High Quality Rendering of Large Point-based Surfaces.

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science (by Research)
in
Computer Science and Engineering

by

Naveen Kumar Reddy Bolla
200402029
naveenb@research.iiit.ac.in

Center for Visual Information Technology
International Institute of Information Technology
Hyderabad - 500 032, INDIA
June 2010

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "High Quality Rendering of Large Point-based Surfaces" by Naveen Kumar Reddy Bolla, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Adviser: Prof. P. J. Narayanan

“The best solutions are always simple.” – Ivan Sutherland

“Brick walls are there for a reason. The brick walls are not there to keep us out. The brick walls are there to show how badly we want something.” – Randy Pausch (The Last Lecture)

To my Beloved Grand Parents

Shri. Veerabadhra Hanuma Reddy

and

Smt. Veerabadhra Saambrajyamu

whom I lost on either end of writing this document.

Acknowledgments

I would like to thank Dr. P. J. Narayanan for his support and guidance during the past few years. I was fortunate enough to have him as my guide. I sincerely appreciate all his help, generosity, ethusiasm, patience and deep insights over perspective solutions to the problems.

I am also grateful to my fellow lab mates at the CVIT, IIIT Hyderabad for their stimulating company during the last couple of years. The financial support I received from the CVIT during my Masters studies is acknowledged.

Above all I am thankful to my family and friends for their unconditional support and unbounded patience.

Abstract

Points-based models have gained popularity in recent years. The most attractive feature of the point-based primitives is its simplicity. No information about connectivity or topology of the surface is stored explicitly. Adaptive modification and dynamic sampling of the surface do not suffer from complexity and robustness like triangle mesh. The level of detail is simple, fast and almost continuous. However, points based methods suffer from surface normal discontinuities in representation. Shading is also not independent from sampling rate. Point based models can be visualized by rendering the points directly. Each sample in the representation has certain attributes (normal, color, material properties) for rendering and shading them as a continuous surface. Typically, a point sample also has a radius to define an area around it. Such samples are called *surfels* and approximate the shape of the surface linearly in their neighborhood. Since linear splats can only be flat shaded, such representations require a large number of samples for a good shading quality. This slows the rendering due to increase in rendering computation and related memory bus activity.

Point-based rendering suffer from the limited resolution of the fixed number of samples representing the model. At some distance, the screen space resolution is high relative to the point samples, which causes under-sampling. A better way of rendering a model is to re-sample the surface during the rendering at the desired resolution in object space, guaranteeing a sampling density sufficient for image resolution. Output sensitive sampling samples objects at a resolution that matches the expected resolution of the output image. This is crucial for hole-free point-based rendering. Many technical issues related to point-based graphics boil down to reconstruction and re-sampling. A point based representation should be as small as possible while conveying the shape well.

In the first part of this thesis, we present a compact representation for point based models using non-linear surface elements. In the second part, we present a method for fast ray casting of a dynamic surface defined by large number of attributed points called Metaballs.

Algebraic Splats: Higher-order approximations of the local neighborhood have the potential to represent the shape using fewer primitives, simultaneously achieving higher rendering speeds and quality. In this thesis, we present *algebraic splats*, which are low-order polynomials that approximate the local neighborhood of a pointset. We adapt the Moving Least Squares (MLS) approximation to generate algebraic splat representations for a given point set. Quadratic and cubic splats as basic rendering primitive are able to approximate point-based models using 10% or fewer points than linear splats, for equivalent

visual quality. Though rendering a non-linear patch is slower compared to a linear splat, the overall speed of rendering of an object could be faster. The approximation error can also be less using a small number of higher order primitives. We represent a given point set with a user-specified small number of algebraic splats with optimal rendering quality. This is done by decimating the point set and jointly approximating each using a local algebraic surface based on the Moving Least Squares (MLS) approximation.

