

Räsänen, J. 2002. *Surface Splatting: Theory, Extensions and Implementation*. Master's thesis, Helsinki University of Technology.

Rusinkiewicz, S. and Levoy, M. 2000. *QSplat: a multiresolution point rendering system for large meshes.* In Proceedings of the 27th Annual Conference on Computer Graphics and interactive Techniques International Conference on Computer Graphics and Interactive Techniques. ACM Press/Addison-Wesley Publishing Co., New York, NY, 343-352.

Siebert,J.P. Marshall,S.J. 2000. *Human Body 3D imaging by speckle texture projection photogrammetry* Sensor Review. Volume 20, No 3 pp 218-226.

Turk, Levoy, 1994. *Zippered polygon meshes from range images.* In Proceedings of the 21st Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '94. ACM, New York, NY, 311-318.

Zwicker, M., Pfister., H., Baar, J. V., and Gross, M. 2001. *Surface Splatting*. In Computer Graphics, SIGGRAPH 2001 Proceedings, pages 371--378. Los Angeles, CA.

Bibliography

[1] *The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games*. ACM/SPIE Multimedia Computing and Networking (MMCN) Conference, 2006.

[2] Jun Yang 0005, Zhengning Wang, Changqian Zhu, and Qiang Peng. Implicit surface reconstruction from scattered point data with noise. In *International Conference on Computational Science (2)*, pages 57–64, 2007.

[3] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[4] M. Alexa, J. Behr, D. Cohen, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 21–28, Washington, DC, USA, 2001. IEEE Computer Society.

[5] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Computing and rendering point set surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):3–15, January 2003.

[6] Daniel Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Kenny Hoff, Tom Hudson, Wolfgang Stuerzlinger, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manocha. Mmr: an interactive massive model rendering system using geometric and image-based acceleration. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, I3D '99, pages 199–206, New York, NY, USA, 1999. ACM.

[7] Apple. *OpenGL programming guide for Mac OS X*. Apple, Cupertino, CA, USA, December 2007.

[8] ARB. Arb pixel buffer object extension specification.

[9] T.K. Ates and A.A. Alatan. Real-time arbitrary view rendering on gpu from stereo video and time-of-flight camera. In *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON), 2011*, pages 1–4, may 2011.

[10] Adrien Bartoli and Peter Sturm. Structure from motion using lines: Representation, triangulation and bundle adjustment. *Computer Vision and Image Understanding*, 100(3):416–441, dec 2005.

[11] Fabio Bettio, Enrico Gobbetti, Fabio Marton, Alex Tinti, Emilio Merella, and Roberto Combet. A point-based system for local and remote exploration of dense 3d scanned models. In *VAST09: The 10th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, volume Full Papers, pages 25–32, St. Julians, Malta, 11/2009 2009. Eurographics Association, Eurographics Association.

[12] Stephan Bischoff and Leif Kobbelt. Sub-voxel topology control for level-set surfaces. *Computer Graphics Forum*, 22:273–280, 2003.

[13] James F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *SIGGRAPH Comput. Graph.*, 16(3):21–29, 1982.

[14] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1.* Addison-Wesley Professional, 2007.

[15] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Versions 3.0 and 3.1.* Addison-Wesley Professional, seventh edition, July 2009.

[16] Naveen Kumar Bola. High quality rendering of large point-based surfaces. Master's thesis, IIIT, June 2010.

[17] Louis Borgeat, Guy Godin, François Blais, and Christian Lahanier. Gold: interactive display of huge colored and textured models. *ACM Trans. Graph*, 24:869–877, 2005.

[18] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's gpus. In *Point-Based Graphics, 2005. Eurographics/IEEE VGTC Symposium Proceedings*, pages 17 – 141, june 2005.

[19] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern gpus. 2003.

[20] Mario Botsch and Leif Kobbelt. Real-time shape editing using radial basis functions. In *Computer Graphics Forum*, pages 611–621, 2005.

[21] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. 2004.

