APPROVAL SHEET

Title of Thesis: HYBRID 3D-MODEL REPRESENTATION THROUGH QUADRIC METRICS AND HARDWARE ACCELERATED POINT-BASED RENDERING

Name of Candidate: Hanli Ni
Master of Science, 2005

Thesis and Abstract Approved: _____

Dr. Marc Olano
Assistant Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

CURRICULUM VITAE

Name: Hanli Ni.

Degree and date to be conferred: Master of Science, August 2005.

Date of Birth: January 11, 1977.

Place of Birth: Nanjing, P. R. China.

Collegiate institutions attended:

University of Maryland, Baltimore County, M.S. Computer Science, 2005.
Mississippi State University, M.S. Organic Chemistry, 2002.
Nanjing University, Nanjing, B.S. Polymer Chemistry, 1999.

Major: Computer Science.

ABSTRACT

Title of Thesis: HYBRID 3D-MODEL REPRESENTATION THROUGH QUADRIC METRICS AND HARDWARE ACCELERATED POINT-BASED RENDERING

Hanli Ni, Master of Science, 2005

Thesis directed by: Dr. Marc Olano

The expectation for highly realistic 3D images has resulted in 3D models with millions of triangles. Traditional algorithms accelerate rendering speed by taking advantage of coherence within a triangle if the screen projection of the triangle covers multiple pixels. However, as the triangle count increases and the triangle size decreases, screen projections of triangles cover fewer pixels. In some cases, projections are even sub-pixel size, which makes these acceleration algorithms ineffective. Alternatively, points have the advantage of no explicit connectivity and the rendering pipeline can be implemented using the latest 3D hardware functionalities. But sufficient densities and proper sampling patterns are required for points to be effective. Previous research suggests that neither triangles nor points are the ultimate solution in terms of hardware rendering. In this study, we build a hybrid rendering system that takes advantage of both primitives. Our system is composed of two components, preprocessing and rendering. During preprocessing, the system builds a hierarchical hybrid model taking a 3D polygonal model as input. The system breaks the input polygonal model into patches by applying clustering algorithms on the model. Planar patches are rendered using triangles, while patches with high variations are rendered with

points. The clustering algorithm is based on quadric error metrics, which is a good measure of the planarity of surfaces. Other metrics, such as shape, normal and size are used. The point representation of a patch is obtained through regular sampling of the triangle patch. The system uses traditional polygonal hardware rendering pipeline for polygonal patch rendering. We implement point rendering pipeline using hardware vertex and pixel shaders. The final system combines the advantages of both polygonal and point rendering pipelines and achieves relatively high image quality while maintaining interactive rendering speed.

HYBRID 3D-MODEL REPRESENTATION THROUGH QUADRIC METRICS AND HARDWARE ACCELERATED POINT-BASED RENDERING

by
Hanli Ni

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2005

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Triangles have been the *de facto* primitives for 3D surface representations for many years, especially in the realm of real time rendering. Huge amounts of research has been done in polygon surface modeling, simplification, level of detail management and texture mapping. Polygon rendering algorithm development has been accompanied by the support of graphics hardware. The traditional graphics hardware rendering pipeline has a relatively efficient data representation and is able to take full advantage of coherence within triangle screen projections during scan-line rasterization.

However the expectation for highly realistic 3D images has pushed the complexity of triangle models to increase continuously. Currently, realism means millions of triangles. As the triangle count increases and the triangle size decreases, screen projections of triangles cover fewer pixels. In some cases, projections are even sub-pixel size, which makes classic acceleration algorithms less appealing. On the other hand, progress in 3D acquisition

technology has also increased the complexity of available 3D objects. Modern laser 3D scanning devices are able to acquire huge volumes of point data [16]. To avoid the time consuming triangulation process, an alternative surface modeling and rendering primitive is needed.

One of the alternatives is point-based rendering. The idea of using points as rendering primitives was briefly investigated by Levoy and Whitted for the special case of continuous, differentiable surfaces [17]. Recently, point-based rendering has attracted a lot of research attention and algorithms for point data acquisition, processing, rendering and hardware pipeline acceleration were developed [3, 4, 13, 14, 22, 23, 27, 35] . Points have some advantages over triangles as representation and rendering primitives. First, point data has no connectivity overhead, this overhead is especially a problem at high triangle counts. Second, point data has the highest degree of freedom, therefore, more flexible modeling algorithms can be developed. Third, the advances of features in the latest graphics hardware, especially the emergence of vertex and fragment programs, made the efficient hardware point-based rendering possible.

Although point-based rendering has demonstrated its potential application in real-time rendering, it has its limitations. Previous research showed that sufficient densities and proper sampling patterns are required for points to be effective. It is also true that as the most mature and widely used 3D rendering primitive, triangles will continue to be an important part of real-time graphics. It is of great interest to build a hybrid renderer to exploit advantages of both triangles and points. Some rendering systems have explored this area [6, 8]. In this study, the concept of hybrid model representation, a 3D model composed of both triangle

and point patches, is presented. A rendering system is developed to build the hybrid model by assigning different representation primitives to different patches through certain metrics. During rendering, patches are rendered using point or polygon pipelines as appropriate.

Some important problems are addressed in this study. Point sample and its data structure are formally defined. Surface clustering algorithms and metrics are discussed in detail. Our major contribution is to apply the quadric error metric proposed by [9] to do surface clustering and build hybrid models. The quadric error metric was initially used for surface simplification. The thesis discusses the point sampling technique and sampling rate. It also talks about the spatial data structure used to represent the hybrid model. Furthermore, it discusses the boundary conditions between patches, which may leave holes during rendering. Hardware point-based rendering pipeline is implemented and the rendering efficiency results are presented.

The thesis is organized as follows. Chapter 2 talks about some related works on point-based rendering, hybrid rendering systems, hardware implementations and some surface clustering methods. A system overview is first given in Chapter 3. It then talks about the methodology used in this study. Chapter 4 is the results and discussions. We conclude this study in Chapter 5.

Chapter 2

Related Works

2.1 Point-based Rendering

Points have often been used historically to model certain natural phenomenons such as smoke, clouds, dust, fire, water and ‘soft’ objects as trees [25, 26, 30]. This modeling technique is well known as particle systems. Particles generated by the rendering system have a limited lifetime. During their lifetime, particle behaviors are defined through certain physical models. Particles may change position, speed, direction, color and size. This technique has been successfully used as special effects in motion pictures and also in 3D games. In Pixar’s recent animation ‘Finding Nemo’, particle system was used to model and render ocean water [24].

A technical report by Levoy and Whitted [17] first discussed the feasibility of using points as rendering primitives. Their work touched many areas of active research on point-based

3D graphics today. These areas include point data acquisition and preprocessing, point-based data structures and representation, point-based rendering algorithms, texture filtering and shading.

There exist two types of methods to acquire point data, sampling existing geometry and using laser scan devices. Levoy and Whitted pointed out that the surface must be continuous and differentiable in a small neighborhood around each point for proper sampling. Grossman and Dally proved that the surface is adequately sampled if the side length of every triangle resulted from the triangulation of the sampled points is less than the side length of a pixel [13]. Adequately sampled means no holes will appear during rendering from any angle. Pfister et al. obtained point samples from three layered depth images at orthographic views and guaranteed that the maximum distance between adjacent point samples is less than the pixel reconstruction filter [23].

Scanned real objects are becoming popular due to advances in 3D scanning technologies. Models with millions of points can be easily collected with scanning devices [16]. Due to a variety of physical effects and limitations of the model acquisition procedure, raw data sets are prone to noise and distortions. Some preprocessing and resampling of the raw data are usually needed. Pauly and Gross applied spectral analysis to filter and resample point patches [22]. Alexa et al. developed tools to represent point set surfaces with varying sampling density based on differential geometry [2].

Because of the huge number of samples within a single model, proper data structures must be applied to organize point samples. A common strategy is to use hierarchical data struc-

tures, for example a tree structure. Rusinkiewicz and Levoy applied a bounding sphere hierarchy to represent a point cloud [28]. Their algorithm builds up the tree by recursively splitting the node along the longest axis of the bounding box. They quantized point properties such as position and normal to compress the model. Pfister et al. used an octree based representation of point samples [23]. Each octree node is attached with a layered depth cube (LDC) with increasing sampling rate as traveling from root to leaves. Each LDC is composed of three orthogonal layered depth images (LDI) [5]. Gobbetti and Marton used binary trees to represent point sampled models by reordering and clustering points [11]. The tree traveseral picks the right point sampling densities according to the projected size in the image.

After collecting and preprocessing point samples and selecting the proper data structures, the final stage is to develop efficient point-based rendering algorithms. The major challenge from point-based representations is that we need to render continuous surfaces (hole free) from discrete samplings of the underline surfaces (unlike triangle models). One popular technique is called splatting. It was originally studied by Levoy and Whitted in their early technical report and was also mentioned in the context of volume rendering [33]. The basic idea of splatting is that a single point is mapped to multiple pixels on the screen and the color of a pixel is the weighted average of the colors of the contributing points. Pfister et al. proposed a novel technique called visibility splatting to determine visible points and holes [23]. In their technique, points are represented as disks and during rendering, these disks are projected from object space to screen space z-buffer in order to perform the visibility test. Zwicker et al. formally defined the framework of surface splatting using

rigorous mathematical analysis [35]. In their work, the rendering process is a concatenation of sampling, warping and filtering. All these steps are represented as a reconstruction kernel function to improve efficiency. This is actually an extension of the well known texture filtering framework using the anisotropic elliptical weighted average (EWA) filtering proposed by Heckbert [15]. Later Zwicker et al. introduced perspective accurate splatting using homogeneous coordinates to obtain the correct screen projection shape [36].

Grossman and Dally avoided using splatting by ignoring holes during rendering and filling gaps at an image reconstruction stage [13]. Each pixel is assigned a weight between 0 to 1 indicating the confidence of whether or not it is a hole (0 means it is a hole). Then a two phase "pull-push" algorithm is applied to fill the gaps [12]. This is essentially to generate a succession of lower resolution approximations of a image. Their work assumes orthographic views and predefined target resolution and magnification. Max and Ohsaki used a similar method to render trees [21]. To overcome the limitations of viewing parameters, techniques that are able to dynamically adjust the sampling rate on a frame by frame basis have also been developed [1, 31, 32].

2.2 Hardware Accelerated Point-Based Rendering

Surface splatting techniques achieve superior visual quality, however, the high computation cost has limited its application in real-time graphics. In recent years, the increasing efficiency and programmability of modern graphic cards [18, 20] has triggered the development of hardware-based splatting methods. A diagram of a modern graphics pipeline is

shown in Figure 2.1. Todays graphics chips, such as the NVIDIA GeForce and the ATI Radeon replace the fixed function vertex and fragment (including texture) stages with programmable stages. These programmable vertex and fragment engines execute user-defined programs and allow fine control over shading and texturing calculations. A vertex program is run on each incoming vertex from the application and the computed results are passed on to the rasterization stage. Typical computations include vertex transformation, normal transformation and normalization, texture coordinate generation, lighting and etc. A fragment program is run on each incoming pixel from the reaterization stage and the computed results are passed on to display. Typical computations include operations on interpolated values, texture access, texture application, fog and etc.

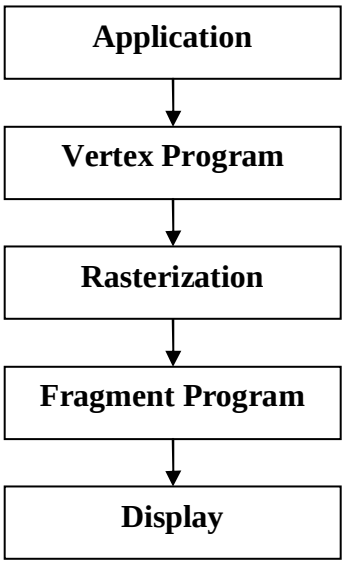


Figure 2.1: The programmable graphics pipeline.

The difficulty of surface splatting is when to blend two splats if two splats are projected to the same pixel. Only closely overlapping splats should be blended when they locally

belong to the same surface; while in other cases when the z-distance between the splats is above a certain threshold, the front-most splat should overwrite the splats behind. To solve this problem, a two pass rendering algorithm is used [3, 14, 27]. The visibility splatting pass only renders the z-buffer with all objects shifted away from the viewer by ε. The second pass renders all splats with filter blending turned on, but it does not alter the z-buffer, thereby blending only those splats that differ by less than ε in depth.

Using the pixel shaders of current graphics hardware allows the rasterization of elliptical splats by rendering just one vertex per splat. Computing the projected size in a vertex shader triggers the rasterization of an image space square. A fragment shader processes each of its pixels and constructs the elliptical shape by discarding pixels outside the ellipse. Extensions of this framework includes perspective accurate splatting [36], using a different affine approximation to the projection; and phong splatting [4], by computing per-pixel lighting.

2.3 Hybrid Rendering Systems

Points and triangles have significant differences in terms of model representation and rendering. Both have their own advantages and disadvantages. If the screen projections cover multiple pixels, algorithms that utilize coherence are very efficient for triangle rendering. If the screen projections are sub pixel sizes, point-based rendering has less overhead and is more flexible. Some work has been done on developing hybrid rendering systems, which are able to build point and triangle hybrid models and have both point and triangle rendering

capabilities. POP is an early system developed to visualize large data sets [6]. The system builds tree model structures by having triangle representation at leaf nodes and point representation at intermediate nodes. The design of the system suggested that triangles at leaves are to ensure the quality of the rendering while points are to speed up the rendering. POP chooses a different number of points and triangles based on viewing location. The closer the model is to the eye, the higher percentage of triangles rendered. The whole system treats points as secondary rendering primitive, mainly for pre-viewing purposes. The most detailed level is still rendered with triangles, which is not very efficient at high triangle counts. It fits the need for visualization applications, but is not suitable for 3D real-time applications. A similar system developed by Coconu and Hege is able to switch between points and triangle rendering based on sampling density [7]. PMR is a system that also uses a hierarchy both in points and triangles [8]. The difference between PMR and POP is that the hierarchy in PMR is built according to the feature geometry in the object space rather than its projection in the screen space. Therefore, the rendering is not affected by viewing parameters. The system chooses a surface representation according to the surface curvature information. For example, flat surfaces are represented by triangles while highly curved surfaces are represented by points. PMR takes as input a point cloud and does the triangulation using Voronoi regions. This system does not take polygonal models. The design philosophy of our rendering system is very close to that of PMR. The idea is that more planar surfaces are more efficiently rendered with triangles. Points are best used for surface areas with high variations. Certain metrics based on planarity measures need to be applied to cluster surfaces for proper representation and rendering. More details on the clustering metric and algorithm are provided in Chapter 3.

2.4 3D Surface Simplification and Clustering Techniques

In this study, we are interested in clustering 3D model surfaces into patches for either point or triangle representation. Although surface simplification is not the focus of this study, it is in some aspect related to surface clustering. Surface simplification is the process of generating simpler versions of detailed geometric surface models. In this study, one surface simplification and clustering technique is of our interest. A quadric error metric was proposed for surface simplification by Garland and Heckbert [9]. Their algorithm is based on pair contraction, which iteratively contracts pairs of adjacent vertices to a single new vertex according to a cost function. The cost function is based on a 4×4 symmetric matrix associated with each vertex. The matrix is related to the sum of distances from the vertex to the planes associated with it. Initially, the error at each vertex is 0 because the planes associated with each vertex pass through it. The error grows as vertex are combined together. This metric is used later on by Garland for surface clustering [10]. The only difference is that no actual contraction is performed in the clustering algorithm. The method is fast and general. Because the metric is a measure of the accumulated distance of planes to a vertex, it is a indication of the planarity of the clustering, which is the criteria this study uses to decide on the patch representation (point or triangle).