Our rendering provides smooth surfaces with normals everywhere. We can render polynomials directly on today's GPUs using ray-tracing because of the semi-implicit nature of the splats in the local reference domain. The major contributions of this work are: (i) A compact and lightweight representation for point geometry using nonlinear surface elements, called algebraic splats; (ii) A multi-resolution representation which provides continuous Level of Detail; and (iii) A ray-tracing approach to render the algebraic splats on the GPU. The method includes an adaptive anti-aliasing step to obtain smooth silhouettes. The David 2mm model can be represented using about 30K (or 0.8% of original) algebraic splats with little or low reduction in visual quality. We can raycast the David model with adaptive antialiasing (3×3 grid) at 130-150 fps and 80-90 fps with shadowing. Our method is faster by a factor of 10 on most models and require about 90% less diskspace in comparision to linear splats.

Metaballs: Since the available computational power is steadily growing, more and more science areas rely on simulations of ever-growing problem sizes producing huge amount of data to be analyzed. Simulation and experimental measurement in life sciences, physics, chemistry, materials, and thermodynamics yield large and often also time-dependent datasets. Interactive visualization is the key service that facilitates the analysis of such datasets and thus enables the researchers in those fields to quickly assess and compare the results of a simulation or measurement, verify and improve their models, and in so doing coming ever closer to understanding how dynamic processes work.

Metaballs (also known as blobs) are popular particle-based implicit surface representations. A set of them can represent smooth deformable, free-form shapes represented as the iso-surface of a scalar field. Current methods can handle only a moderate number of metaballs at interactive rates on commodity GPUs. In this thesis, we present a method to handle a million or more dynamic metaballs at interactive rates. We use a perspective grid as the acceleration structure, which is built in each frame on the GPU using fast primitives. Perspective grids provide perfect coherence to primary rays. We ray trace each grid cell at a time using an adaptation of the marching points methods to metaballs. This method extracts high performance from the SIMD GPUs of today. We show interactive handling of upto a million balls. Our method is also able to handle secondary rays, though at much reduced speeds due to the lack of coherence but is suitable for live visualization of particle-based simulations. The average build time for 1 million primitives is about 38 milliseconds and we can render a million metaballs in about 94 milliseconds. Performance of Secondary rays is not fast due to the lack of coherence, with a rendering time of 150 ms for 1K primitives.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer graphics is a vast field that encompasses almost all graphical aspects, including the generation of images of 3-dimensional scenes. This involves the study of (i) models suitable for manipulating and viewing shapes, (ii) real-time and realistic rendering techniques for these models, and (iii) devices for viewing and interacting with the models. Over the years, computer graphics has gained tremendous importance in several application domains. It has applications in wide range of domains like Computational Biology and Physics, Computer-aided design, Simulation and training, Digital art, Rational Drug Design, Video games, Movies, Information Visualization, Virtual Reality, etc.

Computer graphics relies on an internal model or a mathematical representation of the scene suitable for graphical computations. The model describes the 3D shapes, layout, and materials of the scene. This 3D representation then has to be projected to compute a 2D image from a given viewpoint, which is called *rendering*. Rendering involves projecting the objects, handling visibility and computing their appearance and lighting interactions. Finally, for animated sequence, the motion of objects has to be specified. This process is called graphics pipeline – from scene geometry to image.

The graphics pipeline is meant to accept a certain representation of a three dimensional scene as input and produce a 2D image as output. In early days, this pipeline was implemented in hardware for acceleration and is fixed. Later some of the stages are made programmable for vertex and fragment processing. Vertex and fragment processing can be done independently on the Graphics Processing Unit (GPU). With the introduction of shader model 4.0, an additional geometry processing unit was added to the pipeline. Figure 1.1 shows the Shader Model 4.0 way of rasterization-based rendering as supported by commodity graphics hardware. GPUs are increasingly used for a wide range of problems involving heavy computations in graphics, computer vision, scientific processing, etc.

