SURFACE MODELLING AND RENDERING WITH LINE SEGMENTS

OUYANG XIN

NATIONAL UNIVERSITY OF SINGAPORE

2004

Name: OUYANG XIN
Degree: MASTER OF SCIENCE
Dept: COMPUTER SCIENCE
Thesis Title: SURFACE MODELLING AND RENDERING WITH LINE SEGMENTS

Abstract

Bridging the modelling and rendering gap between the existing triangle and point primitives, we explore the use of line segments as a new primitive to represent and render 3D models. For the task of modelling, we propose two methods to extract hybrid point and line segment models from scanned point clouds, one is (ε, δ) error bounded and one is based on $L^{2,1}$ variational shape approximation. In addition, we also present a method for obtaining pure line segment models from triangle meshes. For the task of rendering, we extend the anti-aliasing theory in texture mapping to anti-aliased line segment rendering, and present an approximation algorithm to render high quality anti-aliased opaque, transparent and textured line segments in 3D models. The anti-aliasing rendering technique is empirically validated by building a software pipeline to render point, line segment as well as hybrid point and line segment models that are acquired using our proposed modelling methods. Experiments show that models comprising of line segments are more effective for modelling and more efficient for high quality rendering as compared to their corresponding pure point models.

Keywords:
I.3.3 [Computer Graphics]: Picture/Image Generation
Anti-aliasing, viewing algorithms
I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling
Curve, surface, solid and object representations
Other keywords: Graphic Primitive, Rendering and modelling system, Point based Graphics, Surface reconstruction

SURFACE MODELLING AND RENDERING WITH LINE SEGMENTS

OUYANG XIN

(B.Comp.(Hons. 1st Class), NUS)

A THESIS SUBMMITED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgments

I would like to thanks my supervisor A/P. Tan Tiow-Seng for his guidance and encouragements in the past two years' time. I would like to thanks Prof. Jurg Nievergelt for his insights and sharing of experiences. I would like to thanks Lim Chi-Wan for working on surface modelling with me and Wong Keen Hon for working on surface rendering with me.

Table of Contents

List of Figures

List of Tables

Summary

Bridging the modelling and rendering gap between the existing triangle and point primitives, we explore the use of line segments as a new primitive to represent and render 3D models. For the task of modelling, we propose two methods to extract hybrid point and line segment models from scanned point clouds, one is (ε, δ) error bounded and one is based on $L^{2,1}$ variational shape approximation. In addition, we also present a method for obtaining pure line segment models from triangle meshes. For the task of rendering, we extend the anti-aliasing theory in texture mapping to anti-aliased line segment rendering, and present an approximation algorithm to render high quality anti-aliased opaque, transparent and textured line segments in 3D models. The anti-aliasing rendering technique is empirically validated by building a software pipeline to render point, line segment as well as hybrid point and line segment models that are acquired using our proposed modelling methods. Experiments show that models comprising of line segments are more effective for modelling and more efficient for high quality rendering as compared to their corresponding pure point models.

Chapter 1.

Introduction

1.1. Surface Primitive Beyond Triangle and Point

1.1.1. The Choice of Surface Primitive

Triangle is the de facto surface primitive of graphics systems ever since the beginning of the computer graphics adventure. Triangle models can effectively represent objects that have large portion of their surfaces being of flat. Examples are the floors of a room, the walls of a building and the boxes piled in a warehouse. However triangle based modelling becomes less adequate and inefficient when used to model objects with highly curved surfaces, such as human faces and ancient Roman statues. Triangle meshes faithful to these objects with high surface details and complex geometry are usually made up from hundreds of thousands to tens of millions small triangles. To fetch such huge amount of triangles from memory to GPU at interactive frame rate requires enormous data bus transferring capacity, exceeding the current graphics card memory bandwidth limit. Rasterizing tiny triangles that are projected onto the screen with footprint sizes less than one pixel does not add much more realism into rendered images, in fact would unnecessarily degrade the performance of the rendering pipeline both at the primitive assembly unit and at the texturing unit.

Point is not fully developed into a surface primitive until in recent four years. Thanks the advance of laser range and optical scanner technology, which makes acquiring mass amount of raw point-wise data accurately from object surfaces becoming feasible and practical. In contrast to triangle meshes, densely sampled point clouds are quite suitable for representing complicated surface geometry and being used when high surface details are to be preserved. By keeping only point data, unnecessary linking edges and covering faces in the triangle meshes that benefit neither modelling nor rendering get removed. This ignorance of explicit connectivity information among vertices helps point clouds achieve representation compactness as well as gain rendering speedups. However it would be a great suffer of efficiency when flat object surface areas that can be fully covered by a single polygon have to be represented with thousands of isolated point samples in a pure point based model. Unlike triangle meshes having triangles spanning among vertices explicitly forming a piecewise linear surface, point cloud by its discrete nature cannot provide a straightforward surface description. This inevitably complicates a number of applications, such as ray tracing which needs to perform intersection tests between rays and surfaces.

Witnessing the emerge of point as a new surface primitive in addition to the longtime predominant – triangle, deliberating on both pros and cons of modelling and rendering with the two primitives for various surfaces, all suggest that depending on applications, appropriate surface primitives should be chosen, adapting to surface geometry, for modelling and rendering different objects' surfaces as well as different regions of one same object surface. So, are the two existing surface primitives – triangle and point, the only two

surface primitives? It makes sense for us to conduct further study in this project to assess the necessities and possibilities to introduce a new surface primitive, a primitive that could potentially complete the role not yet played by triangle and point.

1.1.2. Anti-aliasing High Quality Rendering

Aliasing problem in computer graphics is caused by the disparity of image representation in continuous real world and discrete computer world. Images are represented as continuous signals in the world space, and discrete signals in the screen space. The screen images are created by sampling and quantizing their corresponding world space images. If the sampling frequency is less than the Nyquist frequency, high frequency portions of the continuous signals will masquerade as low frequencies, creating aliasing effects.

To achieve high quality rendering of 3D models, ideally, aliasing effects should be removed completely from final synthesized images. However due to the digital nature of computer systems, aliasing remains and is expected to continue to remain as a problem. Nervelessly, various techniques have since been proposed to mitigate it. Nowadays, high quality anti-aliasing rendering of texture mapped triangle models can be achieved in real time through graphics hardware implementations of both super sampling and anisotropic texture filtering. Recently the Elliptical Weighted Average (EWA) resampling filter – a low pass filter that removes high frequencies from texture images before sampling [Heck89], has been extended for rendering anti-aliased high quality point models [ZPVG01, RPZ02]. Thus, any surface primitive which is equally important in aiding visualizations as triangles and points, should be able to follow the common high quality rendering practice shared by them as well. Introduction of a new surface primitive would

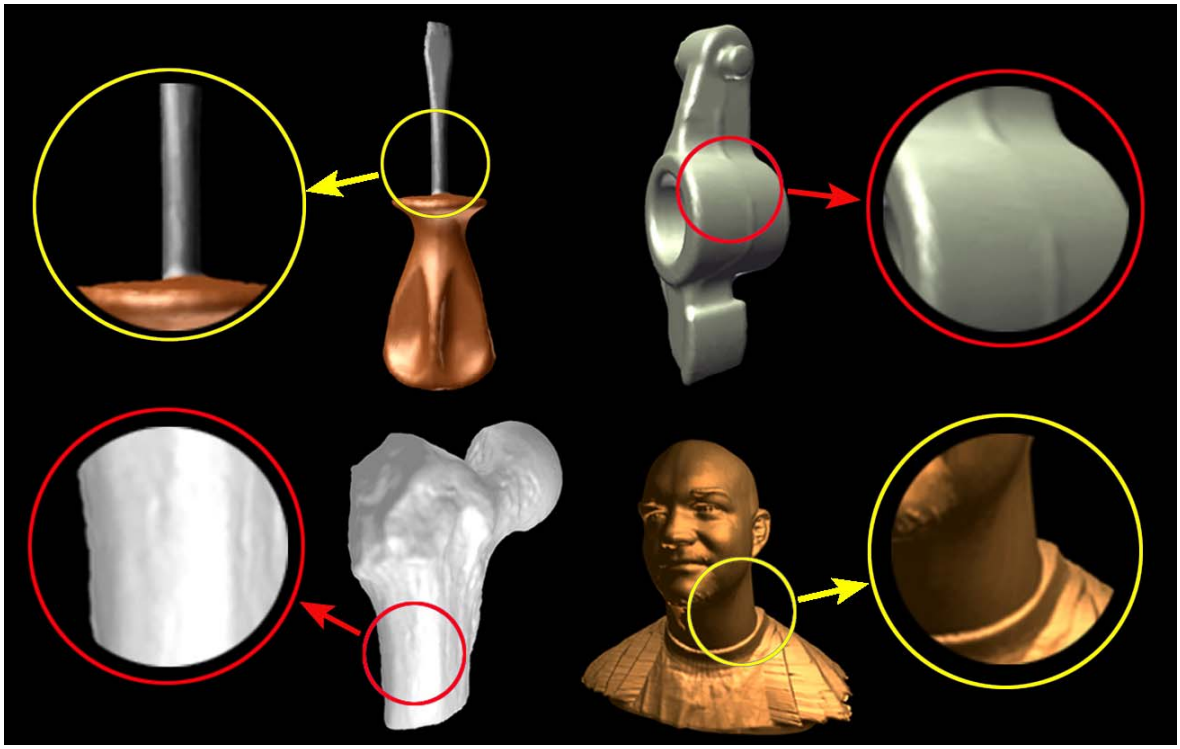

Figure 1: *Line features are observed in both man-built objects, such as the Screw driver and Rocker arm model, as well as irregular shapes, such as the Ball joint and Upper body model. Models in the figure are all rendered using our proposed approximate rendering method.*

require corresponding high quality anti-aliasing rendering techniques be simultaneously developed too.

1.2. Line Segment as A Surface Primitive

1.2.1. Observations

It should not be surprising to observe that lines are prominent features in many types of scenes, in particular of man-built objects; see the Screwdriver and Rocker arm models in Figure 1. Even objects of irregular shape contain features that are best modelled by lines, such as ridges or boundaries of various kinds, see the Ball joint and Upper body models in Figure 1. Line segment is the best primitive candidate to represent these surface regions

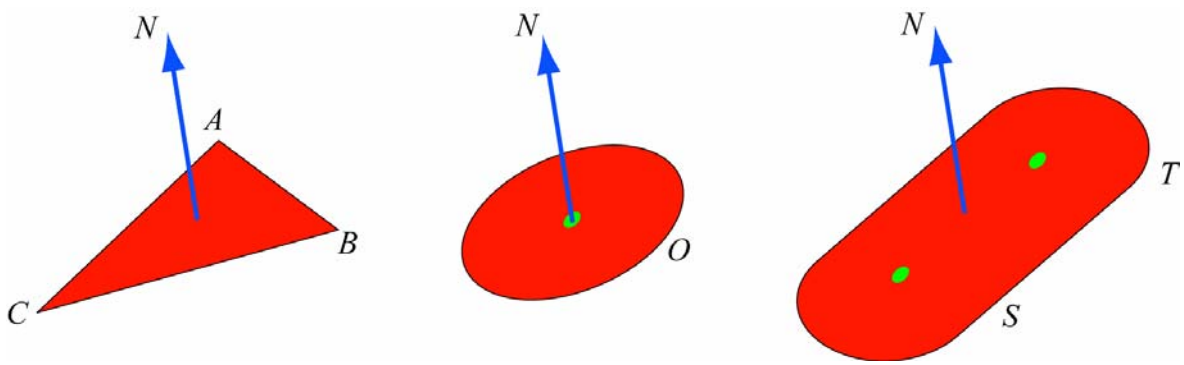

Figure 2: *Triangle primitive, point primitive and line segment primitive.* ***N*** *denotes the normal. The green dot* ***O*** *denotes the point primitive's center. And the two green dots* ***S*** *and* ***T*** *denote the two endpoints of a line segment primitive.*

that are highly stretched in one particular direction. However it is really surprising for us to find out that line segments have only been used as a modelling primitive in very special contexts, such as [DCSD02, LoTa97]. To date, there is no literature work that aims to develop 3D line segments as a full primitive for both surface modelling and rendering.

It is wasteful to ignore lines, a feature that appears so nature, so often. It is wasteful not to make good use of line segments, a possible surface primitive that is as simple as triangle and point (see Figure 2).

It is also interesting to reflect on the fact that the two existing prominent primitives, point and triangle, are simplexes in 0 and in 2 dimensions. What is missing between them is a 1-dimension simplex primitive, line segment is the primitive that rightly fits into this position. Figure 3 shows the Stanford Rabbit modelled by the three primitives respectively. We argue that line segments should be treated as a primitive, rather than as a degenerate special case of triangles or polygons, for the same reason that points are treated as a

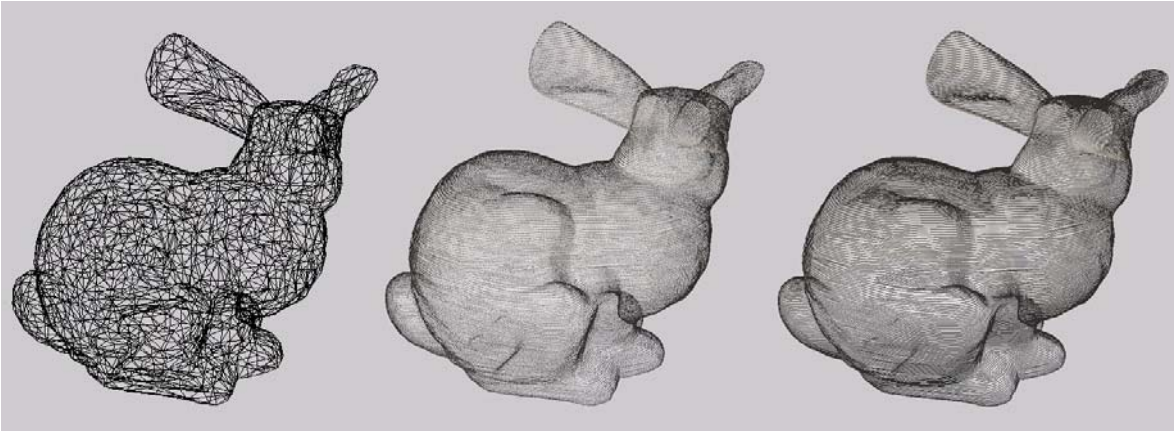

Figure 3: *Stanford Bunny modeled using triangles (left), points (middle) and line segments (right). Here both the line segment model and the point model are generated using our contour plane based line segment extraction algorithm presented in Section 3.3.*

primitive by themselves rather than as tiny triangles or polygons. A line segment is a simpler surface element than a long, thin triangle or rectangle. A lines segment is uniquely defined by its two endpoints and a normal vector that is perpendicular to the line segment itself (see Figure 2).

1.2.2. Motivations

Using line segments as a surface primitive would greatly benefit both the surface modelling process and the surface rendering process. Specifically, there are three advantages brought along.

Effective Representation. While triangles can provide an effective representation for surfaces that are fairly flat and points can provide an effective representation for irregular surfaces that are jagged or highly curved, line segments are capable of effectively modelling surfaces that exhibit strong anisotropy in one particular direction. A typical such surface is the cylindrical surface. The effectiveness of line segment representations implies

geometric faithfulness. The line segment based representation faithfully preserves geometric fidelities of actual surfaces by orienting line segments along surface anisotropic directions.

Compact Representation. Using line segments to model surfaces with regularity along one dimension also means great data savings. Comparing with triangle meshes, line segments achieve representation compactness by keeping only connectivity information between endpoints, discarding all the linking edges in meshes. Comparing with point clouds, very few line segments can often effectively replace quite a number of sequentially aligned points, thus storing far less data.

Efficient Rendering. Line segments can be rendered far more efficiently than sequences of points. Experiments in Section 7.2.3 shows that a maximum of 65.96% speedup can be achieved as compared to rendering pure point models. However, it is unfair to compare the rendering performance of line segments with triangles, as today's graphics pipelines are designed, built and optimized specially for triangle meshes, while high quality anti-aliasing rendering of line segments currently are supported by a software pipeline.

1.3. Objectives

In this project, our objective is to develop line segments into a full primitive for both surface modelling and rendering. For the modelling task, we would study how to obtain line segments from scanned point clouds as well as from triangle meshes. For the rendering task, we would focus on developing high quality anti-aliasing line segment rendering

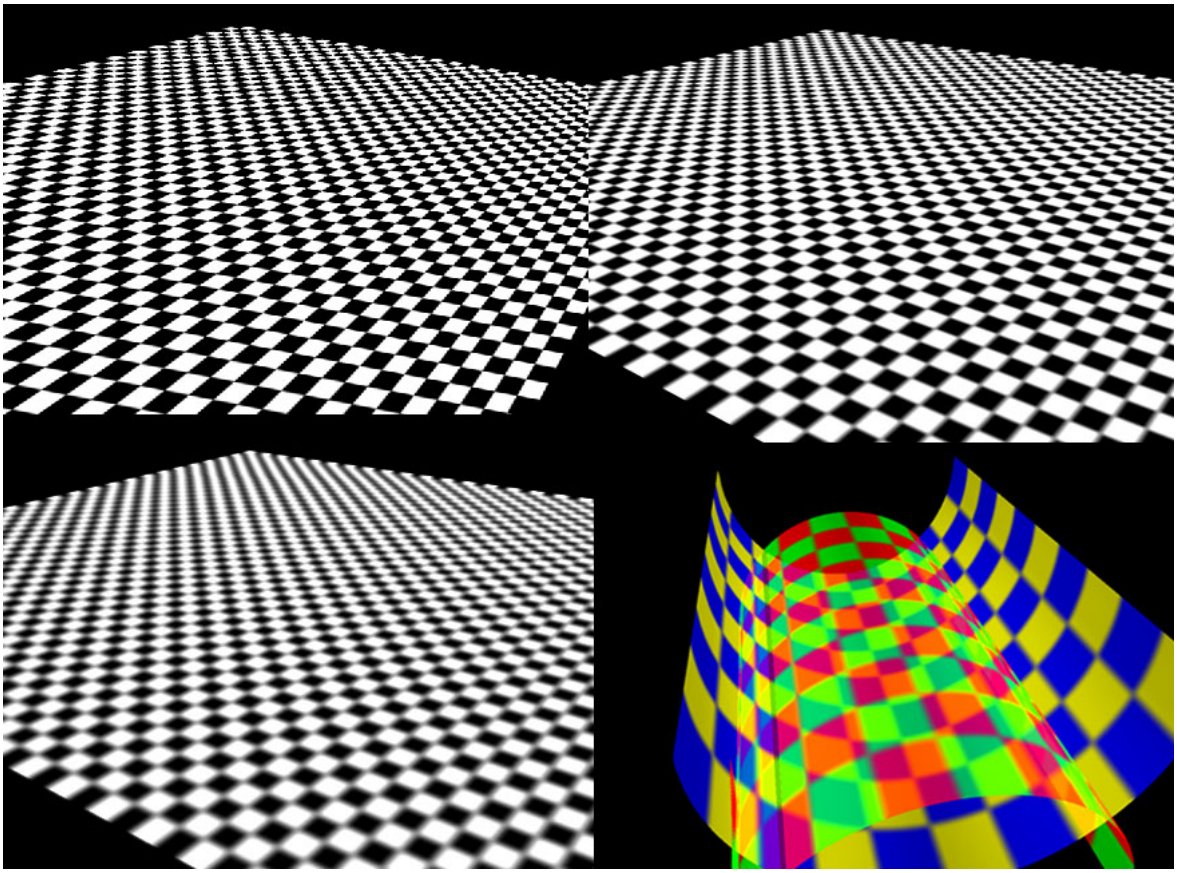

Figure 4: *The checkerboard on the top left is rendered without anti-aliasing. The checkerboard on the top right is rendered from EWA resampled points. Both the opaque checkerboard on the bottom left and the transparent curved checkerboard on the bottom right are rendered from EWA resampled line segments using our approximate rendering method.*

techniques. We need to design corresponding algorithms for both the modelling and the rendering process, study the consequences, and report experiments to show the validity of the approaches we take. We would also describe future directions that promise to make line segment based surface modelling and rendering standard technique in the growing arsenal of computer graphics tools.

1.4. Contributions

Our contribution in this project is the introduction and developing of line segments into a full primitive for both surface modelling and rendering. For the modelling, we propose two methods to extract hybrid point and line segment models from scanned point clouds, one is (ε, δ) error bounded and one is based on $L^{2,1}$ variational shape approximation. We also present a contour plane cutting based method for obtaining pure line segment models from triangle meshes. For the rendering, we extend the anti-aliasing theory in texture mapping [Heck89] to render anti-aliased line segments in 3D models. Though the extension does not result in a closed form solution, we present an approximation method to render high quality anti-aliased opaque, transparent and textured line segments representing 3D models as shown in Figure 4 and Figure 15.

We implement a software graphics pipeline that unifies both high quality point rendering and high quality line segment rendering. Our pipeline can render point models, line segment models as well as hybrid point and line segment models. Our experiments show that the rendered quality of line segment models as well as hybrid point and line segment models are comparable to their corresponding high quality anti-aliased point models. Additionally, there is a significant speed up in the rendering time using hybrid models. This establishes hybrid of points and line segments as a competitive modelling and rendering alternative to pure point models.

1.5. Outlines

This report is organized in such way: In Chapter 2, previous work in various related research fields are extensively surveyed. We discuss in details how these previous work lead us to our problem, how they influence our problem solutions, and in what aspects that our proposed techniques are different from them. In Chapter 3, we discuss how to obtain line segments for surface modelling from both point clouds and triangle meshes. Considerations and problems with each of our proposed methods are also discussed. Mathematical framework for EWA surface rendering procedure appears in Chapter 4. In particular, EWA resampling filters and EWA splatting techniques are examined in full details. In Chapter 5, we show that the mathematical formulation of line segment's EWA resampling filter is of non-closed-form and hence introduce an object space approximation method for rendering line segments. Chapter 6 talks about implementation details of the geometry processing pipeline and the rendering engine. Rendered images, data reduction ratios, performance statistics and image quality comparison results are reported in Chapter 7. Finally, we conclude in Chapter 8 and point out possible future work in Chapter 9.

Chapter 2.

Literature Survey

In this section we give detailed survey of seven research areas that are related to our project work. They are point based surface modelling and rendering, surface anisotropy, modelling and rendering with lines, hybrid surface rendering, texture mapping and anti-aliasing, implicit surface and hardware accelerated rendering.

2.1. Point Based Surface Modelling and Rendering

In 1984, Levoy and Whitted [LeWh85] pointed out that classic modelling primitives, i.e. triangles (or polygons), were less appealing for rendering objects with extremely complex geometry. They suggested decoupling modelling geometry from the rendering process by introducing point as a universal primitive, where each point is associated with a small surface area and a normal for rendering. Using Levoy and Whitted's idea, Rusinkiewicz and Levoy [RuLe00] proposed the QSplat system to render 3D point models. In the QSplat system, points are rendered as screen aligned squares, circles, Gaussian filtered circles, or ellipses. The QSplat system was in fact designed during the course of the Digital Michelangelo Project [LGSF00] to render models consisting of hundreds of millions of scanned points at interactive frame rate.

Hoppe *et al.* [HDD92] described and demonstrated a surface reconstruction algorithm from unorganized point cloud in 1992. Hoppe *et al.*'s algorithm estimates a normal for each

point sample by fitting a best tangent plane into the point's vicinity. With the use of Euclidean Minimum Spanning Trees, points' normals are then adjusted to be consistent. Subsequently a signed distance function is defined based on the estimated normals and best fitted tangent planes, giving isosurfaces. However at the end, surfaces are not rendered directly from points; instead it is a triangle mesh extracted from isosurfaces using the marching cube algorithm is eventually outputted onto the screen.

In 1998, Grossman and Dally [GrDa98] developed the point sample rendering algorithm for real-time rendering of objects with complex geometry. Point samples are acquired from orthographic views without knowing surface topology. Images are synthesized from several views, with each point sample corresponding to one pixel. However each such sampled point pixel contains position and normal data as well. With this approach tears and holes are expected to be visible, Grossman and Dally proposed to fill holes by interpolating from neighbouring pixels.

In 2000, Pfister *et al.* [PZVG00] introduced the surfel, i.e. surface element, paradigm and developed the surfel point rendering pipeline. In this proposal, sampled points are associated with tangent circular disks in object space, thus ellipses on screen after the orthographic projection. However to achieve efficiency and gain hardware support, it is the partially axis aligned bounding boxes of the ellipses that are eventually rendered into graphics buffers. One year later, in 2001, Zwicker *et al.* [ZPVG01, Zwic03] extended Heckbert's EWA filter [Heck89], deriving a rigorous mathematical formulation of screen space EWA resampling filter for irregular point data. Based on this newly derived filter,

they developed another point rendering technique called surface splatting. Now not the bounding rectangles, but the EWA low pass filter filtered ellipses get rendered into buffers. Surface splatting technique is capable of producing high quality anti-aliased images from point samples. Together with it, a set of other point based graphics techniques were simultaneously or subsequently developed in ETH (Federal Institute of Technology), including EWA volume splatting [ZPBG01, ZPBG02, Zwic03], spectral processing of point clouds [PaGr01, Paul03], PointShop3D [ZPKG02], free form shape modelling of point clouds [PKKG03], point cloud simplification [PGK02], feature extraction of point clouds [PKG03] and many other point cloud based applications.

In 2002, Ren *et al.* [RPZ02] proposed a hardware implementation method for the surface splatting technique, called object space EWA splatting. In this proposal, EWA prefiltering is performed by deforming texture mapped surfel polygons. By exploiting the programmability of the vertex shader of the latest graphics hardware, it is Ren *et al.*'s method for the first time achieves high quality anti-aliasing rendering of point clouds at satisfactory real time frame rate.

Wand *et al.* [WFPH01] presented an output-sensitive point modelling algorithm. Points are sampled dynamically and randomly from triangulated object surfaces. Stamminger and Drettakis [StDr01] suggested using a hierarchical sampling scheme which adapts sample densities locally according to the projected sizes in the image to generate point samples. In both [WFPH01] and [StDr01], the emphasis is placed on how to use points to model objects with complex or procedural geometry so as to achieve data saving as well as

rendering efficiency. And both [WFPH01] and [StDr01] are open to either the rendering method in [PZVG00] or the approach taken in [GrDa98] with hole filling.

Kalaiah and Varsheney [KaVa01, KaVa03a] observed that the variations of normals in the vicinities of point samples play an important role in human perception of images' visual quality. Therefore they proposed to store curvature information estimated from each sample point's neighbourhood with the point as well. Such points are named as differential points by them. Differential points are rendered as normal mapped rectangles using graphics hardware. However the total number of different normal maps is infinite. Thus they proposed to quantize the normal maps into 256 different types, and select the normal map that most closely approximates a sample point's neighbourhood's curvatures for rendering. They reported that this rendering approach results better visual quality than splatting based rendering methods.

In the year 2003, besides [KaVa03a], Kalaiah and Varsheney [KaVa03b] also proposed another point modelling and rendering technique which is based on statistical analysis. In this method, point clouds are group into elliptical point clusters via an octree based hierarchical principal component analysis (PCA). By doing so, object models are partitioned into ellipses, it is only the ellipses' information that is to be stored. During rendering, random points are generated on the ellipses' surfaces using trivariate Gaussian random number generator. Each point is to occupy only one screen pixel, thus the random number generator's parameters have to be tuned so as to assure enough points would be

produced. Kalaiah and Varsheney claimed that the statistical point sampling method provides orders of magnitudes saving of data storage.

Alexa *et al.* [ABCF03] used Levin's moving least square method [Levi98, Levi03] to define a smooth manifold surface from a set of points closed to the original surface. With this definition, points can be up sampled and down sampled with bounded approximation errors. Based on this definition, a small polynomial patch embedding local differential geometry can be associated with each point as well. Points are rendered as individual pixels onto the screen. However most likely the number of scanned point samples is insufficient to fully cover the projected screen area of an object. Alexa *et al.* thus suggested sampling more points in points' neighbourhoods from the associated local polynomial patches so as to fill tears and holes. The polynomial patches are locally 2.5D, and thus can be parameterized onto tangent planes of the points. Sampling densities are adjusted by increasing or decreasing the resolutions of parameterization grids that reside on the tangent planes.

In our work, we model object surfaces with a set of unorganized points and line segments that are close and faithful to actual object surfaces as in [RuLe00, HDD92, ZPBG01, ZPBG02, Zwic03]. We do not artificially increase modelling points or line segments dynamically, procedurally or statistically [WFPH01, ABCF03, StDr01, KaVa03b]. We follow the direction of [PZVG00, ZPVG01, RPZ02], developing our high quality anti-aliasing line segment rendering technique. We extend the anti-aliasing theory in texture mapping [Heck89] for rendering anti-aliased line segments contained in 3D models.

Unlike the approach taken in [GrDa98, WFPH01, StDr01, KaVa03b, ABCF03] which renders points as screen pixels and fill tears and holes later, we associate a local surface with each line segment.

2.2. Surface Anisotropy

Before we dive into the details of surface anisotropy, the first question that should be answered is what surface anisotropy is. Surface anisotropy is an intrinsic geometric property of surfaces. It indicates the direction along which a surface region is smooth; it follows the smallest eigenvector and eigenvalue of the curvature tensors of the surface region. However in general, in the context of surface modelling, surface anisotropy refers to surface elements that are intentionally stretched in order to capture some physical phenomena in object models [TART04]. Specifically, in the context of surface meshing or remeshing, anisotropic remeshing refers to align or stretch surface mesh elements with a certain direction field [ACDL03].

The study of surface anisotropy helps to reveal problems and advance visualization techniques pertaining to quite a number of graphics applications. First and the most important, anisotropic surface elements naturally, faithfully and economically model surfaces that exhibit anisotropies. Second, often, elongated mesh elements with large aspect ratios are desired to be generated for turbulent flow simulation problems to improve computation accuracies as well as to better capture transient phenomena [JaSh01, FrAl03]. Third, surprisingly, it is discovered that stretched long and thin triangles are good for linear interpolation [Ripp92]. Finally, it is reported in [BoKo01] that aligning stretched triangles

along sharp features of surfaces help to mitigate aliasing artifacts and improve image quality.

For a given 2D triangle mesh, Bossen and Heckbert [BoHe96] suggested using a 2×2 positive definite, symmetric tensor as anisotropic metric to quantify desired mesh element shapes and sizes. The metric value for a triangle is calculated by taking the average of the metric values evaluated at the triangle's three vertices. Bossen and Heckbert modified the Delaunay triangulation criterion to take this metric into account. The new anisotropic Delaunay triangulate swaps edges to maximize the minimal angle in the normalized space defined by the metric.

Similar to [BoHe96], Li *et al.* [LTU99] also used a 2×2 matrix as the metric tensor to quantify the desired size and shape of a triangle in a 2D triangle mesh near a particular selected point. However Li *et al.* further developed and defined a set of operations on this tensor metric, such as expanding, rotation, summation, subtraction, union and so on. Based on these operations, the distance between two metric measures can be computed, Lipschitz property of anisotropic spacing function can be measured and similar to [BoHe96], the condition of anisotropic Delaunay triangulation can be reformulated to reflect the capturing of anisotropy of meshings. Noting that the 2×2 matrix tensor metric can be geometrically realized as 2D ellipses, Li *et al.* thus designed an advancing front based ellipse biting scheme to pack the ellipses on the meshes. Since the packing of the ellipses respects the underlying control space, the anisotropic mesh can thus be generated from the set of ellipses center vertices by checking anisotropic Delaunay triangulation conditions. In 2000,

Yamakawa and Shimada [YaSh00] extended *Li et al*'s work by proposing an ellipsoidal bubble packing algorithm to generate high quality anisotropic tetrahedral meshes.

In 1997, Garland and Heckbert [GaHe97] proposed a quadric error metric based surface simplification algorithm. Quadric error is a heuristic to characterize the surface geometric error. For a vertex in a triangle mesh, its quadric error is evaluated as a matrix defined as the sum of squared distances to its incident triangles. Later in 1999, Heckber and Garland [HeGa99, Garl99] proved that when triangle areas reduce to zero on a differential surface, the quadric error based mesh simplification algorithm would generate triangles with aspect ratio that is the square root of the ratio of principle curvatures of the curvature tensor at the particular surface point in consideration. The triangles' aspect ratios are optimal in the sense of L^2 geometric error; they in fact capture surface anisotropies of the covered surface regions.

Two latest works in anisotropic meshing and remeshing [ACDL03, CAD04] were presented in 2003 and 2004 SIGGRAPH conferences respectively. Based on the observation that surface anisotropies generally follow in the directions of surface minimum curvatures, Alliez *et al*. [ACDL03] proposed to use surface curvature directions to drive the anisotropic remeshing process. By tracing and intersecting minimum and maximum curvature lines on object surfaces, object surfaces can be remeshed by having quads placed in anisotropic surface regions and triangles placed in spherical surface regions.