Chapter 3

Methodology

3.1 Research Goal and Contribution

The goal of this study is to build a hybrid real-time rendering system. The system is composed of two parts, the preprocessor and the renderer (each part is shown within dashed lines in Figure 3.1). Shaded blocks represent key algorithms in this study. Given a 3D triangle model as input, our system is able to cluster the model surfaces into patches. The clustering is based on the planarity of the surface (§3.2). Either points or triangles are selected as the representation for each individual patch. Patch sampling algorithms are used to obtain point data. Then, the system organizes these patches into an octree. The rendering of the model is a tree traversal process. A traditional polygon rendering pipeline is used for triangle patch rendering. A point-based rendering pipeline applying splatting techniques is used for point patch rendering. A hardware accelerated implementation is provided for the

point-based rendering pipeline, utilizing hardware vertex and fragment shaders in modern graphics boards, to achieve interactive frame rates.

The contributions of this study are two-fold. First, a two phase clustering algorithm based on Garland's quadric error metrics [10] is used to generate surface patches and build hybrid models. Patch representation is independent of the viewing parameters because patches are generated based on the object space surface geometric information. Our method successfully identifies highly varied areas (point representation) on a 3D model from those planar areas (triangle representation). Second, unlike previous hybrid renderers, our rendering system treats points and triangles as equivalent primitives (most previous hybrid rendering systems treat points as the previewing primitive). The renderer is able to choose the best representation to ensure both rendering quality and speed. One extension of this renderer could be level of detail (LOD) management. There could be a 3D model hierarchy composed of a tree with different LODs associated with each node, the system could pick the appropriate set of nodes (a cut through the tree) for rendering according to a cost function. The cost function could have input as viewing parameters and other related parameters. We implement the complete point-based rendering pipeline from point sampling to hardware point-based acceleration, which parallels the traditional triangle rendering pipeline. Our rendering system gives the user great flexibility to choose the desired primitive to render parts of a 3D model.

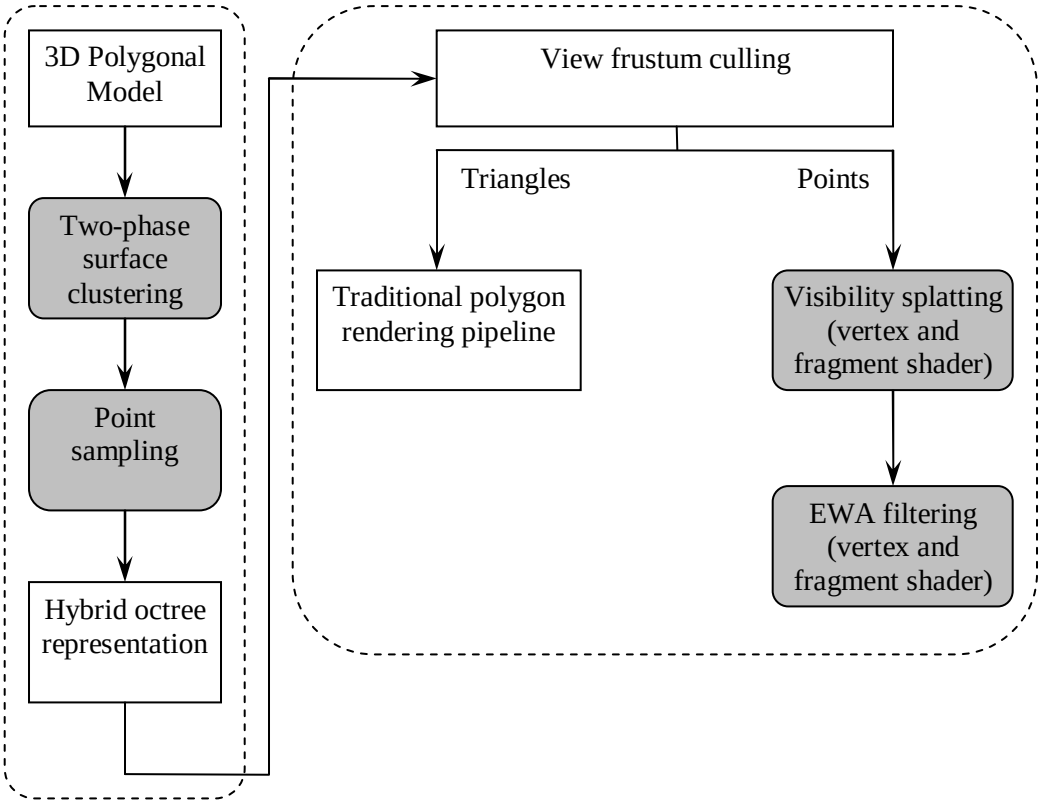


Figure 3.1: Schematical overview of the hybrid renderer and rendering algorithm.

3.2 Surface Clustering Algorithms

In this section, we lay out the foundation of Garland's surface simplification and clustering algorithm. First, we define the quadric error metrics used in both surface simplification and clustering. Then we describe our two phase clustering algorithm.

3.2.1 Quadric Error Metrics and Surface Simplification

One common strategy of surface simplification is called **iterative edge contraction**. The basic idea is that during each iteration, an edge is selected and the incident vertices are replaced by a new vertex with proper updating of the edges associated with the old vertices (Figure 3.2). The position of the new vertex is decided either by picking one of the old vertices or by finding the optimal position that represents a minimum modification of the surface. Both need a cost function to evaluate the effect of the contraction operation. Therefore, picking the right cost function is the key to the success of this type of simplification algorithm. Naturally, we want to contract vertices that are similar. To do that, we need a measure to quantify similarity. One heuristic is the planarity of the local environment of the vertex. A quadric error metric [9] is an efficient way to measure planarity.

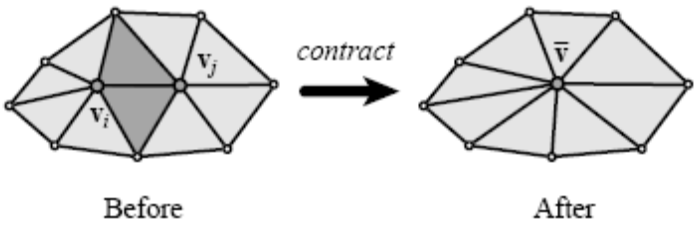


Figure 3.2: Edge contraction example [9].

First, a set of planes are associated with each vertex in the model. The initial selection of planes are the incident triangles of the vertex. Each plane is defined by

$$\mathbf{n}^{\mathbf{T}}\mathbf{v} + d = 0 \tag{3.1}$$

where $\mathbf{n} = \begin{bmatrix} a & b & c \end{bmatrix}^{\mathbf{T}}$ is a unit normal (i.e., $a^2+b^2+c^2=1$) and d is a scalar constant. The square distance of a vertex $\mathbf{v} = \begin{bmatrix} x & y & z \end{bmatrix}^{\mathbf{T}}$ to the plane is given by

$$D^2(\mathbf{v}) = (\mathbf{n}^{\mathbf{T}}\mathbf{v}+d)^2 = (ax+by+cz+d)^2 \tag{3.2}$$

The error at each vertex $\mathbf{v}$ is defined as the sum of squared distances of the set of planes P associated with the vertex

$$Err(\mathbf{v}) = \sum_i D_i^2(\mathbf{v}) = \sum_i (\mathbf{n}_i^{\mathbf{T}}\mathbf{v}+d_i)^2 \tag{3.3}$$

It is noted that initially the error at each vertex is zero because each vertex is the intersection of the associated planes. For any contraction $(\mathbf{v_1}, \mathbf{v_2}) \rightarrow \overline{\mathbf{v}}$, a new error needs to be derived. Ideally, the union of the planes from each vertex needs to be computed and the error needs to be re-calculated by Equation 3.3. To do this, we need to keep tracking a list of planes for each vertex (new or original). This has two disadvantages. First, the amount of storage increases as the simplification goes. Second, the computation cost is proportional to the number of planes associated with each vertex. It would be nice to have both constant storage and computation for each iteration. Garland [9] proposed to use the summation of the error calculated by Equation 3.3 for each vertex to represent the error of the new vertex. To see how this is practical, we can rewrite Equation 3.1 as follows:

$$D^2(\mathbf{v}) \quad = \quad (\mathbf{n}^{\mathbf{T}}\mathbf{v}+d)^2$$

$$\begin{aligned} &= (\mathbf{v^T n} + d)(\mathbf{n^T v} + d) \\ &= (\mathbf{v^T n n^T v} + 2d\mathbf{n^T v} + d^2) \\ &= (\mathbf{v^T}(\mathbf{n n^T})\mathbf{v} + 2(d\mathbf{n})^{\mathbf{T}}\mathbf{v} + d^2) \end{aligned} \tag{3.4}$$

We can define a *quadric* Q as a triplet

$$Q = (\mathbf{A}, \mathbf{b}, c) \tag{3.5}$$

where $\mathbf{A}$ is a 3×3 matrix, $\mathbf{b}$ is a vector and c is a scalar. The quadric assigns a value $Q(\mathbf{v})$ to every vertex $\mathbf{v}$ by the equation

$$Q(\mathbf{v}) = \mathbf{v^T A v} + 2\mathbf{b^T v} + c \tag{3.6}$$

Comparing Equation 3.6 to Equation 3.4, we have

$$D^2(\mathbf{v}) = Q(\mathbf{v}) \tag{3.7}$$

where $Q = (\mathbf{n n^T}, d\mathbf{n}, d^2)$. Q can be represented by a single 4×4 matrix

$$\begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}$$

This is called *fundamental error quadric* [9]. Therefore, the error defined in Equation 3.3 can be rewritten as

$$Err_Q(\mathbf{v}) = \sum_i D_i^2(\mathbf{v}) = \sum_i Q_i(\mathbf{v}) = Q(\mathbf{v}) \tag{3.8}$$

Each vertex is associated with a quadric matrix and the quadric Q for any new vertex $\overline{\mathbf{v}}$ from a contraction of an edge $(\mathbf{v}_i, \mathbf{v}_j)$ is $Q = Q_i + Q_j$. And the cost of contraction is $Q(\overline{\mathbf{v}}) = Q_i(\overline{\mathbf{v}}) + Q_j(\overline{\mathbf{v}})$. By doing this, the cost of storage and computation for each vertex at every iteration is constant. We can find $\overline{\mathbf{v}}$ by solving $\overline{\mathbf{v}} = -\mathbf{A}^{-1}\mathbf{b}$, which gives the optimal solution if $\mathbf{A}$ is not singular. The simplification algorithm is summarized in Algorithm 3.1.

Require: Initialize the quadric Q_i for each input vertex i and a heap H for sorting contraction cost
1: **for** each vertex pair $(\mathbf{v}_i, \mathbf{v}_j)$ **do**
2: Compute $Q = Q_i + Q_j$
3: Compute $\overline{\mathbf{v}}$
4: Compute the contraction cost $Q(\overline{\mathbf{v}}) = Q_i(\overline{\mathbf{v}}) + Q_j(\overline{\mathbf{v}})$
5: Place pair in H keyed on cost $Q(\overline{\mathbf{v}})$
6: **end for**
7: **repeat**
8: Remove the pair $(\mathbf{v}_i, \mathbf{v}_j)$ from the top of the heap.
9: Perform contraction $(\mathbf{v}_i, \mathbf{v}_j) \rightarrow \overline{\mathbf{v}}$
10: Set the new quadric $Q = Q_i + Q_j$, update the remaining pairs and the heap
11: **until** the desired approximation is reached

Algorithm 3.1: Simple surface simplification using quadric metric

3.2.2 Surface Clustering Using Quadric Metric

Garland [10] also applied the quadric error metric in surface clustering. Rather than doing the actual contraction, the modified algorithm groups similar surfaces together. The output of the algorithm is a hierarchy of surfaces (Figure 3.3). Initially, each triangle face is represented as a leaf node in the hierarchy. For each iteration, the algorithm combines two

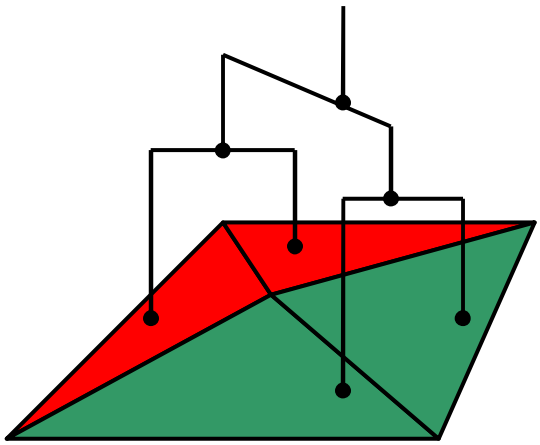


Figure 3.3: A tipical face hierarchy of a patch with 4 triangles. Initially each triangle is a cluster. Then red triangles form a cluster and green triangles form another cluster. Finally the two clusters are combined into the root cluster.

nodes and forms a new node based on the error metrics. There are two differences from Algorithm 3.1. First, Q is calculated as the distances from different vertices to the best fitting plane of the vertices. Second, the final result is averaged. Assuming each node has a set of faces $\{f_1,...,f_n\}$, a set of vertices $\{\mathbf{v}_1,...,\mathbf{v}_k\}$ and a least square best fit plane to this set of vertices, the fit error of a given plane $\mathbf{nv^T}+d=0$ is

$$E_{fit}=\frac{1}{k}\sum_{i=1}^{k}(\mathbf{n^T}\mathbf{v}_i+d)^2 \tag{3.9}$$

where $\mathbf{n}$ is the normal of the plane and d is the offset. Similar to Equation 3.4, the equation above can be rewritten as

$$(\mathbf{n^T}\mathbf{v}+d)^2=\mathbf{n^T}(\mathbf{v}_i\mathbf{v}_i^\mathbf{T})\mathbf{n}+2d\mathbf{v}_i^\mathbf{T}\mathbf{n}+d^2 \tag{3.10}$$

If we define a quadric

$$P_i = (\mathbf{A}_i, b_i, c_i) = (\mathbf{v}_i\mathbf{v}_i^{\mathbf{T}}, \mathbf{v}_i, 1) \tag{3.11}$$

then Equation 3.10 can be represented as $P_i(\mathbf{n}, d)$. The total error is then

$$E_{fit} = \frac{1}{k}\sum_i P_i(\mathbf{n}, d) = \frac{1}{k}\left(\sum_i P_i\right)(\mathbf{n}, d) \tag{3.12}$$

Therefore, we can represent the cost of combining two nodes by the sum of the quadrics of each node $\left(P_i + P_j\right)(\mathbf{n}, d)$.