1.1 Geometry Representation

Smooth curves and surfaces must be generated and represented in many computer graphics applications. Complex shapes are converted to simple primitives before rendering. The representation of three dimensional scene (geometry) varies depending on application. Over the years, several ways of

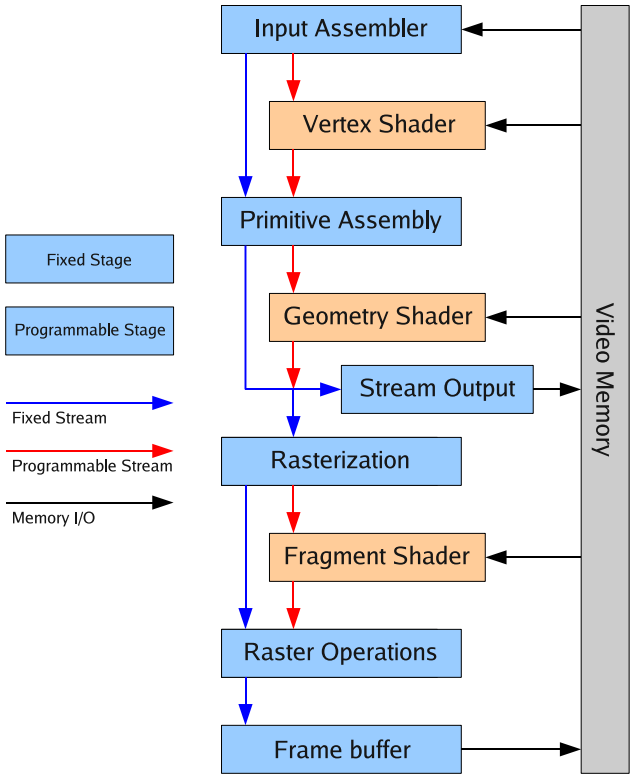


Figure 1.1: The graphics pipeline in Shader Model 4.0

representing geometry has been proposed – polygonal meshes, parametric surfaces, implicit surfaces, points, volume data, constructive solid geometry (often a procedural rendering), etc. Triangles or polygon meshes are most conventional representations and are popular in many application domains. Volume geometry, with voxels as primitives, is popular with medical imaging. Micro-polygons and subdivison surfaces are popular in film production. Metaballs are popular in molecular dynamics and fluid simulations. All these representations have coexisted because each representation offers a unique advantage such as speed of rendering, ease of modeling, flexibility of editing, etc.

The 3D scene content can be created in a variety of ways. The advancements in scanning technologies and three-dimensional digital photography help us capture the geometry and appearance of complex real world objects. Artists can create 3D models or scenes using software like Maya, 3DS Max, etc. The scene content or geometry is stored in one of the representations. These representations are loaded by rendering programs to create the final image.

Procedural geometry, on the other hand, involves on-the-fly creation of arbitrarily accurate shape from compact descriptions, usually in the form of implicit equations. Procedural objects are converted to piecewise linear models using a polygonal mesh. This results in a loss in resolution and incurs computational overhead. Triangulated mesh requires high memory bandwidth from the CPU to the GPU and huge video memory for storage. Procedural geometry can save both bandwidth and memory requirements drastically. Computationally, the overhead of tessellation is removed and visually, the

geometry is rendered exactly at all resolutions. Procedural geometry is also compact in representation needing less bandwidth to send to the GPU

Many classical geometric entities can be directly used as primitives, e.g, cubes, cylinders, spheres and cones. A sphere, for example, can be simply described by the coordinates of its center and its radius. More complex mathematical entities permit the representation of complex and smooth objects. Spline patches and NURBS are the most popular. Non-uniform rational basis spline (NURBS) is a mathematical model commonly used in computer graphics for generating and representing curves and surfaces which offers great flexibility and precision for handling both analytic and free form shapes. A NURBS curve is defined by its order, a set of weighted control points, and a knot vector. NURBS curves and surfaces are generalizations of both B-splines and Bzier curves and surfaces, the primary difference being the weighting of the control points which makes NURBS curves rational (non-rational B-splines are a special case of rational B-splines). Whereas Bezier curves evolve into only one parametric direction, usually called s or u, NURBS surfaces evolve into two parametric directions, called s and t or u and v. They are, however harder to manipulate since one does not directly control the surface but so called control points that are only indirectly related to the final shape. Complex shapes are converted to simple primitives before rendering. For Rendering 2D Splines or NURBS are converted to triangles. The direct rendering of the parameteric surfaces is slow.