Later in 2004, Cohen-Steiner *et al.* [CAD04] proposed a so called $L^{2,1}$ geometric error metric, which is related to the L^2 metric in [HeGa99, Garl99], to measure the variations of surface normals. Cohen-Steiner *et al* proved that the $L^{2,1}$ metric can effectively capture the asymptotic behavior of surface elements. Given a target number of surface elements, the $L^{2,1}$ metric based surface remeshing algorithm in [CAD04] would produce meshes that best approximate object surfaces and best capture surface anisotropies. Specifically, the remeshing algorithm remeshes object surfaces by first iteratively clustering given mesh triangles in the manner of minimizing the total $L^{2,1}$ metric error and subsequently extracts polygons and triangles from the clusters.

Labelle and Shewchuk [LaSh03] pointed out that the anisotropic Delaunay triangulation algorithms in [BoHe96, LTU99] could neither guarantee the termination of edge flipping nor assure to produce a unique anisotropic mesh configuration. In [LaSh03], Labelle and Shewchuk generalized the multiplicatively weighted Voronoi diagrams into the definition of anisotropic Voronoi diagrams, the kind of Voronoi diagrams specially catering for the generation of long and skin triangles. Unfortunately, the dual of anisotropic Voronoi diagrams do not necessarily correspond to anisotropic Delaunay triangulations. Labelle and Shewchuk proved that in 2D, only under the circumstances in which the sites could see all their entire Voronoi cells, would the anisotropic Voronoi diagrams be guaranteed to be dualizeable.

One of our objectives, modelling object surfaces with line segments, is by no means anything else, but to mine surface curvature information so as to capture as much surface

anisotropies as possible, and properly orientate and place line segment in identified anisotropic surface regions. Our shape approximation based line segment extraction algorithm in Section 3.2 thus makes use of the $L^{2,1}$ metric [CAD04] to hierarchically cluster sampled surface points into clusters that would satisfy a prescribed maximum normal deviation tolerance. Due to the proved effectiveness of the $L^{2,1}$ metric in capturing surface anisotropies, the clusters obtained in our algorithm would thus be able to not only satisfy a normal deviation tolerance but also well capture surface anisotropies.

2.3. Modelling and Rendering With Lines

Directly using surface lines to model and render object surfaces attracts a lot of attentions in the computer graphics field. Sousa and Prusinkiewicz [SoPr03] presented a method for rendering 3D models in the line-drawing style. They first extract feature lines from object surfaces and then use a non-photorealistic renderer to draw the lines on screen. The feature lines can be classified into five different types. They are silhouette, boundary, crease, cap and pit edges. During rendering, these curved lines are segmented into small line segments and smoothed. Chains of line segments with varying path, length, thickness, gaps and closures are drawn to create perceptually convincing images.

Rossl and Kobbelt [RoKo00] presented an interactive line art drawing system for illustrating 3D models. Normal and curvatures are computed for every vertex of a triangle mesh. The mesh model is then projected into buffers, with normal and curvature linearly interpolated for every rasterized pixel. The enhanced 2D view of the object model is subsequently segmented into regions based on the analysis of curvature information, and

streamlines are traced through pixels. User can sketch some references lines to aid the system to deduce well oriented and aligned streamlines. By exploiting the special structure of the streamlines, shadings and hatches can be easily added to create visual pleasing images.

In [GIHL00], Girshick *et al.* argued that principle curvature lines should be used for 3D surface drawings. This is backed by psychological studies which suggest that lines in principle curvature directions can communicate shapes better than lines in other directions. Girshick *et al.* used short line segments to denote principle curvature directions. These short line segments are then traced, trained together and smoothed. Subsequently, by employing standard non-photorealistic stroke drawing techniques, images of 3D models get emerged on the screen.

In 2000, McNamara *et al.* [MMJ00] described a high quality anti-aliasing line segment rendering algorithm. In their method, a surrounding rectangle is associated with each line segment. Distance functions are defined on all the four rectangle edges. Upon rasterizing a line segment, signed distances from any fragment within its associated rectangle to the four rectangle edges are calculated and combined. The result value is used as an index to access a pre-computed intensity table. It is the convolution results between a filter and a prototypical line segment at various distances that are stored in the intensity table. To anti-aliase line segments' endpoints, two extra anti-aliased OpenGL points are added. McNamara *et al.* reported that this algorithm generates smooth anti-aliased line segments.

Many papers treat the 2D problem of anti-aliasing line segments on screen. A good reference on the desirable characteristics for an anti-aliased 2D line segment is described in [Nels96].

In our work, we develop 3D line segments into a full primitive for both surface modelling and rendering. We extract line segments directly from input point clouds or triangle meshes for surface modelling and render high quality photorealistic line segments. While in [SoPr03, RoKo00], curved feature lines or traced streamlines are only used as an intermediate modelling primitive to describe the shape of surfaces. And eventually these lines have to be broken into short line segments and rendered as non-photorealistic strokes. As [GIHL00], our line segment extraction algorithms also prefer line segments that orient along minimum principle curvature directions.

Both [MMJ00] and our rendering task share the same goal, to render high quality anti-aliased line segments; however there are three distinct differences. First, in our work, each line segment is associated with a piece of local surface. The size of a line segment is determined by the surface area the line segment represents. While in [MMJ00], the size of a line segment is the width of the line segment to be rasterized onto the screen. Second, in [MMJ00], the intensities to be assigned to rasterized line segments have to be computed fragment by fragment. In our line segment rendering procedure, a pre-computed intensity texture is mapped once for each line segment. Third, unlike [MMJ00] which anti-aliases the endpoints of line segments with OpenGL anti-aliased points, in our work, the pre-computed weight texture anti-aliases the whole line segment's rasterized footprint.

2.4. Hybrid Surface Rendering

Chen and Nguyen [ChNg01] introduced a hybrid point and polygon rendering system called POP. In POP, a hierarchical tree structure, same as the one used in QSplat [RuLe00], is constructed with triangles as leaf nodes and points as non-leaf nodes. In POP, points are represented as bounding spheres as in QSplat. During rendering, depending on screen contributions, point non-leaf nodes are used for displaying when objects are far away, and triangle leaf nodes are selected when objects are nearby. With the use of points, rendering speed gets accelerated and with the use of triangles, surface details are preserved.

Dey and Hudson [DeHu02] proposed another hybrid point and polygon rendering system called PMR. In PMR, an octree based spatial hierarchy is used. For each leaf node of the hierarchy, several versions of points and triangles of different level of details are stored. A metric measure reflecting surface local feature size is used to decimate points from highest level of details to the lowest. Triangles are meshed from point set using Delaunay triangulation. During rendering, for each leaf node, depending on projected pixel size, an appropriate version of points and triangles is selected. For each individual point selected, again depending on projected pixel size, either the point is rendered as a pixel or the triangle umbrella anchored at this point is selected for rendering.

In our hybrid surface rendering solution, a point is rendered as an anti-aliased splat suggested by Zwicker *et al.* [ZPVG01], while in POP points are rendered as spheres, and in PMR points are rendered as screen pixels. Using our line segment rendering method, points and line segments can be seamlessly hybridized together, producing high quality

anti-aliased images. However it is noticed that the hybrid rendering approach taken by both POP and PMR could produce aliasing artifacts. Both POP and PMR build hierarchical LOD structures to organize the primitives. The LOD supported hybrid renderings in both systems put their emphasis on performance. This is different from our approach. We handle a set of unorganized points and line segments.

2.5. Texture Mapping and Anti-aliasing Techniques

Here, we survey several prefiltering based anti-aliasing techniques. Mip-map [Will83] is the most widely used such method. It gains full hardware support. For Mip-map, a texture pyramid is pre-computed storing several versions of an input texture image. Resolutions of the textures in the pyramid decrease from bottom to top. Two such textures are chosen for each backward projected screen pixel, one is of higher resolution and one lower. Isotropic filters are then used to sample the textures, and the linearly interpolated result of the sampled texture values is assigned as the pixel's color intensity. Due to the use of isotropic filtering, Mip-map performs poor for pixels that are backward anisotropically projected – the anisotropic texture filtering problem.

The NIL-maps method [FoFi88] was then proposed to remedy this problem. For NIL-maps, a set of basis functions are used to substitute the space variant filter. These basis functions are convoluted with the input texture image, and resulted convoluted textures are used to build the pyramid structures as in Mip-map. The biggest problem with the NIL-maps approach is that a large number of basis functions are needed to approximate an arbitrary filter. Heckbert's EWA [Heck89] resampling filter proposal attacks the anisotropic texture

filtering problem directly by sampling texels that lie within the filter's footprint. Although the footprint of the EWA resampling filter is known to be anisotropic, be elliptical, the computation required knowing the footprint's size and orientation is still too expensive to afford.

To achieve efficiency of anisotropic texture filtering, three methods that employ a set of isotropic filters to replace the anisotropic filter have since been proposed [SKS96, MFPJ99, MPFJ99, CDK04]. In the footprint assembly method [SKS96], pixels are treated as 1 pixel wide rectangles, and isotropic filters are sampled along the major axis of the backward projected parallelograms. In the Feline method [MFPJ99, MPFJ99], pixels are treated as circles with radii equaling to 1 pixel, and isotropic filters are sampled along the major axis of the backward projected ellipses. Mccormack *et al.* reported that images anti-aliased using the Feline method achieve high visual quality comparable to those using EWA resampling filter. Chen *et al.* [CDK04] pointed that both the footprint assembly method and Feline method could suffer oversampling in lower resolution texture and undersampling in higher resolution texture, of the texture pyramid. They thus suggest adjusting the sampling rate at two different levels of the texture pyramid differently. Reduce the number of samples used in the lower resolution texture and spread the samples around within the footprint area. Similarly, increase the number of samples used in the higher resolution texture and spread them around as well.

In our work, we do not anti-aliase any texture image. Our objective is to anti-aliase surface primitives, in particular line segments. Scanned points, as well as the line segments that are

extracted from the scanned points scatter all over object surfaces. They are samples taken from a continuous object surface. Thus anti-aliasing these surface primitives during rendering is a task as challenging as anti-aliasing texture images. We make the same choice as Zwicker *et al.* [ZPBG01, ZPBG02, Zwic03] for rendering anti-aliased points, a EWA resampling filter for line segments is formulated and developed in our work to provide anti-aliase for surface line segments. We note that EWA resampling filter is well recognized as the best efficient software based anti-aliasing solution [MPFJ99].

2.6. Implicit Surface

McCormack and Sherstyuk [McSh98] expanded the set of skeletal primitives that can be used to construct convolution surfaces. In [McSh98], points, line segments, polygons, arcs and planes all can be used to model and render convolution surfaces. McCormack and Sherstyuk also presented an analytic method based ray tracing algorithm to visualize the convolution surfaces. In [Sher99, JiTa02], discussions on the choice of kernel function used in convolution surfaces are presented. The kernel functions include Gaussian, inverse linear, inverse squared, Cauchy and quartic functions.

Levin proposed a point set surface definition using Moving Least Square (MLS) approximation method [Levi98, Levi03], so called MLS surface. The MLS surface is a set of stationary points that would be mapped to their selves by a weighted minimum least square function. The MLS surface is defined procedurally using the MLS projection operator. The MLS surface is define locally, MLS projection would be applied only using a local reference domain. The MLS surface is proved to be C^{∞} smooth [Levi03]. To

visualize MLS surface, Adamson and Alexa [AdAl03] developed a ray tracing method. Using similar projection procedure like MLS surface, Amenta and Yong [AmYo04] gave a surface definition for points with known normals. The MLS surface definition is currently the most rigorous and most widely accepted point set surface definition proposal.

In our work, we need to know what kernel function should be used for the line segment's EWA resampling filter. For points, it is the Gaussian function that is being used. We propose using line segment field function with Gaussian kernel as the kernel for line segments' EWA resampling filter, since it is proved in [JiTa02] that the line Gaussian function is of closed form. One challenging question posted is how to define continuous object surface for line segment models, or mixed point and line segment models as like either the convolution surface in [McSh98] or MLS surface in [Levi98, Levi03]. In this project we do not address this problem. It is left as part of our future work.

2.7. Hardware Accelerated Rendering

The high quality point rendering algorithm initially developed in [ZPVG01] is software based. Since then, several attempts have been made to port this software pipeline into graphics hardware. Ren *et al.*'s work [RPZ02] is the first such attempt. Ren *et al.* programmed in the vertex shader, and precisely compute the rotation matrix and scaling matrix that are defined by the point's object space EWA resampling filter. The drawback with their approach is that, for every Gaussian weight texture mapped rectangle, the same piece of vertex program has to be executed once for each vertex. That is four times for every sampled surface point, creating a lot of computation redundancies.

Ren *et al.*'s hardware implementation method is an object space based approach [RPZ02]. In the following years, there emerge three other hardware implementation proposals [CoHe02, BoKo03, ZRBD04]. All are screen space based approaches, all make use of the NV_point_spirte feature provided by the latest graphics cards and all are implemented in the pixel shader. In all the three approaches [CoHe02, BoKo03, ZRBD04], point sprites that bound the screen projections of sampled surface points are rasterized. However the three methods rasterize the actual elliptical point projections differently and assign the Gaussian weight texture to the rasterized pixels differently. Coconu and Hege [CoHe02] used the multiplication between a circular Gaussian and a liner approximation of Elliptical Gaussian to approximate the point's screen space EWA resampling filter. It was reported that their approximation method yields satisfactory results in practice.

In [BoKo03], Botsch and Kobbelt used circular discs to represent sampled object surface points, and developed a simple yet efficient approximation method to render these discs into elliptical point splats on screen. During the rasterization of the point sprites, z depth values of pixels within the point sprites are computed via bilinear interpolation. And subsequently, the pixels with their corresponding object space positions known are checked to see whether locating inside or outside the elliptical point splats. In Botsch and Kobbelt's method, it is simply a Gaussian weight texture mapped elliptical point splat that is finally rasterized on the screen. This point splat is not filtered by any low pass filter. It is questionable whether high quality anti-aliasing rendering can in fact be achieved in [BoKo03].

Zwicker *et al.* [ZRBD04] observed that Heckbert's EWA resampling theory [Heck89] leads to series artifacts under extreme perspective projections, due the affine transformation assumption and local affine approximation assumption used. In fact, the EWA splatting technique is only perspective correct at sampled points, and wrong with regarding to the shape of the splats. Based on the fact that conics are closed under perspective projections, Zwicker *et al.* [ZRBD04] thus derived a new formulation of the EWA resampling filter that gives perspective correct splat shapes. Upon rasterization, the multiplication results between the locations of pixels within point sprites and 2×2 square matrices derived from the new EWA resampling filter are computed and compared with a predefined threshold to check whether the pixels lie inside the resampling filters or not. However, in [ZRBD04], the centers of the EWA resampling filters usually do not coincide with the actual screen projection locations of sampled surface points.

From the above discussions, it is not difficult to see that none of the existing hardware methods implements the EWA splatting technique perfectly. [CoHe02, BoKo03] are approximation methods. The method in [ZRBD04] would deviate the centers of resampling filters a lot away from actual screen projection locations of sampled points. Although the implementation in [RPZ02] follows the EWA resampling theory exactly, it is so far the slowest method reported achieving merely 2M-3M splats per second. All the methods strive to sit the EWA splatting technique on top of the existing triangle oriented hardware rendering pipeline. We think, ideally, only when dedicated hardware is developed for the EWA splatting technique, would all the various hardware constraints and

restrictions be removed completely and make practical implementations simple and straightforward. Thus in our work, we have only developed a software pipeline for rendering line segments, we leave the hardware implementation development on existing graphics hardware as part of our future work.

Chapter 3.

Surface Modelling with Line Segments

Today, point cloud models can be obtained efficiently and accurately using laser range and optical scanners. And triangle meshes can be obtained either via triangulating scanned point clouds or through the hands of model designers and computer artists with the aid of modelling software tools, such as Maya and 3D Studio Max. However line segment based models so far can only be artificially generated for certain special types of objects, for example, a set of equally spaced and parallel line segments that are parameterized on the surface of an elliptical cylinder. To develop line segments as a full primitive for both surface modelling and rendering, the question of how to model a given object surface with line segments has to be answered in the first place.

In this section, we study how to model object surfaces with line segments. We take scanned point clouds as well as triangle meshes as input raw surface data. We design line segment extraction algorithms from different perspectives by formulating the surface line segment modelling problem differently and solving the problem with different emphases. Our line segment extraction algorithms extract hybrid point and line segment models from point clouds and pure line segment models from triangle meshes. We also talk about the error measures used in the line segment extraction algorithms, discuss how the scanning of point clouds in actual practices affects the line segment extraction algorithms and explain the reasons why certain other seeming possible approaches are not implemented by us.

This section is organized in the following way: two methods for extracting surface line segments from point clouds are presented in Section 3.1 and 3.2. Section 3.1 gives a Euclidean distance and normal deviation error bounded surface line segment extraction algorithm. While in Section 3.2, a shape approximation based surface line segment extraction algorithm is proposed. In Section 3.3, we discuss how to cut contour planes through triangle meshes to obtain pure line segment based models.

3.1. (ε,δ) Error Bounded Line Segment Extraction

In this subsection, we introduce a Euclidean distance and normal deviation error bounded surface line segment extraction algorithm, called (ε,δ) error bounded line segment extraction algorithm. We further organize our discussions into five parts. In Section 3.1.1, we formulate the problem. In Section 3.1.2, we define what ε and δ are. We then point out that our problem is NP-hard in Section 3.1.3. In Section 3.1.4, we present the details of our algorithm. Lastly, we discuss the problems associated with our proposed algorithm in Section 3.1.5.

Before we proceed to the details of our discussions, it should be pointed out first that both our error bounded line segment extraction algorithm and the shape approximation based line segment extraction algorithm presented in next subsection target to obtain hybrid point and line segment models from scanned point clouds. As we know, due to the complexity and diversity of surface geometry of both natural and man-built objects, most likely, a single choice of surface primitive among point, line segment and triangle for modelling

would not give the overall best surface representation. To achieve the goals of representation compactness and representation effectiveness, surface primitives should be mixed in use. Since objects exhibit anisotropies along certain directions only in parts of their surfaces, line segments should be used together with other surface primitives to give hybrid surface representations that adapt to surface geometry. As we develop our high quality anti-aliasing line segment rendering technique based on the EWA resampling theory, thus the hybridization of points and line segments in use for surface modelling is our best primitive combination choice. In this project we focus on the hybrid point and line segment surface modelling and rendering problem.

3.1.1. Problem Formulation

The past decade has seen the emergence of a vast literature on surface reconstruction from a point cloud [ACDL00, AGJ00]. The standard assumption is that the given point cloud is sampled from a uniquely defined object surface which is to be reconstructed. Appropriate assumptions about surface properties and sampling density have led to reconstruction algorithms that respect surface topology and error bounds. Here we look at the problem from a less stringent point of view. Rather than attempting to reconstruct a uniquely defined surface, we ask first, what surfaces are compatible with the given point cloud within some margin of error, and second, how to construct or approximate an arbitrary surface that respects the stated error bounds.

In our study, the unknown surface is to be represented by a set of mixed points and line segments. A line segment can be viewed as the compressed form of a sequence of well aligned sampled points, given that the sampled points do to locate too far away from the

line segment and the normals of the sampled points are almost equal to the normal of the line segment. Thus here, we formulate the surface line segment modelling problem as searching for a set of surface points and line segments that are close to original sampled point cloud with both Euclidean distance and normal deviation errors bounded within prescribed tolerances.

3.1.2. (ε,δ) Errors

Let $\mathcal{P}=\{\boldsymbol{p}_1,\boldsymbol{p}_2,\boldsymbol{p}_3..,\boldsymbol{p}_n\}$ be a given sampled point cloud. We use $\boldsymbol{n}_i$ to denote the normal vector associated with each sampled point $\boldsymbol{p}_i$ and $\mathcal{N}_d(\boldsymbol{p}_i)$ to denote the set of local delaunay neighbours of $\boldsymbol{p}_i$. The normal vectors cannot be acquired through scanning. We derive the normals using Hoppe's method [HDD92]. Refer to Section 6.1.1 for details.

3.1.2.1. (ε,δ) Error Definitions and (ε,δ)- Line Segment

We define a (ε,δ)-*line segment* $L(\boldsymbol{p}_i)$ anchored at a sampled point $\boldsymbol{p}_i\in\mathcal{P}$ as a line segment lying on the plane P_i that passes through $\boldsymbol{p}_i$ and perpendicular to the $\boldsymbol{p}_i$'s normal vector $\boldsymbol{n}_i$. Let $\mathcal{Q}=\{\boldsymbol{q}_1,\boldsymbol{q}_2,\boldsymbol{q}_3..,\boldsymbol{q}_m\}$ be the set of points replaced by a line segment $L(\boldsymbol{p}_i)$. Then all the points in $\mathcal{Q}$ are sufficiently near to $L(\boldsymbol{p}_i)$ and have nearly equal normals (see figure 5). Specifically, let $\mathcal{Q}'=\{\boldsymbol{q}_1',\boldsymbol{q}_2',\boldsymbol{q}_3'..,\boldsymbol{q}_m'\}$ denote the set of the points on $L(\boldsymbol{p}_i)$ that are nearest to the points in $\mathcal{Q}$, with each $\boldsymbol{q}_j'$ corresponding to a $\boldsymbol{q}_j$. Then every $\boldsymbol{q}_j\in\mathcal{Q}$ satisfies an Euclidean distance error tolerance ε such that $\left\|\boldsymbol{q}_j-\boldsymbol{q}_j'\right\|\le\varepsilon$. And the deviation between the normal vector $\boldsymbol{n}_i$ of point $\boldsymbol{p}_i$ and $\boldsymbol{n}_j$ of point $\boldsymbol{q}_j$ is also bounded by a normal

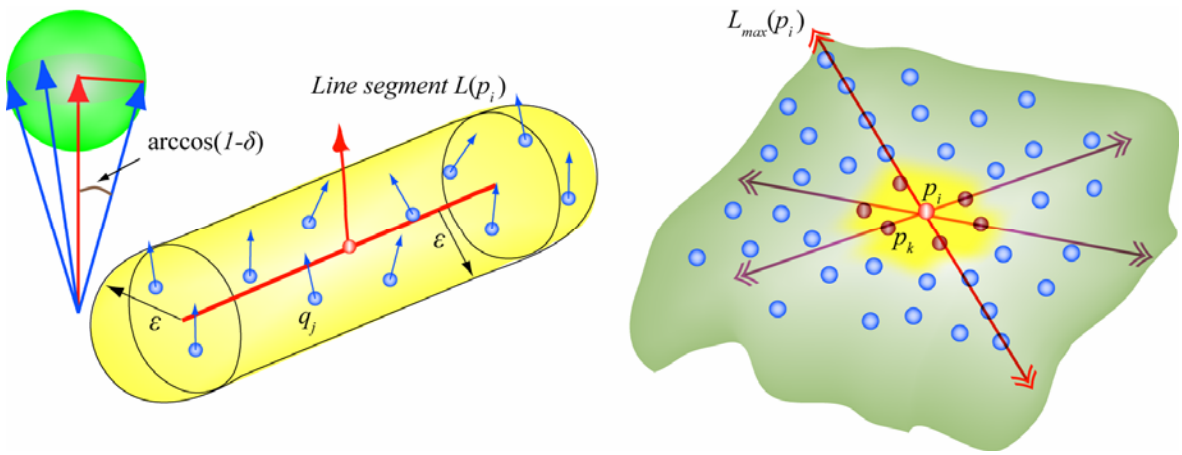

Figure 5: *The left half shows how the* (ε,δ) *error is defined for the line segment* $L(\boldsymbol{p}_i)$ *that passes through the point* $\boldsymbol{p}_i \in \mathcal{P}$. $\boldsymbol{p}_i$ *is represented with a red ball and* $\boldsymbol{q}_i \in Q$ *are colored blue. The right half shows how* $L_{\boldsymbol{max}}(\boldsymbol{p}_i)$ *is computed.* $\boldsymbol{p}_i$*'s neighborhood is colored yellow, and crimson balls are used to denote neighbouring points* $\boldsymbol{p}_k \in \mathcal{N}_d(\boldsymbol{p}_i)$. $L_{\boldsymbol{max}}(\boldsymbol{p}_i)$ *is given along the red line segment with double arrowheads. And other two pink line segments denote other shorter* (ε,δ)*-line segment*.

error tolerance δ such that $1-\boldsymbol{n}_i \cdot \boldsymbol{n}_j \le \delta$. The maximal (ε,δ)*-line segment* $L_{max}(\boldsymbol{p}_i)$ is the (ε,δ)*-line segment* anchored at $\boldsymbol{p}_i$ that replaces the largest number of sampled points belonging to $\mathcal{P}$.

3.1.2.2. Discussions on Error Measures

There exist various kinds of error measures used to quantify the differences between two point sampled object surfaces. Hausdorff distance h_d is defined as the maximum of the set of minimum Euclidean distances measured from a chosen point in one point cloud to all points in another point cloud. The Hausdorff distance h_d is the most straightforward error measure for comparing two point clouds. In [PGK02], the continuous surface defined by one point cloud is constructed using the MLS surface definition proposal. Distances from

points in another point cloud to the constructed continuous surfaces are point-wise computed. The maximum such distance Δ_{max} and the average Δ_{avg} serve as the error measures of surface differences. Recently, Wu and Kobbelt [WuKo04] presented a method to approximate sampled point clouds with circular or elliptical object space point splats. Their method guarantees the maximum distance between sampled points and substitute point splats is bounded by a prescribed Euclidean distance error ε_d. Apart from that their method also ensures the set of point splats would form a hole-free surface.

Throughout the surface modelling and rendering process, there are in general three approaches that can be taken to make a fair comparison between two sampled object surfaces. The first approach is to compare the two sets of sampled surface primitives directly. The Hausdorff distance h_d error measure is such an approach. Our (ε, δ) error measure also falls into this category. The second approach is to compare the two constructed object surfaces. It does not mater whether the object surfaces are constructed to be continuous or are just consisting of discrete splats. However both parties in the comparison should be constructed using the same surface construction algorithm. Since the MLS projection operator projects sampled surface points exactly onto the MLS surface, the Δ_{max} and Δ_{avg} error measures used in [PGK02] fall into this second approach category. The third approach is to compare the two rendered object surface images. This happens at the end of the surface rendering process. In our experiments, we compare the rendering quality of pure line segment models and hybrid point and line segment models with pure point models.

We disagree that the ε_d error measure used in [WuKo04] would make a fair comparison between two object surfaces. As one party of the comparison is a point could, and the other one is a reconstructed object surface comprising point splats. This explains the reason why we opt not to implement a similar method in [WuKo04] to extract surface line segments.

3.1.3. NP Hard Problem

Given the prescribed (ε, δ) error tolerances, one goal reasonably to set is to search for a hybrid point and line segment surface representation that maximize the number of sampled points replaced by line segments. Unfortunately this problem turns out to be NP-hard. In fact it can be mapped into a minimum dominating set problem. This issue has also been noticed in [CAD04, WuKo04]. Therefore we propose a greedy algorithm solution.

3.1.4. The Line Segment Extraction Algorithm

We compute for each point $\boldsymbol{p}_i \in \mathcal{P}$ its maximal (ε, δ)-*line segment* $L_{max}(\boldsymbol{p}_i)$. For practical reason, this computation starts with forming (ε, δ)-*line segment* $L_k(\boldsymbol{p}_i)$ that passes close to a neighbouring point $\boldsymbol{p}_k \in \mathcal{N}_d(\boldsymbol{p}_i)$ having location $\boldsymbol{p}_k$ satisfying $\|\boldsymbol{p}_k - \boldsymbol{p}_i\| \le \varepsilon$ and normal $\boldsymbol{n}_k$ satisfying $1 - \boldsymbol{n}_i \cdot \boldsymbol{n}_k \le \delta$. Then it progressively extends each such $L_k(\boldsymbol{p}_i)$ whenever possible to further represent neighbours of points already included in $L_k(\boldsymbol{p}_i)$. Let us denote the set of candidate points to be replaced by $L_k(\boldsymbol{p}_i)$ using $\mathcal{S}(L_k(\boldsymbol{p}_i))$. Points in $\mathcal{S}(L_k(\boldsymbol{p}_i))$ are checked with the (ε, δ) error tolerances in increasing order of their minimum distances to $L_k(\boldsymbol{p}_i)$. The candidate point $\boldsymbol{p}_e \in \mathcal{S}(L_k(\boldsymbol{p}_i))$ that is nearest to $L_k(\boldsymbol{p}_i)$ is projected onto $L_k(\boldsymbol{p}_i)$. If $\boldsymbol{p}_e$ is bounded by the (ε, δ) error tolerances,

one of the endpoints of $L_k(\boldsymbol{p}_i)$ would be further extended to the projected position of $\boldsymbol{p}_e$, otherwise one side of $L_k(\boldsymbol{p}_i)$ stops growing further. Among all the possible $L_k(\boldsymbol{p}_i)$ computed for $\boldsymbol{p}_i$, the one representing the largest set of points is chosen as the maximal (ε,δ)-*line segment* $L_{max}(\boldsymbol{p}_i)$ (see Figure 5).

The set of maximal (ε,δ)-*line segment* $L_{max}(\boldsymbol{p}_i)$ are then sorted in a heap H in the decreasing order of the number of points they represent. The $L_{max}(\boldsymbol{p}_i)$ representing the largest number of points is the first $L_{max}(\boldsymbol{p}_i)$ removed from H and outputted as a line segment. Points in $\mathcal{Q}$ being replaced are then removed from subsequent processes and the priorities of remaining $L_{max}(\boldsymbol{p}_i)$ in H are also updated reflecting the removal of $\mathcal{Q}$ from $\mathcal{P}$. The maximal (ε,δ)-*line segment* $L_{max}(\boldsymbol{p}_i)$ that represent too few points (3 or less in our experiments) are rejected and are not converted into line segments. Any point left over after line segments have been extracted remains a surface primitive of type "point". See Figure 23 and 24 for the hybrid point and line segment models obtained.

3.1.5. Observations, Problems and Discussions

Although our (ε,δ) error bounded line segment extraction algorithm produces hybrid point and line segment representations that are close and faithful to scanned point clouds, it does not guarantee the rendered surface images would not have holes. However in practice, this is not a serious problem. We find that by simply increasing the radius of the reconstruction filter ε amount, holes will become invisible in almost all rendered images.

We observe that in most scanned models, sampled points are arranged quite regularly on object surfaces. In some flat surface regions, points are almost being spaced at equal distance away from their neighbours. Our (ε, δ) error bounded line segment extraction algorithm takes great advantage of this regularity. In fact the success of the algorithm relies on it too much. If some random noises are present in the point cloud, the error bounded algorithm could perform quite badly.

Another problem with our (ε, δ) error bounded algorithm is that it is too sensitive to the point cloud's scanning direction. We observe that a large number of line segments extracted orientate in the direction along which sampled surface points were actually scanned. This is especially noticeable when the Euclidean distance error tolerance ε is set very small. In some of these situations, the line segments found are actually the "virtual lines" that are beamed from the scanner onto the object surface.

3.2. Shape Approximation Based Line Segment Extraction

Using our (ε, δ) error bounded line segment extraction algorithm, extracted points and line segments are placed very close to the scanned point cloud bounded by tight ε and δ error tolerances. It is the scanned point cloud that the constructed hybrid surface representation is faithful to, not necessarily the actual surface geometry. Furthermore the (ε, δ) error bounded line segment extraction algorithm extracts the maximal $(\varepsilon, \delta)\text{-}\textit{line segment}\ \ L_{max}(\boldsymbol{p}_i)$ that is only the longest line segment local to each individual sampled point $\boldsymbol{p}_i$. The line segments do not necessarily follow the surface anisotropic direction. The line segments may also intersect with each other, creating overlapping. We

observe that seldom an isolated line segment is used to model object surfaces alone. In general, it would be a bunch of nearly parallel line segments that are employed for modelling anisotropic surface regions. Thus fitting line segments onto object surfaces one by one is an inefficient line segment extraction process.

In this subsection we propose another line segment extraction algorithm, called shape approximation based line segment extraction algorithm. This second line segment extraction algorithm produces hybrid point and line segment models that are faithful to surface geometry. With the new algorithm, line segments are no longer being processed separately, they are extracted in bunches from anisotropic surface regions.

We organize our subsequent discussions as such: in Section 3.2.1 we reformulate our line segment extraction problem. In Section 3.2.2, we talk about surface normal variations and $L^{2,1}$ error metric. We then compare two clustering algorithms in Section 3.2.3. One is a greedy algorithm and the other one is a hierarchical iterative optimization algorithm. In Section 3.2.4, we present the details of our line segment extraction algorithm. Finally, we discuss the problems with this approach in Section 3.2.5.

3.2.1. Problem Formulation

In the (ε, δ) error bounded line segment extraction algorithm, the Euclidean distance error tolerance ε and normal deviation error tolerance δ are enforced strictly. Essentially the line segment extraction problem becomes only a primitive replacement problem – replacing sets of sequentially aligned points that meet the ε and δ error tolerances with

line segments. Here we look at the problem from a different perspective. We treat the line segment extraction problem as a shape approximation problem that aims to extract, retain and explicitly represent the primal surface anisotropic information. Specifically, we formulate the surface line segment modelling problem as searching for a set of points and line segments that adapt to surface geometry, approximating the surface shape with the approximation error controlled by a prescribed normal variation tolerance.