[22] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient high quality rendering of point sampled geometry. In *In: EGRW '02: proceedings of the 13th eurographics workshop on rendering*, pages 53–64, 2002.

[23] Tamy Boubekeur, Florent Duguet, and Christophe Schlick. Rapid visualization of large point-based surfaces. In M. Mudge, N. Ryan, and R. Scopigno, editors, *Proceedings of the International Symposium on Virtual Reality, Archeology and Cultural Heritage (VAST)*. Eurographics, Eurographics, November 2005.

[24] John Branch, Flavio Prieto, and Pierre Boulanger. Automatic hole-filling of triangular meshes using local radial basis function. In *Proceedings of the Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)*, 3DPVT '06, pages 727–734, Washington, DC, USA, 2006. IEEE Computer Society.

[25] Morgan Brown. lapl-pyr-recon. `http://sepwww.stanford.edu/~morgan/texturematch/Gif/lapl-pyr-recon.gif`. Retrieved April 28, 2012, from http://sepwww.stanford.edu.

[26] F. Bruls, S. Zinger, and L. Do. Multi-view coding and view synthesis for 3dtv. In *Consumer Electronics (ICCE), 2011 IEEE International Conference on*, pages 685 –686, jan. 2011.

[27] Peter J. Burt and Edward H. Adelson. A multiresolution spline with application to image mosaics. *ACM Trans. Graph.*, 2(4):217–236, 1983.

[28] Peter J. Burt and Edward H. Adelson. The laplacian pyramid as a compact image code. pages 671–679, 1987.

[29] Marco Callieri, Federico Ponchio, Paolo Cignoni, and Roberto Scopigno. Virtual inspector: a flexible visualizer for dense 3d scanned models. *IEEE Computer Graphics and Applications*, 28(1):44–55, Jan.-Febr. 2008.

[30] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 67–76, New York, NY, USA, 2001. ACM.

[31] J.C. Carr, W.R. Fright, and R.K. Beatson. Surface interpolation with radial basis functions for medical imaging. *Medical Imaging, IEEE Transactions on*, 16(1):96 –107, feb. 1997.

[32] Hsin-Jung Chen, Feng-Hsiang Lo, Fu-Chiang Jan, and Sheng-Dong Wu. Real-time multi-view rendering architecture for autostereoscopic displays. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 1165 –1168, 30 2010-june 2 2010.

[33] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 279–288, New York, NY, USA, 1993. ACM.

[34] Wei Chen, Liu Ren, Matthias Zwicker, and Hanspeter Pfister. Hardware-accelerated adaptive ewa volume splatting. In *Proceedings of the conference on Visualization '04*, VIS '04, pages 67–74, Washington, DC, USA, 2004. IEEE Computer Society.

[35] Zhi-Quan Cheng, Yan-Zhen Wang, Bao Li, Kai Xu, Gang Dang, and Shi-Yao Jin. A Survey of Methods for Moving Least Squares Surfaces. *Proceedings of Point Based Graphics*, 2008.

[36] C.H. Chien, Y.B. Sim, and J.K. Aggarwal. Generation of volume/surface octree from range data. In *Computer Vision and Pattern Recognition, 1988. Proceedings CVPR '88., Computer Society Conference on*, pages 254 –260, jun 1988.

[37] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory management and simplification of huge meshes. *Visualization and Computer Graphics, IEEE Transactions on*, 9(4):525 – 537, oct.-dec. 2003.

[38] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Batched multi triangulation. In *Proceedings IEEE Visualiza-*

tion, pages 207–214, Conference held in Minneapolis, MI, USA, October 2005. IEEE Computer Society Press.

[39] James H. Clark. Hierarchical geometric models for visible-surface algorithms. In *SIGGRAPH '76: Proceedings of the 3rd annual conference on Computer graphics and interactive techniques*, pages 267–267, New York, NY, USA, 1976. ACM.

[40] Richard D. Corbett. Point-based level sets and progress towards unorganized particle based fluids. Technical report, Memorial University of Newfoundland, 2005.