1.2 Point Based Graphics

In the past decade, points received a lot of attention as rendering primitives. The overheard of managing, processing and manipulating very large polygonal mesh connectivity information has led many researchers question the simplicity of polygonal primitive and its future as graphics primitive. Points may not replace the triangles as graphics primitives, but have the ability to complement other primitives. The most conventional method for modeling 3D geometry is the use of polygons. An object is approximated by a polygonal mesh, that is a set of connected polygons. Most of the time, triangles are used for simplicity and generality. They are the most popular primitives for representing scene geometry. Graphics hardware and pipeline are optimized for processing polygonal geometry. However, points as a rendering primitive received a lot of attention in the past few years. The idea of using points as surface primitives was proposed by Levoy and Whitted in 1985. The new found interest in points is mainly due to the advancements in 3D scanning technology and the growing complexity of the polygonal meshes. We have also seen triangle meshes becoming denser, with most of the triangles projecting to less than a pixel in the screen. The screen resolution is not growing as fast as density of the triangles. The points can exploit this situation better.

The objective of point-based rendering is to display the point representation on the screen as continuous surfaces. The motivation for point-based rendering includes its efficiency at rendering very complex objects and environments, simplicity of rendering algorithms and emergence of 3D scanning devices that produce very dense point clouds that need to be directly visualized.

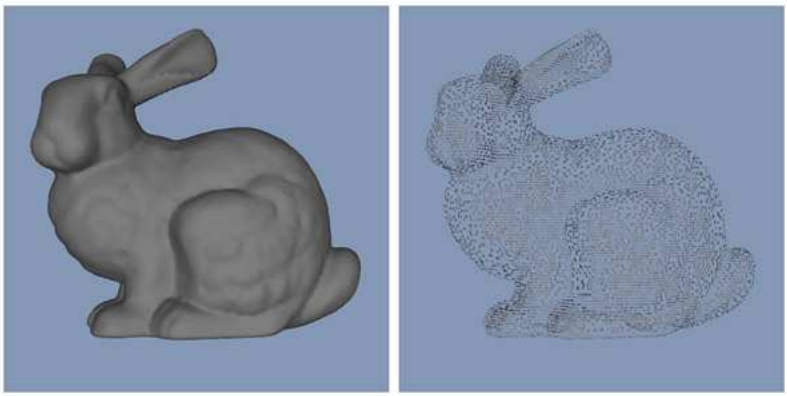


Figure 1.2: Hole free rendering and Just Points

The term point-based modeling is used for point-based surface representation, processing and editing. Point-based modeling techniques are motivated by the need to have a complete geometry processing pipeline from acquisition (3D scanning) to visualization (point-based rendering). Point-based surface representations allow working with point clouds as if they formed a surface: we can ray-trace them or perform smoothing and any surface modifications. A big advantage of points for modeling purposes is the ease of object topology modification.

Points are the simplest of all graphics primitives. No information about connectivity or topology is stored explicitly. The lack of this information is both its strength as well as weakness. Many 3D acquisition methods generate point clouds as output. Points are natural representation for 3D acquisition systems. Triangle meshes are the result of the surface reconstruction algorithms and require prior assumptions about topology and sampling. From a rendering point of view, triangle meshes may not be suitable as meshes are becoming more and more complex while the typical screen resolution does not grow as fast. The steadily increasing performance of CPUs and graphics hardware, the cheap memory, and the wide availability of range-scanning devices result in the acquisition and generation of massive highly detailed geometry data – models consisting of several millions of triangles are common nowadays. When the number of triangles exceeds the number of pixels on the screen, rendering such massive datasets leads to triangles whose projected area is less than one pixel. In this situation, the traditional incremental rasterization methods become inefficient because of the expensive triangle setup. Hence, points seem to qualify as a better suited rendering primitive for such highly complex models. Adaptive modification and dynamic sampling of the surface do not suffer from the complexity and robustness like triangle mesh. The independent nature of the points provides a inbuilt level of detail (LOD), which is simple, fast and almost continuous – making it possible to stream points progressively. Despite its advantages, point based methods suffer from surface normal discontinuities in representation and while representing flat and/or less curvature areas. Shading is not independent from the sampling rate.