In this shape approximation based approach, we lift up the ε and δ error tolerances imposed in our first error bounded algorithm and do not insist in sticking closely to the original scanned point cloud any more. Instead, the geometric faithfulness of the hybrid representation now truly becomes the primary goal we pursue. To be in tune with the actual surface geometry, prior to extracting line segments, we segment the object surface into clusters having different geometric characteristics first. By doing so, anisotropic point clusters would be identified, and line segments can then be extracted effectively and efficiently from points residing in these anisotropic clusters. Eventually, extracted line segment together with leftover points form a shape approximation of the object surface.

3.2.2. Normal Variation and $L^{2,1}$ Shape Error Metric

Surface anisotropy indicates the direction along which a surface region is smooth. Smoothness implies the least variation of surface normals. We thus choose normal as the criteria for grouping sampled points. In the final output clusters of our line segment extraction algorithm, for a seed point $\boldsymbol{p}_i \in \mathcal{P}$ and the point cluster C_i generated with $\boldsymbol{p}_i$, the variation between the normal $\boldsymbol{n}_j$ of any point $\boldsymbol{p}_j \in C_i$ and the seeding normal $\boldsymbol{n}(C_i)$ of

C_i must be less than the normal variation tolerance δ_n. That is $\left\|\boldsymbol{n}(C_i) - \boldsymbol{n}_j\right\|^2 \le \delta_n$. And similar to [CAD04], we define the $L^{2,1}$ shape approximation error measure for the point cluster C_i as $L^{2,1}(C_i) = \sum_{\boldsymbol{p}_j \in C_i} \left\|\boldsymbol{n}(C_i) - \boldsymbol{n}_j\right\|^2$. The $L^{2,1}(C_i)$ accounts for the total amount of normal variations in C_i. Obviously $L^{2,1}(C_i)$ is upper bounded by $\|C_i\| \cdot \delta_n$.

3.2.3. The Clustering Algorithm

We develop two algorithms to segment P into point clusters. The first one is a greedy clustering algorithm. It groups points in P into clusters through flooding. The normal variation tolerance δ_n is used as the flooding criteria. δ_n is capable of effectively capturing surface anisotropies. Given a fixed normal variation tolerance, it comes straightforward that more points in the anisotropic direction of a surface region will be collected into the point cluster.

The second algorithm is modified from the shape error metric $L^{2,1}$ based distortion minimized clustering algorithm. Proved in [CAD04], the $L^{2,1}$ metric is quite effective in capturing surface anisotropies. We actually turn the original algorithm in [CAD04] into a new hierarchical iterative algorithm. Instead of looking for the best way of clustering that minimizes the overall $L^{2,1}$ error measure for a fixed number of clusters, we search for a set of clusters having their maximal normal variations controlled by δ_n while at the same time still attaining a small total normal variation. We give details of the two clustering algorithms in Section 3.2.3.1 and 3.2.3.2 respectively.

3.2.3.1. Greedy Clustering

Let $\mathcal{M}_{\text{cov},i}$ denote the 3×3 covariance matrix computed for every $\boldsymbol{p}_i \in \mathcal{P}$. Refer to Section 6.1.1 for how to compute $\mathcal{M}_{\text{cov},i}$. And let λ_0, λ_1 and λ_2 be the three eigenvalues of $\mathcal{M}_{\text{cov},i}$ and order them as $\lambda_0 \le \lambda_1 \le \lambda_2$. Define the surface variation measure at $\boldsymbol{p}_i$ as $V_s(\boldsymbol{p}_i) = \frac{\lambda_0}{\lambda_0+\lambda_1+\lambda_2}$. The value of $V_s(\boldsymbol{p}_i)$ indicates the flatness of the surface region near $\boldsymbol{p}_i$. We compute $V_s(\boldsymbol{p}_i)$ for all the points in $\mathcal{P}$ and flood the points into $\mathcal{P}$ in the increasing order of the values of the their $V_s(\boldsymbol{p}_i)$. For every point $\boldsymbol{p}_i$ being picked up for flooding, we create a new point cluster C_i with $\boldsymbol{p}_i$ as the seed and $\boldsymbol{n}(C_i) = \boldsymbol{n}_i$ as the seeding normal. We then expand the cluster C_i through the set of neighbouring points of C_i, denoted as $\mathcal{N}_d(C_i)$. The neighbour point $\boldsymbol{p}_k \in \mathcal{N}_d(C_i)$ that has not yet been included into any other point cluster, is checked against the normal variation tolerance δ_n. If $\left\|\boldsymbol{n}(C_i) - \boldsymbol{n}_k\right\|^2 \le \delta_n$, then $\boldsymbol{p}_k$ is added into C_i and updated as a new boundary point of C_i. The status of $\boldsymbol{p}_k$ is also set as already being flooded over. The greedy algorithm terminates when all the points in $\mathcal{P}$ are marked as flooded. At the end a set of point clusters $\mathbb{C}$ is returned.

3.2.3.2. Hierarchical Distortion Minimized Clustering

We randomly pick up two points $\boldsymbol{p}_s$ and $\boldsymbol{p}_t$ from $\mathcal{P}$ and create two new point clusters C_s and C_t. C_s has $\boldsymbol{p}_s$ as its seed and $\boldsymbol{n}(C_s) = \boldsymbol{n}_s$ as its seeding normal. Similarly, C_t has $\boldsymbol{p}_t$ as its seed and $\boldsymbol{n}(C_t) = \boldsymbol{n}_t$ as its seeding normal. We then grow C_s and C_t through their neighbouring points. Let us denote the set of neighbouring points of C_s and C_t using $\mathcal{N}_d(C_s \cup C_t) = \mathcal{N}_d(C_s) \cup \mathcal{N}_d(C_t)$. For each point $\boldsymbol{p}_k \in \mathcal{N}_d(C_s \cup C_t)$ that has not yet been

assigned to any cluster, we keep track of its neighbouring clusters. $\boldsymbol{p}_k$ may have only one neighbouring cluster, either C_s or C_t, or both of them. For each such pair of neighbouring point and cluster $(\boldsymbol{p}_k, C_b)$ where $b = s$ or t, we compute their normal variation $V_n(\boldsymbol{p}_k, C_b) = \|\boldsymbol{n}_k - \boldsymbol{n}(C_b)\|^2$. And then push $(\boldsymbol{p}_k, C_b)$ into a priority queue PQ with $V_n(\boldsymbol{p}_k, C_b)$ as the key value. The $(\boldsymbol{p}_k, C_b)$ pair having the smallest $V_n(\boldsymbol{p}_k, C_b)$ is then popped out from PQ and the point $\boldsymbol{p}_k$ is assigned to the cluster C_b. $\boldsymbol{p}_k$ is updated as a new boundary point of C_b and the status of $\boldsymbol{p}_k$ is set already being assigned. This pushing and popping operation continues until all points in $\mathcal{P}$ are assigned to a cluster.

After the first iteration of clustering, although the point $\boldsymbol{p}_i \in \mathcal{P}$ is assigned to the reachable cluster C_b with which it has smaller normal variation $V_n(\boldsymbol{p}_i, C_b)$, the total amount of normal variation $\sum_{b=r,s} \sum_{\boldsymbol{p}_j \in C_b} V_n(\boldsymbol{p}_j, C_b)$ over the object surface has not yet been minimized. As in [CAD04], we need to reseed for each C_b and repeat the clustering process until certain stopping criteria are met. For each C_b we compute the average normal of all points belonging to it, $\boldsymbol{n}'(C_b) = \frac{\sum_{\boldsymbol{p}_j \in C_b} \boldsymbol{n}_j}{\|C_b\|} \Big/ \left\| \frac{\sum_{\boldsymbol{p}_j \in C_b} \boldsymbol{n}_j}{\|C_b\|} \right\|$. The point $\boldsymbol{p}'_b \in C_b$ where $b = s$ or t, that has the least normal variation with $\boldsymbol{n}'(C_b)$ is chosen as the new seed of C_b. And $\boldsymbol{n}'(C_b)$ is used as the new seeding normal in the next iteration of clustering. The stopping criterion is either the seeds of all C_b remain unchanged or the number of clustering iterations has reached a specified limit.

For each point cluster C_b where $b = s$ or t, we also record its maximum normal variation $V_{n,\max}(C_b) = \max(\|\boldsymbol{n}_j - \boldsymbol{n}(C_b)\|^2)$ where $\boldsymbol{n}_j$ is the normal of the point $\boldsymbol{p}_j \in C_b$. If $V_{n,\max}(C_b) > \delta_n$, it means the cluster C_b does not satisfy the normal variation tolerance constraint. Thus we apply the iterative distortion minimized clustering algorithm to the point cluster C_b again, further splitting C_b into two smaller clusters. At the end, the algorithm will return a set of clusters $\mathbb{C}$, such that $V_{n,\max}(C_i) \le \delta_n$ for any $C_i \in \mathbb{C}$.

3.2.3.3. Discussions on Clustering Algorithms

Both the greedy clustering algorithm and the hierarchical distortion minimized clustering algorithm can effectively form anisotropic clusters on point clouds. And the hierarchical algorithm generally produces fewer clusters than the greedy algorithm. See our experiment results in Section 7.2.1. This is as being expected. Through several iterations of seeding and reseeding, the hierarchical algorithm adjusts the point clusters to optimal shape that efficiently fit to the surface geometry. While the greedy algorithm processes every sampled point once and only once, thus much of the clustering depends on the surface variation heuristic V_s used, which may be good locally to each individual point but could be bad when need to be aware of the nieghbouring points too. However we find that less number of clusters does not help the distortion minimized clustering algorithm extract more line segments (check Section 7.2.1 for the detailed explanations).

At first glance, it seems to be very promising to estimate a minimum curvature vector for every sampled point in input point cloud and use it to collect points into anisotropic point

clusters. This is backed by the fact that the direction of surface anisotropy follows the surface's minimum curvature vector. However this method turns out to be infeasible in practice. The reality is that none of the existing curvature calculation methods is capable of estimating the curvature information of point cloud data reliably. The point cloud's scanning direction and scanning pattern can badly affect the accuracy of curvature computations. Refer to our discussions in Section 6.1.2 for more details.

3.2.4. The Line Segment Extraction Algorithm

3.2.4.1. Constructing Local Coordinate Systems

We try to extract line segments from every point cluster $C_i \in \mathbb{C}$ that owns more than 3 points. We first calculate the centroid of points in C_i, $\boldsymbol{c}(C_i) = \frac{\sum_{\boldsymbol{p}_j \in C_i} \boldsymbol{p}_j}{\|C_i\|}$. Then we compute the householder matrix of $\boldsymbol{n}(C_i)$, $\mathcal{M}_h(\boldsymbol{n}(C_i))$, where $\boldsymbol{n}(C_i)$ is the seeding normal of C_i. From the second and third columns of $\mathcal{M}_h$, we obtain the other two bases $\boldsymbol{x}(C_i)$ and $\boldsymbol{y}(C_i)$. $\boldsymbol{n}(C_i)$, $\boldsymbol{x}(C_i)$ together with $\boldsymbol{y}(C_i)$ form a local coordinate system centered at $\boldsymbol{c}(C_i)$. We then project all the points in C_i along $\boldsymbol{n}(C_i)$ onto the xy plan P_{xy}. Let us denote the set of projected points as $C_i^{'}$ with each $\boldsymbol{p}_j^{'} \in C_i^{'}$ corresponding to a $\boldsymbol{p}_j \in C_i$. To prevent the surface region represented by C_i from being folded in the projection, we need to ensure no $\boldsymbol{p}_j$'s normal $\boldsymbol{n}_j$ spans more than 90° away from $-\boldsymbol{n}(C_i)$. Thus the value of the normal variation tolerance δ_n is set to be no less than 2.

3.2.4.2. Determining Surface Anisotropic Direction

Points in most of C_i and $C_i^{'}$ are unevenly distributed. This irregularity causes the biggest trouble to the line segment extraction process. In the (ε, δ) error bounded algorithm, we attack the problem by greedily fitting line segments. Here since we have already known that the point cluster C_i is anisotropically shaped, we can regularize the points towards the anisotropic direction and subsequently extract line segments from sequences of regularized points.

We compute the minimum area enclosing rectangle $R_a(C_i^{'})$ of points in $C_i^{'}$ using the rotating caliper algorithm suggested in [Tous83]. Let us denote the direction of the longer edges of $R_a(C_i^{'})$ as $\boldsymbol{d}(R_a(C_i^{'}))$. Essentially $\boldsymbol{d}(R_a(C_i^{'}))$ gives the shape anisotropic direction $\boldsymbol{d}_s(C_i)$ of C_i. That is $\boldsymbol{d}_s(C_i) = \boldsymbol{d}(R_a(C_i^{'}))$. Refer to [KFR04] for a recent interesting discussion on the shape anisotropy concept. Let us use $\boldsymbol{d}_f(C_i)$ to denote the surface anisotropic direction of C_i. Since the point cluster C_i is anisotropically formed, according to δ_n and $L^{2,1}(C_i)$, the surface shape of C_i would be stretched along $\boldsymbol{d}_f(C_i)$. That is to say, in general for C_i, its shape anisotropic direction $\boldsymbol{d}_s(C_i)$ agrees with its surface anisotropic direction $\boldsymbol{d}_f(C_i)$.

However this is not necessarily always be the case. The short yet wide cylindrical surface patch has its shape anisotropy direction 90° deviated away from its surface anisotropy direction. To prevent such insistency, we also estimate the minimum curvature direction

$\boldsymbol{k}_{\min}(\boldsymbol{p}_i)$ for every point $\boldsymbol{p}_j \in C_i$ and assign their average as the minimum curvature direction of C_i, $\boldsymbol{k}_{\min}(C_i) = \frac{\sum_{\boldsymbol{p}_i \in C_i} \boldsymbol{k}_{\min}(\boldsymbol{p}_i)}{\|C_i\|} \Big/ \left\| \frac{\sum_{\boldsymbol{p}_i \in C_i} \boldsymbol{k}_{\min}(\boldsymbol{p}_i)}{\|C_i\|} \right\|$. We then check whether $\boldsymbol{d}_s(C_i)$ is consistent with $\boldsymbol{k}_{\min}(C_i)$. If $\boldsymbol{k}_{\min}(C_i)$ differs from $\boldsymbol{d}_s(C_i)$ for more than 45°, we assign $\boldsymbol{k}_{\min}(C_i)$ to $\boldsymbol{d}_f(C_i)$. That is $\boldsymbol{d}_f(C_i) = \boldsymbol{k}_{\min}(C_i)$. Otherwise we use the shape anisotropic direction as the surface anisotropic direction, $\boldsymbol{d}_f(C_i) = \boldsymbol{d}_s(C_i)$. The reason why we do not use $\boldsymbol{k}_{\min}(C_i)$ straightforwardly as the surface anisotropic direction is again because none of existing methods can estimate the curvature information accurately. Refer to Section 6.1.2 for detailed discussions.

3.2.4.3. Computing Average Distances

For convenience, we then rotate the *xy* plan P_{xy} around $\boldsymbol{n}(C_i)$, and align the $\boldsymbol{y}(C_i)$ axis with the surface anisotropic direction $\boldsymbol{d}_f(C_i)$. Let us denote the set of local Delaunay neighbours of every point $\boldsymbol{p}_j^{'} \in C_i^{'}$ as $\mathcal{N}_d^{'}(\boldsymbol{p}_j^{'})$, where $\boldsymbol{p}_k^{'} \in \mathcal{N}_d^{'}(\boldsymbol{p}_j^{'})$ is the projection of $\boldsymbol{p}_k \in \mathcal{N}_d(\boldsymbol{p}_j)$ on P_{xy}. We compute the projected average distance on the $\boldsymbol{x}(C_i)$ basis $d_{avg,x}(\boldsymbol{p}_j^{'})$ from every $\boldsymbol{p}_j^{'} \in C_i^{'}$ to its neighbours $\mathcal{N}_d^{'}(\boldsymbol{p}_j^{'})$, $d_{avg,x}(\boldsymbol{p}_j^{'}) = \frac{\sum_{\boldsymbol{p}_k^{'} \in \mathcal{N}_d^{'}(\boldsymbol{p}_j^{'})} (\boldsymbol{p}_j^{'} - \boldsymbol{p}_k^{'}) \cdot \boldsymbol{x}(C_i)}{\|\mathcal{N}_d^{'}(\boldsymbol{p}_j^{'})\|}$.

And similarly $d_{avg,y}(\boldsymbol{p}_j^{'}) = \frac{\sum_{\boldsymbol{p}_k^{'} \in \mathcal{N}_d^{'}(\boldsymbol{p}_j^{'})} (\boldsymbol{p}_j^{'} - \boldsymbol{p}_k^{'}) \cdot y(C_i)}{\|\mathcal{N}_d^{'}(\boldsymbol{p}_j^{'})\|}$. Subsequently we obtain the projected average distance on the $\boldsymbol{x}(C_i)$ between any two neighbouring points of $C_i^{'}$, $d_{avg,x}(C_i^{'}) = \frac{\sum_{\boldsymbol{p}_j^{'} \in C_i^{'}} d_{avg,x}(\boldsymbol{p}_j^{'})}{\|C_i^{'}\|}$.

And similarly $d_{avg,y}(C_i^{'}) = \frac{\sum_{\boldsymbol{p}_j^{'} \in C_i^{'}} d_{avg,y}(\boldsymbol{p}_j^{'})}{\|C_i^{'}\|}$.

3.2.4.4. Constructing 2D Grid Structures

We use a nonuniform 2D grid structure $G(C_i^{'})$ to regularize points in $C_i^{'}$. $G(C_i^{'})$ is aligned with the two bases $\boldsymbol{x}(C_i)$ and $\boldsymbol{y}(C_i)$ and centered at $\boldsymbol{c}(C_i)$. We find the $\boldsymbol{x}(C_i)$ and $\boldsymbol{y}(C_i)$ bases aligned minimum enclosing rectangle $R_s(C_i^{'})$ of the points in $C_i^{'}$. $R_s(C_i^{'})$ gives the region in P_{xy} that is covered by the grid structure $G(C_i^{'})$. Use $d_x(R_s(C_i^{'}))$ and $d_y(R_s(C_i^{'}))$ to denote the length of the edges of $R_s(C_i^{'})$ that are parallel to $\boldsymbol{x}(C_i)$ and $\boldsymbol{y}(C_i)$ respectively. We cut $G(C_i^{'})$ uniformly into $c_x = \left\lceil \frac{d_y(R_s(C_i^{'}))}{d_{avg,y}(C_i^{'})} \right\rceil$ number of x-slices along the $\boldsymbol{x}(C_i)$ basis.

Similarly we also cut $G(C_i^{'})$ into $c_y = \left\lceil \frac{d_x(R_s(C_i^{'}))}{d_{avg,x}(C_i^{'})} \right\rceil$ number of y-slices along the $\boldsymbol{y}(C_i)$ basis. However the y-slices are not cut uniformly in terms of the distance. Instead we allocate equal number of points in every y-slice. That is $\frac{\|C_i\|}{c_y}$ points per y-slice. Such way of cutting y-slices ensures the spacing of line segments extracted will match the density of points in C_i. We number x-slices from 1 to c_x along the positive $\boldsymbol{x}(C_i)$ direction and number y-slices from 1 to c_y along the positive $\boldsymbol{y}(C_i)$ direction. The mth y-slice is addressed as $ySlice_m$ and similarly the nth x-slice is addressed as $xSlice_n$. And now any gird square in $G(C_i^{'})$ can be address as $g_{m,n}$, where $1 \le m \le c_y$ and $1 \le n \le c_x$.

For each grid square $g_{m,n} \in G(C_i^{'})$, we keep track of all points in $C_i^{'}$ that lie within it. If it is the case that some point lies inside $g_{m,n}$, we then mark $g_{m,n}$ as being occupied. We keep

one more tag with $g_{m,n}$, we mark $g_{m,n}$ as being influenced if $g_{m,n}$ is influenced by some nearby point. We associate an influence zone $Z(\boldsymbol{p}_j^{'})$ with every point $\boldsymbol{p}_j^{'} \in C_i^{'}$. $Z(\boldsymbol{p}_j^{'})$ is defined as a 2D rectangle centered at $\boldsymbol{p}_j^{'}$. For each $Z(\boldsymbol{p}_j^{'})$, we set the length of its edges parallel to $\boldsymbol{x}(C_i)$ be $2 \cdot d_{avg,x}(\boldsymbol{p}_j^{'})$ and the length of its edges parallel to $\boldsymbol{y}(C_i)$ be $2 \cdot d_{avg,y}(\boldsymbol{p}_j^{'})$. Any $g_{m,n}$ that is overlapped with $Z(\boldsymbol{p}_j^{'})$ is marked as being influenced by $\boldsymbol{p}_j^{'}$. By keeping a record of the tags "occupied" and "influenced", essentially we obtain a 2D description of the surface geometry represented by the point cluster C_i in $G(C_i^{'})$.

Not all the surface geometry information is recorded in $G(C_i^{'})$, in fact only partially. But that is sufficient for our purpose. Extracting line segments from a fully reconstructed object surface is left as part of our future work. Given a discrete set of points, we need to know how to grow the line segments through them. For two sampled points, we need to decide whether a line segment could pass through both of them or not, as there might be a surface gap in-between them. In the (ε, δ) error bounded algorithm, we extend the line segments in a try and error manner. Refer to Section 3.1.4 for the details. Here, via reading the geometry information stored in $G(C_i^{'})$, we can easily know how to grow the line segments. By checking the tags "occupied" and "influenced", we are able to tell which part of the surface region represented by C_i is continuous, around which part there might be a hole, and whether the line segments have already reached the boundary of C_i thus they should be stopped from growing longer.

3.2.4.5. Tracing Out Line Segments

Since the $\boldsymbol{y}(C_i)$ axis has already been aligned with the surface anisotropic direction $\boldsymbol{d}_f(C_i)$, we are to extract line segments from the y-slices. Each $ySlice_m$ contains a sequence of n grid squares $G(ySlice_m) = \{g_{m,1}, g_{m,2}, g_{m,3} .., g_{m,n}\}$. We trace every $G(ySlice_m)$ along the positive $\boldsymbol{y}(C_i)$ direction. We attempt to extract a line segment from each sequence of consecutive grid squares $G_{p,q}(ySlice_m) = \{g_{m,p}, g_{m,p+1}, g_{m,p+2} .., g_{m,q}\}$ that are all marked as being influenced, where $1 \le p \le q \le n$.

Let us denote the set of points contained in all gird squares of $G_{p,q}(ySlice_m)$ as $C_i^{'}(G_{p,q}(ySlice_m))$ and their corresponding points in C_i as $C_i(G_{p,q}(ySlice_m))$, where $C_i^{'}(G_{p,q}(ySlice_m)) \subseteq C_i^{'}$ and $C_i(G_{p,q}(ySlice_m)) \subseteq C_i$. If $\left\| C_i(G_{p,q}(ySlice_m)) \right\| < 4$, we would just leave the points in $C_i(G_{p,q}(ySlice_m))$ as they are. Otherwise, we find the points $\boldsymbol{p}_s^{'}, \boldsymbol{p}_e^{'} \in C_i^{'}(G_{p,q}(ySlice_m))$ that have the smallest and largest $\boldsymbol{y}(C_i)$ coordinate respectively. Then we use the two points $\boldsymbol{p}_s, \boldsymbol{p}_e \in C_i(G_{p,q}(ySlice_m))$ that correspond to $\boldsymbol{p}_s^{'}$ and $\boldsymbol{p}_e^{'}$ as the endpoints of the line segment $L(G_{p,q}(ySlice_m))$ extracted from $G_{p,q}(ySlice_m)$. Let us denote the unit direction vector of $L(G_{p,q}(ySlice_m))$ as $\boldsymbol{d}(L(G_{p,q}(ySlice_m))) = \frac{\boldsymbol{p}_s - \boldsymbol{p}_e}{\|\boldsymbol{p}_s - \boldsymbol{p}_e\|}$. We then compute the average normal of all points in $C_i(G_{p,q}(ySlice_m))$, $\boldsymbol{n}_{avg}(G_{p,q}(ySlice_m)) = \frac{\sum_{\boldsymbol{p}_k \in C_i(G_{p,q}(ySlice_m))} \boldsymbol{n}_k}{\left\| C_i(G_{p,q}(ySlice_m)) \right\|} \Big/ \left\| \frac{\sum_{\boldsymbol{p}_k \in C_i(G_{p,q}(ySlice_m))} \boldsymbol{n}_k}{\left\| C_i(G_{p,q}(ySlice_m)) \right\|} \right\|$. Denote the dot product between $\boldsymbol{n}_{avg}(G_{p,q}(ySlice_m))$ and $\boldsymbol{d}(L(G_{p,q}(ySlice_m)))$ as

$d_{\boldsymbol{n},\boldsymbol{d}} = \boldsymbol{n}_{avg}(G_{p,q}(ySlice_m)) \cdot \boldsymbol{d}(L(G_{p,q}(ySlice_m)))$. Eventually the normal of the line segment $L(G_{p,q}(ySlice_m))$ is computed out as $\boldsymbol{n}(L(G_{p,q}(ySlice_m))) = \frac{\boldsymbol{n}_{avg}(G_{p,q}(ySlice_m)) - d_{\boldsymbol{n},\boldsymbol{d}} \cdot \boldsymbol{d}(L(G_{p,q}(ySlice_m)))}{\left\| \boldsymbol{n}_{avg}(G_{p,q}(ySlice_m)) - d_{\boldsymbol{n},\boldsymbol{d}} \cdot \boldsymbol{d}(L(G_{p,q}(ySlice_m))) \right\|}$. $\boldsymbol{n}(L(G_{p,q}(ySlice_m)))$ is perpendicular to the line segment $L(G_{p,q}(ySlice_m))$. See Figure 25 and 26 for the hybrid point and line segment models obtained.

3.2.5. Observations, Problems and Discussions

Our shape approximation based line segment extraction algorithm produces hybrid point and line segment representations that are truly faithful to the surface geometry represented by the scanned point cloud. In this algorithm, a normal variation tolerance δ_n is used to control the maximum shape approximation error. Due to the approximation nature of this approach, in case when δ_n is quite large, a lot of surface geometry details could become lost in the new hybrid models and surface topology could be altered as well. Like the first (ε, δ) error bounded algorithm, this second line segment extraction algorithm could not guarantee a hole in rendered surface images either. We found that often there are slightly bigger visible holes. However better than the first greedy algorithm, this second algorithm performs far less dependent on the scanning process of point clouds.

We have seen work on tracing curvature lines on triangle meshes [CAD04, MaKo04]. To extract line segments from point clouds, it seems to be quite promising to similarly trace curvature lines on point clouds first and subsequently break the curvature lines into straight line segments. However there are two difficulties that hinder us from taking this approach. First, to trace curvature lines, the surface must be known in advance. Both [CAD04] and

[MaKo04] work on triangle meshes, so they do have a complete surface description. However we have no idea of what the surface represented by a discrete set of points looks like. Or otherwise surface reconstruction algorithms must be used to recover the surface first. However this would make the problem much more complicated. We leave the problem of extracting line segments from reconstructed point set surfaces as part of possible future work. The second problem still lies on the curvature estimation of point cloud data. Without being able to estimate curvature information accurately, it would not be of too much meaning to trace out the curvature lines first.

3.3. Contour Plane Based Line Segment Extraction

In this subsection we describe an algorithm that uses contour planes to cut out line segments from triangle meshes. Our previous two algorithms in Section 3.1 and 3.2 extract hybrid point and line segment models from point clouds, while this contour plane based algorithm is able to extract pure line segment models from triangle meshes. The extracted line segment model would be faithful to the given triangle mesh. Since the triangle mesh gives a complete surface description, the rendered images using the line segment model could be hole-free. The length of extracted line segments is dependant on the size of triangles used in the input mesh.

Let d be a given value with the constraint that it is smaller than twice the cutoff radius of the Gaussian kernel used in the reconstruction and low pass filter of our rendering algorithm (see Section 4 and 5 for detailed discussions on filtering). Let us denote the input triangle mesh as $\mathcal{M}$. We choose two set of parallel contour planes $\mathcal{P}_{\parallel}$ and $\mathcal{P}_{=}$ to intersect

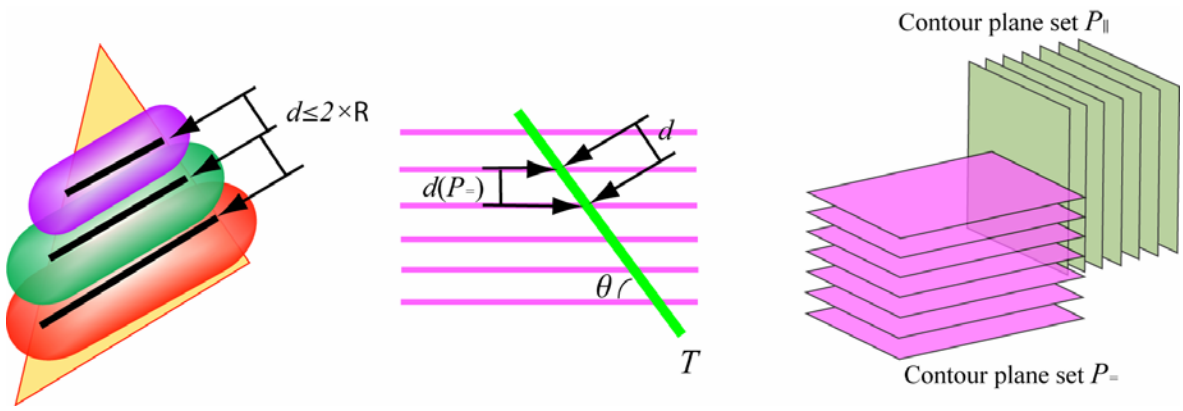

Figure 6: *The left and middle parts illustrate how we compute the spacing distance* $d(\mathcal{P}_{=})$ *between contour planes of* $\mathcal{P}_{=}$. *The right part shows the two set of perpendicular contour planes* $\mathcal{P}_{=}$ *and* $\mathcal{P}_{\|}$ *that are used to cut triangle meshes.*

$\mathcal{M}$ for line segments (see Figure 6). Any contour plane in $\mathcal{P}_{\|}$ is perpendicular to all the contour planes in $\mathcal{P}_{=}$, and consequently any contour plane in $\mathcal{P}_{=}$ is perpendicular to all the contour planes in $\mathcal{P}_{\|}$ as well. The orientations of the contour planes in $\mathcal{P}_{\|}$ and $\mathcal{P}_{=}$ are chosen arbitrarily without any preference, neither are they aligned with the bases of the coordinate system of $\mathcal{M}$.

For each triangle $T \in \mathcal{M}$, we first try intersecting it using contour planes in $\mathcal{P}_{=}$. With the distance constraint d, we need to set the spacing distance $d(\mathcal{P}_{=})$ between contour planes in $\mathcal{P}_{=}$ appropriately. Let θ be the intersected slope between the triangle T and contour planes in $\mathcal{P}_{=}$. $d(\mathcal{P}_{=})$ must satisfy $d(\mathcal{P}_{=}) \le d \times \sin(\theta)$ (see Figure 6). We thus set the value of $d(\mathcal{P}_{=})$ as $d(\mathcal{P}_{=}) = \frac{d}{2^{\lceil \log_2(1/\sin(\theta)) \rceil}}$. With $d(\mathcal{P}_{=})$ determined, we obtain one line segment from each intersection between T and the contour planes in $\mathcal{P}_{=}$. The normal of the intersected line segment inherits the normal of T. Sufficient number of line segments

would be extracted using $d(\mathcal{P}_{=})$ if θ is not too small. In case θ is quite small, which implies T is nearly parallel to the contour planes in $\mathcal{P}_{=}$, we then choose the other contour plane set $\mathcal{P}_{\|}$ to intersect with T instead. Eventually the collection of line segments extracted from all the triangles in $\mathcal{M}$ form a line segment model of $\mathcal{M}$.

To extract a textured line segment model from a textured triangle mesh, texture coordinates need to be assigned to the endpoints of line segments too. This can be done easily. We simply have the texture coordinates of the line segments' endpoints bilinearly interpolated from the texture coordinates of the triangle's vertices, the triangle from which the line segments are extracted.

Points can also be obtained using our contour plane based algorithm. We obtain hybrid point and line segment models by converting extracted line segments that are short than a specified length into points, and obtain pure point models by converting all extracted line segments into points. For the same reason to ensure that no hole will exist in the model, distance between two extracted points must also be less than d. Thus given an extracted line segment of length l, we convert it into $\lceil \frac{l}{d} \rceil + 1$ points at regular interval.

Chapter 4.