[41] Wagner Toledo Correa. *New techniques for out-of-core visualization of large datasets*. PhD thesis, Princeton, NJ, USA, 2004. AAI3110226.

[42] Andrew Corrigan and H. Quynh Dinh. Computing and rendering implicit surfaces composed of radial basis functions on the gpu, 2005.

[43] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press. to appear.

[44] D. E. Crispell. *A continuous probabilistic scene model for aerial imagery*. PhD thesis, Brown University, 2010.

[45] C. Csuri, R. Hackathorn, R. Parent, W. Carlson, and M. Howard. Towards an interactive high visual complexity animation system. In *SIGGRAPH '79: Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, pages 289–299, New York, NY, USA, 1979. ACM.

[46] Brian Curless. From range scans to 3d models. *SIGGRAPH Comput. Graph.*, 33(4):38–41, 2000.

[47] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 303–312, New York, NY, USA, 1996. ACM.

[48] Boguslaw Cyganek and J. Paul Siebert. *An Introduction to 3D Computer Vision Techniques*. Wiley-Blackwell, September 2009.

[49] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 657–662, New York, NY, USA, 2003. ACM.

[50] Jonathan Cohen David Luebke, Martin Reddy. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers Inc., 2003.

[51] Peter H. N. de With and Svitlana Zinger. Free-viewpoint rendering algorithm for 3d tv. In *2nd International Workshop of Advances in Communication*, pages 19–23, 2009.

[52] Rosen Diankov and Ruzena Bajcsy. Real-time adaptive point splatting for noisy point clouds. In *GRAPP (GM/R)'07*, pages 228–234, 2007.

[53] Huong Quynh Dinh, Greg Turk, and Greg Slabaugh. Reconstructing surfaces by volumetric regularization using radial basis functions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24:1358–1371, October 2002.

[54] Luat Do, S. Zinger, Y. Morvan, and P.H.N. de With. Quality improving techniques in dibr for free-viewpoint video. In *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video, 2009*, pages 1 –4, may 2009.

[55] Zhiqiang Du and Qiaoxiong Li. A new method of storage and visualization for massive point cloud dataset. In *22nd CIPA Symposium*, 2009.

[56] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineev-Weinstein. Roaming terrain: Real-time optimally adapting meshes. In *Visualization '97., Proceedings*, pages 81 –88, oct. 1997.

[57] Sajid Farooq and J. Paul Siebert. Pyramidal multi-view pbr - a point-based algorithm for multi-view multi-resolution rendering of large data sets from range images. In Alpesh Ranchordas, João Pereira, and Paul Richard, editors, *GRAPP*, pages 211–216. INSTICC Press, 2009.

[58] Haitham Fattah, Gerardo Aragon-camarasa, and J. Paul Siebert. Towards binocular active vision in a robot head. In *Towards Autonomous Robotic Systems*. TAROS, 2008.

[59] Shachar Fleishman, Daniel Cohen-Or, Marc Alexa, and Cláudio T. Silva. Progressive point set surfaces. *ACM Trans. Graph.*, 22:997–1011, October 2003.

[60] John Fujii, editor. *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, New York, NY, USA, 2005. ACM.

[61] Thomas A. Funkhouser. Database management for interactive display of large architectural models. In *Proceedings of the conference on Graphics interface '96*, pages 1–8, Toronto, Ont., Canada, Canada, 1996. Canadian Information Processing Society.

[62] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 247–254, New York, NY, USA, 1993. ACM.

[63] Thomas A. Funkhouser, Carlo H. Séquin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Proceedings of the 1992 symposium on Interactive 3D graphics*, I3D '92, pages 11–20, New York, NY, USA, 1992. ACM.

[64] FutureMark Corporation. 3DMark06 v1.1.0 - [Computer software]. `http://www.futuremark.com/benchmarks/3dmark06/`, 2006. Retrieved April 28, 2012, from http://www.futuremark.com.

[65] FutureMark Corporation. 3DMark06 Whitepaper v1.0.2 [Whitepaper]. `http://www.futuremark.com/benchmarks/3dmark06/`, 2006. Retrieved April 28, 2012, from http://www.futuremark.com.