One challenge of the modified algorithm is to find the best fitting plane of the vertices. The technique used in Garland's paper is based on principal component analysis (PCA). The basic idea is to find the plane by first using the covariance matrix

$$\mathbf{Z} = \frac{1}{k-1}\sum_{i=1}^{k}(\mathbf{v}_i - \overline{\mathbf{v}})(\mathbf{v}_i - \overline{\mathbf{v}})^{\mathbf{T}} \tag{3.13}$$

where $\overline{\mathbf{v}} = \left(\sum_i \mathbf{v}_i\right) \Big/ k$. The eigenvector with the least eigenvalue is the normal of the plane. It can be proved that the average of all the positions of the vertices is a point on the plane. In practice, Garland adds two more metrics to do the clustering. The first is E_{dir}, a measure of the deviation of the plane normal $\mathbf{n}$ from surface normals:

$$E_{dir} = \frac{1}{w}\sum_i w_i(1 - \mathbf{n}^{\mathbf{T}}\mathbf{n}_i)^2 \tag{3.14}$$

where w_i is the area of face f_i and $w = \sum_i w_i$ is the total area of the face cluster. This metric avoids surface folding back within a cluster. It can also be presented as a quadric. The

second is E_{shape}. This error metric is to control the shape of the cluster, which makes the cluster as circular as possible by taking into account the area w and perimeter ρ relationship of the cluster. The clustering algorithm is shown in Algorithm 3.2.

Require: Initialize the quadric P_i for each input face i and a heap H for sorting contraction cost
1: **for** each node pair $(\mathbf{node}_i, \mathbf{node}_j)$ **do**
2: Compute the best fit plane $\mathbf{nv}^{\mathbf{T}} + d = 0$ for the vertex set $V_{node_i} \cup V_{node_j}$
3: Compute the cost of merging $\left(P_i + P_j\right)(\mathbf{n}, d)$
4: Place pair in H keyed on cost
5: **end for**
6: **repeat**
7: Remove the pair $(\mathbf{node}_i, \mathbf{node}_j)$ from the top of the heap.
8: Create a new node in the hierarchy
9: Set the new quadric P, update the remaining pairs and the heap
10: **until** the desired level of clustering is reached

Algorithm 3.2: Simple surface clustering using quadric metric

We assume the input model M initially having n valid node pairs. The goal is to produce a cluster hierarchy having m cluster trees. Each iteration combines two nodes and forms a new node. We also assume that the maximum node degree is bounded by a constant k, this gives an upper bound on updating the neibouring node pairs after pair contraction. Constructing all the initial quadrics takes $O(n)$ time. Placing all the resulting candidates in a heap requires $O(n \log n)$ time. Thus, the total complexity of initialization is $O(n \log n)$.

For each iteration i of the clustering, we need to select the minimum cost pair, contract it, and update the local neighborhood. Selecting the minimum cost pair takes $O(\log(n-2i))$. Contracing the pair and updating the neighborhood require $O(k \log(n-2i))$. The total complexity for each iteration is therefore $O(\log(n-2i))$. Summing over all iterations, the

total cost for the clustering is

$$\log n + \log(n-2) + \log(n-4) + \ldots + \log m$$
$$\leq \log n + \log(n-1) + \log(n-2) + \ldots + \log m$$

which is simply

$$\log \frac{n!}{m!} = \log n! - \log m! = O(n \log n - m \log m)$$

Thus, the overall complexity of the clustering algorithm is $O(n \log n)$.

3.2.3 Two-phase Clustering

Based on Algorithm 3.2, we use a modified algorithm to find the best representations of the input model. The heuristic is that the higher the variation of the surface, the more efficient it is to represent and render the surface using points. A quadric error metric is a good measure of the planarity of the surface (§3.2.1), which is an indication of the surface variation. The algorithm goes in two phases. The first phase is exactly the same as Algorithm 3.2. The algorithm clusters surfaces based on a predefined planarity error threshold. The maximum size of a cluster is also set in phase one. All three error metrics (E_{fit}, E_{dir} and E_{shape}) are turned on in phase one. Before the second phase, the algorithm marks all the clusters resulted in phase one. Because the error measure is set in the first phase, the clusters at the highly varied areas of the model tend to be smaller than those at the planar areas of the model. The second phase tries to combine these small clusters, and outputs clusters in similar shapes and sizes. Therefore, in this phase, E_{shape} is turned on, E_{fit} and E_{dir} are

turned off. We add a new error measure E_{size}. This is a measure of the size of the patch, which applies more penalty on merging larger patches. The newly formed clusters will be unmarked. The output of the two-phase clustering algorithm is also a surface hierarchy. The marked clusters will be represented using triangles, while unmarked clusters will be represented using points. The algorithm is summarized in Algorithm 3.3.

Require: Set the planarity error threshold, set the maximum size of a cluster
1: turn on E_{fit}, E_{dir} and E_{shape} {start phase one}
2: Run Algorithm 3.2
3: Mark all the resulting clusters
4: Turn off E_{fit} and E_{dir}, add E_{size} {start phase two}
5: Run Algorithm 3.2
6: Unmark all the newly created clusters

Algorithm 3.3: Two-phase surface clustering

3.2.4 Level of Detail (LOD) Control within Clusters

The two-phase clustering algorithm (§3.2.3) does not change the geometry of the input model. The cluster hierarchy is at the highest LOD. During rendering, it is not always necessary to render the cluster using the highest level of detail. For example, if the screen projection of the cluster is relatively small, it is enough to render the cluster at a lower LOD. It will be nice to have some sort of LOD control within clusters. In this section, we concentrate on LOD on triangle clusters. we talk about LOD on point clusters in Section 3.3.

For each triangle cluster, we could build a series of surface patches with decreasing LODs. For simplification purposes, we could use Algorithm 3.1 directly. However, Algorithm 3.1 does not preserve boundaries during the simplification process. It could be possible to leave

holes between clusters if the boundary edges are contracted during simplification. Garland [9] provides a solution to preserve boundaries of models such as terrain height fields, which could be used here. First, we will need to mark all the boundary edges during clustering. For each boundary edge, we could generate a perpendicular plane through this edge. This plane can be easily converted into a quadric. Then Algorithm 3.1 could be applied on this cluster. Because the high penalty imposed by the added planes, boundary edges will avoid being the target of edge contraction. This technique is used to preserve color discontinuities of a surface patch by Garland [9].

3.3 Point Sampling Algorithms

In this section, we describe the point sampling algorithm we use to sample the patches from the two phase clustering algorithm in the previous section. We first define point samples, and then talk about the algorithm and the sampling pattern and sampling density.

3.3.1 Point Sample Definition

To represent a 3D surface with points, each point is actually an oriented disk (Figure 3.4). First, we need three scalars to represent the spatial coordinates of a point and three scalars to represent the color of the point (one scalar for alpha transparency can also be added). A normal vector is also needed. We need a radius r_p to represent the size of the disk. To perform anisotropic EWA filtering, two tangential vectors (S, T) are needed. The normal

and the two tangential vectors form a local coordinate system at the point. Tangent vectors can be calculated on the fly during rendering (§3.4.1), therefore, we don't have to save these two vectors. The complete data representation of a point sample with the above information is called a surface element or a surfel [23]. Therefore, a surfel is a multidimensional representation. Other information can also be incorporated, such as texture coordinates. Further more the most accurate representation of a point sample is not by symmetric disk, but by an ellipse, whose axis correspond to the two tangent vectors.

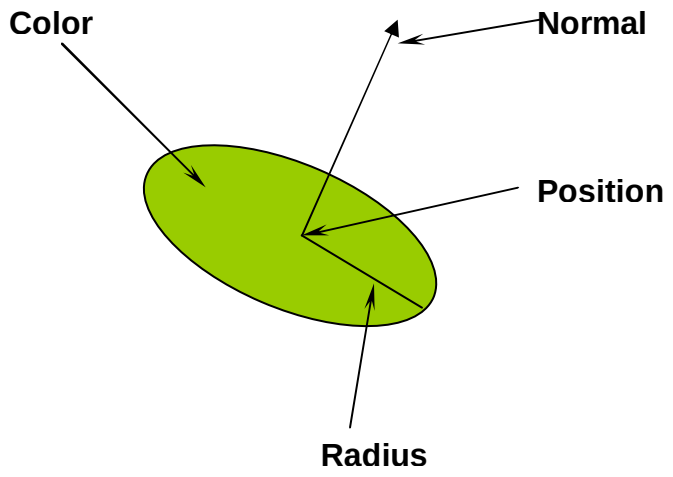


Figure 3.4: Surfel definition.

3.3.2 Surface Sampling

The output of the two-phase clustering algorithm (§3.2.3) is a set of triangle patches. Our heuristic is that patches generated during phase two represent highly varied areas on the surface, therefore, it is more efficient to render them by points. The goal of this section is to find a surfel representation of the triangle geometry by doing sampling. Surface sampling

needs to meet two requirements. First, the sampling rate has to be high enough to record the details of the model. Second, we need to choose the radius of the surfel disk carefully to ensure a waterproof surface reconstruction.

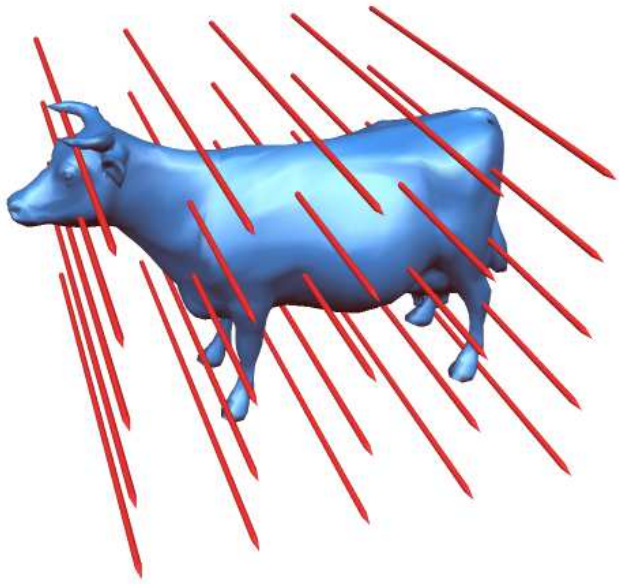


Figure 3.5: Concept view of the sampling algorithm. This figure only shows sampling rays from one face of the bounding box.

We use a method similar to the one used by Pfister et al. [23]. The basic idea is to sample the patch from three faces of the bounding box of a patch through ray casting (Figure 3.5). At each intersection point between a ray and the patch, a surfel is created with its position and normal. Because triangle patches are used, we take advantage of barycentric coordinates of each triangle to do normal interpolation. If necessary, we also do the barycentric coordinates interpolation to calculate texture coordinates and color values of each surfel. Perturbation of the surface normal or of the geometry for bump and displacement mapping can be performed on the geometry before sampling or during ray casting using procedural shaders. To simplify the sampling process, we use the axis aligned bounding box to

generate the sampling grid. The detailed sampling algorithm is listed below

```
Require: Initialize the Surfel Cloud (a list of surfels) of the patch
 1: Generate the axis aligned bounding box of the patch
 2: Compute the sampling distance (h) of the patch
 3: for each face of the bounding box do
 4:     Generate the sampling grid
 5:     for each ray on the grid do
 6:         for each triangle in the patch do
 7:             Compute the intersection of the ray and the triangle
 8:             if intersection is in the triangle then
 9:                 Generate a surfel with the intersection as the position
10:                 Compute normal, texture coordinate, color and etc through interpolation
11:                 Set the radius of the surfel
12:                 Add the surfel into the Surfel Cloud
13:             end if
14:         end for
15:     end for
16: end for
```

Algorithm 3.4: Surface sampling through ray casting

Line 3 of Algorithm 3.4 only needs to be executed three times because parallel faces of the bounding box produce the same sampling grid. If we assume the bounding box is a cube and there are n^2 (n is the number of rays per side of the grid) rays per sampling grid; also we assume the number of triangles in a patch is m, then the complexity of the sampling algorithm is $O(mn^2)$. One way to improve the efficiency of this algorithm is to build an octree for each triangle patch. This saves some cost on ray-triangle intersection calculation. Because the sampling is a preprocessing step, it does not affect the rendering speed. For simplicity, we do not use the octree for each patch. We call the resulting sampled patches surfel clouds.

Line 2 of Algorithm 3.4 computes the sampling distance (h) of the patch. This value is used to set the sampling grid distance. As was observed by Lischinski and Rappoport [19], the

imaginary triangle mesh generated by this sampling process has a maximum side length $s_{\max}$ of $\sqrt{3}h$. The minimum side length $s_{\min}$ is 0 when two or three sampling rays intersect at the same surface position. If $s_{\max}$ is less than the radius of the surfel (surface reconstruction filter, see Section 3.4.1), we can guarantee a waterproof point sampled surface [13]. Originally, we choose a global sampling distance (h) that is the same across all the triangle patches. The problem with this is that some patches are under sampled, with a resulting loss of surface detail. Therefore, we let the algorithm adjust the sampling distance per patch. The sampling algorithm calculates the average side length (l) of triangles in a patch and set the sampling distance as $h = \frac{l}{c}$, where c is a constant scaling factor. A typical choice of c is 4. The geometric interpretation is that we make sure there are more than one surfels per triangle after sampling (see detailed results in Chapter 4).

Another issue we consider when applying the sampling algorithm is the boundaries between surfel cloud patches and triangle patches. The concern is the possibility of leaving holes at the boundary area. The actual rendering result exhibits this situation very rarely (We explain the reason in Chapter 4). Therefore, we do not increase the sampling rate along the boundary areas between triangle patches and surfel cloud patches.

In section 3.2.4, we discussed the LOD control for triangle patches. It is also possible to do LOD control for surfel cloud patches. Our sampling grid is regular and we assume the number of sampling rays for each side of the grid is a power of 2. A fully sampled patch (highest LOD) has n rays per side and grid distance h. To get a patch that is one level less detail, we could have $\frac{n}{2}$ rays by picking every second ray in the first sampling grid and double the grid distance h. By doing this recursively, a series of LOD patches could be

generated until one of the three sampling grids has only one ray.

3.4 Point-based Rendering Algorithms

As discussed in Section 3.1, the rendering system organizes patches into an octree representation. The rendering of the model is a tree traversal process. Traditional polygon rendering pipeline is used for triangle patch rendering. Point-based rendering pipeline applying a splatting technique is used for surfel cloud patch rendering. In this section, we first give a brief introduction to EWA and then discuss in detail the hardware accelerated multi-pass point-based rendering algorithm.

3.4.1 EWA Splatting Framework

Elliptical weighted average (EWA) filtering was first proposed by Heckbert [15] for application in texture mapping. Zwicker et al. formalized the framework in the application of point-based rendering [35]. It is essentially an anisotropic filtering technique that in theory avoids aliasing.

Let P_k be a set of points which defines a 3D surface. Each point has a position and a normal. As discussed in Section 3.3.1, the point set is essentially a set of surfels. We assign a color coefficient w_k for each point. We need to define a continuous texture function f_c on the surface represented by P_k. To do this, each point is associated with a radial function r_k at the center of the point. We call these basis functions at each point reconstruction filters, due

to the fact that they form a continuous surface from the discrete point set P_k. If there is a point on the surface with local coordinates u, then the continuous function $f_c(u)$ is defined as the weighted sum:

$$f_c(u) = \sum_{k \in N} w_k r_k(u - u_k) \tag{3.15}$$

where u_k is the local coordinates of point P_k.

This is illustrated in Figure 3.6. Point Q is a point anywhere on the surface. A local parameterization of the surface in a small neighborhood of Q can be constructed. The color value of Q is the sum of the contributions of its local neighbors. Because the basis function of each point has a cut off radial range, point Q lies in a small number of basis functions.