Recent advances in 3D model-acquisition technologies, such as laser scanning, have led us to a stage where we can now scan more accurately (at sub-micron levels) as well as at great distances (even

entire cities, in some cases). This has led to the emergence of massive and highly detailed 3D point-cloud data. Such large point-cloud datasets have inspired new research direction in their representation and rendering. Many technical issued related to point-based graphics boil down to reconstruction and resampling. A point based representation should be as small as possible while conveying the shape well.

Direct rendering of points leaves the holes in the rendered image for closeup views. Point based methods require a denser sampling compared to triangle meshes. They must cover the whole surface to assure hole-free reconstruction of the surface. Point-based rendering suffer from the limited resolution of the fixed number of samples representing the model. At some distance, the screen space resolution is high relative to the point samples, which causes under-sampling. A better way of rendering/representing a model is to resample the surface during the rendering at the desired resolution in object space, guaranteeing the sampling density is sufficient with respect to image space resolution. The output sensitive sampling samples objects at a resolution that matches the expected resolution of the output image. This is crucial for hole-free point-based rendering. A common and effective means for this sampling is ray-tracing.

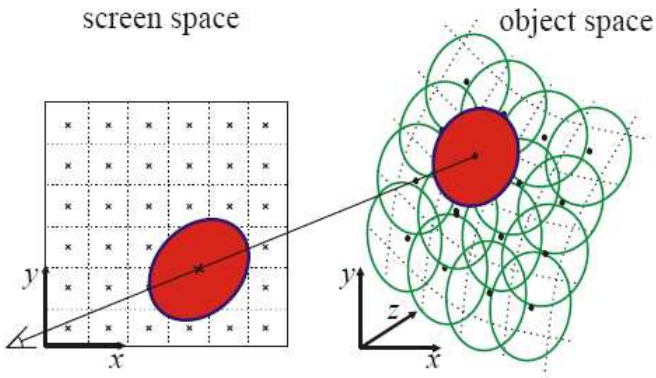


Figure 1.3: Splatting

1.2.1 Point Primitive and Splatting

The situation is shown in Figure 1.3. Each point sample has a position, surface normal and shading attributes. If we assign an area to the point sample, it becomes a surface element or a surfel. A surfel represents a piece of surface rather than a simple point sample. We do not distinguish surfels from points and we use the two terms interchangeably. The surfel can be expressed by a point and a radius, assuming that the surfel represents a circular area in object space. The surfel areas must fully cover the surface to assure hole-free reconstruction.

Splatting assumes that a surface area is assigned to each surfel in object space; it reconstructs the continuous surface by drawing an approximation of the projection of the surfel, which is an ellipse for circular surfels. The projected surfel shape is called a splat. For splatting algorithms the point density

is often specified as a per-surfel radius r which is set to a value which assures that the surfels cover the surface completely.

1.3 Metaballs

Metaballs, also known as blobby objects, is a type of implicit modeling objects. We can think of a metaball as a particle surrounded by a density field, where the density attributed to the particle (its influence) decreases with distance from the particle location. A surface is implied by taking an isosurface through this density field - the higher the isosurface value, the nearer it will be to the particle. The powerful aspect of metaballs is the way they can be combined. By simply summing the influences of each metaball on a given point, we can get very smooth blendings of the spherical influence fields. They have been used widely to visualize the results of particle based simulations, fluids, and other dynamic objects. Such simulations use a very large number of particles, usually in the order of millions.