Mathematical Framework for Surface Rendering

In this section, a rigorous mathematical framework for EWA point based rendering procedure is presented. Two alternative approaches exist on top of this common framework – screen space EWA splatting and object space EWA splatting. This framework is an evolvement of Heckbert's EWA resampling theory [Heck89], extending texture mapping to point rendering. This framework defines a generic rendering paradigm. Our rendering pipeline for surface line segment primitive is also founded on it.

In Section 4.1 we discuss Heckbert's EWA resampling theory in great details. Then we talk about the EWA splatting techniques in Section 4.2. Formulation of screen space EWA splatting and Object space EWA splatting appear in Section 4.2.1 and 4.2.2 correspondingly. Most of mathematical derivations here are combined and modified from [Heck89, ZPVG01, RuLe00].

4.1. EWA Resampling Filter

Images and textures are digitized and sampled at certain frequency in the world space of a computer. Then they are warped onto screen and sampled to the screen grid before being displayed out. Aliasing is caused throughout the course of both the first sampling and the second sampling. To avoid aliasing artifacts, raising sampling frequency is what we can do for the first sampling practice; for the second round of sampling, we prefilter the warped

images before we sample. The prefiltering process stops high frequencies and passes only low frequencies, thus enabling the subsequent sampling to meet the Nyquist criterion. The entire process of warping, prefiltering and sampling is referred as resampling.

Ideal resampling is a resampling process that inputs a properly sampled signal, warps it, and outputs a properly sampled signal, minimizing information loss [Heck89]. It is consisting of four steps: reconstructing, warping, prefiltering and sampling.

1. Reconstruct the continuous signal from the discrete input signal
2. Warp the domain of the continuous signal
3. Prefilter the warped, continuous signal
4. Sample this signal to produce the discrete output signal

The mathematical formulation for ideal resampling process is derived as followings.

Given an input signal $f(\boldsymbol{u})$, a forward mapping from object space to screen space $\boldsymbol{x} = m(\boldsymbol{u})$ where the object space coordinates are $\boldsymbol{u} = (u, v)$ and the screen space coordinates are $\boldsymbol{x} = (x, y)$, the inverse mapping $\boldsymbol{u} = m^{-1}(\boldsymbol{x})$, a reconstruction filter $r(\boldsymbol{u})$ and a prefilter $h(\boldsymbol{x})$, it is the output signal $g(\boldsymbol{x})$ we want to compute. Signal progresses through the four resampling steps as bellowing:

Discrete input signal	$f(\boldsymbol{u})$
Reconstructed input signal	$f_c(\boldsymbol{u}) = f(\boldsymbol{u}) \otimes r(\boldsymbol{u}) = \sum_{k \in \mathbb{Z}^2} f(\boldsymbol{k})\ r(\boldsymbol{u} - \boldsymbol{k})$

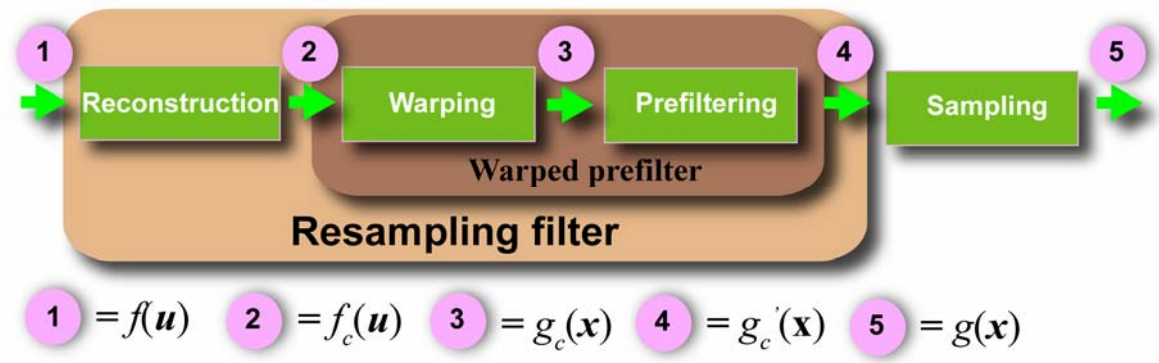

Figure 7: *EWA resampling filter block diagram.*

Warped signal	$g_c(\boldsymbol{x}) = f_c(m^{-1}(\boldsymbol{x}))$
Continuous output signal	$g_c'(\boldsymbol{x}) = g_c(\boldsymbol{x}) \otimes h(\boldsymbol{x}) = \int_{\mathbb{R}^2} g(\boldsymbol{t})\ h(\boldsymbol{x}-\boldsymbol{t})\, d\boldsymbol{t}$
Discrete output signal	$g(\boldsymbol{x}) = g_c'(\boldsymbol{x})\ i(\boldsymbol{x})$

In step 1, we convolute the discrete input signal $f(\boldsymbol{u})$ with the reconstruction filter $r(\boldsymbol{u})$ producing the continuous signal $f_c(\boldsymbol{u})$ in object space. In step 2, we warp the domain of the $f_c(\boldsymbol{u})$ from object space to screen space using the inverse mapping $\boldsymbol{u} = m^{-1}(\boldsymbol{x})$. We get the warped continuous signal $g_c(\boldsymbol{x})$ in screen space. In step 3, we convolute $g_c(\boldsymbol{x})$ with the prefilter $h(\boldsymbol{x})$. This gives us the low pass filtered continuous signal $g_c'(\boldsymbol{x})$ in screen space. Finally, we sample $g_c'(\boldsymbol{x})$ with the impulse function $i(\boldsymbol{x})$ giving us the discrete output signal $g(\boldsymbol{x})$. (See Figure 7)

Expand the above signal progression procedure in reverse order, for every $\boldsymbol{x}$ aligned to screen pixel grid we has

$$
\begin{aligned}
g(\boldsymbol{x}) &= g_c^{'}(\boldsymbol{x}) \\
&= \int_{\mathbb{R}^2} g_c(\boldsymbol{t})\ h(\boldsymbol{x}-\boldsymbol{t})\, d\boldsymbol{t} \\
&= \int_{\mathbb{R}^2} h(\boldsymbol{x}-\boldsymbol{t}) \left(\sum_{k\in\mathbb{Z}^2} f(\boldsymbol{k})\ r(m^{-1}(\boldsymbol{t})-\boldsymbol{k}) \right) d\boldsymbol{t}\,. \\
&= \sum_{k\in\mathbb{Z}^2} f(\boldsymbol{k})\ \rho(\boldsymbol{x},\boldsymbol{k})
\end{aligned}
$$

$$\text{where}: \rho(\boldsymbol{x},\boldsymbol{k}) = \int_{\mathbb{R}^2} h(\boldsymbol{x}-\boldsymbol{t})\ r(m^{-1}(\boldsymbol{t})-\boldsymbol{k})\, d\boldsymbol{t}$$

$\rho(\boldsymbol{x},\boldsymbol{k})$ is called resampling filter [Heck89]. $\rho(\boldsymbol{x},\boldsymbol{k})$ specifies the weight contributed by an input sample $\boldsymbol{k}$ in object space to an output sample $\boldsymbol{x}$ in screen space.

Substitute $\boldsymbol{t} = m(\boldsymbol{u})$ into $\rho(\boldsymbol{x},\boldsymbol{k})$, yields

$$\rho(\boldsymbol{x},\boldsymbol{k}) = \int_{\mathbb{R}^2} h(\boldsymbol{x}-m(\boldsymbol{u}))\ r(\boldsymbol{u}-\boldsymbol{k}) \left|\frac{\partial m}{\partial \boldsymbol{u}}\right| d\boldsymbol{u}$$

where: $\left|\frac{\partial m}{\partial \boldsymbol{u}}\right|$ is the determinant of Jocabian matrix $\begin{Bmatrix} x_u & x_v \\ y_u & y_v \end{Bmatrix}$.

Next we make two assumptions. The first one is local affine approximation assumption, which states that, in object space, for $\boldsymbol{u}$ in the neighborhood of $\boldsymbol{u}_o$, given $\boldsymbol{u}_o = m^{-1}(\boldsymbol{x}_o)$ we have $m_{\boldsymbol{u}_o}(\boldsymbol{u}) = \boldsymbol{x}_o + J_{\boldsymbol{u}_o} \cdot (\boldsymbol{u}-\boldsymbol{u}_o)$ where $J_{\boldsymbol{u}_o} = \frac{\partial m}{\partial \boldsymbol{u}}(\boldsymbol{u}_o)$ is the value of Jacobian matrix $J = \begin{Bmatrix} x_u & x_v \\ y_u & y_v \end{Bmatrix}$ evaluated at $\boldsymbol{u}_o$. The second one is affine mapping assumption made for both forward mapping m and backward mapping m^{-1}. This gives us the formula

$l^{-1}(\boldsymbol{x}-\boldsymbol{y}) = m_{\boldsymbol{u}_o}^{-1}(\boldsymbol{x}) - m_{\boldsymbol{u}_o}^{-1}(\boldsymbol{y})$ where l and l^{-1} are linear functions. Keep in mind that linear mappings can be expressed in terms of Jacobians, $l(\boldsymbol{u}) = J_{\boldsymbol{u}_o} \cdot \boldsymbol{u}$ and $l^{-1}(\boldsymbol{x}) = J_{\boldsymbol{u}_o}^{-1} \cdot \boldsymbol{x}$.

Substitute formulas implied by above two assumptions into the resampling filter $\rho(\boldsymbol{x},\boldsymbol{k})$, $\rho(\boldsymbol{x},\boldsymbol{k})$ get further simplified in the neighborhood of $\boldsymbol{x} = \boldsymbol{x}_o$ as following,

$$
\begin{aligned}
\rho(\boldsymbol{x},\boldsymbol{k}) &= \rho_{\boldsymbol{u}_o}(\boldsymbol{x}_o,\boldsymbol{k}) \\
&= \int_{\mathbb{R}^2} h(\boldsymbol{x} - m_{\boldsymbol{u}_o}(\boldsymbol{u}))\ r(\boldsymbol{u}-\boldsymbol{k}) \left|J_{\boldsymbol{u}_o}\right| d\boldsymbol{u} \\
&= \int_{\mathbb{R}^2} h'(l^{-1}(\boldsymbol{x} - m_{\boldsymbol{u}_o}(\boldsymbol{u})))\ r(\boldsymbol{u}-\boldsymbol{k})\, d\boldsymbol{u} \qquad \text{where: } h'(\boldsymbol{u}) = \left|J_{\boldsymbol{u}_o}\right| h(J_{\boldsymbol{u}_o} \cdot \boldsymbol{u}) \quad . \\
&= \int_{\mathbb{R}^2} h'(m_{\boldsymbol{u}_o}^{-1}(\boldsymbol{x}) - \boldsymbol{u})\ r(\boldsymbol{u}-\boldsymbol{k})\, d\boldsymbol{u} \\
&= \rho'_{\boldsymbol{u}_o}(m_{\boldsymbol{u}_o}^{-1}(\boldsymbol{x}) - \boldsymbol{k}) \qquad \text{where: } \rho'_{\boldsymbol{u}_o}(\boldsymbol{u}) = h'_{\boldsymbol{u}_o}(\boldsymbol{u}) \otimes r(\boldsymbol{u})
\end{aligned}
$$

Now the output signal $g(\boldsymbol{x})$ at $\boldsymbol{x} = \boldsymbol{x}_o$ can be expressed as a convolution of the discrete input signal with the resampling filter,

$$
\begin{aligned}
g(\boldsymbol{x}) &= \sum_{k \in \mathbb{Z}^2} f(\boldsymbol{k})\ \rho'(m^{-1}(\boldsymbol{x}) - \boldsymbol{k}) \\
&= \sum_{k \in \mathbb{Z}^2} f(\boldsymbol{k})\ \rho'_{\boldsymbol{u}_o}(m_{\boldsymbol{u}_o}^{-1}(\boldsymbol{x}) - \boldsymbol{k}). \\
&= (f \otimes \rho'_{\boldsymbol{u}_o})(m_{\boldsymbol{u}_o}^{-1}(\boldsymbol{x}))
\end{aligned}
$$

Finally, we replace both reconstruction filter and prefilter with Gaussians, yielding $r(\boldsymbol{u}) = g_{V_r}(\boldsymbol{u})$ and $h(\boldsymbol{x}) = g_{V_h}(\boldsymbol{x})$ where V_r and V_h are the variances of these Gaussians.

Since elliptical Gaussian is closed under convolution, this makes it possible for us to write resampling filter at $\boldsymbol{u} = \boldsymbol{u}_o$ into a much simpler form,

$$\begin{aligned}\rho^{'}(\boldsymbol{u}) &= \rho^{'}_{\boldsymbol{u}_o}(\boldsymbol{u}) \\ &= h^{'}_{\boldsymbol{u}_o}(\boldsymbol{u}) \otimes r(\boldsymbol{u}) \\ &= g_{J^{-1}_{\boldsymbol{u}_o} V_h J^{-1^T}_{\boldsymbol{u}_o}}(\boldsymbol{u}) \otimes g_{V_r}(\boldsymbol{u}) \\ &= g_{J^{-1}_{\boldsymbol{u}_o} V_h J^{-1^T}_{\boldsymbol{u}_o} + V_r}(\boldsymbol{u})\end{aligned}.$$

We notice that the resampling filter $\rho^{'}(\boldsymbol{u})$ is in fact of elliptical Gaussian shape. Heckbert names it Elliptical Weighted Average (EWA) filter. Elliptical Gaussians are closed under affine warps and convolution, so they form a very convenient class of filters for image resampling [Heck89]. EWA resampling filter can be implemented efficiently at the same time minimize aliasing.

4.2. EWA Splatting

Point rendering procedure is equivalent to the resampling process of textures. However, different from texture functions which are 2D functions, surface functions are of 3D. We need to define surface function from point cloud firstly before we can actually resample it.

Let us use $\mathcal{N}(\boldsymbol{q})$ to denote the set of neighbouring sampled points of an arbitrary object surface point $\boldsymbol{q}$. We construct a 2D local parameterization $\mathbb{S}$ in the neighborhood of $\boldsymbol{q}$. Let $\boldsymbol{u} \in \mathbb{S}$ be the local coordinate of $\boldsymbol{q}$, and $\boldsymbol{u}_k \in \mathbb{S}$ be the local coordinate of some sampled point $\boldsymbol{p}_k \in \mathcal{N}(\boldsymbol{q})$. The continuous surface function can thus be defined

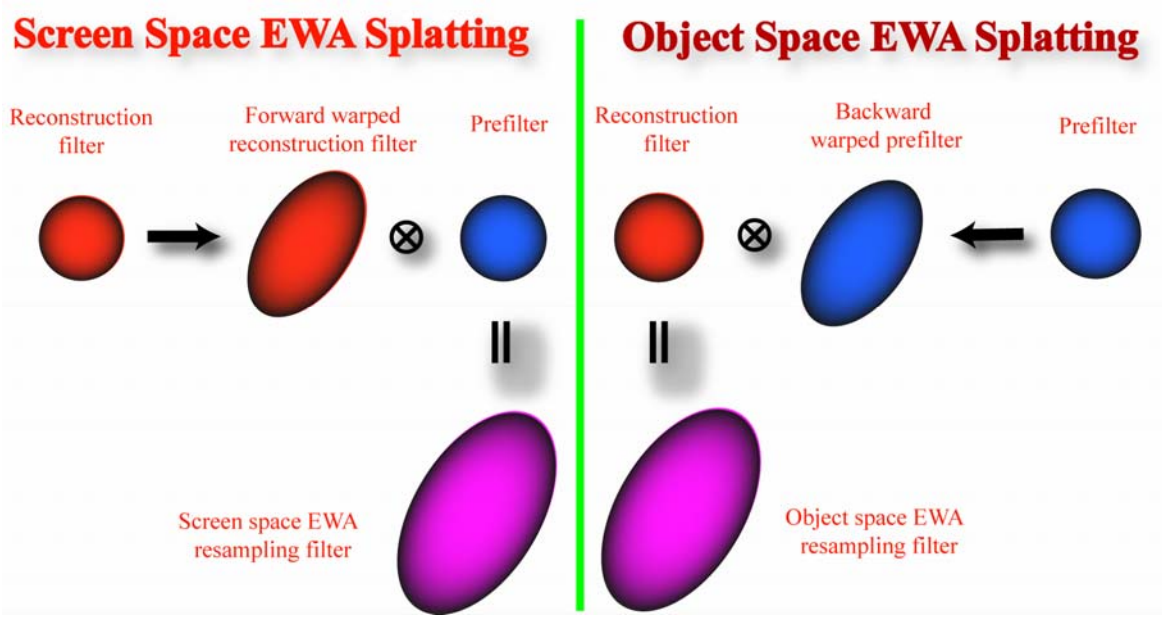

Figure 8: *Two forms of EWA splatting, in object space and screen space respectively. Reconstruction filters are colored red and prefilters are colored blue.*

as $f_c(\boldsymbol{u}) = \sum_{\boldsymbol{p}_k \in \mathcal{N}(\boldsymbol{q})} w_k \; r_k(\boldsymbol{u} - \boldsymbol{u}_k)$, where w_k is the color contribution of $\boldsymbol{p}_k$ to $\boldsymbol{q}$ and r_k is the reconstruction filter of $\boldsymbol{p}_k$.

Thereafter, we warp surface function $f_c(\boldsymbol{u})$ to screen space using forward mapping $\boldsymbol{x} = m(\boldsymbol{u})$, obtaining the continuous screen space signal $g_c(\boldsymbol{x}) = f_c(m^{-1}(\boldsymbol{x}))$. Next we apply prefiltering operation to $g_c(\boldsymbol{x})$, resulting in the low pass filtered continuous function $g_c^{'}(\boldsymbol{x}) = g_c(\boldsymbol{x}) \otimes h(\boldsymbol{x}) = \int_{\mathbb{R}^2} g_c(\boldsymbol{t}) \; h(\boldsymbol{x} - \boldsymbol{t}) \, d\boldsymbol{t}$. Lastly, we sample $g_c^{'}(\boldsymbol{x})$ with impulse function $i(\boldsymbol{x})$, getting $g(\boldsymbol{x}) = g_c^{'}(\boldsymbol{x}) \; i(\boldsymbol{x})$.

Combine reconstruction, warping and prefiltering in different order, we obtain EWA resampling filters belonging to different domains. This leads to two different ways to resample and splat points (See Figure 8).

4.2.1. Screen Space EWA Splatting

We write output function $g_c(\boldsymbol{x})$ aligned to pixel grid as a weighted sum of screen space resampling filter as following,

$$\begin{aligned} g_c(\boldsymbol{x}) &= g_c^{'}(\boldsymbol{x}) \\ &= \sum_{\boldsymbol{p}_k \in \mathcal{N}(\boldsymbol{q})} w_k \rho_k(\boldsymbol{x}) \\ \text{where:} \rho_k(\boldsymbol{x}) &= (r_k^{'} \otimes h)(\boldsymbol{x} - m_k(\boldsymbol{u}_k)) \\ r_k^{'}(\boldsymbol{x}) &= r_k(m^{-1}(\boldsymbol{x})) \end{aligned} .$$

Here, $\boldsymbol{x}$ corresponds to the screen projection of object surface point $\boldsymbol{q}$. The resampling filter $\rho_k(\boldsymbol{x})$ for the point $\boldsymbol{p}_k$ at $\boldsymbol{u}_k$ is combined from the warped reconstruction filter $r_k^{'}(\boldsymbol{x})$ and the prefilter $h(\boldsymbol{x})$ in screen space. Replace both reconstruction filter and prefilter with Gaussian kernels, $r(\boldsymbol{u}) = g_{V_k^r}(\boldsymbol{u})$ and $h(\boldsymbol{x}) = g_{V_k^h}(\boldsymbol{x})$. Denote the value of Jacobian matrix evaluated at $\boldsymbol{u}_k$ as J_k. Then with the use of local affine approximation assumption and affine mapping assumption, we get $\rho_k(\boldsymbol{x})$ further simplified, $\rho_k(\boldsymbol{x}) = \frac{1}{|J_k^{-1}|} g_{J_k V_k^r J_k^T + V_k^h}(\boldsymbol{x} - m_k(\boldsymbol{u}_k))$, which is named as screen space EWA resampling filter in [ZPBG01]. The point rendering procedure in screen space works in such a way:

project every sample point $\boldsymbol{p}_k$ at $\boldsymbol{u}_k$ from object space onto screen space, and splat the resampling filter $\rho_k(\boldsymbol{x})$ at $\boldsymbol{x}_k$.

4.2.2. Object Space EWA Splatting

Let us rearrange the screen space EWA resampling filter $\rho_k(\boldsymbol{x})$ as following,

$$\begin{aligned}
\rho_k(\boldsymbol{x}) &= (r_k^{'} \otimes h)(\boldsymbol{x} - m_k(\boldsymbol{u}_k)) \\
&= (r_k^{'} \otimes h)(m_k(m_k^{-1}(\boldsymbol{x})) - m_k(\boldsymbol{u}_k)) \\
&= (r_k^{'} \otimes h)(J_k \cdot (m_k^{-1}(\boldsymbol{x}) - \boldsymbol{u}_k)) \\
&= (r \otimes h_k^{'})(\boldsymbol{u} - \boldsymbol{u}_k) \\
&= \rho_k^{'}(\boldsymbol{u}) \\
\text{where: } h_k^{'}(\boldsymbol{u}) &= |J_k| \, h(J_k \cdot \boldsymbol{u})
\end{aligned}.$$

The output function $g_c(\boldsymbol{x})$ aligned to pixel grid can then be formulated as

$$\begin{aligned}
g_c(\boldsymbol{x}) &= g_c^{'}(\boldsymbol{x}) \\
&= \sum_{\boldsymbol{p}_k \in \mathcal{N}(\boldsymbol{q})} w_k \rho_k^{'}(\boldsymbol{u})
\end{aligned}.$$

The resampling filter $\rho_k^{'}(\boldsymbol{u})$ for each point $\boldsymbol{p}_k$ at $\boldsymbol{u}_k$ is combined from the reconstruction filter $r(\boldsymbol{u})$ and the warped prefilter $h_k^{'}(\boldsymbol{u})$ in object space. We still use Gaussian kernels for the reconstruction filter and prefilter. With the two affine assumptions, we get $\rho_k^{'}(\boldsymbol{u})$ simplified, $\rho_k^{'}(\boldsymbol{u}) = g_{V_k^r + J_k^{-1} V_k^h J_k^{-1^T}}(\boldsymbol{u} - \boldsymbol{u}_k)$ which is called object space EWA resampling filter. The point rendering process in object space works in the following way: splat the

resampling filter $\rho_k^{'}(\boldsymbol{u})$ at each sample point $\boldsymbol{p}_k$ in object space and project every splat onto screen.

Chapter 5.

Surface Rendering with Line Segments

In this section, we discuss how to render surface line segment primitive by introducing a mathematical formulation of the object space EWA resampling filter for line segments. However our derivation shows that the resampling filter has no close form, which is also true in screen space. To overcome this theoretical difficulty, we propose an approximation method to splat the resampling filter in object space instead [Wong03]. Our method is constructed on top of the common EWA surface rendering framework. Our method can be implemented easily and efficiently. Further more the rendering results in our experiments show that there is no visible image quality scarification.

Although we only implement our approximation method for object space line segment EWA splatting, we believe that, with little modifications, the screen space line segment EWA splatting can be similarly implemented as well. In our formulation of the line segments' resampling filter, each line segment is allowed to be assigned with only one color. However practices tell us rendering linear texture mapped line segments can still produce good quality images. We describe such a texturing algorithm for line segments as well.

We organize our presentation in this section as such: in Section 5.1 we step by step derive the mathematical formulation of the object space EWA resampling filter for line segments.

In Section 5.2, we first illustrate the object space EWA splatting procedure of line segments. Then we describe the geometric observations we have made from this splatting procedure. Subsequently we give out the details of our rendering algorithm. Lastly we describe how to render texture mapped line segment models.

5.1. Object Space EWA Resampling Filter for Line Segments

This subsection is split into two parts. In the first part we work out the resampling filter's formula. In the second part we show the steps for evaluating the formula, hence concluding the none-existence of the closed form. We use the method presented in Ren *et al.*'s work [RPZ02] to compute the Jacobian matrix.

5.1.1. Mathematical Formulation

For any arbitrary object surface point $\boldsymbol{q}$, we construct a 2D local parameterization $\mathbb{S}$ in its neighborhood. Let $\boldsymbol{u}$ be the local coordinate of $\boldsymbol{q}$. There exists a set of sampled surface lines segment $\mathbb{L}$ that lie in $\boldsymbol{q}$'s neighborhood. Let L_k represent a line segment belonging to L. For a point in L_k, we use $\boldsymbol{p}_k(t)$ to denote, where t is the Euclidian distance from $\boldsymbol{p}_k(t)$ to L_k's endpoint $\boldsymbol{o}$ which is chosen as L_k's starting endpoint, and $t \in [0, l_k]$. Here l_k denotes the length of L_k. Let $\boldsymbol{u}_k(t)$ be the local coordinate of point $\boldsymbol{p}_k(t)$. Having $r(\boldsymbol{u})$ representing the reconstruction filter for point, the reconstruction filter $R_k(\boldsymbol{u})$ of a line segment L_k is calculated by integrate $r(\boldsymbol{u})$ along the line segment, $R_k(\boldsymbol{u}) = \int_0^l r(\boldsymbol{u} - \boldsymbol{u}_k(t))dt$. The object surface function can then be defined by summing

$R_k(\boldsymbol{u})$ for $\mathbb{L}$. That is $f_c(\boldsymbol{u}) = \sum_{L_k \in \mathbb{L}} w_k \int_0^l r(\boldsymbol{u} - \boldsymbol{u}_k(t))dt$ where w_k is the value of L_k's color contribution to surface points. (See Figure 9)

We then warp surface function $f_c(\boldsymbol{u})$ to screen space using forward mapping $\boldsymbol{x} = m(\boldsymbol{u})$, obtaining the continuous screen space signal $g_c(\boldsymbol{x}) = f_c(m^{-1}(\boldsymbol{x}))$. Next we apply prefiltering operation to $g_c(\boldsymbol{x})$, resulting in the low pass filtered continuous function $g_c^{'}(\boldsymbol{x}) = g_c(\boldsymbol{x}) \otimes h(\boldsymbol{x}) = \int_{\mathbb{R}^2} g_c(\boldsymbol{\xi})\ h(\boldsymbol{x} - \boldsymbol{\xi})\, d\boldsymbol{\xi}$. Lastly, we sample $g_c^{'}(\boldsymbol{x})$ with impulse function $i(\boldsymbol{x})$, getting $g(\boldsymbol{x}) = g_c^{'}(\boldsymbol{x})\ i(\boldsymbol{x})$.

Expand above formulas of the four rendering steps in reverse order; we then obtain the continuous output function $g_c(\boldsymbol{x})$ aligned to screen pixel grid as following,

$$
\begin{aligned}
g_c(\boldsymbol{x}) &= g_c^{'}(\boldsymbol{x}) \\
&= \int_{\mathbb{R}^2} \sum_{L_k \in \mathbb{L}} w_k \left(\int_0^l r(m^{-1}(\boldsymbol{\xi}) - \boldsymbol{u}_k(t))dt \right)\ h(\boldsymbol{x} - \boldsymbol{\xi})\, d\boldsymbol{\xi} \\
&= \sum_{L_k \in \mathbb{L}} w_k \int_0^l \int_{\mathbb{R}^2} r(m^{-1}(\boldsymbol{\xi}) - \boldsymbol{u}_k(t))\ h(\boldsymbol{x} - \boldsymbol{\xi})\, d\boldsymbol{\xi}\ dt \quad . \\
&= \sum_{L_k \in \mathbb{L}} w_k \rho_k(\boldsymbol{x})
\end{aligned}
$$

$$\text{where:} \rho_k(\boldsymbol{x}) = \int_0^l \int_{\mathbb{R}^2} r(m^{-1}(\boldsymbol{\xi}) - \boldsymbol{u}_k(t))\ h(\boldsymbol{x} - \boldsymbol{\xi})\, d\boldsymbol{\xi}\ dt$$

Substitute the object space to screen space mapping $\boldsymbol{\xi} = m(\boldsymbol{u})$ into $\rho_k(\boldsymbol{x})$, yields

$$\rho_k(\boldsymbol{x}) = \int_0^l \int_{\mathbb{R}^2} r(\boldsymbol{u} - \boldsymbol{u}_k(t))\ h(\boldsymbol{x} - m(\boldsymbol{u}))\ \left|\frac{\partial m}{\partial \boldsymbol{u}}\right| d\boldsymbol{u}\ dt$$

where: $\left|\frac{\partial m}{\partial \boldsymbol{u}}\right|$ is the determinant of Jocabian matrix $\frac{\partial m}{\partial \boldsymbol{u}}$.

The general forward mapping $m(\boldsymbol{u})$ in $\rho_k(\boldsymbol{x})$ can then be replaced locally by $m_{(k,t)}(\boldsymbol{u})$ which is the forward mapping defined at every point $\boldsymbol{p}_k(t)$ of line L_k. This gives us

$$\rho_k(\boldsymbol{x}) = \int_0^l \int_{\mathbb{R}^2} r(\boldsymbol{u} - \boldsymbol{u}_k(t))\ h(\boldsymbol{x} - m_{(k,t)}(\boldsymbol{u}))\ \left| J_{(k,t)} \right| d\boldsymbol{u}\ dt$$

where: $J_{(k,t)} = \frac{\partial m_{(k,t)}}{\partial \boldsymbol{u}}$.

Use affine mapping assumption and local affine approximation assumption made, we can rewrite $\rho_k(\boldsymbol{x})$ as bellowing,

$$\begin{aligned}
\rho_k(\boldsymbol{x}) &= \int_0^l \int_{\mathbb{R}^2} r(\boldsymbol{u} - \boldsymbol{u}_k(t))\ h'(l^{-1}(\boldsymbol{x} - m_{(k,t)}(\boldsymbol{u}))\, d\mathbf{u}\ dt \\
&= \int_0^l \int_{\mathbb{R}^2} r(\boldsymbol{u} - \boldsymbol{u}_k(t))\ h'(m_{(k,t)}^{-1}(\boldsymbol{x}) - \boldsymbol{u})\, d\boldsymbol{u}\ dt \\
&= \int_0^l \rho'_{(k,t)}(\, m_{(k,t)}^{-1}(\boldsymbol{x}) - \boldsymbol{u}_k(t))\, dt
\end{aligned}$$

where: $h'(\boldsymbol{u}) = \left| J_{(k,t)} \right| h(J_{(k,t)} \cdot \boldsymbol{u})$

where: $\rho'_{(k,t)}(\boldsymbol{u}) = h'(\boldsymbol{u}) \otimes r(\boldsymbol{u})$

We then replace the point reconstruction filter and prefilter with Gaussian kernels, $r(\boldsymbol{u}) = g_{V_r}(\boldsymbol{u})$ and $h(\boldsymbol{x}) = g_{V_h}(\boldsymbol{x})$ where V_r and V_h are the variances of these Gaussians. We get

$$
\begin{aligned}
h^{'}(\boldsymbol{u}) &= \left|J_{(k,t)}\right| h(J_{(k,t)} \cdot \boldsymbol{u}) \\
&= \left|J_{(k,t)}\right| g_{V_h}(J_{(k,t)} \cdot \boldsymbol{u}), \\
&= g_{J_{(k,t)}^{-1} V_h J_{(k,t)}^{-1^T}}(\boldsymbol{u})
\end{aligned}
$$

and

$$
\begin{aligned}
\rho_{(k,t)}^{'}(\boldsymbol{u}) &= h^{'}(\boldsymbol{u}) \otimes r(\boldsymbol{u}) \\
&= g_{J_{(k,t)}^{-1} V_h J_{(k,t)}^{-1^T}}(\boldsymbol{u}) \otimes g_{V_r}(\boldsymbol{u}). \\
&= g_{J_{(k,t)}^{-1} V_h J_{(k,t)}^{-1^T} + V_r}(\boldsymbol{u})
\end{aligned}
$$

Now substitute our newly derived $\rho_{(k,t)}^{'}(\boldsymbol{u})$ back into $\rho_k(\boldsymbol{x})$, we obtain the following equivalence relation,

$$
\begin{aligned}
\rho_k(\boldsymbol{x}) &= \int_0^l g_{J_{(k,t)}^{-1} V_h J_{(k,t)}^{-1^T} + V_r}(m_{(k,t)}^{-1}(\boldsymbol{x}) - \boldsymbol{u}_k(t))\, dt \\
&= \int_0^l g_{J_{(k,t)}^{-1} V_h J_{(k,t)}^{-1^T} + V_r}(\boldsymbol{u} - \boldsymbol{u}_k(t))\, dt \qquad . \\
&= \rho_k^{'}(\boldsymbol{u})
\end{aligned}
$$

Thus we have formulated $\rho_k^{'}(\boldsymbol{u})$ – the object space EWA resampling filter of line segment L_k . $\rho_k^{'}(\boldsymbol{u})$ is combined from reconstruction filter $r(\boldsymbol{u})$ and L_k ’s warped prefilter $\int_0^l g_{J_{(k,t)}^{-1} V_h J_{(k,t)}^{-1^T}}(\boldsymbol{u})\, dt$ which is referred as line segment’s field function with Gaussian kernel in the area of convolution surface [Sher99].