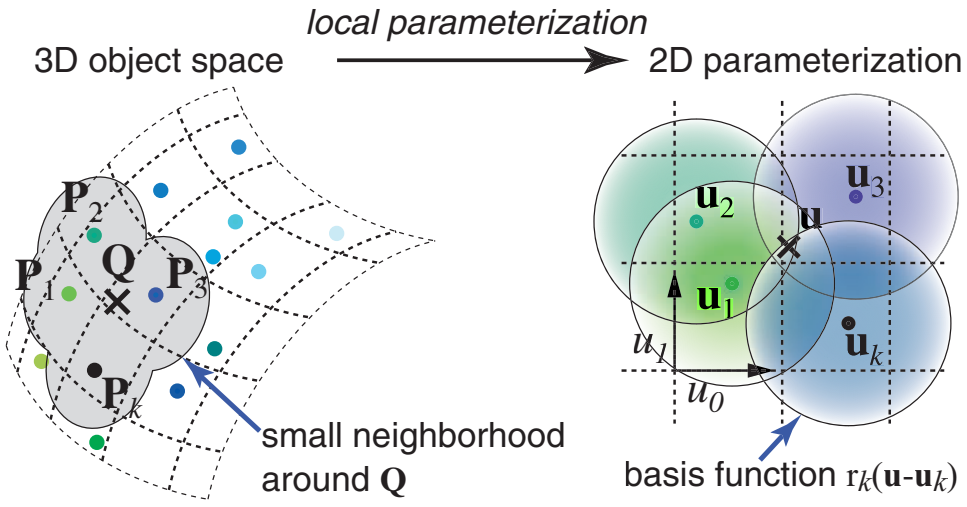


Figure 3.6: Defining a texture function on the surface of a point sampled 3D model [35].

During rendering, each basis function r_k is warped from object space to screen space. It is possible that some of the screen projection of the basis functions will be sub-pixel sizes, creating aliasing artifacts. Therefore, we need to band-limit the warping of r_k by convolving it with a prefilter h. h is related to the Nyquist limit of the screen pixel grid. Usually h is the

identity matrix for unit-spaced pixels. This output function can be written as a weighted sum of screen space resampling filter $\rho_k(x)$:

$$g_c(x) = \sum_{k \in N} w_k \rho_k(x) \tag{3.16}$$

where

$$\rho_k(x) = (r'_k \otimes h)(x - m_k(u_k)) \tag{3.17}$$

where m_k is the local affine approximation of the projective mapping $x = m(u)$ for the point u_k. This approximation is given by the Taylor expansion of m at u_k:

$$m_k(u) = m(u_k) + J_k(u - u_k) \tag{3.18}$$

where J_k is the Jacobian $J_k = \frac{\partial m}{\partial u}(u_k)$.

We choose elliptical Gaussians as the basis functions (reconstruction filters) and prefilters. Gaussians are closed under affine mappings and convolution, which means the resampling filter is also a Gaussian. If the 2D elliptical Gaussian is defined as:

$$G_V(x) = \frac{1}{2\pi |V|^{\frac{1}{2}}} e^{-\frac{x^T V^{-1} x}{2}} \tag{3.19}$$

where $V \in \Re^2$, the object space resampling filter $\rho'_k(x)$ can be represented as:

$$G_{V'_k + J_k^{-1}(J_k^{-1})^T}(u - u_k) \tag{3.20}$$

where V_k^r is:

$$\begin{bmatrix} R^2 & 0 \\ 0 & R^2 \end{bmatrix} \tag{3.21}$$

and R is the maximum distance between surfels, which is related to surface sampling rate. In our case, R is the same as the grid sampling distance h (§3.3.2). The screen space resampling filter $\rho_k'(u)$ can be represented as:

$$\frac{1}{\left|J_k^{-1}\right|} G_{J_k V_k^r J_k^T + I}(x - m_k(u_k)) \tag{3.22}$$

Now we need to evaluate the Jacobian J_k. If we assume that the transformation from object space to camera space only contains uniform scaling, rotation and translation, a point u in camera space can be represented as:

$$P^c(u) = O + u_S \cdot S + u_T \cdot T \tag{3.23}$$

where $O = (O_x, O_y, O_z)$ is the point's position in camera space, $S = (S_x, S_y, S_z)$ and $T = (T_x, T_y, T_z)$ are the tangent vectors in camera space [27]. Next, we map the points from camera space to screen space. This includes the projection to the image plane by perspective division, followed by a scaling with a factor η to screen coordinates (viewport transformation). The scaling factor η is determined by the view frustum and computed as follows:

$$\eta = \frac{h_{vp}}{\frac{2t}{n}} \tag{3.24}$$

where h_{vp} is viewport height, t and n are the standard parameters of the viewing frustum. Hence, screen space coordinates (x_0, x_1) of the projected point are computed as (c_0 and c_1 are given translation constants):

$$\begin{aligned} x_0 &= \eta \cdot \tfrac{O_x + u_S \cdot S_x + u_T \cdot T_x}{O_z + u_S \cdot S_z + u_T \cdot T_z} + c_0 \\ x_1 &= -\eta \cdot \tfrac{O_y + u_S \cdot S_y + u_T \cdot T_y}{O_z + u_S \cdot S_z + u_T \cdot T_z} + c_1 \end{aligned} \tag{3.25}$$

We approximate the Jacobian at the center of the point, therefore, it is the partial derivative of Equation. 3.25 evaluated at $(u_S, u_T) = (0,0)$:

$$J_k = \begin{bmatrix} \frac{\partial x_0}{\partial u_S} & \frac{\partial x_0}{\partial u_T} \\ \frac{\partial x_1}{\partial u_S} & \frac{\partial x_1}{\partial u_T} \end{bmatrix} (0,0) \tag{3.26}$$

which is:

$$\eta \cdot \frac{1}{O_Z^2} \begin{bmatrix} S_x O_z - S_z O_x & T_x O_z - T_z O_x \\ S_y O_z - S_z O_y & T_y O_z - T_z O_y \end{bmatrix} \tag{3.27}$$

Figure 3.7 shows the process.

3.4.2 Hardware Implementation

Due to the recent development of graphics hardware, the hardware implementation of point-based splatting algorithms based on EWA filtering is possible. To achieve this, we employ hardware vertex and fragment shaders. The implementation is a multi-pass rendering process, namely, visibility splatting, EWA filtering and normalization. Our implemen-

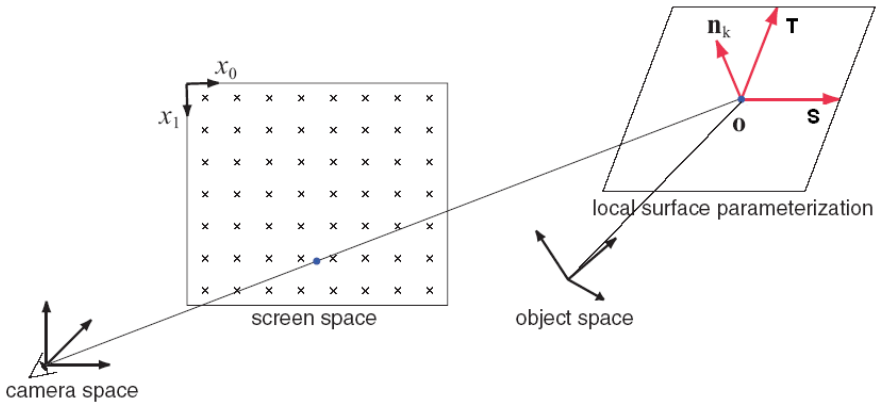


Figure 3.7: Calculating the Jacobian [27].

tation is mostly based on the hardware point-based pipeline proposed by Guennebaud and Paulin [14].

Visibility Splatting

To render water-proof surfaces with point samples, we need to guarantee the overlapping between surfel projections. Because there is no connection information about point samples, we can only assume that neighboring surfels form the local part of a 3D surface. For each pixel on the screen, we need to blend contributions from neighboring surfels. It should be noted that blending does not happen just because the projection of two surfel on the screen overlap. We assume that it is only possible for a set of surfels to form a local surface if the depth values of these surfels differ by no more than a threshold ε and blending only happens if the condition is true. Therefore, the first pass is to render the correct depth buffer for blending and avoiding artifacts. To ensure correct blending, the depth im-

age should be translated away from the viewpoint by a small offset. First we calculate the screen projection size of each surfel and render a square on the screen. The vertex program calculates the point size according to the surfel normal and view parameters. The equation for point size is defined as:

$$size = 2R \cdot \frac{n}{z_C} \cdot \frac{h_{vp}}{t-b} \tag{3.28}$$

where n, t and b are the standard parameters of the viewing frustum and h_{vp} is the height of the viewport. z_C is the depth value in camera coordinates. R is related to the surface sampling rate. In this implementation, R is the maximum distance between two neighboring surfel centers (§3.4.1).

The rasterization results a $size \times size$ image-space square. The fragment program does the per-pixel depth correction. We use the approach by Bostch et. al [4] to find the object space coordinates of the pixel and each pixel's actual depth. A ray through the eye and the pixel on the near plane intersects with the surfel and yields the corresponding point on the surfel (Figure 3.8). The actual depth can be easily calculated. Because the fragment position is in window coordinates, a 2D transformation of the coordinates is applied to each fragment position before doing ray casting. The transformation is given by:

$$Q_C = \begin{bmatrix} Q_x^{vp} \times \frac{2r}{w_{vp}} - r \\ Q_y^{vp} \times \frac{2t}{h_{vp}} - t \\ -n \end{bmatrix} \tag{3.29}$$

where Q_C is the pixel position in camera coordinates. Q_{vp} is the window position of the pixel. w_{vp} is the width of the viewport. Because we have the intersection point, we can

calculate the true depth of each pixel and normalize the depth between $[-1,1]$. The final step is to add a small offset ε along the ray for proper blending of surfels. We also compute the minimum and maximum depth value and kill all fragments that are not in this range.

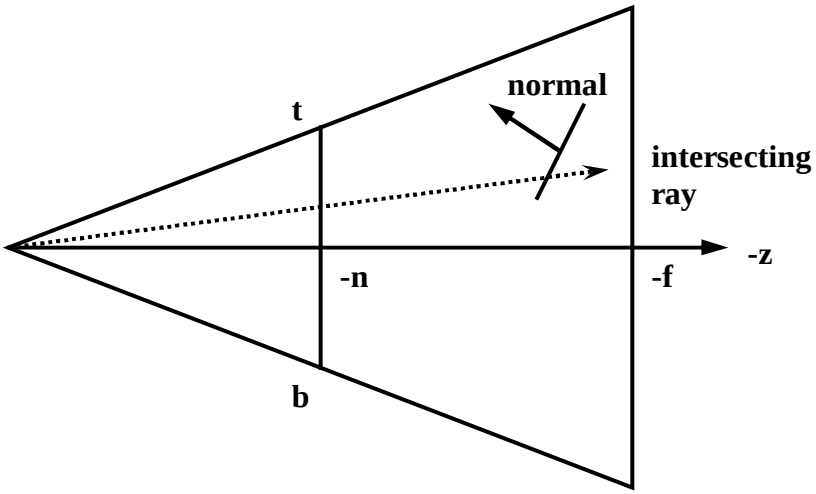


Figure 3.8: Camera space ray casting for per-pixel depth correction.

EWA filtering

The EWA filtering pass is very similar to the previous section. We need to calculate the color value for each surfel. During rendering, blending is turned on. The blending factor for both color RGB and alpha are one. Therefore, the resulting frame buffer pixel value is calculated as:

$$Color^{out}_{RGBA} = Color^{prev}_{RGBA} + Color^{in}_{RGBA} \tag{3.30}$$

where $Color^{prev}_{RGBA}$ is the previous frame buffer value for a pixel, $Color^{in}_{RGBA} and Color^{out}_{RGBA}$ are the incoming and resulting color values respectively. Depth comparison is also turned on. Any incoming fragments with depth values greater than those in the depth buffer are

discarded. In this pass, we don't update the depth buffer because the previous pass already produces the correct depth buffer. In the vertex program, we calculate the screen space resampling filter (Equation. 3.22). The vertex program also calculates the lighting information. Because the Gaussian resampling kernel is computed only for a limited range, we choose a cutoff radius c, such that $\frac{x^T V^{-1} x}{2} < c$ (typically $c = 1$). Therefore, the point size calculation of each surfel becomes:

$$size = 2\sqrt{2c}R \cdot \frac{n}{z_C} \cdot \frac{h_{vp}}{t-b} \tag{3.31}$$

In the fragment program, we test the membership per surfel per pixel using $\frac{x^T V^{-1} x}{2} < c$ and kill any pixel that is outside the surfel ellipse screen projection. The final color is calculated after texture lookup. The color value of each pixel is attenuated by the weight calculated by the Gaussian resampling filter. Therefore, the final frame buffer is the accumulation of the blending of surfels. To avoid color clamping, a scaling factor needs to be applied to each pixel. The scaling factor is chosen to be proportional to $\frac{1}{R^2}$, where R is the radius of the reconstruction filter. Figure 3.9 is a rendering result of 4 surfel disks. The blending effect is clearly shown.

Normalization

Due to the irregular sampling of point models and the truncation of the filtering kernel (when the radius is larger than a threshold c, the contribution of a surfel to a pixel is zero). We can not guarantee that the summation of the weight of each pixel is the same. Therefore,

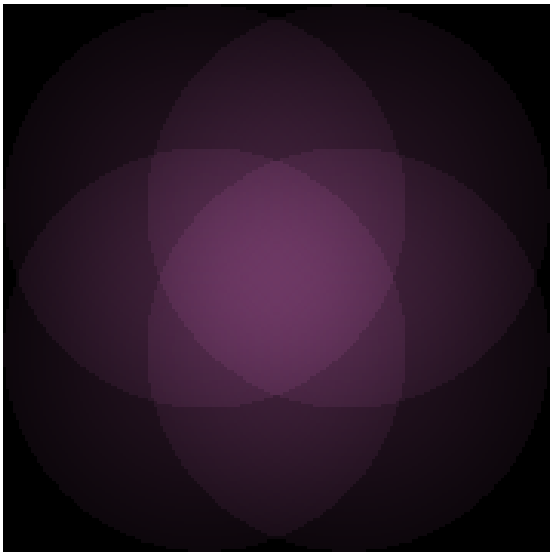


Figure 3.9: Blending effects of the EWA filtering pass of four surfel disks.