[66] Enrico Gobbetti, Dave Kasik, and Sung-eui Yoon. Technical strategies for massive model visualization. In *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, SPM '08, pages 405–415, New York, NY, USA, 2008. ACM.

[67] Enrico Gobbetti and Fabio Marton. Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Comput. Graph.*, 28:815–826, December 2004.

[68] Prashant Goswami, Yanci Zhang, Renato Pajarola, and Enrico Gobbetti. High quality interactive rendering of massive point models using multi-way kd-trees. In *Proceedings of the 2010 18th Pacific Conference on Computer Graphics and Applications*,

PACIFIC GRAPHICS 10, pages 93–100, Washington, DC, USA, 2010. IEEE Computer Society.

[69] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 231–238, New York, NY, USA, 1993. ACM.

[70] Markus Gross and Hanspeter Pfister. *Point-Based Graphics (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[71] J. P. Grossman and William J. Dally. Point sample rendering. In *Rendering Techniques*, pages 181–192. Springer, 1998.

[72] The Khronos Group. The official opengl wiki. `http://www.opengl.org/wiki/Framebuffer_Objects`. Retrieved April 28, 2012, from http://www.opengl.org.

[73] Gaël Guennebaud, Loïc Barthe, and Mathias Paulin. Deferred splatting. *Comput. Graph. Forum*, 23(3):653–660, 2004.

[74] Gaël Guennebaud, Marcel Germann, and Markus H. Gross. Dynamic sampling and rendering of algebraic point set surfaces. *Comput. Graph. Forum*, 27(2):653–662, 2008.

[75] Gaël Guennebaud and Markus Gross. Algebraic point set surfaces. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[76] Mohammad Y. Hajeer, Zhili Mao, Declan T. Millett, Ashraf F. Ayoub, and Jan Paul Siebert. A new three-dimensional method of assessing facial volumetric changes after orthognathic treatment. *The Cleft Palate-Craniofacial Journal*, 42(2):113–120, 2012/04/29 2005.

[77] Akihiro Hayasaka, Takuma Shibahara, Koichi Ito, Takafumi Aoki, Hiroshi Nakajima, and Koji Kobayashi. A passive 3d face recognition system and its performance evaluation. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E91-A:1974–1981, August 2008.

[78] Donald Hearn and M. Pauline Baker. *Computer Graphics C version*. Prentice Hall, 2nd edition, 1997.

[79] Paul S. Heckbert. Fundamentals of texture mapping and image warping. Technical Report UCB/CSD-89-516, EECS Department, University of California, Berkeley, Jun 1989.

[80] Simon Heinzle, Gaël Guennebaud, Mario Botsch, and Markus Gross. A hardware processing unit for point sets. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 21–31, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[81] Simon Heinzle, Johanna Wolf, Yoshihiro Kanamori, Tim Weyrich, Tomoyuki Nishita, and Markus Gross. Motion blur for ewa surface splatting. *Computer Graphics Forum*, 29(2):733–742, 2010.

[82] A. Hilton, A.J. Stoddart, J. Illingworth, and T. Windeatt. Marching triangles: range image fusion for complex object modelling. In *Image Processing, 1996. Proceedings., International Conference on*, volume 1, pages 381 –384 vol.2, sep 1996.

[83] Adrian Hilton. On reliable surface reconstruction from multiple range images. pages 117–126. Springer-Verlag, 1996.

[84] Adrian Hilton, A. J. Stoddart, John Illingworth, and T. Windeatt. Reliable surface reconstruction from multiple range images. In *ECCV '96: Proceedings of the 4th European Conference on Computer Vision-Volume I*, pages 117–126, London, UK, 1996. Springer-Verlag.

[85] Hugues Hoppe. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, New York, NY, USA, 1996. ACM.