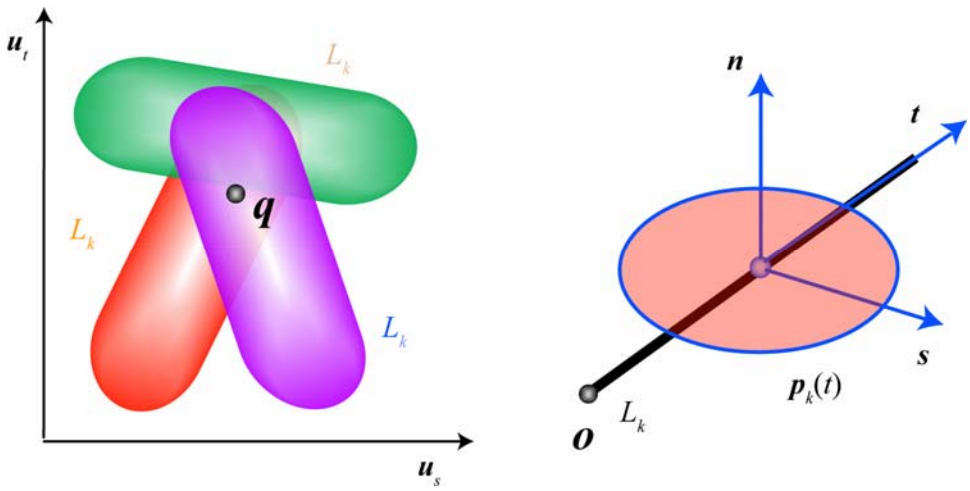

Figure 9: *The left part illustrates the local parameterization around any object surface point* $\boldsymbol{q}$ *, where the line segments* $L_k \in \mathbb{L}$ *are lying in* $\boldsymbol{q}$ *'s neighbourhood. And the right figure illustrates the local parameterization of a point* $\boldsymbol{p}_k(t) \in L_k$ *. The blue circle denotes he boundary of* $\boldsymbol{p}_k(t)$ *'s neighbourhood.*

5.1.2. Evaluation

To evaluate line segment L_k's resampling filter $\rho_k^{'}(\boldsymbol{u})$, we need to determine the Jacobian matrix bound with every point $\boldsymbol{p}_k(t)$ first.

We construct a local parameterization for each point $\boldsymbol{p}_k(t)$. It is the tangent plane given by line segment L_k 's normal $\boldsymbol{n}_k$ where the neighborhoods of all the points $\boldsymbol{p}_k(t)$ in line segment L_k lie on. For convenience we use $\boldsymbol{n}$ to denote $\boldsymbol{n}_k$ in subsequent discussions. Use $\boldsymbol{s}$ and $\boldsymbol{t}$ denote the pair of orthogonal basis vectors of the point $\boldsymbol{p}_k(t)$'s 2D neighborhood. We always align $\boldsymbol{t}$ with the line segment L_k, $\boldsymbol{t}$ pointing from L_k's starting endpoint $\boldsymbol{o}$ to another endpoint. Having $\boldsymbol{s}$ perpendicular to both $\boldsymbol{t}$ and $\boldsymbol{n}$, the triple $(\boldsymbol{s},\boldsymbol{t},\boldsymbol{n})$ thus forms a right hand local 3D coordinate system centered at $\boldsymbol{p}_k(t)$ (See Figure 9) .

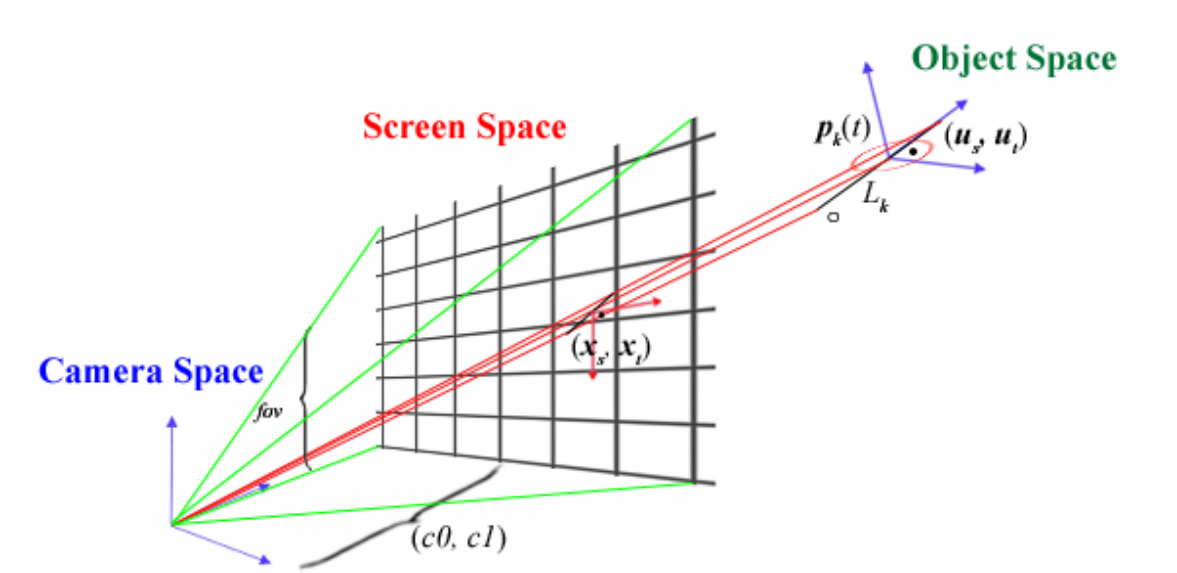

Figure 10: *Computation of the Jacobian matrix* $J_{(k,t)}$ *for point* $\boldsymbol{p}_k(t)$.

A point at $\boldsymbol{u}$ with local coordinate (u_s, u_t) in $\boldsymbol{p}_k(t)$' neighborhood thus corresponds to a point $\boldsymbol{p}_k^o(\boldsymbol{u}) = (\boldsymbol{o} + t \cdot \boldsymbol{t}) + u_s \cdot \boldsymbol{s} + u_t \cdot \boldsymbol{t}$ in object space. Make the assumptions that the object space to camera space mapping only contains uniform scaling $\mathbf{S}$, rotation $\mathbf{R}$, and translation $\mathbf{T}$, this is similar to what Liu has done [RPZ02]. Then the corresponding point in camera space is

$$\begin{aligned}
\boldsymbol{p}_k^c(\boldsymbol{u}) &= \mathbf{R} \cdot \mathbf{S} \cdot \boldsymbol{p}_k^o(\boldsymbol{u}) + \mathbf{T} \\
&= (\mathbf{R} \cdot \mathbf{S} \cdot (\boldsymbol{o} + t \cdot \boldsymbol{t}) + \mathbf{T}) + u_s \cdot \mathbf{R} \cdot \mathbf{S} \cdot \boldsymbol{s} + u_t \cdot \mathbf{R} \cdot \mathbf{S} \cdot \boldsymbol{t} \\
&= \mathbf{R} \cdot \mathbf{S} \cdot \boldsymbol{o} + \mathbf{T} + t \cdot \mathbf{R} \cdot \mathbf{S} \cdot \boldsymbol{t} + u_s \cdot \mathbf{R} \cdot \mathbf{S} \cdot \boldsymbol{s} + u_t \cdot \mathbf{R} \cdot \mathbf{S} \cdot \boldsymbol{t}. \\
&= \tilde{\boldsymbol{o}} + u_s \tilde{\boldsymbol{s}} + (t + u_t) \tilde{\boldsymbol{t}}
\end{aligned}$$

where: $\tilde{\boldsymbol{o}} = (\mathbf{R} \cdot \mathbf{S} \cdot \boldsymbol{o} + \mathbf{T})$; $\tilde{\boldsymbol{s}} = \mathbf{R} \cdot \mathbf{S} \cdot \boldsymbol{s}$; $\tilde{\boldsymbol{t}} = \mathbf{R} \cdot \mathbf{S} \cdot \boldsymbol{t}$

More specifically, the 3D vectors $\tilde{\boldsymbol{o}} = (o_x, o_y, o_z)$, $\tilde{\boldsymbol{s}} = (s_x, s_y, s_z)$, and $\tilde{\boldsymbol{t}} = (t_x, t_y, t_z)$ in camera space. (See Figure 10)

Next we project the camera space point $\boldsymbol{p}_k^c(\boldsymbol{u})$ to screen space point $\boldsymbol{x}$. This includes the projection by perspective division, followed by a scaling with a factor η [RPZ02].

$$\eta = \frac{v_h}{2\tan(\frac{fov}{2})}$$

where: v_h is viewport height .

fov is the field of view of the view frustum

The coordinate of $\boldsymbol{x} = (x_s, x_t)$ is

$$x_s = \eta \cdot \frac{o_x + u_s s_x + (t + u_t)t_x}{o_z + u_s s_z + (t + u_t)t_z} + c_0$$

$$x_t = -\eta \cdot \frac{o_y + u_s s_y + (t + u_t)t_y}{o_z + u_s s_z + (t + u_t)t_z} + c_1 \quad .$$

where: c_0 and c_1 are translation constants

Since every point $\boldsymbol{p}_k(t)$ itself is the center of its own neighborhood, the local coordinate of $\boldsymbol{p}_k(t)$ is thus $(0,0)$. The Jacobian matrix $J_{(k,t)}$ at point $\boldsymbol{p}_k(t)$ consists of partial derivates evaluated at $(0,0)$

$$J_{(k,t)} = \begin{bmatrix} \partial x_s / \partial u_s & \partial x_s / \partial u_t \\ \partial x_t / \partial u_s & \partial x_t / \partial u_t \end{bmatrix}$$

$$= \eta \cdot \frac{1}{(o_z + t \cdot t_z)^2} \begin{bmatrix} s_x o_z - s_z o_x + t(s_x t_z - s_z t_x) & t_x o_z - t_z o_x \\ s_z o_y - s_y o_z + t(s_z t_y - s_y t_z) & t_z o_y - t_y o_z \end{bmatrix} .$$

Once we obtain $J_{(k,t)}$, we then can derive $J_{(k,t)}^{-1}$ and $J_{(k,t)}^{-1^T}$. To simplify subsequent formulas, we let

$$
\begin{aligned}
e_J &= \eta \cdot \frac{1}{(o_z + t \cdot t_z)^2}(s_x o_z - s_z o_x + t(s_x t_z - s_z t_x)) \\
f_J &= \eta \cdot \frac{1}{(o_z + t \cdot t_z)^2}(t_x o_z - t_z o_x) \\
g_J &= \eta \cdot \frac{1}{(o_z + t \cdot t_z)^2}(s_z o_y - s_y o_z + t(s_z t_y - s_y t_z)) \\
h_J &= \eta \cdot \frac{1}{(o_z + t \cdot t_z)^2}(t_z o_y - t_y o_z)
\end{aligned}
.
$$

With e_J, f_J, g_J and h_J substituted, we have followings

$$
\begin{aligned}
J_{(k,t)} &= \begin{bmatrix} e_J & f_J \\ g_J & h_J \end{bmatrix} \\
J_{(k,t)}^{-1} &= \frac{1}{e_J h_J - g_J h_J}\begin{bmatrix} h_J & -f_J \\ -g_J & e_J \end{bmatrix} \\
J_{(k,t)}^{-1^T} &= \frac{1}{e_J h_J - g_J h_J}\begin{bmatrix} h_J & -g_J \\ -f_J & e_J \end{bmatrix}
\end{aligned}
.
$$

Recall previously we have the resampling filter

$$
\begin{aligned}
\rho_k(\boldsymbol{x}) &= \int_0^l g_{J_{(k,t)}^{-1} V_h J_{(k,t)}^{-1^T} + V_r}(m_{(k,t)}^{-1}(\boldsymbol{x}) - \boldsymbol{u}_k(t))\, dt \\
&= \int_0^l g_{J_{(k,t)}^{-1} V_h J_{(k,t)}^{-1^T} + V_r}(\boldsymbol{u} - \boldsymbol{u}_k(t))\, dt \\
&= \rho_k^{'}(\boldsymbol{u})
\end{aligned}
.
$$

For simplicity reason, let's assign $V_{(k,t)} = J_{(k,t)}^{-1} V_h J_{(k,t)}^{-1^T} + V_r$ and $\ell = m_{(k,t)}^{-1}(\boldsymbol{x}) - \boldsymbol{u}_k(t)$, we then have $\rho_k(\boldsymbol{x}) = \int_0^l g_{V_{(k,t)}} \ell \, dt$. Quite often we use identity matrix for the variance matrices of both reconstruction filter and prefilter [Heck89, ZPVG01, RPZ02], $V_r = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $V_h = c_h \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Then we get

$$\begin{aligned} V_{(k,t)} &= J_{(k,t)}^{-1} V_h J_{(k,t)}^{-1^T} + V_r \\ &= \frac{1}{(e_J h_J - g_J f_J)^2} \begin{bmatrix} h_J{}^2 + f_J{}^2 + c_h & -g_J h_J - e_J f_J \\ -g_J h_J - e_J f_J & e_J{}^2 + g_J{}^2 + c_h \end{bmatrix}. \end{aligned}$$

To keep formulas simple and concise, we let $a = \frac{h_J{}^2 + f_J{}^2 + c_h}{(e_J h_J - g_J f_J)^2}$, $b = \frac{-g_J h_J - e_J f_J}{(e_J h_J - g_J f_J)^2}$ and $d = \frac{e_J{}^2 + g_J{}^2 + c_h}{(e_J h_J - g_J f_J)^2}$. We then have $V_{(k,t)} = \begin{bmatrix} a & b \\ b & d \end{bmatrix}$ and

$$\begin{aligned} \rho_k(\boldsymbol{x}) = \rho_k^{'}(\boldsymbol{u}) &= \int_0^l g_{V_{(k,t)}} \ell \, dt \\ &= \int_0^l \frac{1}{2\pi \left| V_{(k,t)} \right|^{\frac{1}{2}}} \exp^{-\frac{1}{2}\ell^T V_{(k,t)}^{-1} \ell} \, dt \\ &= c_{(k,t)} e^{d_{(k,t)} \ell^T \begin{bmatrix} d & -b \\ -b & a \end{bmatrix} \ell} \end{aligned}.$$

where: $c_{(k,t)} = \frac{1}{2\pi \left| V_{(k,t)} \right|^{\frac{1}{2}}}$

$$d_{(k,t)} = -\frac{1}{2(ad - b^2)}$$

$$h_{(k,t)} = \ell^T V_{(k,t)}^{-1} \ell$$

Now, the formula of object space EWA resampling filter for every sampled surface line L_k is of the form $\rho_k(\boldsymbol{x}) = \rho_k^{'}(\boldsymbol{u}) = \int_0^l c_{(k,t)} e^{d_{(k,t)} \cdot h_{(k,t)}} dt$. The only unknown variable left is t and others are constants. We then expand the formulas all the way back, we find $d_{(k,t)} \cdot h_{(k,t)} = \frac{p(t)}{q(t)}$ where $p(t)$ and $q(t)$ are polynomials of degree 6. In general, an exponential whose exponent is a rational function cannot be integrated in closed form. Hence the resampling filter is of non-closed-from.

5.2. Object Space EWA Splatting for Line Segments

Although the straightforward implementation of the resampling filter's formula is unattainable, being aware of the existence of several geometric equivalences between Gaussian textures mapped polygons and Gaussian based filters through the rendering procedure, we propose an approximation method to splat line segment's resampling filter.

5.2.1. Splatting Procedure Illustrated

A line segment's prefilter on screen is integrated from circular Gaussians along the line segment. Theoretically, the prefilter has infinite support; however in practice only a limited range of Gaussian influence is computed. A cutoff radius R is applied to Gaussian function. Different from the screen space to object space mapping $\boldsymbol{u} = m^{-1}(\boldsymbol{x})$ used in the warping stage of point rendering, which is assumed to be affine, we perspective project a line segment back to object space. However, it is the case that we still keep the affine assumptions for the mappings in the neighborhoods of points in the line segment.

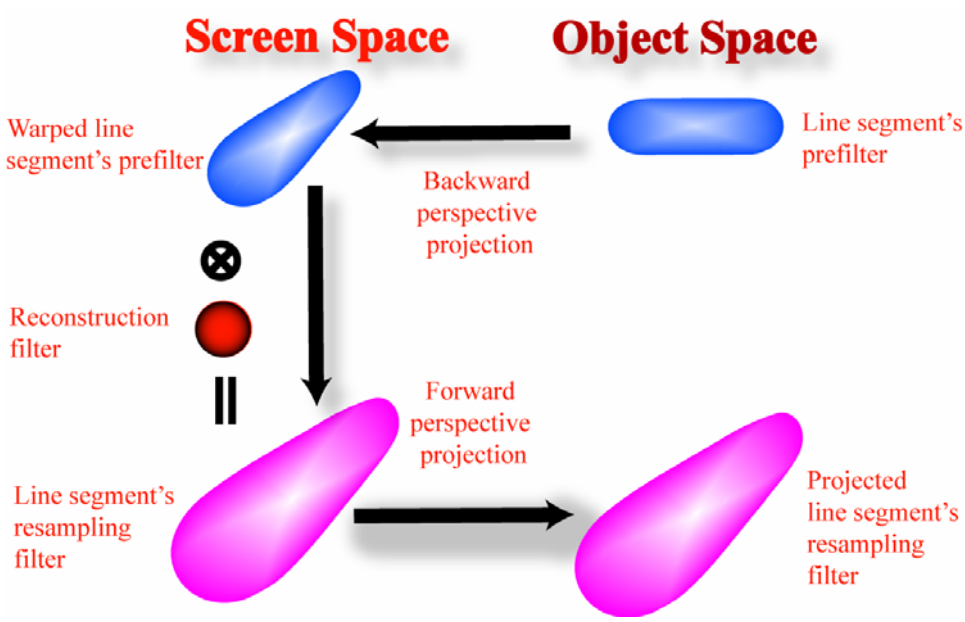

Figure 11: *Object space EWA splatting procedure for line segments.*

Following warping, we convolute the warped prefilter for the line segment with the reconstruction filter which is a circular Gaussian as well, resulting in the splatted resampling filter for line segments in object space. As the same case for the prefilter, the reconstruction filter is also cut within a limited distance. Finally we forward perspective project the line segment splat onto screen. (See Figure 11)

5.2.2. Geometric Observations

The prefilter is characterized by line segment's field function with Gaussian kernel. Examining the formula as well as the graphs plotted, it turns out that the prefilter can essentially be sliced into three portions in terms of its weight distribution. This includes two endpoint portions, and one middle portion. While the endpoint portions are of circular shapes, the middle portion consists of parallel straight line segments connecting endpoint portions. Inside the prefilter, locations that receive the same amount of influences form

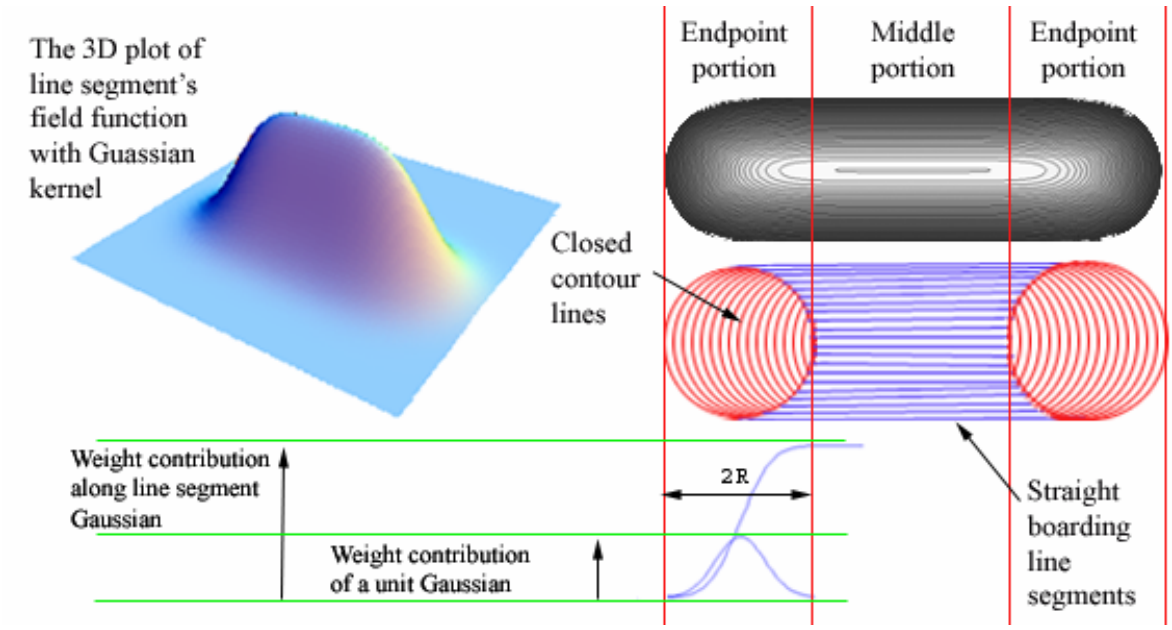

Figure 12: *Geometric observations on line segment's prefilter.*

into closed contour lines. As Gaussians are cut with R, therefore the boundary of the prefilter is enclosed. Geometrically speaking, it is the two most outward line segments of the middle portion and circular arcs of the endpoint portions being tied up together into the prefilter's exterior border. In addition, the two boundary line segments are tangent to both circles. The weight contributions received by points along the line segment are higher than those received by points within endpoint circles. (See Figure 12)

Since we perspective project the prefilter when carrying out warping operation, the straight line segments in the middle portion remain straight in object space, however endpoint circles would be deformed into distorted ellipses. Geometric analysis show that the boundary of the warped prefilter keeps in closed and the two bordering line segments stay tangent to distorted ellipses centered at endpoints [Wong03].

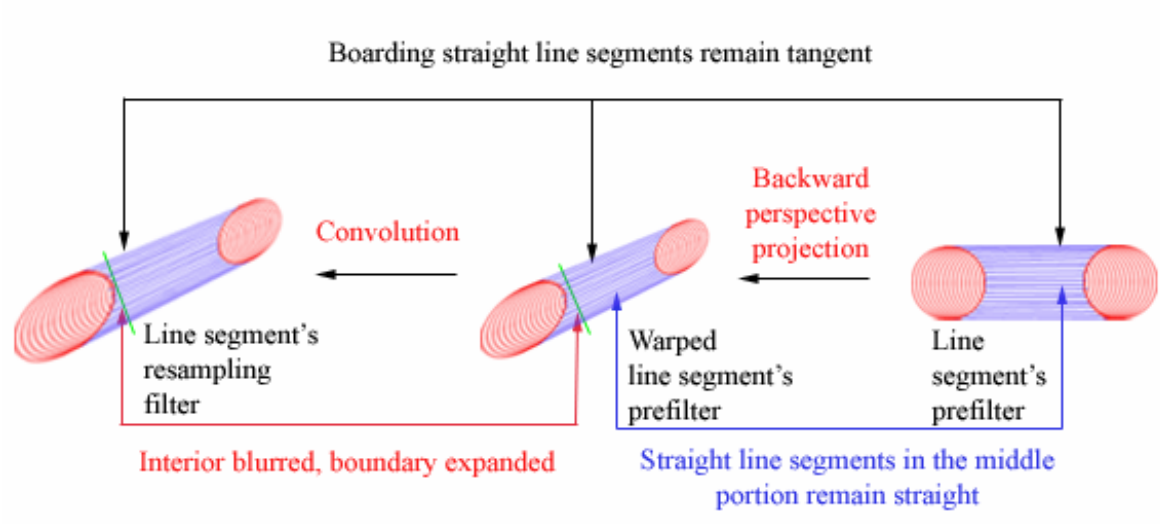

Figure 13: *Geometric observations on the splatting procedure of line segments.*

Convoluting with the reconstruction filter expands and blurs the warped prefilter, yielding the resampling filter. The border of the warped prefilter uniformly stretches out a distance of cutoff radius R. As a result, the closure property of filter's boundary is still preserved; bordering line segments remain tangent lines. (See Figure 13)

5.2.3. The Approximation Method

5.2.3.1. Approximating The Shape

We find endpoints' resampling filters' shapes in object space first. With affine mapping assumption and local affine approximation assumption made around endpoints' neighborhoods, it is the case that endpoints' resampling filters are of elliptical shape, because we use circular Gaussian as point's prefilter on screen. Ren *et al.* propose a method to compute it [RPZ02]. Recall that the formula for point's object space EWA resampling filter is $\rho_k^{'}(\boldsymbol{u}) = g_{V_k^r + J_k^{-1} V_k^h J_k^{-1^T}}(\boldsymbol{u})$. Let $M_k = V_k^r + J_k^{-1} V_k^h J_k^{-1^T}$. We use the method

discussed in 5.1.2 to find endpoints' Jacobian matrices, hence determining M_k. Since in practice we use identity matrix for both the reconstruction filter and prefilter's variance matrices, $V_k^r = I$ and $V_k^h = I$, the matrix M_k is thus a symmetric matrix, and it can be decomposed as following,

$$M_k = Rot(\theta) \cdot \Lambda \cdot \Lambda \cdot Rot(\theta)^T$$

$$\text{where: } Rot(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}.$$

$$\Lambda = \begin{bmatrix} r_0 & 0 \\ 0 & r_1 \end{bmatrix}$$

The rotation matrix $Rot(\theta)$ consists of eigenvectors, and scaling matrix Λ consists of square roots of eigenvalues of M_k. Let $\boldsymbol{u} = Rot(\theta) \cdot \Lambda \cdot \boldsymbol{y}$, which is a linear relationship, we then get $\boldsymbol{y}^T \boldsymbol{y} = \boldsymbol{u}^T M_k^{-1} \boldsymbol{u}$ and the following equation,

$$g_{M_k}(Rot(\theta) \cdot \Lambda \cdot \boldsymbol{y}) = \frac{1}{2\pi \cdot r_0 r_1} e^{-\frac{1}{2}\boldsymbol{y}^T \boldsymbol{y}} = \frac{1}{r_0 r_1} g_I(\boldsymbol{y}).$$

The above suggests that we can obtain an endpoint' resampling filter' elliptical shape by scaling a circle with radius R, r_0 and r_1 in its two orthogonal basis vectors' directions respectively, followed by a counter clockwise rotation of angle θ. Notice that the two ellipses centered at the line segment's endpoints and lie on the plane which is given by the line segment's normal $\boldsymbol{n}_k$.

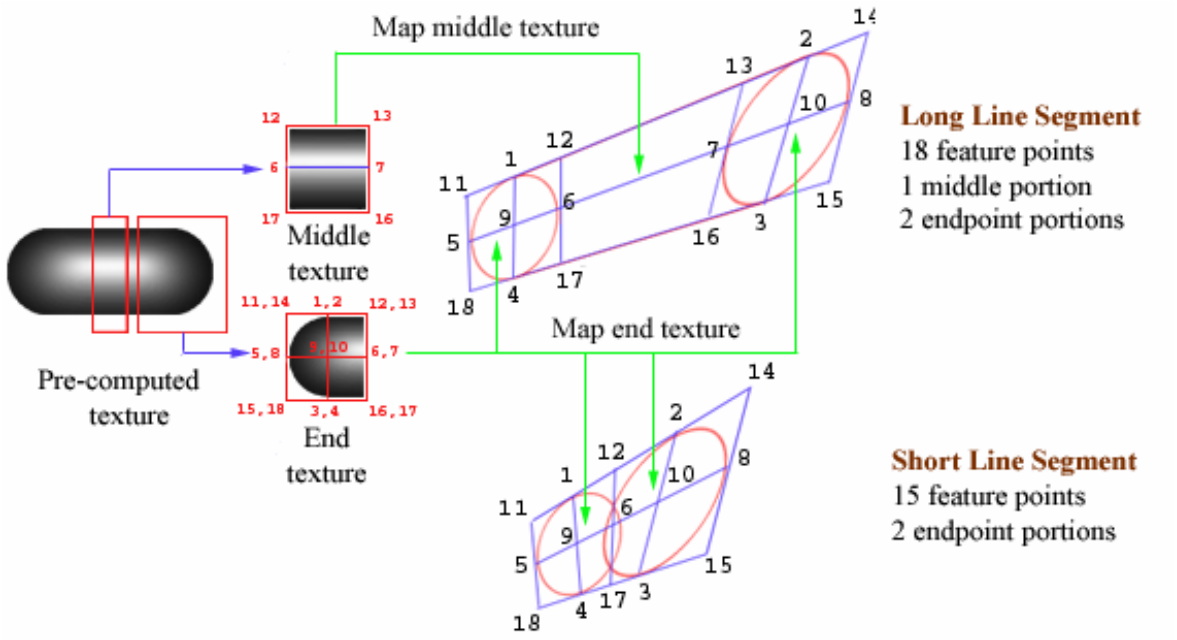

Figure 14: *Long line segment, short line segment and weight texture mapping.*

Next we determine the line segment's resampling filter's bordering line segments. The two boundary line segments are tangent to the ellipses centered at endpoints; this is one of our geometric observations. Wong proposes a method to compute them out by solving this 2D geometry problem in its dual space [Wong03]. Let us number the points where the boundary line segments meet the ellipses 1, 2, 3 and 4. For convenience, we call the two tangent line segments, line 1-2 and line 3-4 afterwards. The line segments that connect tangent points on the same ellipse are referred as crossing line segments; they are line 1-4 and line 2-3 respectively. Line 1-4 intersects with the actual surface line segment at point 9, and point 10 for line 2-3, forming the line 9-10 (See Figure 14).

So far, we only get the boundary of the line segment's resampling filter. From the geometric observations, we know that the middle portion of the resampling filter is made up from Gaussian blurred straight line segments, and endpoint portions are accumulated

from sequences of circular Gaussians. Therefore before we map weight textures to the resampling filter, we need to further cut down the filter's shape. There are two scenarios, depending on whether the two endpoint ellipses intersect with each other or not. If the ellipses overlap, we call it short line segment case; otherwise it is the long line segment case.

In either case, we extend the line 9-10 further longer, obtaining a set of intersection points between the line 9-10 and the two endpoint ellipses. For the long line segment case, we record all the intersection points 5, 6, 7 and 8. For the short line segment case, we only keep intersection points 5 and 8. Next, we compute out a set of straight lines that are tangent to the endpoint ellipses at these intersection points. We then intersect these newly computed tangent lines with line 1-2 and line 3-4, slicing the resampling filter into a number of connected quadrilaterals. For a long line segment case, we have four such newly computed tangent lines. They are line 11-18, line 12-17, line 13-16 and line 14-15. The resampling filter is split into 10 small quadrilaterals. The quadrilateral 11-12-17-18 and 13-14-15-16 bound the line segment's two endpoint portions respectively, while quadrilateral 12-13-16-17 bounds the middle portion. For a short line segment case, we have only two such newly computed tangent lines. They are line 11-18 and line 14-15. Since the two endpoint ellipses intercross, this implies that the surface line segment may be too short such that its middle portion vanishes away, only left with endpoint portions. Thus one extra line is needed to separate the two endpoint portions. We choose the line that connects the intersection points between the two ellipses. This separating line crosses the boarding tangent line segments line 1-2 and line 3-4 at point 12 and 17 respectively. The resampling

filter is thus decomposed into 8 small quadrilaterals. Endpoint portions are bounded by quadrilateral 11-12-17-18 and 12-14-15-17 (see Figure 14).

5.2.3.2. Mapping Weight Textures

The weight texture is generated using line segment field function with Gaussian kernel of cutoff radius R. Let us place a line segment L of length l, where $l > 2R$, in a 2D coordinate system, with L's starting point $\boldsymbol{o}$ coincide with the 2D coordinate system center. We align the line segment along the positive $\boldsymbol{x}$ axis. We compute the following,

$$T_1(x,y) = \frac{1}{2\pi}\sqrt{\frac{\pi}{2}}e^{-\frac{1}{2}y^2}\left(erf\left(\sqrt{\frac{1}{2}(R^2 - y^2)}\right) + erf\left(\frac{1}{\sqrt{2}}x\right)\right)$$

$$T_2(x,y) = \frac{1}{\pi}\sqrt{\frac{\pi}{2}}e^{-\frac{1}{2}y^2}\left(erf\left(\sqrt{\frac{1}{2}(R^2 - y^2)}\right)\right)$$

Here, x gives the $\boldsymbol{x}$ coordinate value. It measures the distance to L's starting point $\boldsymbol{o}$. While y gives the $\boldsymbol{y}$ coordinate value. It is in the direction perpendicular to the line segment L. *erf* denotes the error function. The result of the integration of unit Gaussians from $[0, r]$ is the weight texture for endpoint portion, given by T_1 when $x^2 + y^2 \le R^2$, and by T_2 when $x^2 + y^2 > R^2$, $0 < x < R$ and $-R < y < R$; while the result of the integration of unit Gaussians from $(r, l-r)$ is the weight texture for middle portion, given by T_2 when $R < x < l - R$ and $-R < y < R$.