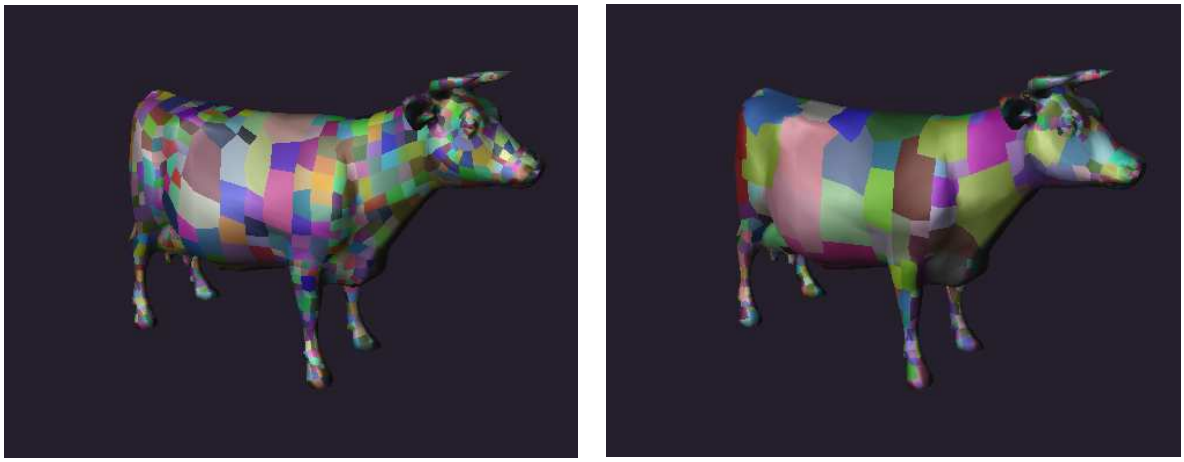
each pixel color is normalized by dividing by the sum of the accumulated contributions, which is stored in the alpha channel of the resulting color value in the second EWA filtering pass. The final result can be rendered as a texture mapped quad [14]. The frame buffer is first copied into a texture. A simple four-sided polygon (rectangle) aligned with the near plane of the viewing frustum is then rendered with the texture. The rectangle is supplied to the hardware by GL_QUADS in OpenGL.

Chapter 4

Results and Discussion

4.1 Surface Clustering

Figure 4.1 shows the result of the Garland's surface clustering algorithm applied on a cow model with 5804 faces. This is actually the phase one of our two-phase surface clustering algorithm. Each triangle in the input model is initialized as a cluster. Each neighboring pair of triangles form an edge. The cost of contraction for each edge is calculated (§3.2) and inserted into a heap. Before clustering, the user needs to either specify the cost threshold or the number of clusters needed. In our system, the user needs to provide the cost threshold because this is an indication of the planarity of the surface. The cost threshold may be negative. Figure 4.1 shows that the higher the threshold the larger the number of clusters. Therefore, higher cost value of an edge implies the two clusters connected by the edge is less planar.



(a) (b)

(c)

Figure 4.1: Phase one clustering result of the cow model (5804 triangles) with different cost thresholds. (a) 0.033, (b) -0.19, (c) -0.96. Each cluster is represented by one color.

Figure 4.2 is the wire frame result of the phase one clustering algorithm. High variation areas in the model, such as the head of the cow, need lots of small triangles to define in order to avoid detail loss. Some of these triangles are close to pixel size. These are the areas that we want to represent using surfel clouds. In phase two, we add the patch size as the new measure for the cost of edge contraction into the total cost and remove the planarity and direction measure (§3.2.3). Before continuing the clustering, the cost of each edge needs to be recalculated and the heap updated. The user also needs to provide the cost threshold for phase two. In this phase, the cost threshold is an indication of the size and shape of the resulting clusters in the model. It should be noted that for some patches, there is no clear choice of triangle rendering pipeline or point rendering pipeline. The cost threshold can be used to control whether or not the user wants more triangle patches or more surfel cloud patches. Higher cost threshold produces more triangle patches while lower cost threshold produces more surfel cloud patches. If the cost threshold equals to the maximum value in the heap, the output is the same triangle model from phase one. If the cost threshold equals to a minimum possible value, the output is a completely surfel cloud model. Therefore, by changing the cost threshold, our system provides the user full control of how the model is to be constructed. The result of the phase two clustering after phase one is shown in Figure 4.3. After this phase, small patches in high variation areas , such as legs and head, are combined together. After the two phase clustering, we have patches of comparable sizes and ready to do the patch sampling and build the octree based hybrid model.

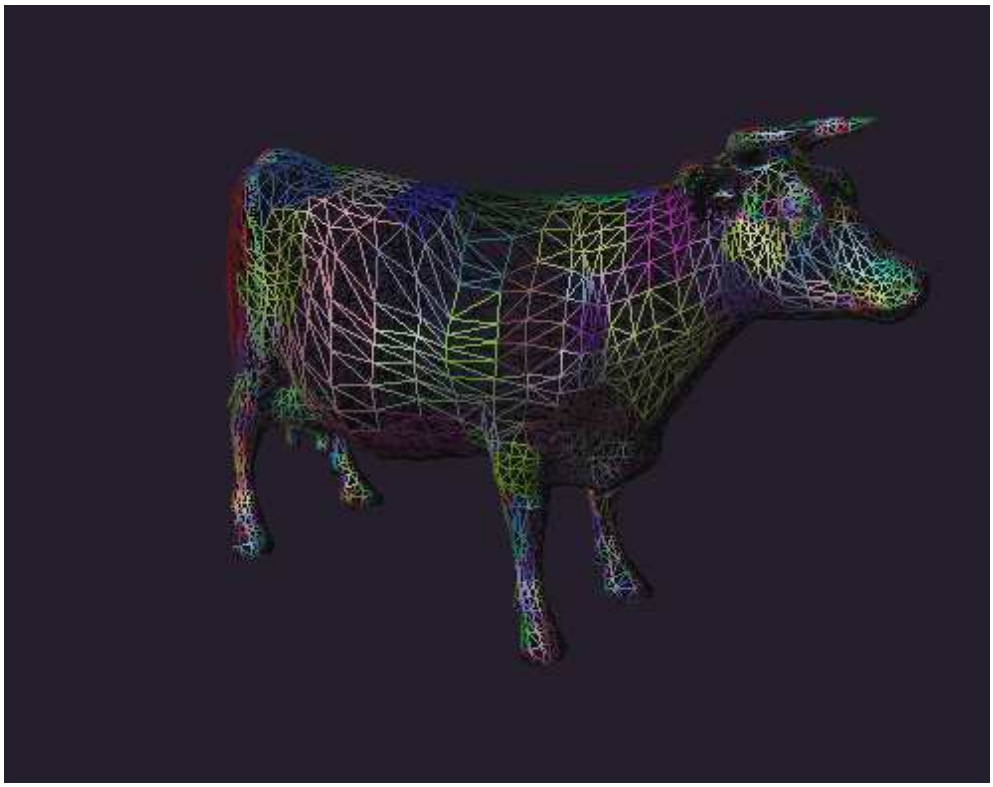
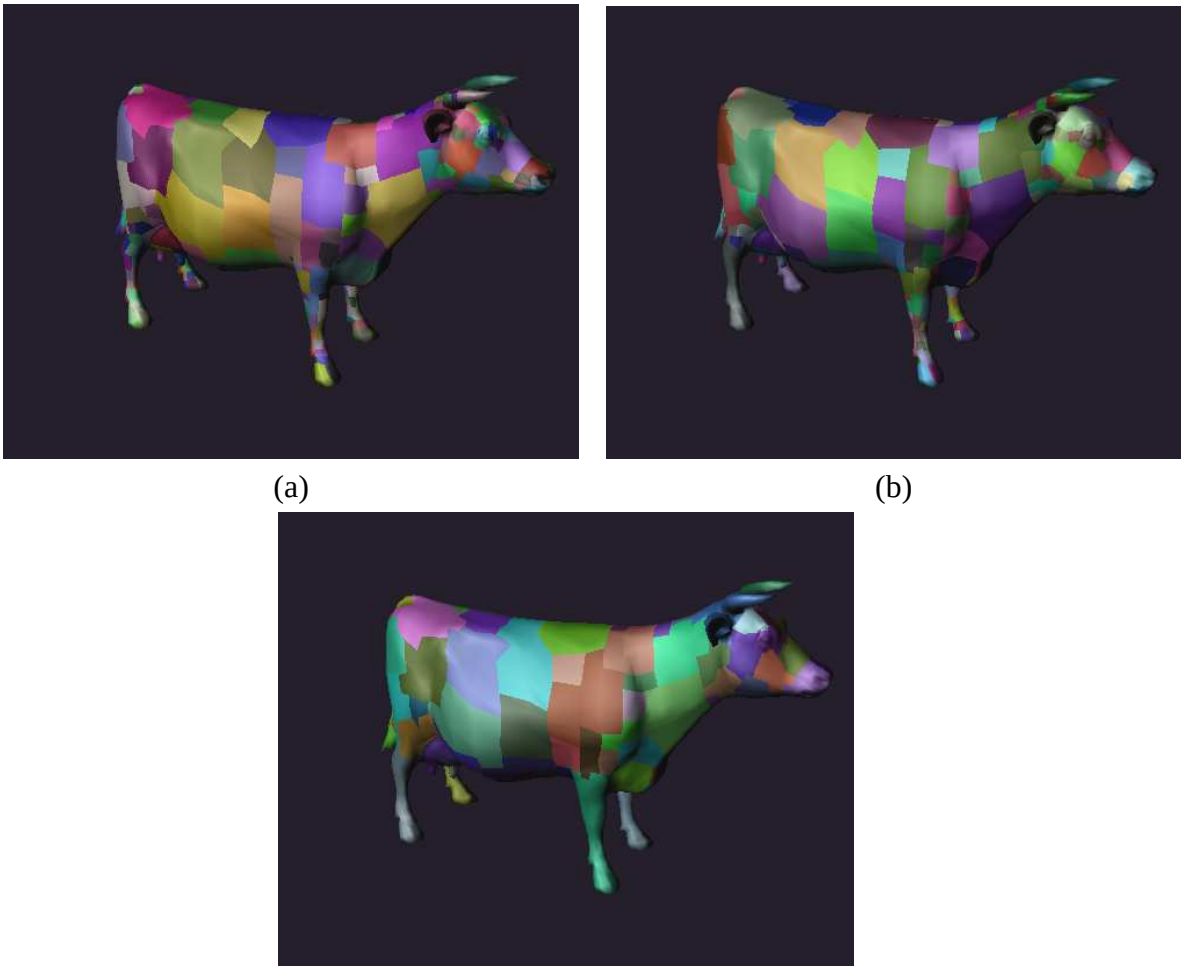


Figure 4.2: Wire frame of the phase one clustering result of the cow model. Cost threshold is -0.19.



(a) (b)

(c)

Figure 4.3: Phase two clustering result of the cow model (5804 triangles) with different cost thresholds. (a) -0.07, (b) -0.16, (c) -0.42. (The cost threshold used in phase one of this example is -0.19, see Figure 4.1(b))

4.2 Surface Sampling

We calculate the axis aligned bounding box for each triangle patch and the average edge length l of the patch (§3.3.2). l is used to evaluate the sampling grid distance in $\frac{l}{c}$, where c is a scale factor. Larger values of c produce more samples in a patch. When c is greater than 1, the triangles in a patch have at least one surfel sample on average. In practice, the value of 4 for c produces nice sampling result. Sometimes the value of $\frac{l}{c}$ can be too big (reduce sampling quality) or too small (increase sampling time). To avoid either situations, we apply a maximum and minimum sampling grid distance. It should be noted that the sampling grid distance is related to the side length of the bounding box. It makes no sense to just provid the grid distance without mentioning the size of the bounding box. All our sampling is done on models that are bounded by a bounding box size of $[1, 1, 1]$. Any models that are bigger or smaller are scaled accordingly before sampling and scaled back for rendering. Figure 4.4 is a rendering of a small percentage of surfel samples of a fully sampled model. Because the body of the cow is aligned with one of the axes, the pattern of regular grid is clearly shown in the figure. It can also be noticed that some surfels have positions close to each other and almost overlap completely. During rendering, this redudnant overlap is the major cause for frame rate decrease in that fragment shader is doing redundant calculation for the same pixels.

The ideal sampling is to have the least overlapping among surfel samples while guaranteeing that there are no holes between surfels. This is a hard problem and is a whole research topic, which we do not address in this study. Wu and Kobelt [34] provides a technique to

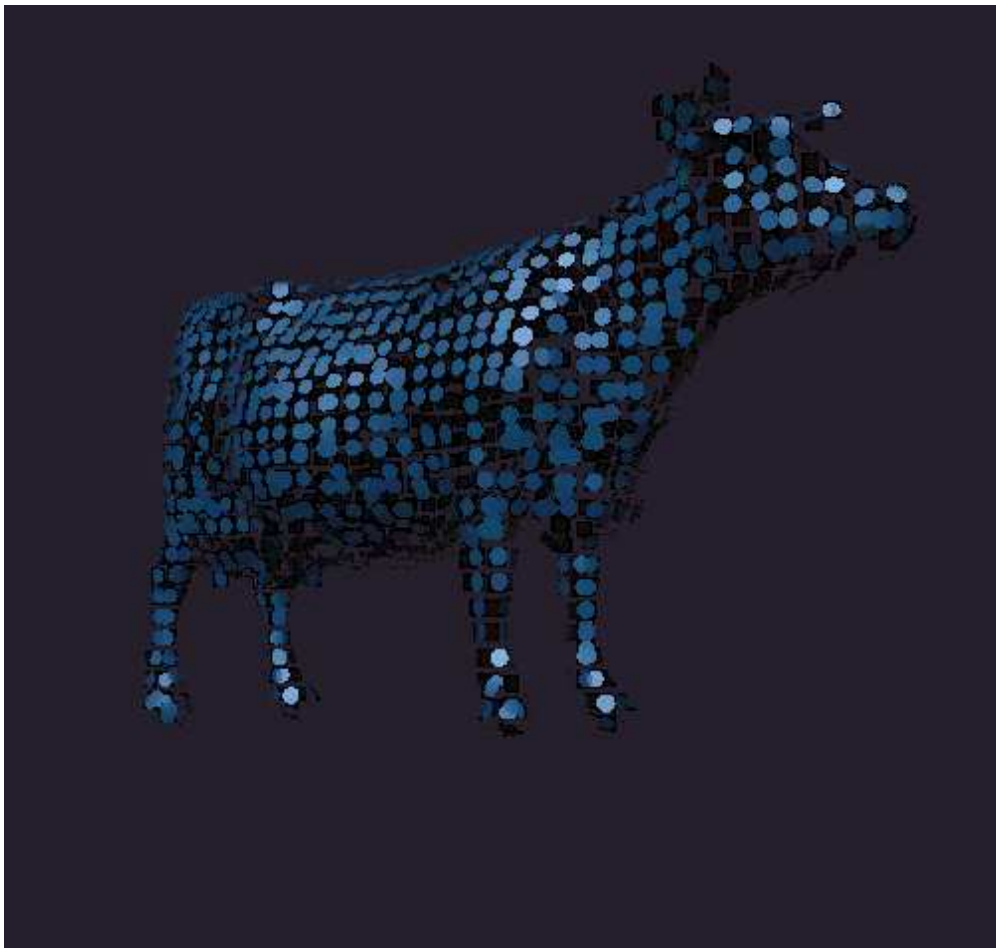


Figure 4.4: A rendering of partial point samples of a fully sampled model.

optimize point sampled models. The technique uses a greedy procedure to first guarantee sufficient overlapping in order to provide a hole-free surface, then it undergoes a global relaxation procedure. The idea is to iteratively replace subsets of surfels by new sets that have fewer elements or at least a better surfel distribution. Further more, Wu and Kobelt use elliptical surfels rather than circular surfels. These elliptical surfels have their major and minor axis aligned with the surface principal curvature directions and scaled according to principal curvature values. If we apply this technique in our sampling step, the model generated will have fewer surfel samples, which could potentially increase the rendering speed.

4.3 Hybrid Rendering Pipeline

We have implemented our hardware accelerated hybrid rendering pipeline with OpenGL and C++ in Microsoft Visual Studio Dot Net 2003. Performance has been measured on a 2.8GHz Intel Pentium4 system with 768 MB memory with Windows XP operating system. A standard OpenGL polygon rendering is used for triangle patch rendering. Hardware vertex and fragment shaders are used to render surfel cloud patches.