[86] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings of the conference on Visualization '98*, VIS '98, pages 35–42, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[87] Dongjin Huang, Jianliang Lan, Wu Liu, and Youdong Ding. High efficiency real time rendering for point-based model on gpu. In *Proceedings of the 2009 International Conference on Multimedia Information Networking and Security - Volume 01*, MINES '09, pages 296–300, Washington, DC, USA, 2009. IEEE Computer Society.

[88] Thomas Hübner, Yanci Zhang, and Renato Pajarola. Multi-view point splatting. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 285–294, New York, NY, USA, 2006. ACM.

[89] Thomas Hübner, Yanci Zhang, and Renato Pajarola. Single-pass multi-view rendering. *IADIS International Journal on Computer Science and Information Systems*, 2(2):122–140, October 2007.

[90] D. Q. Huynh, R. A. Owens, and P. E. Hartmann. Calibrating a structured light stripe system: A novel approach. *Int. J. Comput. Vision*, 33:73–86, September 1999.

[91] Katsushi Ikeuchi, Daisuke Miyazaki, Ryusuke Sagawa, Ko Nishino, Mark D. Wheeler, and Katsushi Ikeuchi. Parallel processing of range data merging. In *Digitally Archiving Cultural Objects*, pages 147–159. Springer US, 2008.

[92] Xiangyang Ju, Tim Boyling, J. Paul Siebert, Nigel McFarlane, Jiahua Wu, and Robin Tillett. Integration of range images in a multi-view stereo system. In *ICPR '04: Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 4*, pages 280–283, Washington, DC, USA, 2004. IEEE Computer Society.

[93] Xiangyang Ju, J.P. Siebert, B.S. Khambay, and A.F. Ayoub. Self-correction of 3d reconstruction from multi-view stereo images. In *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, pages 1606 –1613, 27 2009-oct. 4 2009.

[94] David Laur and Pat Hanrahan. Hierarchical splatting: a progressive refinement algorithm for volume rendering. *SIGGRAPH Comput. Graph.*, 25:285–288, July 1991.

[95] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 131–144, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[96] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical report, University of North Carolina at Chapel Hill, 1985.

[97] Xiaokun Li and W.G. Wee. Overlap elimination for registered range images. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 4297 –4300, nov. 2009.

[98] Xinju Li and I. Guskov. 3d object recognition from range images using pyramid matching. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1 –6, oct. 2007.

[99] H. Pfister Liu Ren and M. Zwicker. Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. *COMPUTER GRAPHICS FORUM*, 21:461–470, 2002.

[100] Tsz-Wai Rachel Lo. *Feature Extraction for Range Image Interpretation using Local Topology Statistics*. PhD thesis, Computing Science, University of Glasgow, January 2009.

[101] Tsz-Wai Rachel Lo and J. Paul Siebert. Local feature extraction and matching on range images: 2.5d sift. *Comput. Vis. Image Underst.*, 113(12):1235–1250, December 2009.

[102] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM.

[103] M. Zwicker L. Kobbelt M. Botsch, A. Hornung. High-quality surface splatting on today's gpus. In *Proceedings of the Eurographics Symposium on Point-Based Graphics*, 2005.

[104] N.A. Manap and J.J. Soraghan. Novel view synthesis based on depth map layers representation. In *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON), 2011*, pages 1–4, may 2011.

[105] Jonathan Marbach. Gpu acceleration of stereoscopic and multi-view rendering for virtual reality applications. In *Proceedings of the 16th ACM Symposium on Virtual*

Reality Software and Technology, VRST '09, pages 103–110, New York, NY, USA, 2009. ACM.

[106] Ricardo Marroquim, Martin Kraus, and Paulo Roma Cavalcanti. Special section: Point-based graphics: Efficient image reconstruction for point-based and line-based rendering. *Comput. Graph.*, 32(2):189–203, 2008.

[107] F. Marton, E. Gobbetti, F. Bettio, J.A.I. Guitian, and R. Pintus. A real-time coarse-to-fine multiview capture system for all-in-focus rendering on a light-field display. In *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON), 2011*, pages 1–4, may 2011.

[108] Leonard McMillan and Gary Bishop. Plenoptic modeling: an image-based rendering system. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 39–46, New York, NY, USA, 1995. ACM.