Now, the task of mapping weight texture to resampling filter becomes mapping weight textures belonging to different portions to corresponding bounding quadrilaterals. For a

long line segment, we need to map the weight texture for the middle portion to the quadrilateral 12-13-16-17, and the weight texture for the endpoint portion to quadrilateral 11-12-17-18 and 13-14-15-16. For a short line segment, we only need to map the texture for the endpoint portion to quadrilateral 11-12-17-18 and 12-14-15-17. Note that, to further reduce the texture mapping errors caused by the affine mapping assumption, we can divide a piece of weight texture into 2 or 4 smaller pieces, and map them to those smaller quadrilaterals, such as the quadrilateral 11-1-9-5 bounded in 11-12-17-18. However we learn from practices that this is only necessary when the line segments are rendered very close to the eyes.

5.2.3.3. Assigning Scaling Factors

Lastly, notice that in the equation $g_{M_k}(Rot(\theta)\cdot\Lambda\cdot\boldsymbol{y}) = \frac{1}{2\pi\cdot r_0\, r_1} e^{-\frac{1}{2}\boldsymbol{y}^T\boldsymbol{y}} = \frac{1}{r_0\, r_1} g_I(\boldsymbol{y})$, we have a scaling factor $\frac{1}{r_0\, r_1}$ in front of the unit Gaussian function, this means we need to scale the basis function for every point along the line segment, before we accumulate them together. By tracing the formulas derived in Section 5.1.2, we find that the scaling factor is non-linear along the line segment. However, to facilitate simplicity and efficiency of implementation, we approximate the scaling factor linearly along the line segment. We compute $\frac{1}{r_0\, r_1}$ at the line segment's two endpoints explicitly, and interpolate it for the remaining points in the line segment. Our results show that there is nearly no visual artifact on account of this approximation approach.

Figure 15: *On the left is a textured line segment plan. In the middle is a cylindrical surface mapped with a cat image. On the right is a line segment human head model rendered with a human face image mapped. This line segment head model is obtained using our contour plane based line segment extraction algorithm.*

5.2.4. Rendering Texture Mapped Line Segments

As what we have already discussed in Section 3.3, we only sample image texture coordinates at the two endpoints of each line segment. Similar to mapping weight scaling factors discussed in Section 5.2.3.3, we linearly interpolate the texture coordinates along the line segment as well. Linearly interpolating texture coordinates may cause some visible artifacts when sampled line segments are too long. We notice that texturing short line segments would result less interpolation errors. Generally, this linear interpolation method maps texture images to line segment models well (see Figure 15).

Chapter 6.

Implementations

6.1. Surface Geometry Processing Pipeline

The surface geometry processing pipeline serves for our (ε, δ) error bounded line segment extraction algorithm in Section 3.1 and the shape approximation based line segment extraction algorithm in Section 3.2. Both of these two line segment extraction algorithms target at obtaining surface line segments from scanned point clouds. However a discrete set of scanned points convey zero connectivity information. Without even knowing the local surfaces nearby the points, none of the surface geometry information needed by the line segment extraction algorithms can possibly be provided. Thus the tasks of this surface geometry processing pipeline are first, establishing the neighbouring connectivity information among the scanned points. Second, compute out the basic surface geometry information required by the line segment extraction algorithms, in particular the surface normal and curvature.

This subsection is organized in such way: we first go through each computation stage in the geometry processing pipeline one by one. We then devote our discussions to the normal and curvature estimation problem. Lastly we talk out some implementation problems and performance issues.

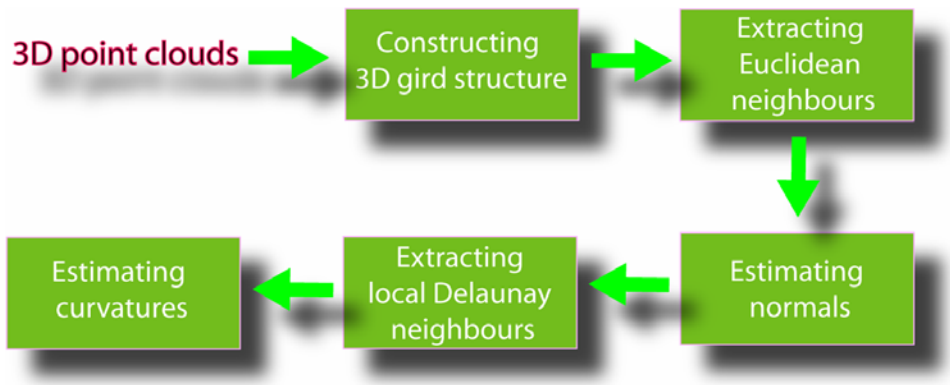

Figure 16: *The geometry processing pipeline.*

6.1.1. Geometry Processing Pipeline Illustrated

The geometry processing pipeline consists of five stages (see Figure 16). They are constructing 3D grids structure, extracting Euclidean neighbours, estimating normals, extracting local Delaunay neighbours and estimating curvatures. The pipeline takes a scanned point cloud $\mathcal{P} = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3 .., \boldsymbol{p}_n\}$ as input.

Constructing 3D Grid Structure. In this stage we build a simple 3D grid structure G^3 over the point cloud $\mathcal{P}$. G^3 serves as a global spatial data structure, providing location information for $\mathcal{P}$. We first find the minimum axis aligned bounding box B of the points in $\mathcal{P}$. B gives the size of G^3. We then split G^3 uniformly into $N \times N \times N$ small cubic grids. The value of N depends on the number of points in $\mathcal{P}$. If $\|\mathcal{P}\|$ is large, we would then also set N to be bigger so as to keep the average number of points in the cubic grid sufficiently small. For each $\boldsymbol{p}_i \in \mathcal{P}$, we determine the cubic gird $g(\boldsymbol{p}_i) \in G^3$ it lies in. For

each cubic gird $g_{r,s,t} \in G^3$, we keep track of all the points in $\mathcal{P}$ that locate within it, $\mathcal{P}(g_{r,s,t})$.

Extracting Euclidean Neighbours. To establish connectivity information among points in $\mathcal{P}$, we try to find the nearest k points $\mathcal{N}_e(\boldsymbol{p}_i)$ for each $\boldsymbol{p}_i \in \mathcal{P}$. We first check whether $\|\mathcal{P}(g(\boldsymbol{p}_i))\| \geq k$. If this is the case then the nearest k points in $\mathcal{P}(g(\boldsymbol{p}_i))$ would be stored into $\mathcal{N}_e(\boldsymbol{p}_i)$. Otherwise, we expand the search space, by including the 26 neighbouring cubic grids $g_{a,b,c} \in G^3$ of $g(\boldsymbol{p}_i)$ as well. Let $n = 1$, $r(\boldsymbol{p}_i)$, $s(\boldsymbol{p}_i)$ and $t(\boldsymbol{p}_i)$ denote $g(\boldsymbol{p}_i)$'s indices to G^3. Here we have $|a - r(\boldsymbol{p}_i)| \leq n$, $|b - s(\boldsymbol{p}_i)| \leq n$ and $|c - t(\boldsymbol{p}_i)| \leq n$. The search continues with the search space further expanded until the k nearest neighbouring points are found or $n > N$. Note $\mathcal{N}_e(\boldsymbol{p}_i)$ may not contain the actual k nearest neighbours, but so found k points in $\mathcal{N}_e(\boldsymbol{p}_i)$ could have already provided sufficient neighboring information to $\boldsymbol{p}_i$.

Estimating Normals. We then use the method in [HDD92] to estimate the normal $\boldsymbol{n}_i$ for each $\boldsymbol{p}_i$. We first compute the centroid of points in $\mathcal{N}_e(\boldsymbol{p}_i)$ as $\boldsymbol{c}_i = \frac{\sum_{\boldsymbol{p}_k \in \mathcal{N}_e(\boldsymbol{p}_i)} \boldsymbol{p}_k}{\|\mathcal{N}_d(\boldsymbol{p}_i)\|}$. We then compute the 3×3 covariance matrix $\mathcal{M}_{\text{cov},i}$ defined on $\mathcal{N}_e(\boldsymbol{p}_i)$, where $\mathcal{M}_{\text{cov},i} = \sum_{\boldsymbol{p}_k \in \mathcal{N}_e(\boldsymbol{p}_i)} (\boldsymbol{p}_k - \boldsymbol{p}_i) \cdot (\boldsymbol{p}_k - \boldsymbol{p}_i)^T$. Let λ_0, λ_1 and λ_2 be the three eigenvalues of $\mathcal{M}_{\text{cov},i}$ ordered as $\lambda_0 \leq \lambda_1 \leq \lambda_2$. Let $\boldsymbol{v}_0$, $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ be the three eigenvectors corresponding to λ_0, λ_1 and λ_2. Then we have $\boldsymbol{n}_i = +\boldsymbol{v}_0$ or $\boldsymbol{n}_i = -\boldsymbol{v}_0$. To fix the direction of $\boldsymbol{n}_i$, as suggest by

Hoppe *et al.*, we find the a point $\boldsymbol{p}_z \in \mathcal{P}$ that has maximum z coordinate first. The direction of $\boldsymbol{p}_z$'s normal $\boldsymbol{n}_z$ can be fixed easily as $\boldsymbol{n}_z$ always points outwards from $\mathcal{P}$. Then we construct a minimum spanning tree (MST) with every pair of neighbouring points $(\boldsymbol{p}_i, \boldsymbol{p}_k)$ as an edge of the MST, where $\boldsymbol{p}_k \in \mathcal{N}_d(\boldsymbol{p}_i)$. And $1-\left|\boldsymbol{n}_i \cdot \boldsymbol{n}_j\right|$ is assigned as the weight of $(\boldsymbol{p}_i, \boldsymbol{p}_k)$. Thus the correct normal direction of $\boldsymbol{n}_z$ can get propagated over $\mathcal{P}$ through the MST, eventually fixing the direction of $\boldsymbol{n}_i$.

Extracting Local Delaunay Neighbours. With the centroid $\boldsymbol{c}_i$ and normal $\boldsymbol{n}_i$ known for each $\boldsymbol{p}_i$, we can fix the best fitting plan P_i of $\mathcal{N}_e(\boldsymbol{p}_i)$. P_i passes through $\boldsymbol{c}_i$ and perpendicular to $\boldsymbol{n}_i$. We then project all $\boldsymbol{p}_k \in \mathcal{N}_e(\boldsymbol{p}_i)$ onto P_i, obtaining a set of projected points $\mathcal{N}_e(\boldsymbol{p}_i^{'})$. Here $\boldsymbol{p}_i^{'}$ corresponds to $\boldsymbol{p}_i$, and each $\boldsymbol{p}_k^{'} \in \mathcal{N}_e(\boldsymbol{p}_i^{'})$ corresponds to the $\boldsymbol{p}_k \in \mathcal{N}_e(\boldsymbol{p}_i)$. We use the 2D delaunay triangulation algorithm provided by CGAL [CGAL04] to obtain the delaunay neighbors of $\boldsymbol{p}_i^{'}$ from $\mathcal{N}_e(\boldsymbol{p}_i^{'})$, denoted as $\mathcal{N}_d(\boldsymbol{p}_i^{'})$. We then collect $\mathcal{N}_d(\boldsymbol{p}_i^{'})$'s corresponding points in $\mathcal{N}_e(\boldsymbol{p}_i)$ into $\mathcal{N}_d(\boldsymbol{p}_i)$. Here $\mathcal{N}_d(\boldsymbol{p}_i)$ denotes the set of local delaunay neighbours of $\boldsymbol{p}_i$.

Estimating Curvatures. Normals are computed as the first derivatives of positions, while curvatures are the second derivatives. Therefore the accuracy of curvature estimation is problematic. In fact simply due to the unreliability of curvature information estimation, quite a number of promising alternative approaches that can be used to extract line

segments from point clouds turn to be difficult to implement. We discuss the curvature estimation problem in more details in the following subsection.

6.1.2. Curvature Estimation

Once the normal $\boldsymbol{n}_i$ have been estimated and the delaunay neighbours $\mathcal{N}_d(\boldsymbol{p}_i)$ have been found out, we then can start to estimate the minimum and maximum curvature vectors $\boldsymbol{k}_{\min}(\boldsymbol{p}_i)$ and $\boldsymbol{k}_{\max}(\boldsymbol{p}_i)$ for $\boldsymbol{p}_i$. We implement the method suggested in [Taub95]. We first compute the matrix $\mathcal{M}_{cur,i} = \sum_{\boldsymbol{p}_k \in \mathcal{N}_d(\boldsymbol{p}_i)} w_{i,k} \cdot k_{i,k} \cdot T_{i,k} \cdot T_{i,k}^{\ T}$. Here, $T_{i,k} = \frac{\boldsymbol{p}_i^{'} - \boldsymbol{p}_k^{'}}{\|\boldsymbol{p}_i^{'} - \boldsymbol{p}_k^{'}\|}$, $T_{i,k}$ is an unit vector pointing from $\boldsymbol{p}_k^{'}$ to $\boldsymbol{p}_i^{'}$. $k_{i,k} = \frac{2 \cdot \boldsymbol{n}_i \cdot (\boldsymbol{p}_k - \boldsymbol{p}_i)}{\|\boldsymbol{p}_k - \boldsymbol{p}_i\|^2}$, $k_{i,k}$ is the direction curvature along $T_{i,k}$. Let $\boldsymbol{p}_{i,k,s}^{'}, \boldsymbol{p}_{i,k,t}^{'} \in \mathcal{N}_d(\boldsymbol{p}_i^{'})$ denote the two projected points on P_i that are incident to both $\boldsymbol{p}_i^{'}$ and $\boldsymbol{p}_k^{'}$. Use $\Delta_{i,k,s}$ and $\Delta_{i,k,t}$ denote the triangles formed by $\boldsymbol{p}_i^{'}$, $\boldsymbol{p}_k^{'}$, $\boldsymbol{p}_{i,k,s}^{'}$ and $\boldsymbol{p}_i^{'}$, $\boldsymbol{p}_k^{'}$, $\boldsymbol{p}_{i,k,t}^{'}$ respectively. Then $w_{i,k} = \frac{area(\Delta_{i,k,s}) + area(\Delta_{i,k,t})}{\sum_{\boldsymbol{p}_k^{'} \in \mathcal{N}_d(\boldsymbol{p}_i^{'})} area(\Delta_{i,k,s}) + area(\Delta_{i,k,t})}$, $w_{i,k}$ measure the amount of contribution of $k_{i,k}$ to $\mathcal{M}_{cur,i}$.

We now construct the householder matrix $\mathcal{M}_h(\boldsymbol{n}_i)$ of $\boldsymbol{n}_i$. Thus from the second and third column of $\mathcal{M}_h(\boldsymbol{n}_i)$, we obtain two unit vector $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$. $\boldsymbol{x}_i$, $\boldsymbol{y}_i$ together with $\boldsymbol{n}_i$ form a 3D local coordinate system on P_i. Examining the construction function of $\mathcal{M}_{cur,i}$, we can see that $\boldsymbol{n}_i$ is an eigenvector of $\mathcal{M}_{cur,i}$. Thus we could have the following,

$$\mathcal{M}_h(\boldsymbol{n}_i)^T \cdot \mathcal{M}_{cur,i} \cdot \mathcal{M}_h(\boldsymbol{n}_i) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & m_{11,i} & m_{12,i} \\ 0 & m_{21,i} & m_{22,i} \end{pmatrix}.$$

Inside the 2×2 matrix $\begin{pmatrix} m_{11,i} & m_{12,i} \\ m_{21,i} & m_{22,i} \end{pmatrix}$, we have $m_{12,i} = m_{21,i}$. This matrix can be diagnoalized, giving a rotation angle θ. Thus the two principle curvature vectors are given by $\boldsymbol{k}_{1,i} = \cos(\theta)\boldsymbol{x}_i - \sin(\theta)\boldsymbol{y}_i$ and $\boldsymbol{k}_{2,i} = \sin(\theta)\boldsymbol{x}_i + \cos(\theta)\boldsymbol{y}_i$. And corresponding two principle curvature values are given by $k_{1,i} = 3m_{11,i} - m_{22,i}$ and $k_{2,i} = 3m_{22,i} - m_{11,i}$. Comparing the values of $k_{1,i}$ and $k_{2,i}$, we can then obtain the maximum principle curvature vector and value at $\boldsymbol{p}_i$, $\boldsymbol{k}_{\max}(\boldsymbol{p}_i)$ and $k_{\max}(\boldsymbol{p}_i)$. As well as the minimum principle curvature vector and value $\boldsymbol{k}_{\min}(\boldsymbol{p}_i)$ and $k_{\min}(\boldsymbol{p}_i)$.

As we have discussed, to our knowledge, there is no curvature estimation algorithm designed for point cloud data. Taubin's curvature estimation method [Taub95] is meant for polyhedral models. There are two reasons why we choose Taubin's method for implementation. First, [Taub95] gives a rigorous mathematical formulation of the curvature computation problem. Second, by reading the derivations and proofs in [Taub95], we find that Taubin's method is actually not that much dependent on the underlying polyhedral models. Only the weighting factor $w_{i,k}$ is determined from triangle areas. In fact it is quite adaptable to point cloud data.

We have also seen some other curvature estimation method on triangle or polyhedral meshes, such as [CoMo03]. Cohen-Steiner and Morvan's algorithm is quite widely accepted and recognized. Their method is used in [ACDL03] to trace curvature lines on

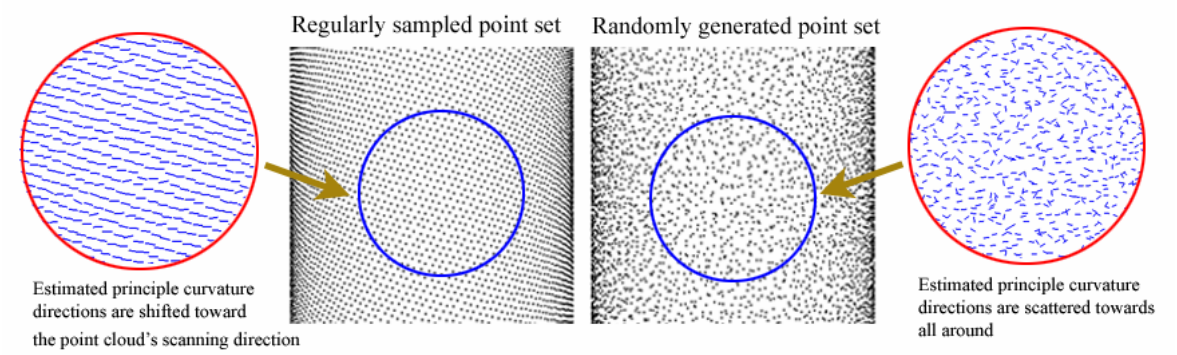

Figure 17: *Problems with estimating principle curvature vectors from point cloud data.*

triangle meshes. However in [ACDL03], Gaussian filters are still used to smooth the estimated principle curvature vectors to remove inaccuracies and noises.

It is not surprising to find out that the performance of the curvature estimation algorithms is quite dependant on the local surfaces that they are applied to. In our case of processing scanned point clouds, the input point cloud data's scanning direction and scanning pattern could affect the curvature estimation algorithm a lot. For some regularly sampled point clouds, estimated principle curvature vectors may all be shifted towards the point cloud's scanning direction (see Figure 17). Given a set of randomly sampled points on an elliptical cylinder surface, we can see the estimated principle curvature vectors are scattered towards all around, not really being aligned with the actual surface minimum curvature direction (see Figure 17).

6.1.3. Implementation Problems and Discussions

In the implementation, we find out that the value of k – the number of nearest Euclidean neighbours, is hard to be set properly. If k is too small, the connectivity information among points may only be partially explored. If k is too large, quite often the

neighbouring points counted would include scanned points locating at nearby surfaces as well, or even from opposite surfaces. And when k is large, estimated surface attributes would behave as if have been low pass filtered. This could result in removing too much surface geometry details and creating a lot of blurring in rendered surface images.

6.2. Surface Rendering Engine

In this subsection, we first talk about design considerations of our rendering engine's architecture. After that, we go through the entire rendering pipeline in the front-end to back-end order, stage by stage. Computation methods and algorithms used in each rendering step are discussed. We then give an extensive coverage of the visibility and blending algorithm in a separate subsection, where the modified Z^3 algorithm is presented and compared with delta-z-buffer algorithm. We end this section with a discussion on miscellaneous implementation problems we have encountered in the course of developing the engine.

6.2.1. Design Considerations

The surface rendering engine should support all the necessary fundamental 3D graphics rendering operations. It should support basic culling operations such as view frustum culling and back face culling. It should support basic transformation operations such as projective and affine mapping. It should also support basic lighting features, such as point light source and local illumination model [SeAk91].

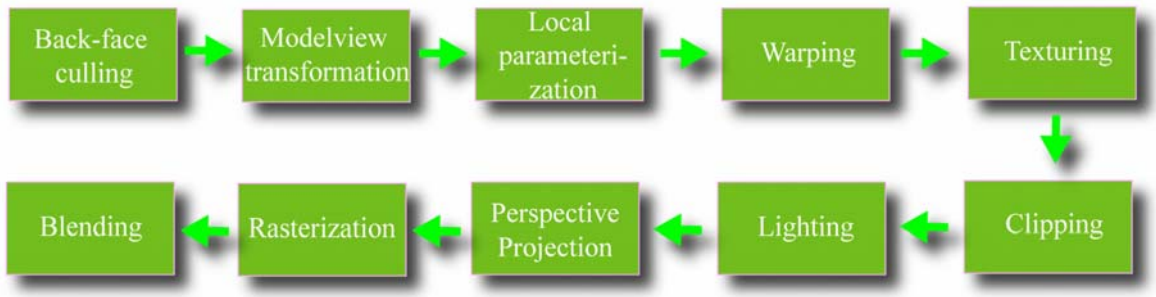

Figure 18: *The surface rendering pipeline.*

The surface rendering engine should support hybrid rendering, accepting a set of basic surface primitives. It should support direct rendering of both surface point primitive and line segment primitive. With our approximation method, while a point's resampling filter is only a weight texture mapped quadrilateral, 2 to 10 such quadrilaterals are used for one line segment's resampling filter. These make quadrilateral best primitive candidate for our pipeline. This is different from both the OpenGL pipeline which uses triangle as the basic pipeline primitive, as well as the Surfel rendering pipeline where point is the pipeline primitive.

6.2.2. Rendering Pipeline Illustrated

Altogether the rendering pipeline is consisting of 10 stages (See Figure 18). They are back-face culling, modelview transformation, local parameterization, warping, texturing, clipping, lighting, perspective projection, rasterization and blending. The pipeline takes either a surface point primitive or a surface line segment primitive as input. The data attributes associated with each point are its center, normal and color. For a line segment, this includes two endpoints, normal and color.

Back-face Culling. In the stage of back face culling, a surface primitive is culled away, only if it is facing back from the camera. This test is done by computing the dot product between the primitive's normal vector and a distance vector which points from the camera space center to either a point's center or a line segment's starting point.

Modelview Transformation. Next we multiply every surface primitive's normal, and center or endpoints with the modelview matrix. This transforms surface primitives from object space into camera space.

Local Parameterization. In the pipeline stage local parameterization, we construct a local 3D coordinate system for every surface primitive. For a point primitive, the center of the local coordinate system coincides with the point's center, and the $\boldsymbol{z}$ axis aligned with the point's normal. We use a simple yet efficient method similar to the householder transformation to compute the other two orthogonal basis vectors $\boldsymbol{x}$ and $\boldsymbol{y}$. Let $\boldsymbol{x} = (a, b, c)$, it works by checking three if conditions. If $a \neq 0$, then $\boldsymbol{x} = (b/a, -1, 0)$ and $\boldsymbol{y} = (c, (b \times c)/a, -a - (b \times b/a))$. If $a = 0$ and $b \neq 0$, then $x = (-1, a/b, 0)$ and $\boldsymbol{y} = (-(a \times c)/b, -c, (a \times a)/b + b)$. If $a = 0$ and $b = 0$, then $\mathbf{x} = (1, 0, 0)$ and $y = (0, 1, 0)$. For a line segment primitive, the coordinate center is placed at the line segment's starting point $\boldsymbol{o}$, the $\boldsymbol{z}$ axis is aligned with the line segment's normal, $\boldsymbol{y}$ axis is aligned with the line segment itself pointing to the other endpoint, and $\boldsymbol{x}$ axis is computed by finding the cross product of $\boldsymbol{y}$ and $\boldsymbol{z}$.

Warping. We then want to warp surface primitives. This is done within every primitive's own local coordinate system. For a point, we compute out the local coordinates of the four corner vertices of its bounding quadrilateral by using the rotation matrix $Rot(\theta)$ and scaling matrix Λ. For a line segment, we locate the 15 or 18 points, as discussed in our approximation method in 5.2.3. With these points known, we then can slice a surface primitive's object space EWA resampling filter into a set of interconnected bounding quadrilaterals. From warping onwards, all subsequent pipeline operations are defined on these bounding quadrilaterals and convex polygons that are clipped from them. At the end of the warping stage, we need to rewrite these locally defined quadrilaterals' coordinates with reference to the camera space. This is easily implemented using straightforward 3D coordinate mapping.

Texturing. Now we need to map weight textures onto the quadrilaterals. With the support to bilinear interpolation by the rasterizer, the color, camera space coordinate, texture coordinate and other numerical attribute values can be derived from respective values assigned to corner vertices. As we have discussed in Section 5.2.3.2, the task of weight texture mapping now becomes to assign appropriate texture coordinates to the vertices of each quadrilateral. (See Figure 14)

A weight scaling factor is also assigned to each quadrilateral vertex at the texturing stage. Similar to the case of texture coordinate, weight scaling factor is bilinearly interpolated as well. Let's use a long line segment as an example. In fact we only compute the scaling factor values for point 9, $\frac{1}{r_0 r_1}$ and point 10, $\frac{1}{r_0^{'} r_1^{'}}$. With our approximation method, along

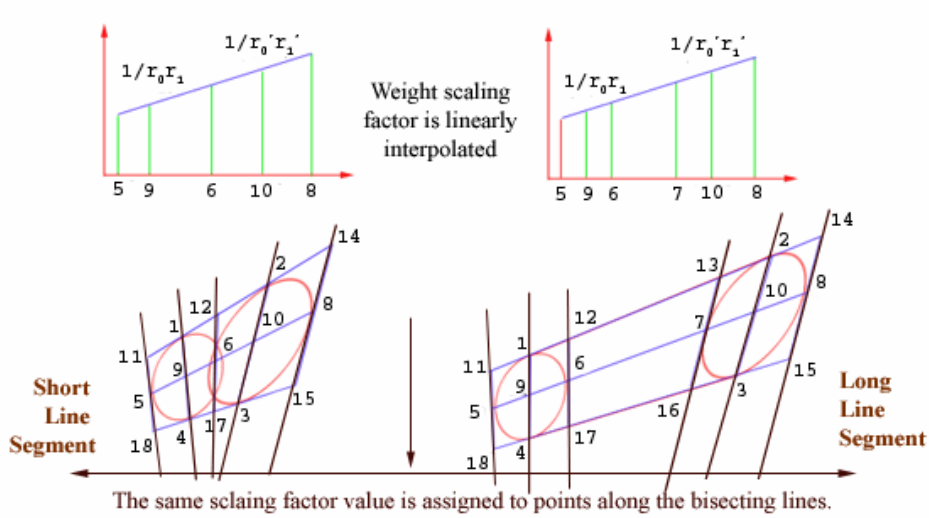

Figure 19: *Implementation of the weight scaling factor interpolation.*

the line 5-8, the scaling factor is linearly interpolated. So we deduce the scaling factor values of point 5, 6, 7 and 8 from $\frac{1}{r_0 r_1}$ and $\frac{1}{r_0' r_1'}$. Points along the lines that intersect line 5-9, including line 11-18, line 1-4, line 12-17, line 13-16, line 2-3 and line 14-15, are considered being scaled by almost the same amount. This is not clearly true from our geometry observations and approximation method in 5.2.2. However in order to devise a workable solution, we approximate those points' scaling factor value in such way. Our rendering results show that almost no loss of image quality with this approximation. Thus we assign point 5's scaling factor value to point 11 and 18, point 9's to point 1 and 4 and so on (See Figure 19).

Mapping an image texture to line segments is implemented more or less the same as assigning weight scaling factors. There is only a small difference. To match with textured triangle meshes, we only interpolate texture coordinates in-between an extracted line segment's two endpoints and extrapolate the texture coordinates outwards at the endpoints.

For example, for a textured short line segment, texture coordinates of endpoints 9 and 10 are assigned to the every point of the set {1, 4, 5, 9, 11, 18} and {2, 3, 8, 10, 14, 15} respectively.

Clipping. Moving forward to the clipping step, we clip every quadrilateral to our six view frustum planes. Notice, our view frustum clipping is done in camera space. This is different from the implementation provided by OpenGL, which is done in clipping coordinate system after perspective projection. The clipping space is bounded by a cube, and each of the six faces of the cube is parallel to one of the coordinate system's three basis axes. Clipping to these axis-aligned faces usually saves time, and can be implemented efficiently with hardware support. Since OpenGL takes hardware acceleration advantages, this is the most efficient approach for OpenGL. However, our rendering pipeline is software based; we are not able to benefit from graphics card acceleration features. In addition perspective projecting every quadrilateral to clipping space incurs extra time burden. Therefore very likely, a software implemented camera space clipping may be superior to a software implemented clipping space clipping.

Lighting. Now we proceed to the lighting stage. Our pipeline implements Gourand shading, supporting ambient light, diffuse light as well as specular light. Each vertex's color is recalculated, taking illumination and surface material property into account. Our lighting calculation is delayed until after clipping. This is different from OpenGL approach, in which case lighting is done prior to clipping. The reason is that some quadrilaterals may

have been totally discarded or partially discarded due to the view frustum culling, so delayed lighting helps us saving computation time.

Perspective Projection. Next we perspective project all the quadrilaterals as well as the convex polygons which are the results of clipping quadrilaterals, into the normalized device coordinate system. The perspective projection matrix is computed using view frustum parameters, involving z near value, aspect ratio, and field of view.

Rasterization and Blending. Finally we arrive at the last two pipeline stages, rasterization and blending. As the bounding quadrilaterals are also convex polygons, this means only convex polygons are to be rasterized by the rasterizer. Hence, we use a modified scan conversion algorithm for the rasterizer, which is optimized for convex polygon rasterization. Similar to the original scan conversion algorithm, the modified one also keeps track of an edge table and an active edge table. However, at any time, the size of the active edge table is at most two, because a scan line can intersect a convex polygon at most twice, firstly goes into the polygon from one left edge, then leaves away from one right edge. So as the scan line advances from bottom to top along the y axis of the normalized device coordinate system, in stead of updating the active edge table at every y value, we only need to keep a left edge pointer, a right edge pointer, and compute out which edge's y top is first reached by the scan line, and update the active edge table with next left or right edge only when the scan line actually arrives there. In contrast to concave polygon rasterization which requires sorting the active edge table at every y, we do not need to do it any more; therefore the modified rasterization algorithm works more efficiently.

Convex polygons are thus tessellated into pixel fragments, and subsequently these fragments are written into frame buffers one by one. For each such fragment, we bilinearly interpolate its z depth value, color, weight texture coordinate, weight scaling factor value as well as texture image coordinate. Before we write a fragment's color into color buffer, we need to do a visibility test with its z depth value, checking whether the fragment has been occluded away or not. We implement the modified Z^3 algorithm for visibility testing and blending. For each fragment, we also need to compute out its weight contribution which is used later when the fragment is blended with other fragments to form object surface. It is equal to the product of corresponding weigh texture value and weight scaling factor value. Fragment's weight texture value is retrieved from texture memory with its interpolated texture coordinate. Fragment could also be transparent, or semitransparent. Z^3 algorithm supports order independent transparency. We discuss Z^3 algorithm with more details in next subsection.

6.2.3. Visibility and Blending Algorithm

6.2.3.1. The Modified Z^3 Algorithm

For triangle based rendering, visibility algorithm is separated from blending algorithm. Consider the scenario when two fragments arrive at the same pixel location. The Z buffer algorithm compares their z depth values, decides which fragment is in front and which fragment is at back. If the front fragment is opaque, then only the front fragment will be written into the frame buffer, and the back fragment is occluded away. The blending

algorithm will blend the front and back fragments into one fragment only if the front fragment is transparent.

However, for point based rendering as well as our line segment based rendering, we cannot simply discard the back fragment in the case when the front fragment is opaque. Because quite often, it is the circumstance that these two fragments are rasterized from surface points or surface line segments sampled next to each other. If the difference of the z depth values between these two fragments is quite small, then the blending algorithm should compress them together to form one piece of surface. Usually we set a z threshold and compare the z difference with the z threshold, if the difference is larger than the threshold, these two fragments are considered coming from two separate pieces of surfaces and are not merged. In addition, like triangle case, the blending algorithm needs to take care of transparency problem as well. As visibility testing algorithm and blending algorithm are highly coherent, we would like to combine them into one instead.