4.3.1 Hardware EWA Rendering

We implemented our EWA algorithm with standard OpenGL ARB_Vertex_Program and ARB_Fragment_Program extensions supported by Radeon 9x00 from ATI and GeForceFX

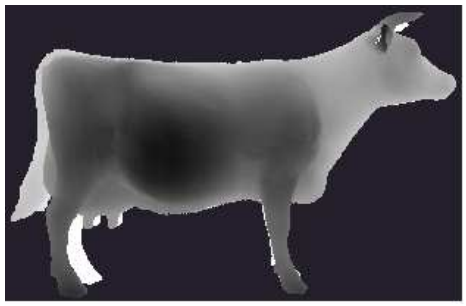
family from NVidia. We tested our implementation on a GeForceFX 5950 Ultra GPU. We attempted to also run our shaders on an ATI 9800 Pro GPU without success. The reason is that one of the internal variables defined in the ARB_Fragment_Program specification, *fragment.position*, is not supported by ATI boards. This variable stores the window coordinates of each pixel. We need this information to do ray-surfel intersection (§3.4.2, Equation. 3.29). It is noted that both vertex and fragment programs are written in low level assembly-like code. We also have shaders written in the latest OpenGL Shading Language (GLSLang). GLSLang is a high level shading language. Although the code is significantly shorter than that written in low level assembly language, the current compiler provided by NVidia is not able to produce efficient low level code from high level code. Therefore, high level code is only able to run in software mode rather than in hardware.

	Visibility Splatting	EWA Filtering	Normalization
Vertex Shader	37	86	-
Fragment Shader	8	11	3

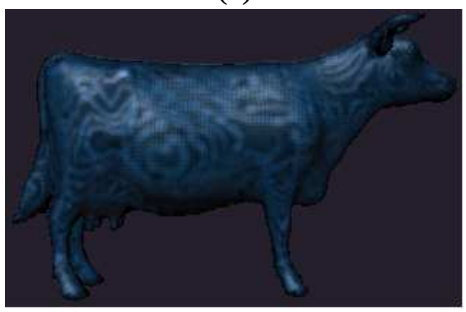
Table 4.1: Number of instructions needed for each pass.

Table 4.1 lists the number of instructions for each pass of the hardware rendering. The vertex shaders are much longer than the fragment shaders. We move as much calculation to the vertex shader as possible. The reason is that during rendering many fragments are rasterized needlessly while OpenGL point primitive handles an axis aligned square on the window. Even though the fragment programs are very simple, we observe a slowdown of two-fold in comparison with the case where the fragment programs are disabled.

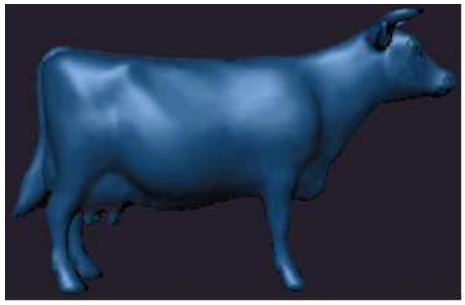
Figure 4.5 shows the rendering results of each pass of the cow model. Figure 4.5(a) is the



(a)



(b)



(c)

Figure 4.5: Rendering results from different passes (a) visibility splatting, (b) EWA filtering, (c) Normalization.

resulting depth buffer of the visibility splatting pass. The depth buffer is transferred away from the camera for an offset ε (§3.4.2). It is very important to choose an appropriate ε. This is illustrated in Figure 4.6. Assume the surfels in the dashed ellipse forms the local surface, we want to blend these surfels together during rendering. The single surfel to the right of the dashed ellipse does not belong to the surface formed by the surfels in the ellipse locally, therefore, no blending of this surfel with those in the ellipse during rendering. The dashed lines show the actual depth buffer offset. Because we set the OpenGL depth

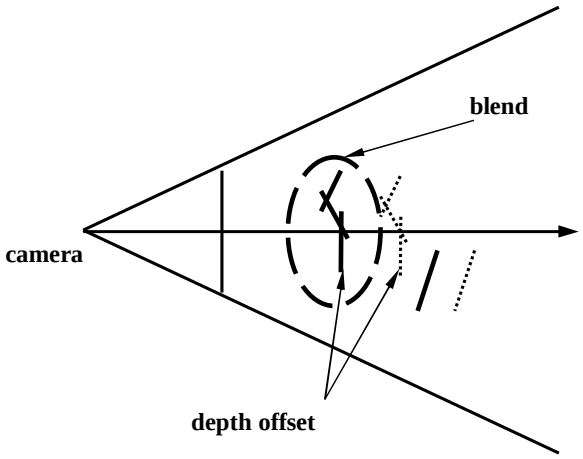
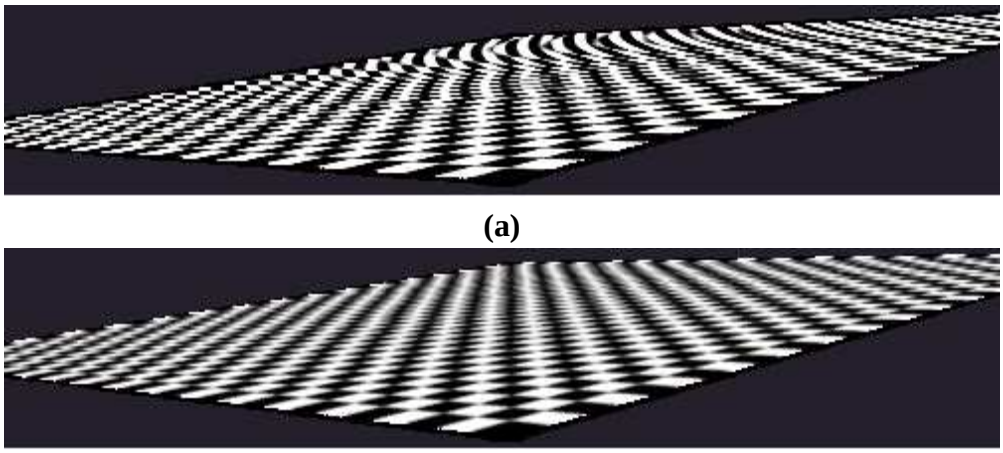


Figure 4.6: Applying the depth offset ε during visibility splatting.

comparison function to less than or equal, surfels in the ellipse will be blended successfully during rendering. If we choose an offset that is too big, the hardware will blend the single surfel on the right, causing over blending. If we choose an offset that is too small, we can not guarantee the blending of surfels inside the ellipse, causing under blending. In practice, we choose the same offset value as the sampling grid distance and allow the user to apply a scale factor to the grid distance desired (scale factor is between 0.75 to 1.5). Figure 4.5(b) is the result of the EWA filtering pass. The pattern of the sampling grid and some banding effects can be seen clearly. Because the surfels defining the model are not uniformly distributed on the surface, we can not guarantee that the sum of contribution from surfels to each pixel is the same across the frame buffer. Therefore, we need a third pass to do the normalization, which is just to divide the color value of each pixel by the sum of contribution of each pixel. The result is shown in Figure 4.5(c).

We compare the texture quality of the standard Gaussian texture filtering function (without applying the prefilter, see Section 3.4.1) to that of EWA filtering in Figure 4.7. In Figure 4.7(a), the texture function exhibits significant aliasing at high frequency area. Due to the



(a)

(b)

Figure 4.7: Checker board rendering using (a) without EWA filtering, (b) with EWA filtering.

convolution of the warped surface reconstruction filter and the screen prefilter (§3.4.1), we ensure that the screen projection size of each surfel is at least as big as a screen pixel. Further more, the screen projection of each surfel is an ellipse, which has its major and minor axis aligned according to the 3D surfel orientation. This provides very good anisotropic texture filtering. Figure 4.7(b) shows the same square rendered with EWA filtering (163k surfels), which is nicely antialiazed.

As discussed in Chapter 1, sufficient densities and proper sampling patterns are required for points to be effective. Figure 4.8 shows rendering results from a series of point sampled squares. It is clearly shown that higher sampling rate provides superior rendering quality. From classic sampling theory, low sampling rate on a high frequency signal causes aliasing. We see this happens in Figure 4.8(a), where we get a lower frequency texture mapped image while the actual texture signal has higher frequency. On the other hand, high sampling rate affects the rendering speed. Table 4.2 lists frame rates of rendering surfel clouds of different sampling rates. Our system maintains interactive frame rate (20 fps) with 320k surfels and

very high rendering quality. In this checker board example, using a grid distance of 0.007 achieves both good rendering quality and fast rendering speed.

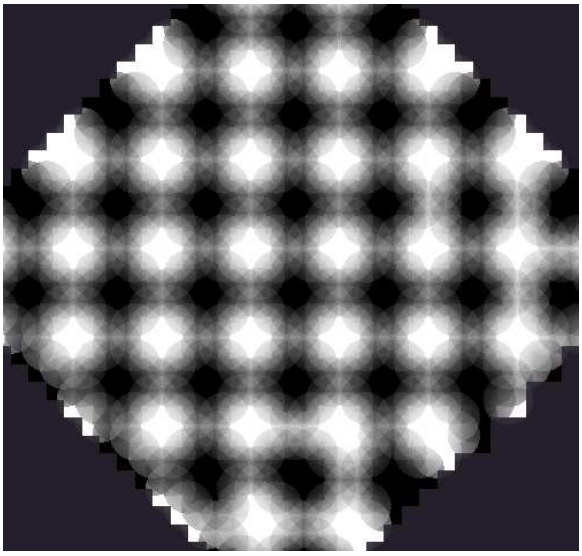
Sampling Grid distance	0.005	0.007	0.01	0.02	0.05	0.1
Number of surfels (k)	320	163	80	20	3	< 1
fps	20	31	37	64	83	154

Table 4.2: Rendering performance of texture mapped checker board on our system on a NVdia GeForce FX 5950 Ultra. The frame buffer resolution is 512×512.

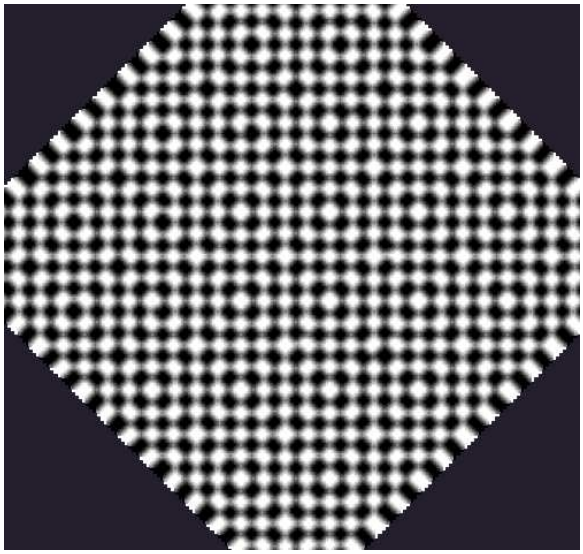
4.3.2 Hybrid Model Rendering

After clustering and sampling, the rendering system organizes patches into an octree representation. The rendering of the model is a tree traversal process. Traditional polygon rendering pipeline is used for triangle patch rendering. Point-based rendering pipeline applying EWA filtering techniques is used for point patch rendering (§3.1). As showed in Section 4.1, the two phase clustering algorithm is able to identify highly varied areas on a model and the sampling algorithm is able to convert these areas into surfel representations for rendering.

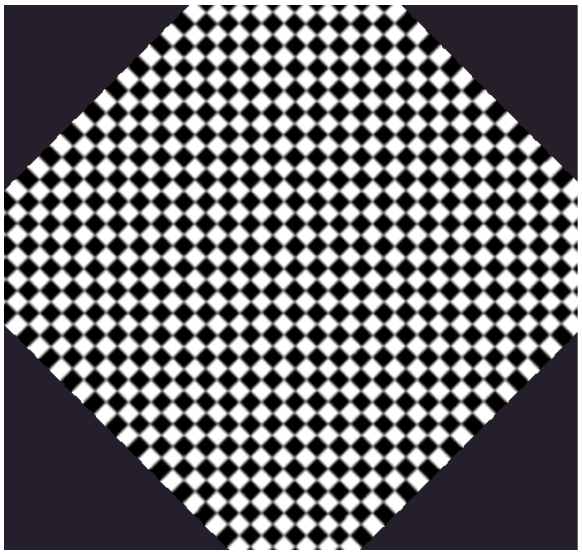
The patch boundary problem is mentioned in Section 3.3.2. It is possible that holes exist between triangle patches and surfel cloud patches. It is very rare in the actual rendering results. The reason holes may exist is that there is no connection information between triangle patches and surfel cloud patches. If the sampling rate is not high enough, there could be pixels in the boundary area that do not get contributions from surfels. However, we maintain a maximum sampling grid distance (§4.2), which guarantees adequate sampling



(a) Grid distance (0.1), 784 surfels.



(b) Grid distance (0.02), 20k surfels.



(c) Grid distance (0.007), 163k surfels.

Figure 4.8: Checkboard rendering with different sampling rate (a) low rate, (b) medium rate, (c) high rate.

for all patches. We observed that during rendering, holes may exist (although very rare) at the corner of a surfel cloud, which is surrounded by triangle patches (Figure 4.9). Because of the regular sampling grid, the sampling algorithm is likely to miss samples at the corners of a patch, especially when the angle of the corner is less than 90 degrees. This problem could be easily solved by adding surfel samples at each corner of the surfel cloud.

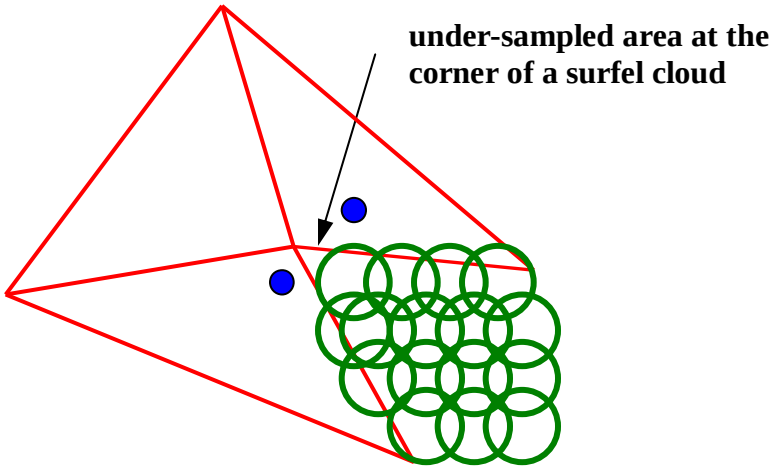


Figure 4.9: Missing surfel samples at the corner of a surfel cloud may leave holes during rendering. The red lines are part of surrounding triangle patches. The two blue dots indicate the sampling grid position at the boundary, which miss the surfel cloud patch. This leaves empty space at the the tip of the corner.

During hardware surfel rendering, the OpenGL point primitive renders a square for each surfel on the screen, therefore, the actual surfel cloud patch size on the screen is bigger than that of the corresponding triangle patch. This creates an overlapping between triangle patches and surfel cloud patches. Because we add a depth offset (§3.4.2) to each surfel, at the boundary overlapping area, triangle patches have smaller depth values than those of surfel cloud patches. Therefore, triangle patch boundary usually overwrites surfel cloud boundary. This is clearly shown in Figure 4.11. On the other hand, the boundary between surfel cloud patches are blurred due to blending. The higher sampling rate, the less blurred

the boundary appears.