[109] Y. Mori, N. Fukushima, T. Fujii, and M. Tanimoto. View generation with 3d warping using depth information for ftv. In *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video, 2008*, pages 229 –232, may 2008.

[110] Yannick Morvan. Acquisition, compression and rendering of depth and texture for multi-view video. In *PhD thesis*, 2009.

[111] Guangyu Mu, Miao Liao, Ruigang Yang, Dantong Ouyang, Zhiwen Xu, and Xiaoxin Guo. Complete 3d model reconstruction using two types of depth sensors. In *Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE International Conference on*, volume 1, pages 162 –166, oct. 2010.

[112] S. K. Nayar and Y. Nakagawa. Shape from focus. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(8):824–831, 1994.

[113] N. Neophytou and K. Mueller. Gpu accelerated image aligned splatting. In *Volume Graphics, 2005. Fourth International Workshop on*, pages 197 – 242, june 2005.

[114] Christof Nuber, Ralph W. Bruckschen, Bernd Hamann, and Kenneth I. Joy. Interactive visualization of very large medical datasets using point-based rendering. volume 5029, pages 27–36. SPIE, 2003.

[115] NVIDIA. *Using Vertex Buffer Objects*. NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050, March 2003.

[116] The University of Arizona. Hirise - high resolution imaging science experiment. `http://hirise.lpl.arizona.edu`. Retrieved April 28, 2012, from http://hirise.lpl.arizona.edu.

[117] Opensource. Wikipedia, the free encyclopaedia. `//http://www.wikipedia.org`. Retrieved April 28, 2012, from http://www.wikipedia.org.

[118] Stanley Osher and James A. Sethian. Fronts propagating with curvature dependent speed: Algorithms based on hamilton-jacobi formulations. *JOURNAL OF COMPUTATIONAL PHYSICS*, 79(1):12–49, 1988.

[119] A. C. Oztireli, G. Guennebaud, and M. Gross. Feature Preserving Point Set Surfaces based on Non-Linear Kernel Regression. *Computer Graphics Forum*, 28(2):493–501, April 2009.

[120] Jordi Pagès and Joaquim Salvi. Coded light projection techniques for 3d reconstruction. 2005.

[121] Renato Pajarola. Efficient level-of-details for point based rendering. In *Conference on Computer Graphics and Imaging*, pages 141–146, 2003.

[122] Renato Pajarola, Miguel Sainz, and Patrick Guidotti. Confetti: Object-space point blending and splatting. *IEEE Transactions on Visualization and Computer Graphics*, 10:598–608, 2004.

[123] Renato Pajarola, Miguel Sainz, and Roberto Lario. Xsplat: External memory multiresolution point visualization. pages 628–633. IASED VIIP, 2005.

[124] Ziyuan Pan, Y. Ikuta, M. Bandai, and T. Watanabe. Bandwidth-efficient user dependent transmission for multi-view video. In *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON), 2011*, pages 1–4, may 2011.

[125] O. Petrik and L. Vasa. Improvements of mpeg-4 standard famc for efficient 3d animation compression. In *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON), 2011*, pages 1–4, may 2011.

[126] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: surface elements as rendering primitives. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[127] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[128] Kari Pulli, Tom Duchamp, John Mcdonald, Linda Shapiro, and Werner Stuetzle. Robust meshes from multiple range maps. In *In Proceedings of International Conference on Recent Advances in 3-D Digital Imaging and Modeling*, pages 205–211, 1997.

[129] Kari Antero Pulli. *Surface reconstruction and display from range and color data*. PhD thesis, 1997. AAI9819292.

[130] He Qiang, Zhang Shusheng, Bai Xiaoliang, and Zhang Xin. Hole filling based on local surface approximation. In *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, volume 3, pages V3–242 –V3–245, oct. 2010.

[131] Hongxing Qin, Jie Yang, and Yue Min Zhu. Novel meshless method for point set surface processing, April 2008.