Before we list out the details of Z^3 algorithm, we discuss the buffers supported by our engine first. Altogether there are three different kinds of buffers used in our rendering pipeline; they are color buffer, z buffer and accumulation buffer. Color buffer is used to store fragments' color values, which includes four channels – red, green, blue and alpha. Accumulation buffer is used to store fragment's weigh contributions. Weight contribution value is within the range $[0,1]$. Z buffer is used to store fragments' z depths. Buffers are grouped into layers. Each layer has only one color buffer, one z buffer and one accumulation buffer.

Let us use n to denote the number buffer layers available, c to denote color, a to denote alpha value, w to denote weight contribution and z to denote z depth value. The modified Z^3 algorithm operates in the following way:

1. For a newly added fragment f at pixel location l
    a. ***Fragment Insertion.*** We first search all the existing fragments at l, across the n buffer layers, to check whether the difference of the z depth value between f and any existing fragment e is less than the z threshold or not. If this turns out to be the case, then f is merged with e, and its color and weight contribution are added into e's corresponding buffer entries, $c_e = c_e + c_f w_f$ and $w_e = w_e + w_f$. We keep e's z depth value unchanged.
    b. ***Fragment Allocation.*** Otherwise, we check whether the number of buffer layers used at l has already exceeded the limit of available layers n or not. If not yet, then we allocate a new layer for inserting f. Usually we keep all the fragments at l sorted in the increasing order of z. If f is opaque, then all the fragments behind it are occluded away, consequently buffer entries occupied by them are freed, and additional layers at l become available.
    c. ***Fragment Compression.*** Otherwise, if we have already used up all the available n layers, then we merge the nearest two layers, so that one layer can be freed for storing fragment f. We use subscript f for the front layer to be merged, b for the back layer to be merged and m for the merged layer, the

formulas for compression are $c_m = c_f a_f + c_b a_b (1 - a_f)$, $a_m = a_f + a_b (1 - a_f)$ $z_m = (z_f + z_b)/2$ and $w_m = 1$.

2. After all the fragments have been added, we combine the n buffer layers into one layer in the back-to-front order. Formulas for computing color and alpha values are similarly defined as those used at fragment compression step; however z depth and weight contribution information is no longer needed to be kept, hence discarded. It is the final combined layer where the image to be displayed on screen is stored.

Our results show that Z^3 algorithm works appropriately well for a variety of different models, at different viewing distances, resulting smooth surface blending, correct transparency accumulating. In OpenGL, with the standard A-buffer algorithm, to correctly render translucent objects, especially when the complexity of the scene increases, a lot of objects exist in the environment, we must always place objects into the rendering context in the back-to-front manner. This restriction makes programming very tricky, and is difficult to satisfy in an animated scene, where characters keep moving around.

With use of multiple layers, the Z^3 algorithm sort transparent objects in the buffers before combining them, to certain extent, providing order independent transparency. There also exists a hardware solution of order independent transparency problem, however it is a multi-pass algorithm, and because the blending nature of point and line rendering, our pipeline cannot be supported by existing hardware, therefore multi-pass algorithm is too costly for us [NVID01].

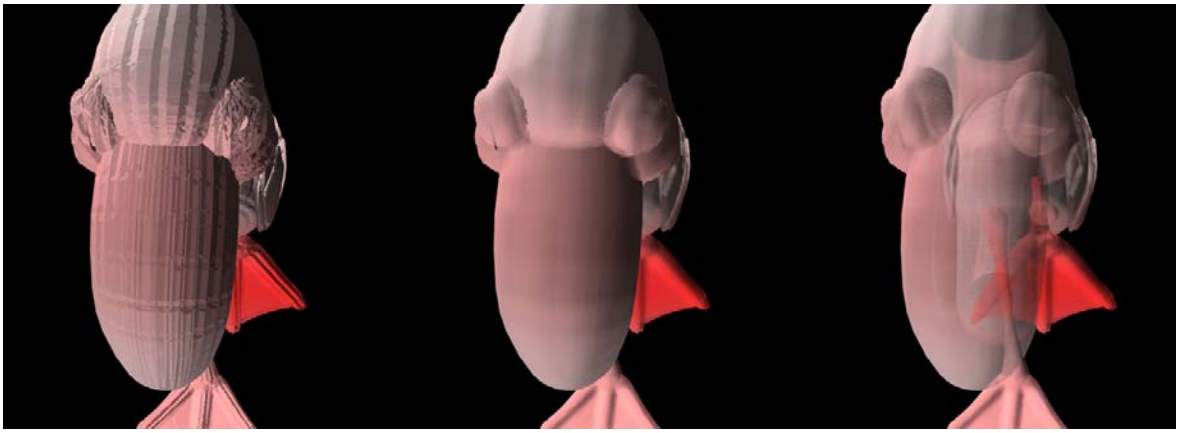

Figure 20: *The flamingo model rendered with different z thresholds. The z threshold value for the flamingo model on the left is set to be 0, thus there is no blending at all. While the z threshold value for the flamingo model on the right is set to be 0.3, causing too much blending. The flamingo's body can even been seen through. And the flamingo model in the middle is rendered with z threshold value of 0.01 which gives an adequate blending.*

In our implementation, we encounter two problems with the modified Z^3 algorithm. The first one is the setting of the z threshold value. Z^3 algorithm only supports one fixed z threshold value. According the algorithm, two fragments are treated as coming from the same piece of surface and merged as long as the difference between their z depth values is less the z threshold, regardless the actual values of their z depths. However the fact with perspective projection is that, for a set of equally distanced points along one line, after projection, points with smaller z values in original eye space are placed further apart from each other then those having larger z values. To make sure that fragments from the same surface are always blended together, the z threshold should be set adaptively with respect to the fragments' actual z values, if they are near to our eye, a relatively larger z threshold should be chosen, if they are far away, use a smaller z threshold value is more accurate. In our implementation, we let the z threshold to be 0.001 (See Figure 20).

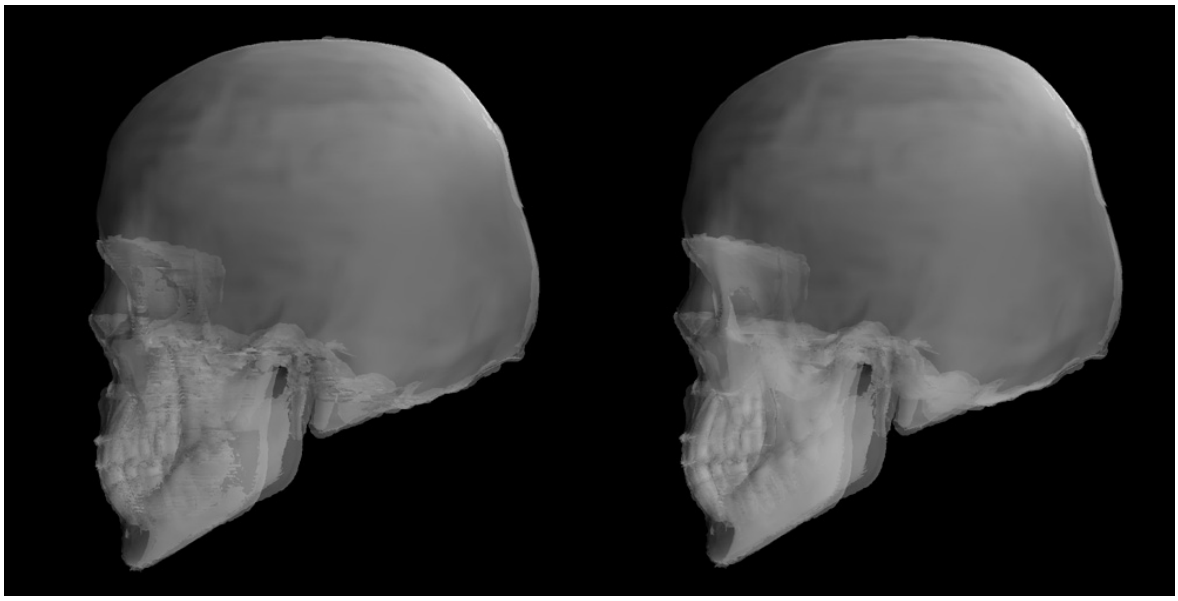

Figure 21: *A transparent skull model is rendered using different number of buffer layers. On the left, 4 layers are used which is much less than the depth complexity of the skull, thus the surfaces are wrongly blended, creating visible artifacts. While for the skull model on the right, we allocate 10 layers of buffers, which is larger than the skull's depth complexity, thus surfaces are correctly blended.*

The second problem with Z^3 algorithm is the decision of the number of buffer layers to be used. Z^3 algorithm supports only a fixed number of buffer layers. As mentioned before order independent transparency is only supported to certain extent, it is only accurate when the depth complexity of a transparent object is less than the number of buffer layers. As specified in the algorithm, when two fragments are merged in the "fragment compression" step, the merged fragment's depth value is set as the average of the two being merged, this practice inevitably causes loss of the depth information, which could be crucial for subsequently arriving fragment. Because it may happen that the next coming fragment lie just in-between the two already merged fragments, as a result, transparency is calculated wrongly, visual artifacts could be introduced (See Figure 21). Meanwhile we cannot use too many layers of buffers in Z^3 algorithm either, because the more buffer layers used, the

heavier the cost burden incurred by sorting becomes, evidently slowing down the whole rendering process. In our implementation we use 4 buffer layers.

6.2.3.2. The Delta-Z-Buffer Algorithm

The delta-z-buffer algorithm proposed by Peng *et al.* [PHY01], is primarily designed for point based rendering. Like Z^3 algorithm, it also uses multi buffer layers, however it allows a specifically calculated z threshold assigned for each point individually. Peng *et al.* calls the z threshold a different name delta z. Inside the z buffer, besides storing a fragment's z depth value, two extra depth values are also collected, zmax and zmin, where zmax is computed by adding delta z to the z depth value and zmin is obtained by subtracting delta z from z depth value. A newly come fragment is considered to lie on the same surface with the existing fragment, only if its z depth value falls inside the z range set by the existing fragment's zmin and zmax.

The algorithm assumes that around a point's neighborhood, all the fragments being rasterized have the same delta z value. Delta z value is conservatively estimated by finding $\max(z \text{ of } \mathbf{T}(\boldsymbol{p} - \boldsymbol{v} \cdot r) - \mathbf{T}(\boldsymbol{p}), z \text{ of } \mathbf{T}(\boldsymbol{p} + \boldsymbol{v} \cdot r) - \mathbf{T}(\boldsymbol{p}))$, where $\mathbf{T}$ denotes the transformation function of the perspective projection, $\boldsymbol{p}$ denotes the position of the point center, $\boldsymbol{v}$ stands for the unit vector pointing form the point center to the eye, and r is the radius that bounds the neighborhood of the point. We know, the resampling filter of a point is of elliptical shape, therefore the value of r is actually equal to the length of the major axis the ellipse. Before perspective projection, along the z direction, the two most outwards fragments are of the same distance away from the point center. However after projection, as what we have discussed, their z differences varies, so in order to guarantee that no fragment will be

mistakenly treated coming from different piece of surface when being blended, we should use the maximal distance difference as delta-z.

A blending algorithm almost the same as delta-z-buffer algorithm is implemented in Pointshop 3D [ZPKG02]. It shows that fragments are merged more accurately, creating less blending errors, comparing with the Z^3 algorithm which is used by earlier surface splatting technique. However this algorithm does not fit well to the hybrid rendering paradigm of our surface engine. Because applying the method of computing z threshold to line, it is assumed that nearby the line segment, there is only one delta-z. But a line segment could be very long, hence the assumption cannot be held. It could only be true around a very small neighborhood. A possible way to apply delta-z-buffer algorithm to line segment rendering is to compute z threshold at the two endpoint of a line segment, and interpolate it along the line. However this could be too costly, interpolation could be inaccurate and introduces blending errors. Therefore we choose the modified Z^3 algorithm for implementation, which benefits us with its simplicity and efficiency.

6.2.4. Implementation Problems

The rasterizer of a hardware graphics pipeline, such as OpenGL pipeline, implements various rasterization algorithms, for example the middle point algorithm, at very low cost. These 2D algorithms help points, lines, curves and other 2D primitives appear more smooth and nicer when displaying on the screen. As our entire rendering pipeline is software implemented, incorporating these algorithms could significantly slow down the rendering process. So we choose only to implement the essential modified scan conversion algorithm for convex polygons. Hence, some artifacts may pop up when rasterizing very

thin polygons, edges could appear very jagged. However this problem is minor, no obvious loss of image quality due to it is observed.

In our approximation method, to get the tangent lines of two ellipses, we need to solve quadratic equations. Floating number errors emerge when the ellipses either too small or too near to each other. For example, a small number less than 10^{-6} may appear to be a denominator, hence causes the result of a division extremely large, and the intersection point is in turn miscalculated as outside the view frustum. To avoid and reduce these numerical errors, we increase the number of bits used for floating numbers, replacing 64 bit double numbers with 128 bit long double numbers. However some errors still cannot be removed. We find that most of these problems occur at the silhouettes of the model, when the neighborhoods of points or line segments lie nearly parallel to our eyesight. Therefore even we ignore these wrongly computed points and lines completely, not projecting them onto the screen at all; no apparent visual artifact will be resulted noticeable to our eyes.

Chapter 7.

Experiments and Results

7.1. Experiment Goals and Settings

The goal of our experiments is to validate the modelling and rendering approaches we have taken. We are to verify that the line segment based modelling could indeed achieve representation compactness. We are to verify that rendering line segments would truly be more efficient than rendering sequence of points. And also important, we are to compare the quality of the rendered images using our proposed anti-aliasing line segment rendering algorithm with those using EWA point rendering method to show our approach could produce equivalent high quality rendering results. We implemented our line segment extraction algorithms in Section 3 and the proposed rendering pipeline described in Section 5 in C/C++, and performed the experiments on a P4 3.0 GHz with 2GB RAM PC. Note that the rendering results of point models are also from our own implementation of the theory given in [RPZ02, ZPVG01] rather than the probably more optimized version of [ZPKG02].

7.2. Point Cloud Based Experiments

We experiment with both the (ε, δ) error bounded line segment extraction algorithm and the shape approximation based line segment extraction algorithm to extract hybrid point and line segment models from pure point models. For the (ε, δ) error bounded algorithm,

ε is set to be 5% of the average distance between two neighbouring points in the model and δ is assigned to be of 0.004, which is equivalent to about 5.1264 degree of angular difference. For the shape approximation based algorithm, we let the value of δ_n be 0.008, matching exactly with the value of δ, as it can be derived that $\delta_n = 2\delta$.

In Table 1, 2 and 3, we list down the statistics collected from three different experiment settings that compare our solutions with the pure point cloud data approach. Table 1 corresponds to the comparison between pure point models and hybrid point and line segment models obtained using the (ε, δ) error bounded algorithm. And both Table 2 and Table 3 report the comparisons between pure point models and hybrid point and line segment models produced with our shape approximation based line segment extraction algorithm. While in Table 2, we use the greedy clustering algorithm in Section 3.2.3.1 to obtain anisotropic point clusters, it is the distortion minimized clustering algorithm in Section 3.2.3.2 that is used for the experiment reported in Table 3.

In all the three tables, we first list out the number of points and line segments obtained using respective line segment extraction algorithms. Then between every two corresponding images of pure point models and hybrid point and line segment models, we show their mean square error (*mse*, measuring the difference) and the normalized cross-correlation measure (*nccm*, measuring the similarity with 1 means identical image) [ASS02] in the table. Both *mse* and *nccm* are image quality comparison measures which are obtained by rendering color images (each channel has 256 values) of size 512x512 pixels for 50 different viewpoints chosen around each of the model without any priori knowledge.

Models	Points	Hybrid Point and Line Segments				
	# Points	# Points	# Line Segments	Quality		Speedup
				mse	nccm	
Armadillo	172974	153169	2793	0.0016	1.0	7.27%
Ball joint	137062	89060	6202	0.0036	0.9999	23.77%
Golf club	209779	52287	12887	0.0042	0.9993	57.25%
Igea artifact	134345	75565	6823	0.0035	0.9999	31.38%
Male	303380	90457	6993	0.0101	1.0	30.19%
Rabbit	67038	46047	2789	0.0042	0.998	21.35%
Rocker arm	40177	27132	1502	0.0015	1.0	22.07%
Santa	75781	62360	1849	0.0019	0.9998	12.57%
Screwdriver	27152	17479	1058	0.0039	0.9998	21.80%
Teeth casting	116604	65250	4287	0.003	1.0006	41.99%

Table 1: *Hybrid point and line segment model obtained using* (ε, δ) *error bounded algorithm.*

Models	Points	Hybrid Point and Line Segments				
	# Points	# Points	# Line Segments	Quality		Speedup
				mse	nccm	
Armadillo	172974	131284	8391	0.0043	0.9995	10.64%
Ball joint	137062	68831	11616	0.0057	0.9993	28.65%
Golf club	209779	27335	13391	0.0086	0.9972	65.96%
Igea artifact	134345	52920	12391	0.0051	0.9995	38.13%
Male	303380	72863	12065	0.0151	0.9992	33.93%
Rabbit	67038	27844	6406	0.006	0.9994	35.10%
Rocker arm	40177	14609	2607	0.0047	0.9991	43.29%
Santa	75781	48922	4650	0.0043	0.9991	22.26%
Screwdriver	27152	11126	2050	0.0072	0.9994	34.03%
Teeth casting	116604	43852	5734	0.0069	0.9994	55.06%

Table 2: *Hybrid point and line segment model obtained using the shape approximation based line segment extraction algorithm with greedy clustering.*

Models	Points	Hybrid Point and Line Segments				
	# Points	# Points	# Line Segments	Quality		Speedup
				mse	nccm	
Armadillo	172974	141864	6720	0.0043	0.9995	10.18%
Ball joint	137062	79741	11075	0.0049	0.9994	24.01%
Golf club	209779	37754	19149	0.0058	0.9993	55.75%
Igea artifact	134345	62780	12800	0.0047	0.9998	31.69%
Male	303380	86106	10814	0.0136	0.9995	27.54%
Rabbit	67038	35037	5974	0.0054	0.9997	23.38%
Rocker arm	40177	17342	3347	0.0045	0.9996	36.45%
Santa	75781	53335	4249	0.0041	0.9995	18.33%
Screwdriver	27152	12689	2311	0.0073	0.9989	35.81%
Teeth casting	116604	54407	8316	0.0068	0.9985	43.19%

Table 3: *Hybrid point and line segment model obtained using the shape approximation based line segment extraction algorithm with hierarchical distortion minimized clustering.*

Models	*Number of Clusters*	
	Greedy Clustering	*Hierarchical Distortion Minimized Clustering*
Armadillo	*56173*	*49211*
Ball joint	*20378*	*17807*
Golf club	*5576*	*5892*
Igea artifact	*17062*	*14627*
Male	*22297*	*20107*
Rabbit	*7784*	*7710*
Rocker arm	*7574*	*6941*
Santa	*19357*	*17767*
Screwdriver	*4564*	*4310*
Teeth casting	*18467*	*17003*

Table 4: *Number of clusters generated from the greedy clustering and hierarchical distortion minimized clustering algorithm.*

Lastly we show the speedup of the rendering performance of using hybrid point and line segment models compared with using pure point models.

7.2.1. Clustering Algorithms

The experiment results show that generally the hierarchical distortion minimized clustering algorithm produces less number of clusters than the greedy clustering algorithm (Table 4). Intuitively this suggests larger clusters would be formed by the hierarchical distortion minimized algorithm, consequently the hierarchical distortion minimized algorithm should be likely to extract more line segments than the greedy approach. However by examining the number counts of the extracted points and line segments listed in Table 2 and 3, we find that conversely, it is the greedy clustering algorithm which always generates more line segments for replacing points in all the ten tested point models used in our experiment.

Figure 22: *The Rocker arm model segmented using the hierarchical distortion minimized clustering algorithm and the greedy clustering algorithm are shown on the left and right respectively. It can be seen that there are more large clusters formed on the right Rocker arm model than on the left one.*

Further investigation reveals the fact that although in average, the clusters generated by the greedy clustering algorithm contain less number of points, there are more large clusters formed. For example, for the Rocker arm model, with the greedy clustering, there are 66 clusters having size larger than 100 points are found, accounting for the replacement of 13818 points. However using the hierarchical distortion minimized clustering algorithm, there are just 16 clusters with size larger than 100 points found, replacing merely 2578 points in total. Figure 22 gives a screenshot of the Rocker arm model being segmented using these two different clustering algorithms.

7.2.2. Effective and Compact Representation

The statistics data in Table 1, 2 and 3 shows that large amount of points in the scanned point models are substituted by the line segments extracted using our proposed line segment extraction algorithms. In fact, the hybrid point and line segment model provides an effective yet more compact model representation alternative to the pure point cloud representation. Let us just consider the essential geometry information associated with the point and line segment primitive. That is two 3D vectors are used to store the location and the normal of each point and three 3D vectors are used to store the two endpoints and the normal of each line segment. It can be computed out that the hybrid models in Table 1 gain an average 34.03% data reduction ratio comparing to their respective point clouds. And this average number is even higher for hybrid models in Table 2 and 3, 47.51% and 39.85% respectively. A maximum 77.39% data saving ratio is achieved by the hybrid Golf club model in Table 2. Screenshots of the extracted hybrid point and line segment models can be found in figure 23, 24, 25 and 26.

7.2.3. Efficient High Quality Rendering

Figures in Table 1, 2 and 3 also show significant speedups are achieved to render hybrid point and line segment models as compared with their corresponding point models. The hybrid models in Table 1 achieve an average 26.96% speedup, and this number is 36.71% and 30.63% for the hybrid models in Table 2 and Table 3 respectively. A maximum speedup of 65.96% is also achieved by the hybrid Golf club model in Table 2. This is in line with the fact that this same hybrid Golf Club model also owns the maximum data reduction ratio.

The quality of the rendered images of the hybrid point and line segment models can be evaluated both numerically and visually. It can be observed that the values of *mse* are all quite near to 0 and the values of *nccm* are near to 1. These suggest that the rendered hybrid point and line segment models and their corresponding point models indeed have nearly the same quality. Figure 27 shows the rendering outcome of the hybrid models.

7.3. Triangle Mesh Based Experiments

The point and line segment models used in this experiment are all obtained using our contour plane based line segment extraction algorithm from triangle meshes. We render each model in Table 5 at 20 different viewpoints chosen around the model without any priori knowledge. Altogether 180 images of point models and 180 images of pure line segment models are used. These images are of size 512x512 pixels. As in Section 7.2.3, the rendering quality is evaluated both visually and numerically. Figure 28 shows

Models	***Number of Primitives***		***Image Differences***	
	# Points	*# Line Segments*	*Mean*	*Standard Deviation*
Bunny	*128642*	*26834*	*0.13*	*2.91*
Flamingo	*126938*	*27876*	*0.12*	*2.70*
Al	*163205*	*46595*	*0.09*	*2.17*
Gargoyle	*146704*	*46106*	*0.09*	*1.88*
Dragon	*241893*	*78305*	*0.19*	*3.62*
Buddha	*338709*	*115883*	*0.31*	*4.45*
Horse	*76601*	*26518*	*0.02*	*1.56*
Skull	*156434*	*57247*	*0.04*	*1.83*
Skeleton	*221856*	*87838*	*0.17*	*3.89*
		Average	*0.13*	*2.78*
		Maximum	*0.31*	*4.45*

Table 5: *Pure point and line segment models obtained using the contour plane based line segment extraction algorithm. Their rendered image differences are also computed out.*

rendering outcome of line segment models in Table 5. We compute the mean of the absolute (pixel) differences between two images rendered using line segments and points. These differences are calculated from a composite of red, green and blue channels (each channel has 256 values). Table 5 shows the average of the means from the 20 different viewpoints per model and the average of the standard deviations. We find that the means of the absolute differences are nearly zero with an average of 0.13 and the average standard deviations of 2.78 (second last row in the table). The maximum of such mean is 0.31 and the maximum standard deviation is 4.45 (last row in the table). The numerical results suggest that the rendered line segment and point models have nearly the same quality again.

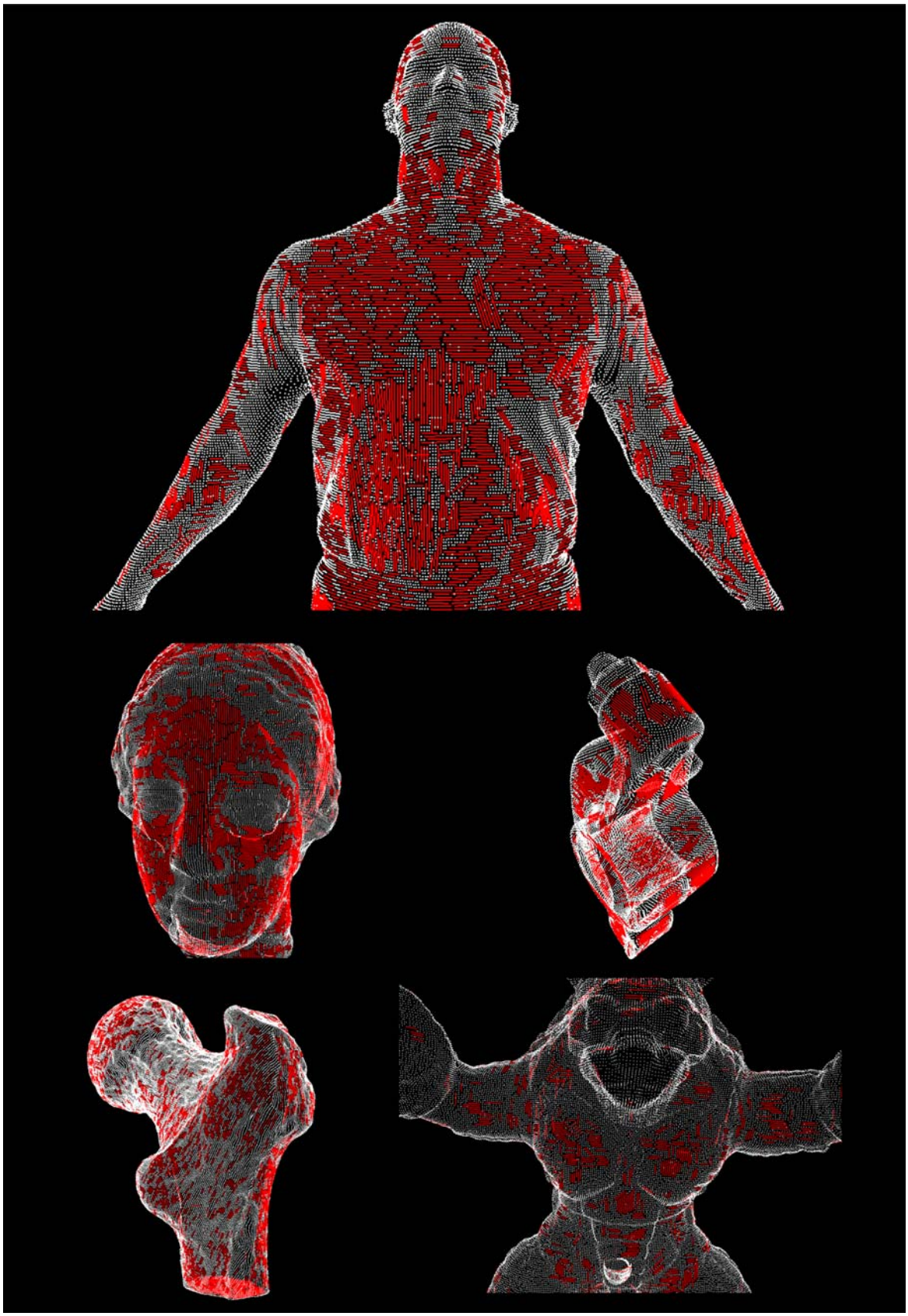

Figure 23: *Hybrid Man, Igea artifact, Rocker arm, Ball joint and Armadillo models extracted using the* (ε, δ) *error bounded line segment extraction algorithm.*

Figure 24: *Hybrid Golf club, Rabbit, Screwdriver, Santa and Teeth casting models extracted using the* (ε, δ) *error bounded line segment extraction algorithm.*

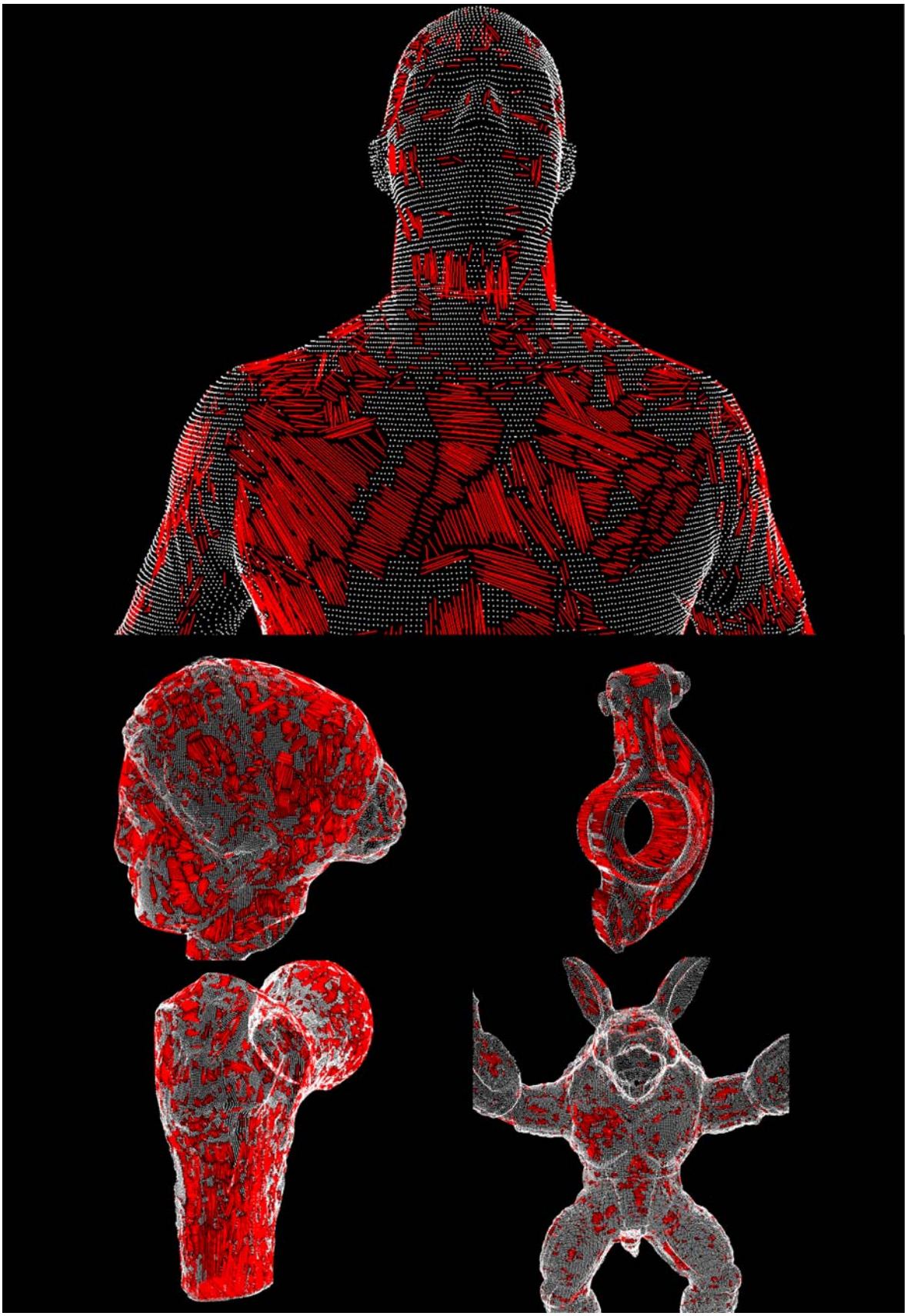

Figure 25: *Hybrid Man, Igea artifact, Rocker arm, Ball joint and Armadillo models extracted using the shape approximation bounded line segment extraction algorithm.*