Figure 4.10: Global sampling rate. The grid distance is 0.01. The frame buffer resolution is 512×512.

Because our sampling algorithm adjusts the sampling rate for each patch, the loss of detail is unnoticeable for most cases. Note in Figure 4.11 that the horns and ears of the cow is fully detailed. The result of using a global sampling rate is shown in Figure 4.10. The loss of detail at the horns and ears of the cow is obvious. It can also be noticed that there is more blur between patches on the head of the cow than that of Figure 4.11.

Table 4.3 (also see Figure 4.12 and Figure 4.14) lists the preprocessing time and rendering performance of our hybrid rendering system. Some of the rendering results are shown in Figure 4.16 - 4.20. Our system achieves interactive rendering speed (over 20 fps) for medium sized models (less than 100,000 triangles). The clustering algorithm takes almost linear time in terms of the input triangle count. As the triangle count increases, the sampling time dominates the total preprocessing time (Figure 4.12). Table 4.4 (also see Figure 4.13) lists the rendering performance of the corresponding triangle models. It is clearly

shown that triangle rendering pipeline has much higher performance than that of the hybrid system. Note that the bottom row of Table 4.3 shows the surfel contribution per pixel in the frame buffer. We see a very high overlapping of surfels indicating that the sampling algorithm produces a lot of redundant surfels. This is the major cause for the reduction of rendering performance. It is also noted that the overlapping of surfels increases as the surfel count increases (Figure 4.14). During rendering the system does the convolution operation (§3.4.1) which guarantees the screen projection of each surfel to be at least one pixel size, therefore, we have higher overlapping for smaller surfels (sub-pixel screen projection before convolution), which is the case at high surfel counts. It should be mentioned here that our sampling algorithm does not optimize surface sampling distribution. This is especially true for patches that are not aligned with any of the major axes. By eliminating the redundancy of surfels, we could achieve significantly higher rendering speed. Figure 4.15 is a visualizaiton of the overlapping of surfels in the framebuffer. As mentioned in previous sections, the rendering speed is affected by the number of surfels and the degree of overlapping (§4.3.1). Increasing the surfel quantity reduces the rendering speed. Reducing the redundant overlapping between surfels in a model is essentially decreasing the number of surfels in the model. Another factor to rendering speed is the screen projection size of the surfel. The larger the screen projection, the more calculation needed for fragments. Therefore, moving the object closer to the camera also reduces the rendering speed.

(a) Front view.

(b) Back view.

(c) Polygon patches.

(d) Surfel cloud patches.

Figure 4.11: Rendering results of the cow hybrid model on a NVdia GeForce FX 5950 Ultra. The frame buffer resolution is 512×512.

Model	bones	cow	dragon	bunny	terrain
triangle count	617	1533	9552	13038	22517
surfel count	43269	51731	145712	308550	1070316
clustering time	1.72	2.59	24.42	30.58	81.61
sampling time	14.13	77.31	132.63	182	555.73
fps	63	69	29	17	2
surfels per pixel	10.9	11.2	20.7	15.8	40.1

Table 4.3: Preprocessing time and Rendering performance of hybrid models on our system on a NVdia GeForce FX 5950 Ultra. The frame buffer resolution is 512×512.

Model	bones	cow	dragon	bunny	terrain
triangle count	4204	5804	50761	69451	199114
fps	968	906	130	92	10

Table 4.4: Rendering performance of triangle models on our system on a NVdia GeForce FX 5950 Ultra. The frame buffer resolution is 512×512.

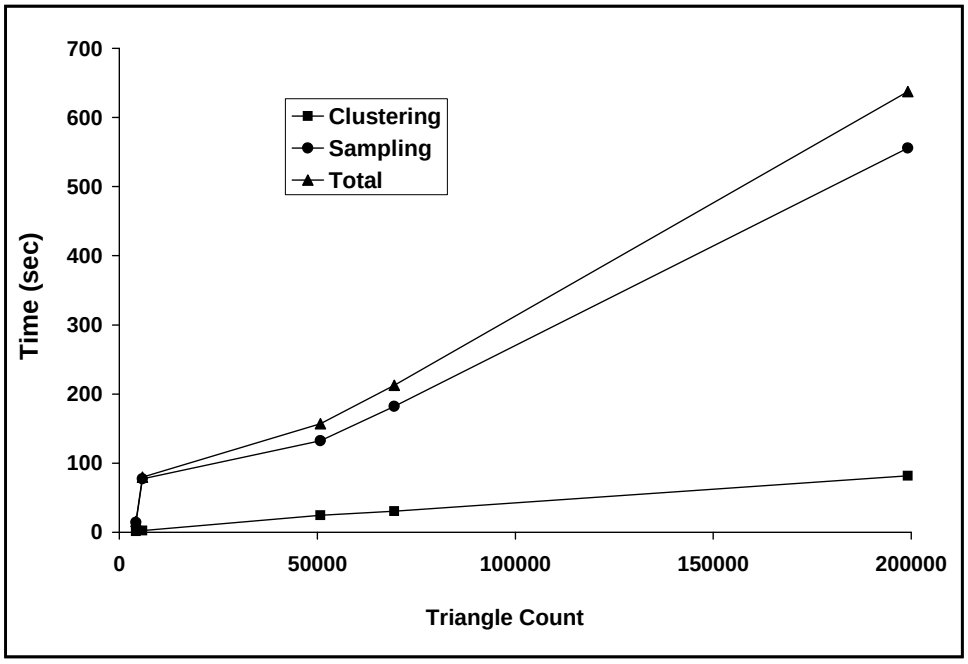


Figure 4.12: Running times for preprocessing of different models.

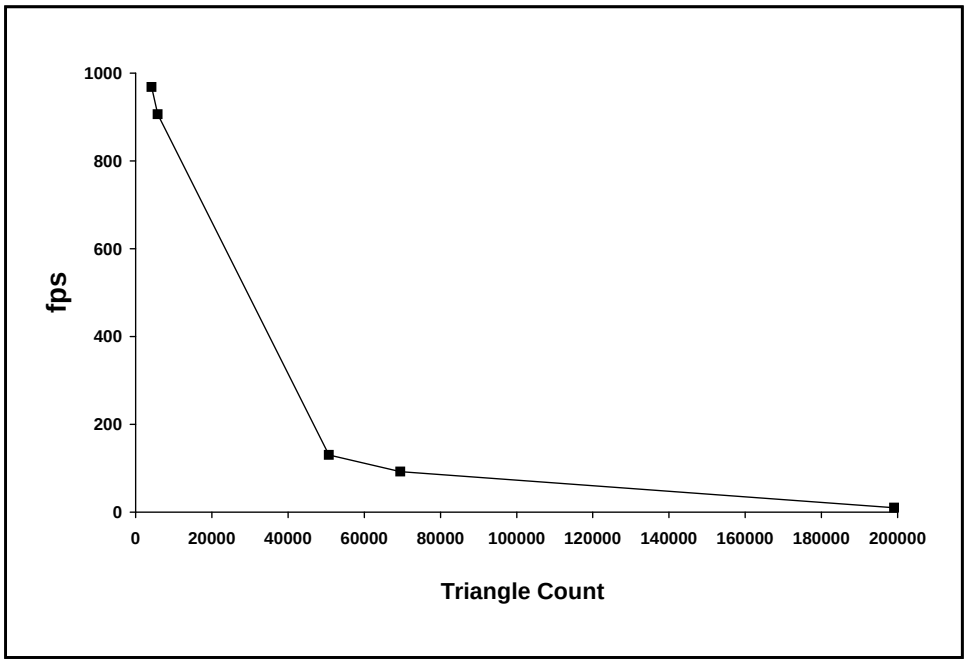


Figure 4.13: Rendering performances of different triangle models.

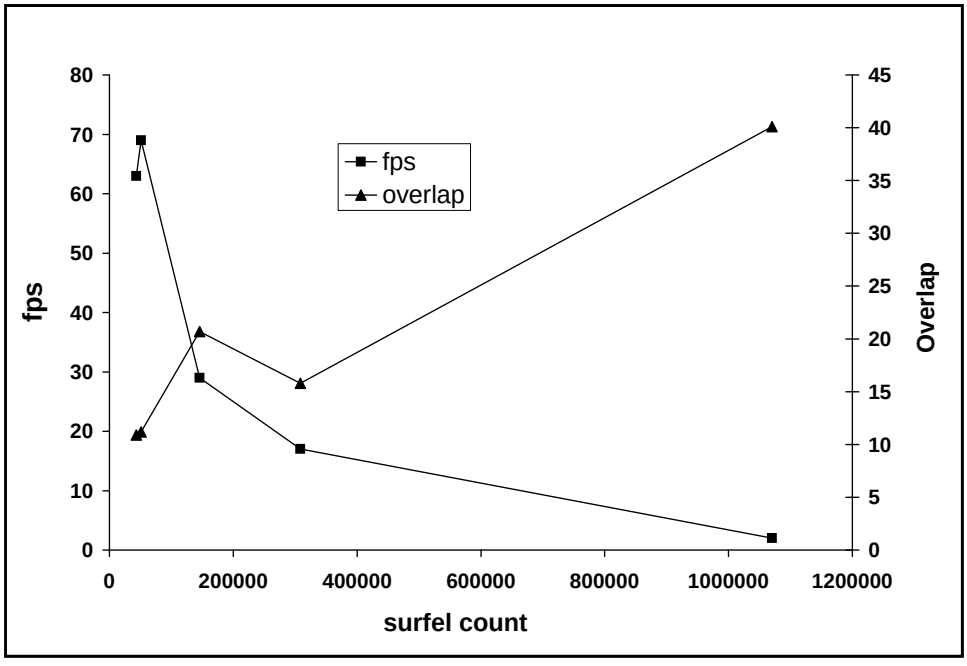


Figure 4.14: Rendering performances of different hybrid models. The performances are plotted with respect to surfel counts. Overlap indicates the average number of surfels contributing to each pixel in the framebuffer.

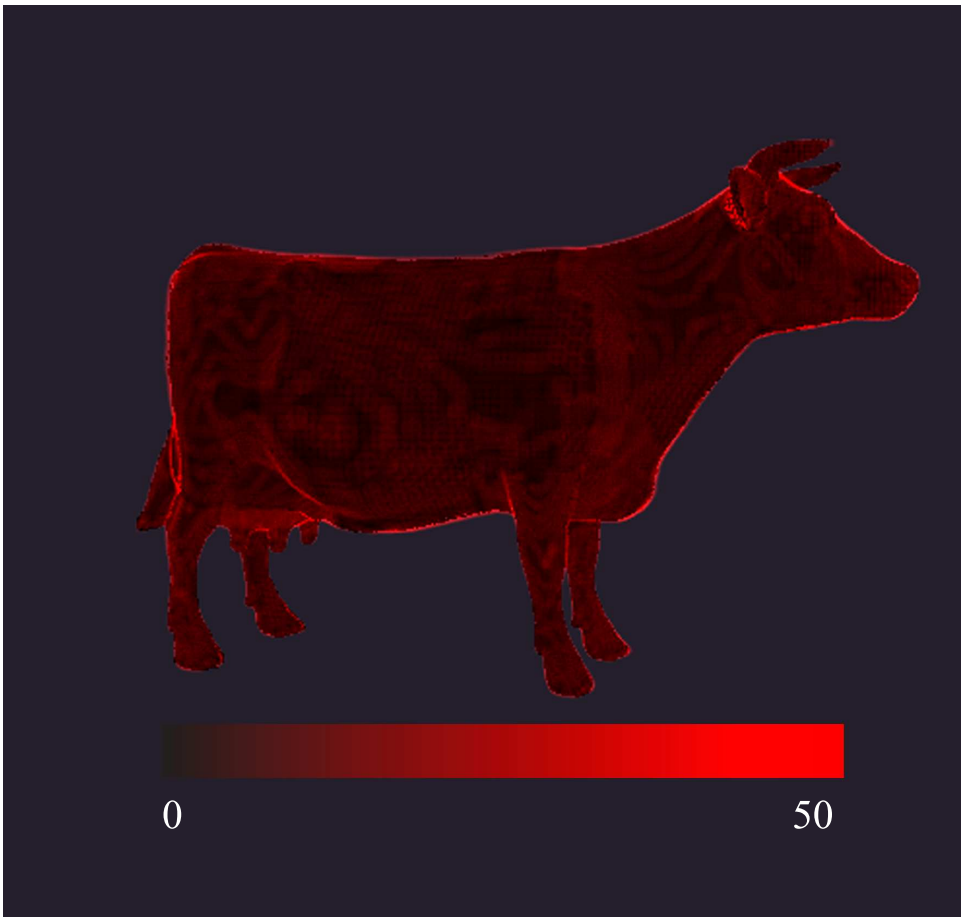
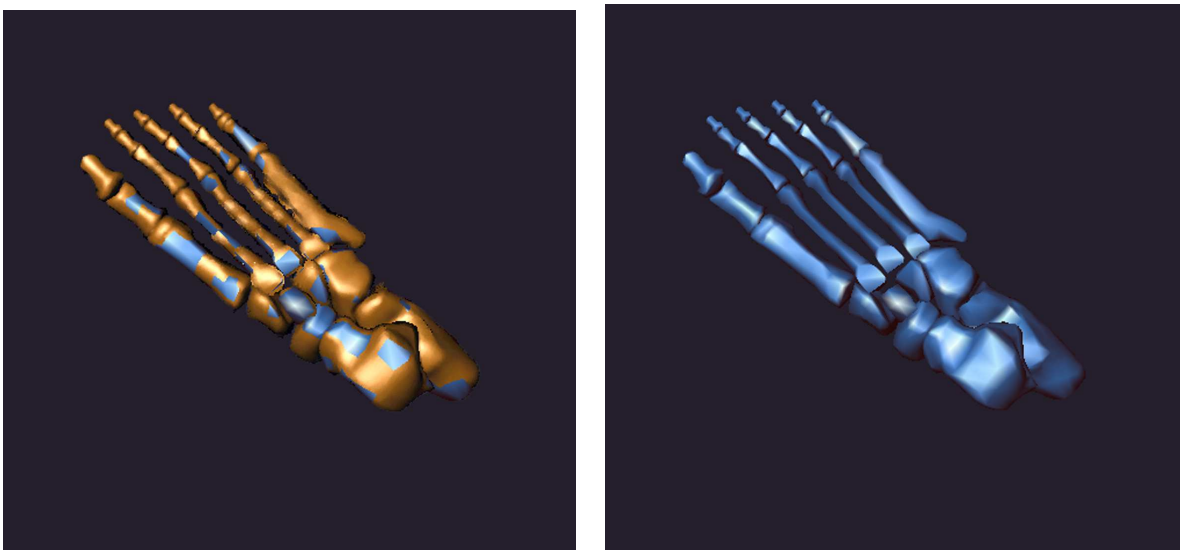


Figure 4.15: Visualization of the number of surfels contributing to each pixel in the frame-buffer. The higher the red color the larger the number of surfels contributing.