[132] Dan Ginsburg Randi J. Rost, Bill Licea-Kane. *OpenGL Shading Language*. Addison-Wesley Professional, seventh edition edition, July 2009.

[133] Jussi Räsänen, Lauri Savioja, and Timo Aila. Surface splatting: Theory, extensions and implementation. Master's thesis, Helsinki University of Technology, 2002.

[134] Michael K. Reed and Peter K. Allen. 3-d modeling from range imagery: An incremental method with a planning component. In *Image and Vision Computing*, pages 76–83, 1999.

[135] W. T. Reeves. Particle systems–a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[136] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. In *Computer Graphics Forum*, pages 461–470, 2002.

[137] Patrick Reuter, Ireneusz Tobor, Christophe Schlick, and Sébastien Dedieu. Point-based modelling and rendering using radial basis functions. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 111–118, New York, NY, USA, 2003. ACM.

[138] Patrick Reuter, Ireneusz Tobor, Christophe Schlick, and Sébastien Dedieu. Point-based modelling and rendering using radial basis functions. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '03, pages 111–118, New York, NY, USA, 2003. ACM.

[139] Tom's Hardware review team. Tom's hardware - the authority on tech. `http://www.tomshardware.com`. Retrieved April 28, 2012, from http://www.tomshardware.com.

[140] Rico Richter and Jürgen Döllner. Out-of-core real-time visualization of massive 3d point clouds. In *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '10, pages 121–128, New York, NY, USA, 2010. ACM.

[141] Marcos Rodrigues, Robert Fisher, and Yonghuai Liu. Special issue on registration and fusion of range images. *Comput. Vis. Image Underst.*, 87:1–7, July 2002.

[142] J Rossignac and P Borrel. Multi-resolution 3d approximations for rendering complex scenes. In *Modeling in Computer Graphics: Methods and Applications*, pages 455–465, 1993.

[143] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph.*, 14:110–116, July 1980.

[144] Szymon Rusinkiewicz and Marc Levoy. Qsplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[145] Szymon Rusinkiewicz and Marc Levoy. Streaming qsplat: A viewer for networked visualization of large, dense models, 2001.

[146] Martin Rutishauser, Markus Stricker, and Marjan Trobina. Merging range images of arbitrarily shaped objects. In *In Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 573–580, 1994.

[147] B. Sabata, F. Arman, and J.K. Aggarwal. Segmentation of 3d range images using pyramidal data structures. In *Computer Vision, 1990. Proceedings, Third International Conference on*, pages 662 –666, dec 1990.

[148] Miguel Sainz and Renato Pajarola. Point-based rendering techniques. *Comput. Graph.*, 28(6):869–879, 2004.

[149] R. Schaback. *A Practical Guide To Radial Basis Functions*. Electronic Resource, 2007.

[150] H.-J. Schulz, S. Hadlak, and H. Schumann. Point-based tree representation: A new approach for large hierarchies. In *Visualization Symposium, 2009. PacificVis '09. IEEE Pacific*, pages 81 –88, april 2009.

[151] Sebastian Schuon, Christian Theobalt, James Davis, and Sebastian Thrun. High-quality scanning using time-of-flight depth superresolution. *CVPR Workshop on Time-of-Flight Computer Vision 2008*, 2008.

[152] J. Paul Siebert and Stephen J. Marshall. Human body 3d imaging by speckle texture projection photogrammetry. *Sensor Review*, 20:218 – 226, 2000.

[153] Marc Stamminger and George Drettakis. Interactive sampling and rendering for complex and procedural geometry. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 151–162, London, UK, 2001. Springer-Verlag.

[154] G. A. Triantafyllidis, A. Enis Cetin, Aljoscha Smolic, Levent Onural, Thomas Sikora, and John Watson. 3dtv: Capture, transmission, and display of 3d video. *EURASIP Journal on Advances in Signal Processing*, 2009(585216):2, 2009.

[155] Greg Turk and Marc Levoy. Zippered polygon meshes from range images. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 311–318, New York, NY, USA, 1994. ACM.