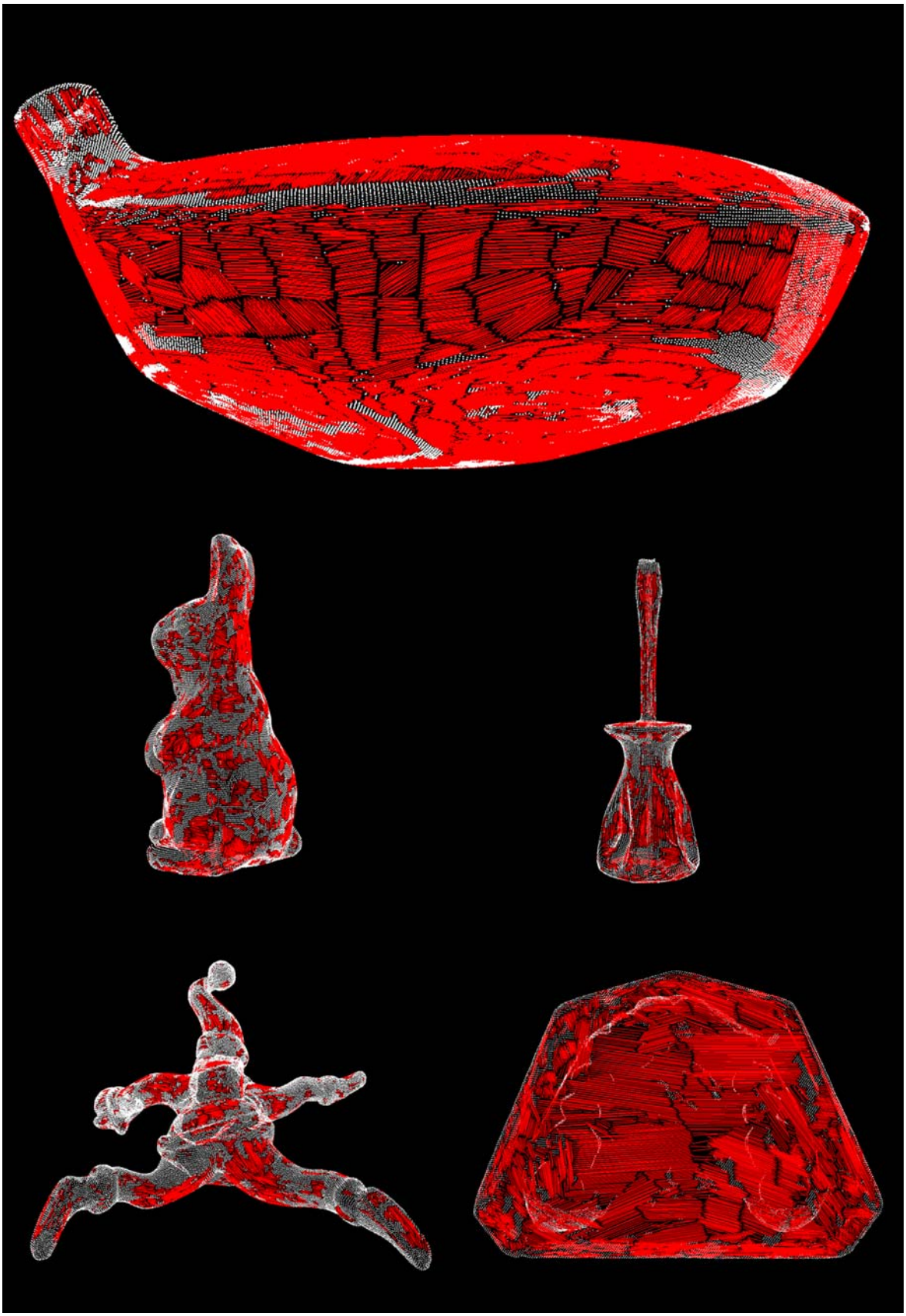

Figure 26: *Hybrid Golf club, Rabbit, Screwdriver, Santa and Teeth casting models extracted using the shape approximation bounded line segment extraction algorithm.*

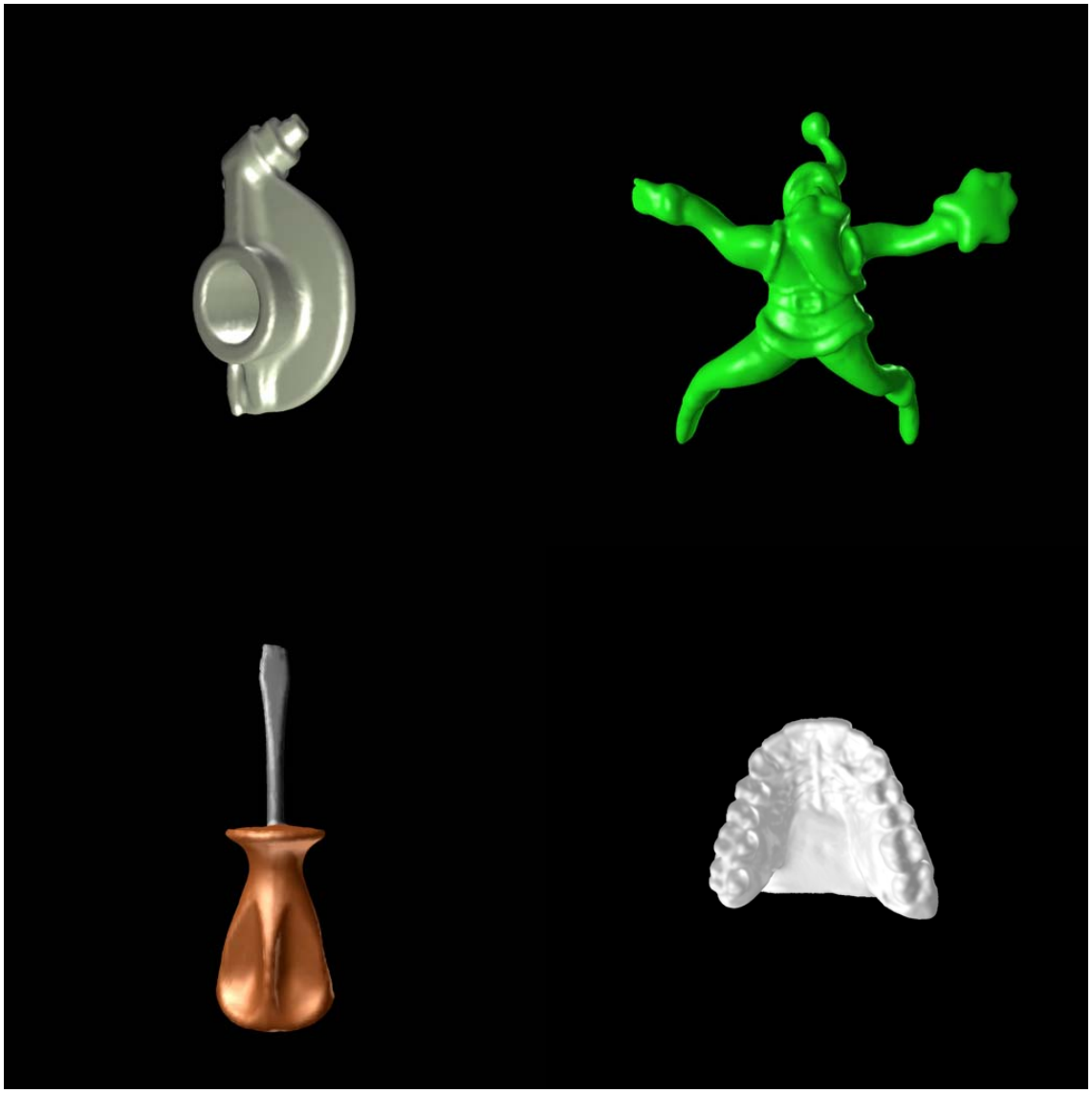

Figure 27: *Hybrid Armadillo, Ball joint, Golf club, Igea artifact, Man, Rabbit, Rocker arm, Santa, Screwdriver, and Teeth casting model rendered using our approximate rendering algorithm in Section 5.*

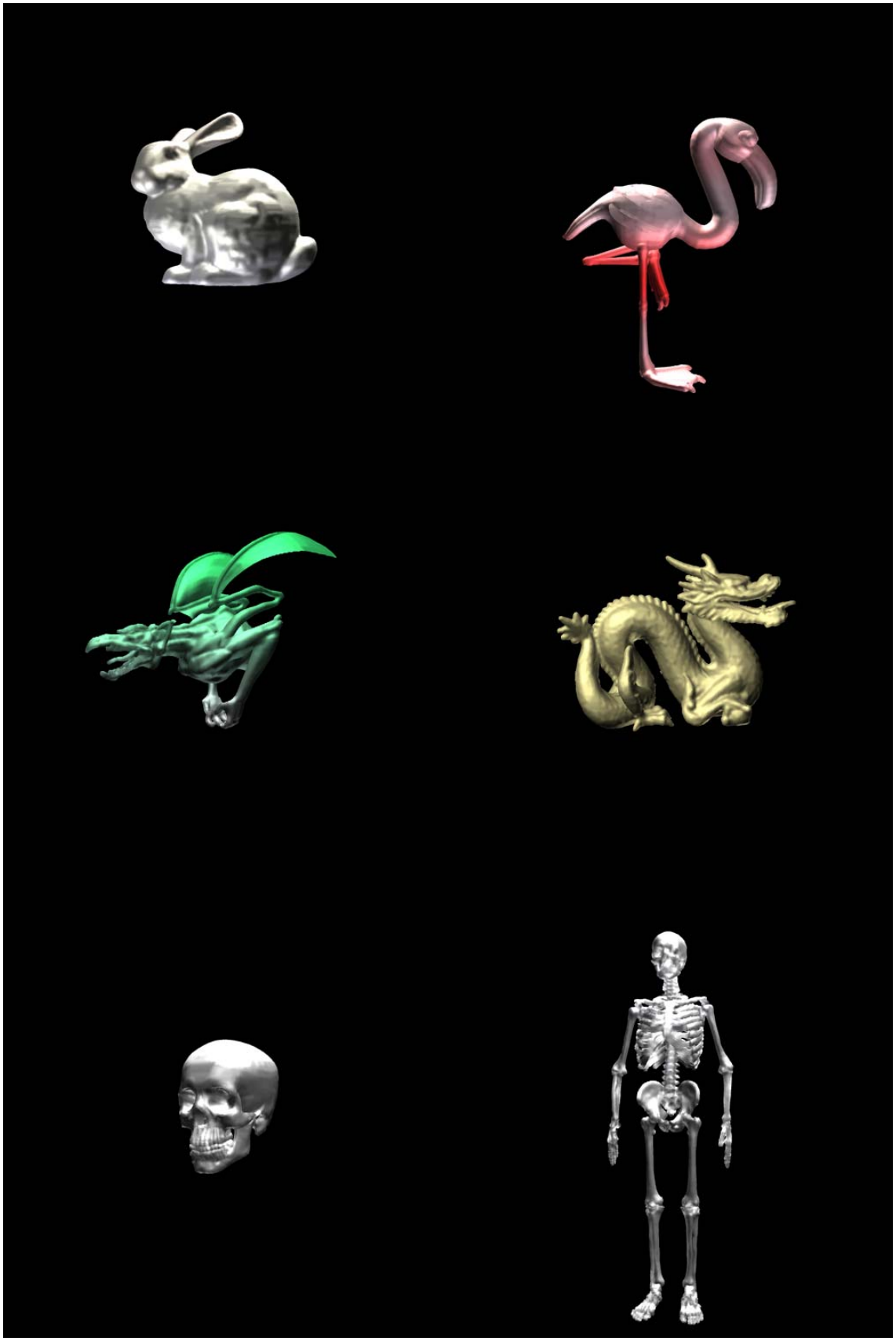

Figure 28: *Line segment Bunny, Flamingo, Gargoyle, Dragon, Skull, Skeleton, AL, Horse, Budda, rendered using our approximate rendering algorithm in Section 5.*

Chapter 8.

Conclusions

This report demonstrates the feasibility of using 3D line segments as a primitive for both surface modelling and rendering. In modelling, we propose one (ε, δ) error bounded and one shape approximation based line segment extraction algorithm for extracting hybrid point and line segment models from scanned point clouds. We present a method for obtaining pure line segment models from triangle meshes as well. In rendering, we extend the anti-aliasing theory in texture mapping to anti-aliased line segment rendering, and present an approximation algorithm to render high quality anti-aliased opaque, transparent and textured line segments in 3D models. Experiments show that the rendered pure line segment models as well as hybrid point and line segment models have the same high quality as their counterparts using points only. As compared to pure point models, hybrid point and line segment models enjoy significant rendering efficiency too.

There are three limitations identified with our current hybrid point and line segment modelling and rendering approach. First, like pure point models, the explicit connectivity information is also absent in hybrid point and line segment models, which makes the model deformation more difficult than it is in surface triangulations. Second, the use of local affine assumptions in the neighborhood of a point or line segment produces artifacts in the presence of highly nonlinear mappings. Lastly, our current formulation of line

segments requires each line segment possess only one normal, which rules out the possibility of using line segments to render ruled surfaces.

Chapter 9.

Future Work

9.1. Line Segment Based Surface Definition

We learn from our hand-on experiences on the (ε, δ) error bounded line segment extraction algorithm and the shape approximation based line segment extraction algorithm that error control is the key issue of extracting line segments from a discrete point cloud. The error measures ε , δ and δ_n used in our line segment extraction algorithm can confine the geometric deviations from the extracted line segments to scanned points. But they are unable to assure the surface formed by the extracted line segments would be really close to the actual object surface. Thus holes can be visible in our rendered images. Thus we are lead to the problem, for a given set of line segments, what continuous surface do they imply? This question is fundamental and important. The answer to it would give a complete surface description for discrete line segments. Recently we have seen a number of proposals on defining point set surfaces [Levi98, Levi03, AmYo04]. In particular, the MLS surface [Levi98, Levi03] receives a lot of attentions, as it is defined both analytically and procedurally. There exists the possibility that line segment based surface can be similarly defined.

9.2. Extract Line Segments from Reconstructed Object Surfaces

Not knowing the shape of the surface spanning across sampled points is one of the two difficulties that hinder us from tracing curvature lines on point cloud data. But if we have the object surface reconstructed first, then the idea of following surface curvature lines is very promising. As not only a continuous surface description is given but also the surface curvatures can be estimated accurately from the implicit functions that define the point set surface, helping us taking out the other difficulty too. A possible approach can be tried out is to use the MLS method [Levi98, Levi03] to reconstruct the surface of the input point cloud part by part and then trace curvature lines across the reconstructed polynomial surface patches. The last step would then be straightening and segmenting extracted curvature lines into sequences of interconnected line segments.

9.3. Hardware Accelerated Line Segment Rendering

Currently the main obstacle that blocks us from porting our line segment rendering algorithm into the hardware pipeline comes from the computation of the tangent line segments on the endpoint ellipses. Our C/C++ code for computing the duals of ellipses and for solving the quadratic equations is of around 1500 lines long. It would be extremely hard to implement the shaders' code counterpart directly, as only a limited set of CPU functionalities is supported by the graphics cards. And our current formulation of the line segment's resampling filter is in object space. However, a hardware implementation of the object space EWA resampling filter only consumes the vertex shaders' computational power, pixel shaders are left hang up without taking any workload. This is not desirable, as the performance bottleneck in our problem is the lack of sufficient computational power.

Given the fact that a number of efficient screen space implementations of the EWA point splatting [CoHe02, BoKo03, ZRBD04] are reported performing faster than the object space approach in [RPZ02], we think likely, a hardware accelerated screen space implementation of the line segment's EWA resampling filter is likely easier to be attained.

9.4. Non-photorealistic Rendering

The hybrid point and line segment models extracted using our shape approximation based line segment extraction algorithm contain bunches of nearly parallel line segments. These line segments are faithfully aligned with surface anisotropic directions. Girshick *et al.* [GIHL00] have already pointed out that the curvature aligned line segments are quite efficient for communicating object shape information to our human beings. We believe there is a great potential for our extracted hybrid point and line segment models being used in the non-photorealistic rendering contexts. By exploiting the nearly parallel nature of the extracted line segments, non-photorealistic rendering techniques can be easily implemented on top of the hybrid model's geometric structure. For example, cross-hatching strokes can be drawn along the nearly parallel line segments.

9.5. Hybrid Surface Modelling and Its Applications

Stemming from our extracted hybrid point and line segment models, there emerges quite a number of thrilling fields that the future research can further explore. The photorealistic rendering technique should continuously strive for speed, searching for a graphics hardware supported rendering solution; and the non-photorealistic rendering technique could explore the faithful yet flexible shape representation given by the hybrid surface

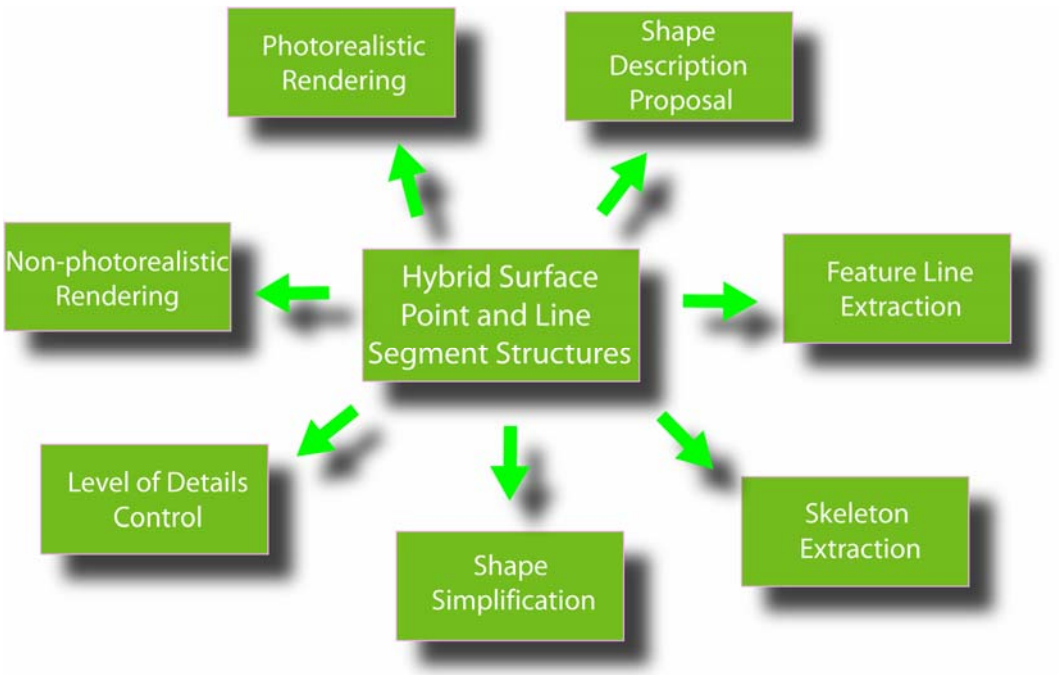

Figure 29: *A summary of future research work.*

modelling structure, so as to better illustrate shape features as well as help to speed up the conceptualization of modelled shapes. Meaning while, research efforts can also be invested with emphasis on 3D geometry processing. As what we have already discussed in Section 9.1, a complete analytic surface description with solid mathematical foundations for hybrid point and line segment models is what we can aim for. Line segments extracted in the hybrid models could be further processed to extract the models' skeleton lines and to reveal salient feature lines of the described object surfaces. There exist the possibilities to extend the use of line segments in shape modelling to level of details controls and simplifications of shapes as well. These might be achieved similarly as what have been discussed in [ChNg01] and [DeHu02]. Figure 29 summarizes all these possible future work.

References

[ABCF03] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin and C.T. Silva. Computing and Rendering Point Set Surfaces. In *Proc. of IEEE Transaction on Visualization and Computer Graphics*, Vol. 9, No. 1, pp. 3-15.

[ACDL00] N. Amenta, S. Choi, T.K. Dey and N. Leekha. A Simple Algorithm for Homeomorphic Surface Reconstruction, In *Proc. of ACM Symposium on Computational Geometry 2000*, pp. 213–222.

[ACDL03] P. Alliez, D. Cohen-Steiner, O. Devillers, B. Levy and M. Desbrun. Anisotropic Polygonal Remeshing. In *Proc of SIGGRAPH 2003*, pp. 485-493.

[AdAl03] A. Adamson and M. Alexa. Approximating and Intersecting Surfaces from Points. In *Proc. of Eurographics Symposium on Geometry Processing 2003*, pp. 245-254.

[AGJ00] U. Adamy, J. Giesen and M, John. New Techniques for Topologically Correct Surface Reconstruction. In *Proc. of IEEE Visualization 2000*, pp. 373–380.

[AmYo04] N. Amenta and J.K. Yong. Defining Point-Set Surfaces. In *Proc. of SIGGRAPH 2004*, pp. 264-270.

[ASS02] I. Avcibas, B. Sankur and K. Sayood. Statistical Evaluation of Image Quality Measures. In *Journal of Electronic Imaging*, vol. 11, no. 2, 2002, pp. 206–223.

[BoHe96] F.J. Bossen and P.S. Heckbert. A Pliant Method for Anisotropic Mesh Generation. In *Proc. of 5th International Meshing Roundatable*, pp 63-67.

[BoKo01] M. Botsch and L. Kobbelt. Resampling Feature and Blend Regions in Polygonal Meshes for Surface Anti-Aliasing. In *Proc. of Eurographics 2001*, pp 402-410.

[BoKo03] M. Botsch and L. Kobbelt. High Quality Point Based Rendering on Modern GPUs. In *Proc. of Pacific Graphics 2003*, 335-343.

[CAD04] D. Cohen-Steiner, P. Alliez and M. Desbrun. Variational Shape Approximation. In *Proc. of SIGGRAPH 2004*, pp. 905-914.

[CDK04] B. Chen, F. Dachille and A.E. Kaufman. Footprint Area Sampled Texturing. In *Proc. of IEEE Transactions on Visualization and Computer Graphics*, Vol. 10, No. 2, pp. 230-240.

[CGAL04] Computational Geometry Algorithm Library. Webpage Address: *http://www.cgal.org*.

[ChNg01] B. Chen and M.X. Nguyen. POP: A Hybrid Point and Polygon Rendering System for Large Data. In *Proc. of IEEE Visualization 2001*, pp. 45–52.

[CoHe02] L. Coconu and H.C. Hege. Hardware Accelerated Point Based Rendering of Complex Scenes. In *Proc. of Eurographics Workshop on Rendering*, pp. 43-52.

[CoMo03] D. Cohen-Steiner and J. Morvan. Restricted Delaunay Triangulations and Normal Cycle. In *Proc. of Symposium on Computational Geometry 2003*, pp. 237-246.

[DCSD02] O. Deussen, C. Colditz, M. Stamminger and G. Drettakis. Interactive Visualization of Complex Plant Ecosystems. In *Proc. of IEEE Visualization 2002*, pp. 219–226.

[DeHu02] T.K. Dey and J. Hudson. PMR: Point to Mesh Rendering, A Feature-Based Approach. In *Proc. of IEEE Visualization 2002*, pp. 155–162.

[FoFi88] A. Fournier and E. Fiume. Constant-Time Filtering with Space-Variant Kernals. In *Proc. of SIGGRAPH 1988*, pp. 229-238.

[FrAl03] P.J. Frey and F. Alauzet. Anisotropic Mesh Adaptation for Transient Flows Simulations. In *Proc. of 12th International Meshing Roundtable*, pp 335-348.

[GaHe97] M. Garland and P. Heckbert. Surface Simplification Using Quadric Error Metrics. In *Proc. of SIGGRAPH 1997*, pp. 209-216.

[Garl99] M. Garland. Quadric-Based Polygonal Surface Simplification. Phd Thesis, Carnegie Mellon University, 1999.

[GIHL00] A. Girshick, V. Interrante, S. Haker and T. Lemoine. Line Direction Matters: An Argument for the Use of Principle Directions in 3D Line Drawings. In *Proc. of 1st International Symposium on Non-photorealistic Animation and Rendering*, pp 43-52.

[GrDa98] J.P. Grossman and W.J. Dally. Point Sample Rendering. In *Proc. of 9th Eurographics Workshop on Rendering 1998*, pp. 181–192.

[HDD92] H. Hoppe, T. DeRose, T. Duchamp. Surface Reconstruction from Unorganized Points. In *Proc. of SIGGRAPH 1992*, pp 71-78.

[Heck89] P. Heckbert. Fundamentals of Texture Mapping and Image Warping. Master Thesis, University of California, Berkeley, 1989.

[HeGa99] P.S. Heckbert and M. Garland. Optimal Triangulation and Quadric-Based Surface Simplification. In *Journal of Computation Geometry: Theory and Applications*, Vol. 14, No. 1, pp 49-65.

[JaSh01] K.E Jansen and M.S. Shephard. On Anisotropic Mesh Generation and Quality Control in Complex Flow Problems. In *Proc. of 10th International Meshing Roundtable*, pp 341-349.

[JiTa02] X. Jin and C.L. Tai. Analytical Methods for Polynomial Weighted Convolution Surfaces with Various Kernels. In *Computer & Graphics*, Vol. 23, No. 3, pp. 437 ~ 447.

[KaVa01] A. Kalaiah and A. Varshney. Differential Point Rendering. In *Proc. of Eurographics Workshop on Rendering Techniques*, pp. 139-150.

[KaVa03a] A. Kalaiah and A. Varshney. Modelling and Rendering of Points with Local Geometry. In *IEEE Transaction of Visualization and Computer Graphics*, Vol. 9, No. 1, pp. 30-42.

[KaVa03b] A. Kalaiah and A. Varshney. Statistical Point Geometry. In *Proc. of Eurographics Symposium on Geometry Processing*, pp. 107-115.

[KFR04] M. Kazhdan, T. Funkhouser and S. Rusinkiewicz. Shape Matching and Anisotorpy. In *Proc. of SIGGRAPH 2004*, pp. 623-629.

[LaSh03] F. Labelle and J.R. Shewchuk. Anisotropic Voronoi Diagrams and Guaranteed-Quality Anisotropic Mesh Generation. In *Proc. of ACM Symposium on Computational Geometry 2004*, pp 191-200.

[Levi03] D. Levin. Mesh-independent Surface Interpolation. In *Geometric Modelling for Scientific Visualization* 2003, pp. 37-49.

[Levi98] D. Levin. The Approximation Power of Moving Least-squares. In *Mathematics of Computations*, Vol. 67, No. 224, pp. 1517-1531.

[LeWh85] M. Levoy and T. Whitted. The Use of Points as a Display Primitive. Technical Report TR 85-022, University of North Carolina at Chapel Hill, 1985.

[LGSF00] M. Levoy, K, Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. In *Proc. of SIGGRAPH 2000*, pp. 131-144.

[LoTa97] K.L. Low and T.S. Tan. Model Simplification using Vertex Clustering. In *Proc. of Symposium on Interactive 3D Graphics 1997*, pp. 75–81.

[LTU99] X.Y Li, S.H. Teng and A. Ungor. Biting Ellipses to Generate Anisotropic Mesh. In *Proc. of 8th International Meshing Roundtable*, pp. 97-108.

[MaKo04] M. Marinov and L. Kobbelt. Direct Anisotropic Quad-Dominant Remeshing. In *Proc. of Pacific Graphics 2004*.

[McSh98] J. McCormack and A. Sherstyuk. Creating and Rendering Covolution Surfaces. In *Computer Graphics Forum*, Vol. 17, No. 2, pp. 113-120.

[MFPJ99] J. Mccormack, K.I. Farkas, R. Perry and N.P. Jouppi. Simple and Table Feline: Fast Elliptical Lines for Anisotropic Texture Mapping. Technical Report, Compaq Western Research Laboratory, 1999.

[MMJ00] R. McNamara, J. McCormack and N.P. Jouppi. Prefiltered Anti-aliased Lines Using Half-Plane Distance Functions. In *Proc. of*

SIGGRRAPH/Eurographics Work Shop on Graphics Hardware 2000, pp 77-85.

[MPFJ99] J. Mccormack, K.I. Farkas, R. Perry and N.P. Jouppi. Feline: Fast Elliptical Lines for Anisotropic Texture Mapping. In *Proc. of SIGGRAPH 1999*, p 243-250.

[Nels96] S.R. Nelson. Twelve Characteristics of Correct Antialiased Lines. In *Journal of Graphics Tools*, vol.1:4, 1996, pp. 1–20.

[NVID01] NVIDIA. Order Independent Transparency, 2001. Webpage Address: *http://developer.nvidia.com/view.asp?IO=order_independent_transparency*.

[PaGr01] M. Pauly and M. Gross. Spectral Processing of Point-sampled Geometry. In *Proc. of SIGGRAPH 2001*, pp 379-386.

[Paul03] M. Pauly. Point Primitive for Interactive Modelling and Processing of 3D Geometry. Phd Thesis, Federal Institute of Technology (ETH) of Zurich, 2003.

[PGK02] M. Pauly, M. Gross and L. Kobbelt. Efficient Simplification of Point-sampled Geometry. In *Proc. of IEEE Visualization 2002*, pp. 163-170.

[PHY01] Q. Peng, W. Hua and X. Yang. A New Approach of Point-Based Rendering. In proc. *IEEE Computer Graphics International 2001*, pp. 275-282.

[PKG03] M. Pauly, R. Keiser and M. Gross. Multi-scale Feature Extraction on Point Sampled Surfaces. In *Proc. of Eurographics 2003*, pp. 281-289.

[PKKG03] M. Pauly, R. Keiser, L.P. Kobbelt and M. Gross. Shape Modelling with Point-sampled Geometry. In *Proc. of SIGGRAPH 2003*, pp. 641-650.

[PZVG00] H. Pfister, M. Zwicker, J. van Baar and M. Gross. Surfels: Surface Elements as Rendering Primitives. In *Proc. of SIGGRAPH 2000*, pp. 335–342.

[Ripp92] S. Rippa. Long and Thin Triangles Can Be Good for Linear Interpolation. In *SIAM Journal of Numerical Analysis*, Vol. 29, No. 1, pp 257-270.

[RoKo00] C. Rossl and L. Kobbelt. Line-Art Rendering of 3D-Models. In *Proc. of Pacific Graphics 2000*, pp 231-239.

[RPZ02] L. Ren, H. Pfister and M. Zwicker. Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering. In *Proc. of Eurographics 2002*, pp. 461–470.

[RuLe00] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proc. of SIGGRAPH 2000*, pp. 343–352.

[SeAk91] M. Segal and K. Akeley. The Design of the OpenGL Graphics Interface. Silicon Graphics Computer Systems, 1991.

[Sher99] A. Sherstyuk. Kernel Functions in Convolution Surfaces: A Comparative Analysis. Technical Report, Monash University, 1999.

[SKS96] A. Schilling, G. Knittel and W. Strasser. Texram: Smart Memory for Texturing. In *Computer Graphics and Application*, Vol. 16, No. 3, pp. 32-41.

[SoPr03] M.C. Sousa and P. Prusinkiewicz. A Few Good Lines: Suggestive Drawing of 3D Models. In *Proc. of Eurographics 2003*, pp. 381-390.

[StDr01] M. Stamminger and G. Drettakis. Interactive Sampling and Rendering for Complex and Procedural Geometry. In *Proc. of Eurographics Workshop on Rendering 2001*, pp. 151-162.

[TART04] The Anisotropic Remeshing Topic Webpage of Meshing Research Center. Web Address: *http://www.andrew.cmu.edu/user/sowen/topics/aniso.html.*

[Taub95] G. Taubin. Estimating the Tensor of Curvature of A Surface from A Polyhedral Approximation. In *Proc. of ICCV 1995*, pp. 902-907.

[Tous83] G.T. Toussaint. Solving Geometric Problems with the Rotating Calipers. In *Proc. of IEEE MELECON 1983*, pp. 1-4.

[WFPH01] M. Wand, M. Fischer, I. Peter, F.M. auf der Heide and W. StraBer. The Randomized z-Buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In *Proc. of SIGGRAPH 2001*, pp. 361-370.

[Will83] L. Williams. Pyramidal Parametrics. In *Proc. of SIGGRAPH 1983*, pp. 1-11.

[Wong03] K.H. Wong. Line Rendering Primitive. Master Thesis, National University of Singapore, 2003.

[WuKo04] J. Wu and L. Kobbelt. Optimized Sub-Sampling of Point Sets for Surface Splatting. In *Proc. of Eurographics 2004*, pp. 643-652.

[YaSh00] S. Yamakawa and K. Shimada. High Quality Anisotropic Tetrahedral Mesh Generation Via Ellipsoidal Bubble Packing. In *Proc. of 9th International Meshing Roundtable*, pp. 263-273.

[ZPBG01] M. Zwicker, H. Pfister, J. van Baar and M.H. Gross. EWA Volume Splatting. In *Proc. of IEEE Visualization 2001*, pp. 29-36.

[ZPBG02] M. Zwicker, H. Pfister, J. Van Baar and M. Gross. EWA Splatting. In *IEEE Transactions on Visualization and Computer Graphics*, Vol. 8, No. 3, pp. 223-238.

[ZPKG02] M. Zwicker, M. Pauly, O. Knoll and M. Gross. Pointshop 3D: An Interactive System for Point-based Surface Editing. In *Proc. of SIGGRAPH 2002*, pp. 322–329.

[ZPVG01] M. Zwicker, H. Pfister, J. van Baar and M. H. Gross. Surface Splatting. In *Proc. of SIGGRAPH 2001*, pp. 371–378.

[ZRBD04] M. Zwicker, J. Rasanen, M. Botsch, C. Dachsbacher and M. Pauly. Perspective Accurate Splatting. In *Proc. of Graphics Interface 2004*, pp 247-254.

[Zwic03] M. Zwicker. Continuous Reconstruction, Rendering and Editing of Point-Sampled Surfaces. Phd Thesis, Federal Institute of Technology (ETH) of Zurich, 2003.