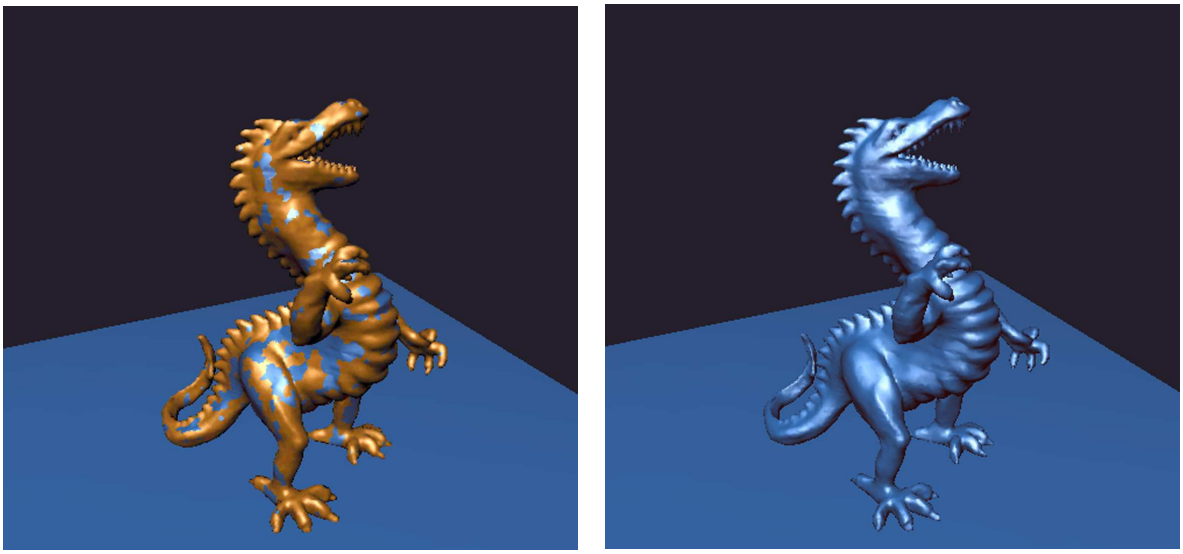


(a) 43269 surfels, 617 triangles. (b) 4204 triangles.

Figure 4.16: Rendering results of the bone model on a NVdia GeForce FX 5950 Ultra. The frame buffer resolution is 512×512.

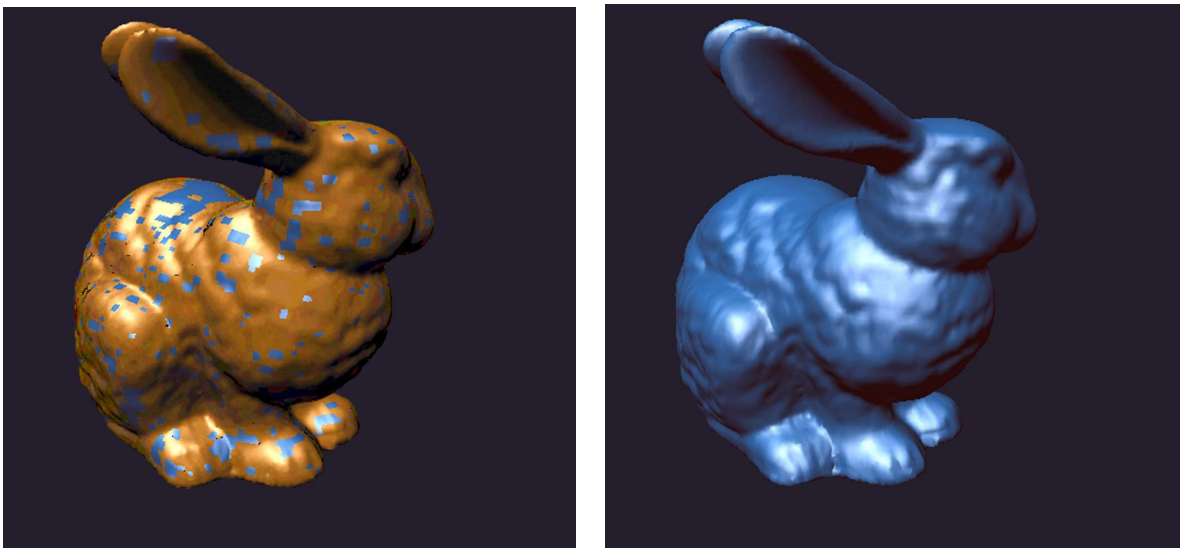
(a) 51166 surfels, 1533 triangles. (b) 5804 triangles.

Figure 4.17: Rendering results of the cow model on a NVdia GeForce FX 5950 Ultra. The frame buffer resolution is 512×512.



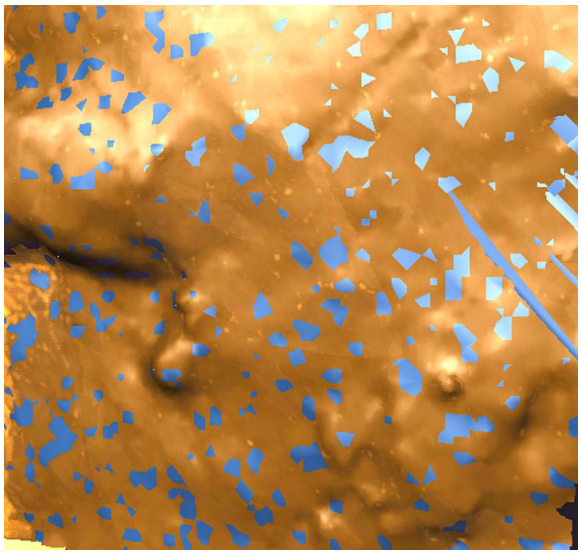
(a) 145712 surfels, 9552 triangles. (b) 50761 triangles.

Figure 4.18: Rendering results of the dragon model on a NVdia GeForce FX 5950 Ultra. The frame buffer resolution is 512×512.

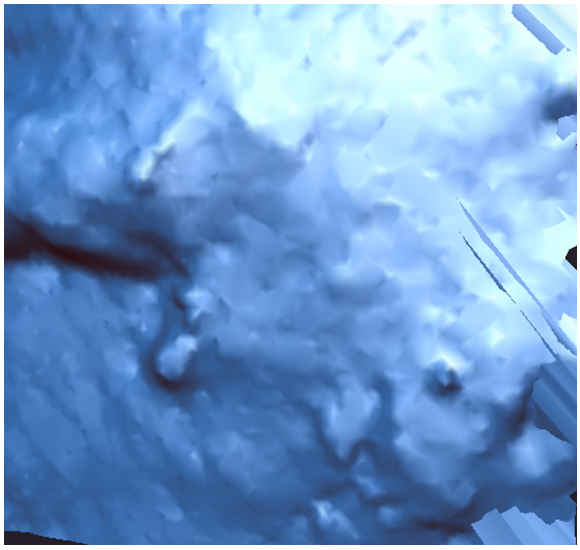


(a) 308550 surfels, 13038 triangles. (b) 69451 triangles.

Figure 4.19: Rendering results of the bunny model on a NVdia GeForce FX 5950 Ultra. The frame buffer resolution is 512×512.



(a) 1070316 surfels, 22517 triangles.



(b) 199114 triangles.

Figure 4.20: Rendering results of the terrain model on a NVdia GeForce FX 5950 Ultra. The frame buffer resolution is 512×512.

Chapter 5

Conclusion and Future Work

5.1 Summary of Contributions

As discussed in Chapter 3, the contributions of this thesis are two folds. First, Garland's quadric error metrics based surface clustering algorithm [10] is modified into a two-phase clustering algorithm to generate patches and build hybrid models. Patch representation is independent of the viewing parameters. Highly varied surface areas are identified by our algorithm and represented by points. Second, points and triangles are considered equivalent primitives in our system and can be used at all levels of details. Unlike most previous hybrid renderers, points are not used only for previewing purposes. We have built a true hybrid real-time rendering system. The traditional polygon rendering pipeline is used for triangle patch rendering. A point-based rendering pipeline applying splatting techniques is used for point patch rendering. A hardware implementation of EWA filtering is provided for the

point-based rendering pipeline, utilizing hardware vertex and fragment shaders in modern graphics boards, to achieve interactive frame rate.

Our system is able to render medium sized models in interactive frame (over 20 fps) on a 2.8GHz Intel Pentium4 system with 768 MB memory with a GeForceFX 5950 Ultra GPU. The two phase clustering algorithm successfully identifies planar and highly varied areas on the surface and the rendering system assigns triangle or point representations for different patches. The sampling algorithm converts triangle patches into point patches and adjusts the sampling rate based on the patch local information. The hardware based EWA filtering achieves superior anisotropic texture filtering while maintains interactive frame rate. The rendering system outputs hole free hybrid models. It is very rarely that holes appear at the boundary areas between triangle patches and surfel cloud patches (§4.3.2). Our system provides the users great flexibility to render any part of a model with either points or triangles while maintaining good rendering quality and speed.

5.2 Future Directions

There are several ways in which this work could be extended in the future. The list below appears particularly important and promising.

- Modern laser 3D scanning devices are able to acquire huge volumes of point data (Chapter 1). These point cloud data are essentially point cloud models. Our current system only takes triangle models as input and converts them into hybrid models. By

incorporating the ability of converting point cloud models into hybrid models, our system will be more flexible.

- The current sampling algorithm is very simple and easy to implement. However, we have lots of overlapping surfels as the result of sampling, which affects the efficiency of the rendering system. We mentioned in Section 4.2, the ideal sampling is to have the least overlapping among surfel samples while guaranteeing that there is no holes between surfels. We could apply some global optimization methods as one of them described by Wu and Kobbelt [34]. The idea is to iteratively replace subsets of surfels by new sets that have fewer elements or at least a better surfel distribution. Further more, we could use elliptical rather than circular surfels. These elliptical surfels can have their major and minor axis aligned with the surface principal curvature directions and scaled according to principal curvature values, which are elliptical in object space [34].

- In Chapter 3, we mentioned of building a LOD tree for each model so that each patch in the model has a series of LOD representations. By doing that, we could apply a cost function during rendering. The cost function will take into account the screen projection size of the patch, the distance of the patch and the surface variation of the patch to select the best representation (point or triangle at certain level of the LOD tree) of the patch for rendering.

- Our current point rendering system sends all surfels of a model to the hardware for rendering. We could improve rendering efficiency by applying some hierarchical data structure to the rendering system. For example, to test the visibility of a patch,

we could use normal cones [29]. We could also find better ways to organize our surfel data to use graphics memory more efficiently.

Bibliography

[1] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Point set surfaces. In *Proceedings fo IEEE Visualization 2001*, pages 21–28, 2001.

[2] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Computing and rendering point set surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):3–15, 2003.

[3] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern GPUs. In *Proceedings of Pacific Graphics*, pages 335–343, 2003.

[4] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. In *Proceedings of Symposium on Point-Based Graphics*, pages 25–32, June 2004.

[5] Chun-Fa Chang, Gary Bishop, and Anselmo Lastra. LDI tree: a hierarchical representation for image-based rendering. In *Proceedings of SIGGRAPH 1999*, pages 291–298. ACM Press, August 1999.

[6] Baoquan Chen and Minh Xuan Nguyen. POP: A hybrid point and polygon rendering system for large data. In *Proceedings of IEEE Visualization*, pages 45–52, October 2001.

[7] Liviu Coconu and Hans-Christian Hege. Hardware-accelerated point-based rendering of complex scenes. In *Proceedings of 13th Eurographics Workshop on Rendering*, pages 43–52, June 2002.

[8] Tamal K. Dey and James Hudson. PMR: Point to mesh rendering, a feature-based approach. In *Proceedings of IEEE Visualization*, pages 155–162, 2002.

[9] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH 1997*, pages 209–216, 1997.

[10] Michael Garland, Andrew Willmott, and Paul S. Heckbert. Hierarchical face clustering on polygonal surfaces. In *Proceedings of the 2001 symposium on interactive 3D graphics*, pages 49–58, 2001.

[11] Enrico Gobbetti and Fabio Marton. Layered point clouds. In *Proceedings of Symposium on Point-Based Graphics*, pages 113–120, 227, June 2004.

[12] Stephen J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Proceedings of SIGGRAPH 1996*, pages 43–54, 1996.

[13] J.P. Grossman and William J. Dally. Point sample rendering. In *Proceedings of 9th Eurographics Workshop on Rendering*, pages 181–192, June 1998.

[14] Gaël Guennebaud and Mathias Paulin. Efficient screen space approach for hardware accelerated surfel rendering. In *Proceedings of 2003 Vision, Modeling and Visualization, Munich*, pages 1–10. IEEE Signal Processing Society, November 2003.

[15] Paul S. Heckbert. *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer science, June 1987.

[16] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3D scanning of large statues. In *Proceedings of SIGGRAPH 2000*, pages 131–144, 2000.

[17] Marc Levoy and Turner Whitted. *The Use of Points as a Display Primitive*. Technical Report TR 85-022. University of North Carolina at Chapel Hill, Computer Science Department, 1985.

[18] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of SIGGRAPH 2001*, pages 149–158. ACM Press, 2001.

[19] D. Lischinski and A. Rappoport. Image-based rendering for non-diffuse synthetic scenes. *Rendering Techniques*, pages 301–314, 1998.

[20] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *Proceedings of SIGGRAPH 2003*. ACM Press, 2003.

[21] Nelson Max and Keiichi Ohsaki. Rendering trees from precomputed z-buffer views. In *Proceedings of 6th Eurographics Workshop on Rendering*, pages 45–54, 1995.

[22] Mark Pauly and Markus Gross. Spectral processing of point-sampled geometry. In *Proceedings of SIGGRAPH 2001*, pages 379–386. ACM Press, July 2001.

[23] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of SIGGRAPH 2000*, pages 335–342. ACM Press, July 2000.

[24] Pixar. Finding Nemo. `http://www.pixar.com/featurefilms/nemo` [Last Acessed Feburary 2005], 2003.

[25] William T. Reeves. Particle systems - A technique for modeling a class of fuzzy objects. In *Proceedings of SIGGRAPH 1983*, pages 359–376. ACM Press, July 1983.

[26] William T. Reeves. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Proceedings of SIGGRAPH 1985*, pages 313–322. ACM Press, 1985.

[27] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings of Eurographics 2002*, 2002.

[28] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of 12th Eurographics Workshop on Rendering*, pages 151–162, June 2001.

[29] Leon A. Shirmun and Salim S. Abi-Ezzi. The cone of normals technique for fast processing of curved patches. In *Proceedings of Eurographics 1993*, pages 261–272, 1993.

[30] Alvy Ray Smith. Plants, fractals and formal languages. In *Proceedings of SIGGRAPH 1984*, pages 1–10. ACM Press, July 1984.

[31] Marc Stamminger and George Drettakis. Interactive sampling and rendering for complex and procedural geometry. In *Proceedings of Eurographics Workshop on Rendering 2001*, pages 151–162, 2001.

[32] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straßer. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. In *Proceedings of SIGGRAPH 2001*, pages 361–370, 2001.

[33] Lee Westover. Footprint evaluation for volume rendering. In *Proceedings of SIGGRAPH 1990*, pages 367–376. ACM Press, August 1990.

[34] Jianhua Wu and Leif Kobbelt. Optimized sub-sampling fo point sets for surface splatting. In *Proceedings of Eurographics 2004*, pages 643–652, 2004.

[35] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *Proceedings of SIGGRAPH 2001*, pages 371–378. ACM Press, July 2001.

[36] Matthias Zwicker, Jussi Rasanen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate spaltting. In *Proceedings of the 2004 Conference on Graphics Interface*, pages 247–254, 2004.