[156] C. W. Urquhart. *The Active Stereo Probe: The Design and Implementation of an Active Videometrics System*. Phd thesis, The Turing Institute, The Department of Computing Science, University of Glasgow, june 1997.

[157] Michael Wand, Alexander Berner, Martin Bokeloh, Philipp Jenke, Arno Fleck, Mark Hoffmann, Benjamin Maier, Dirk Staneker, Andreas Schilling, and Hans peter Seidel. Processing and interactive editing of huge point clouds from 3d scanners. *Computers and Graphics*, 32:204–220, 2008.

[158] Wen-Cheng Wang, Feng Wei, and En-Hua Wu. View dependent sequential point trees. *Journal of Computer Science and Technology*, 21(2):181–188, March 2006.

[159] Lee Westover. Interactive volume rendering. In *VVS '89: Proceedings of the 1989 Chapel Hill workshop on Volume visualization*, pages 9–16, New York, NY, USA, 1989. ACM.

[160] Lee Westover. Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, New York, NY, USA, 1990. ACM.

[161] Tim Weyrich, Simon Heinzle, Timo Aila, Daniel B. Fasnacht, Stephan Oetiker, Mario Botsch, Cyril Flaig, Simon Mall, Kaspar Rohrer, Norbert Felber, Hubert Kaeslin, and Markus Gross. A hardware architecture for surface splatting. *ACM Trans. Graph.*, 26(3):90, 2007.

[162] Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, 1983.

[163] Michael Wimmer and Claus Scheiblauer. Instant points : Fast rendering of unprocessed point clouds. *Memory*, pages 129–136, 2006.

[164] J. Wu, Z. Zhang, and L. Kobbelt. Progressive splatting. *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics*, 0:25–142, 2005.

[165] Jiahua Wu, Robin Tillett, Nigel McFarlane, Xiangyang Ju, J. Paul Siebert, and Paddy Schofield. Extracting the three-dimensional shape of live pigs using stereo photogrammetry. *Computers and Electronics in Agriculture*, 44(3):203 – 222, 2004.

[166] Joris Vanden Wyngaerd and Luc Van Gool. Automatic crude patch registration: toward automatic 3d model building. *Comput. Vis. Image Underst.*, 87:8–26, July 2002.

[167] Qi Xia, M.Y. Wang, and Xiaojun Wu. Orthogonal least squares in partition of unity surface reconstruction with radial basis function. In *Geometric Modeling and Imaging–New Trends, 2006*, pages 28 –33, aug. 2006.

[168] Xiaohui Yang, Ju Liu, Jiande Sun, Xinchao Li, Wei Liu, and Yuling Gao. Dibr based view synthesis for free-viewpoint television. In *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON), 2011*, pages 1 –4, may 2011.

[169] S.-E. Yoon, B. Salomon, R. Gayle, and D. Manocha. Quick-vdr: out-of-core view-dependent rendering of gigantic models. *Visualization and Computer Graphics, IEEE Transactions on*, 11(4):369 –382, july-aug. 2005.

[170] Ruo Zhang, Ping S. Tsai, James E. Cryer, and Mubarak Shah. Shape from shading: A survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(8):690–706, August 1999.

[171] Yanci Zhang and Renato Pajarola. Gpu-accelerated transparent point-based rendering. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 178, New York, NY, USA, 2006. ACM.

[172] Yanci Zhang and Renato Pajarola. Deferred blending: Image composition for single-pass point rendering. *Computer Graphics*, 31(2):175–189, April 2007.

[173] C. Lawrence Zitnick, Sing Bing Kang, Matthew Uyttendaele, Simon Winder, and Richard Szeliski. High-quality video view interpolation using a layered representation. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 600–608, New York, NY, USA, 2004. ACM.

[174] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Ewa volume splatting. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 29–36, Washington, DC, USA, 2001. IEEE Computer Society.

[175] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, New York, NY, USA, 2001. ACM.

[176] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Ewa splatting. *IEEE Transactions on Visualization and Computer Graphics*, 2002.

[177] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate splatting, 2004.