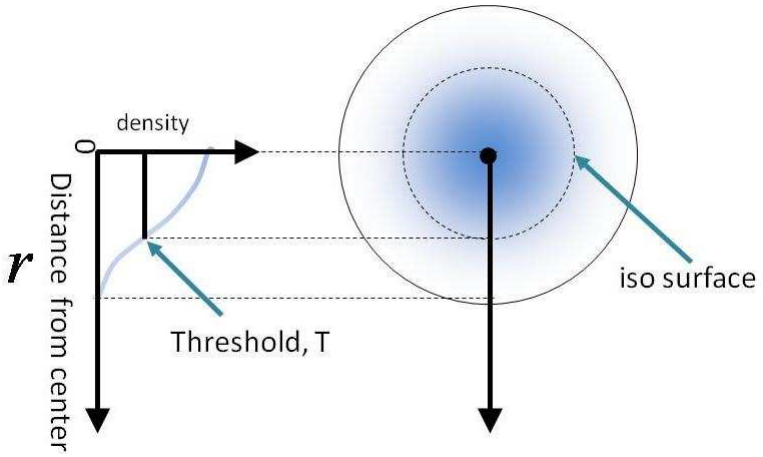


Figure 1.4: A Single Metaball

1.3.1 Definition

A metaball i is a sphere, with a potential field centered at its center p_i. The density field of i is defined by a density function f_i, which monotonically decreases with the distance r from p_i. f_i has a finite support R_i and reduces to zero beyond it (Figure). The isosurface of the density field is defined using a threshold T. The isosurface with more than one metaball is defined by the following equation

$$f(x) = \sum_{i=0}^{N} f_i(x) - T = 0, \tag{1.1}$$

for a threshold T. The bounding sphere of a metaball is a sphere centered at p_i with radius R_i.

1.3.2 Density Functions

There are two classes of density functions. Functions with finite support and infinite support. Infinite support functions, as their name specifies have infinite reach to influence the density field of the space. However, infinite functions are computationally expensive to evaluate because we need to take all metaballs into account. One of the popular infinite density functions is given by Jim Blinn. He used approximating each density field by a Gaussian potential (see Equation 1.2), and using superposition of these potentials to define a surface.

$$f(r) = exp(-ar^2) \tag{1.2}$$

where a is a coefficient and r is distance from the center of a metaball.

With high computational cost involved in dealing with infinite support density functions, some researchers have proposed several polynomial functions with finite supports, such as piecewise quadratic, quartic, and sixth-degree polynomials. While higher degree polynomials produce smoother results, the computational cost is also increased with the degree. Equations 1.3 and 1.4 shows quartic and sixth-order density functions.

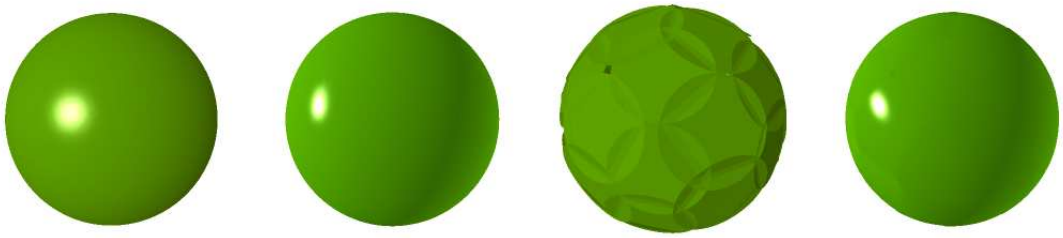
$$f_i(r) = (1 - (\frac{r}{R_i})^2)^2 \tag{1.3}$$

$$f_i(r) = -\frac{4}{9}(\frac{r}{R_i})^6 + \frac{17}{9}(\frac{r}{R_i})^4 - \frac{22}{9}(\frac{r}{R_i})^2 + 1 \tag{1.4}$$

1.4 Contributions Of The Thesis

Algebraic Splats: Points carry all the attributes required for shading. Each sample in the representation has certain attributes – normal, color, material properties – for rendering and shading. Typically a point sample also has a radius that defines an area around it. Such samples are called surfels, the surface elements, approximate the shape of the surface linearly in their neighborhood. Since linear splats are flat shaded, such representations require a large number of samples for good shading quality. The high resolution of the representation leads to increased bandwidth requirements between CPU and GPU. The bandwidth has to be traded with the processing speed.

The demand for large number of splats causes slower rendering due to increase in rendering computation and related memory bus activity. Non-linear patches can approximate the shape of the model well in large neighborhoods. Thus, piecewise non-linear patches can represent the model using fewer number of primitives with the same or better quality than linear primitives. Although rendering a non-linear patch is slower compared to a linear splat, overall speed of rendering for an object could be improved. The approximation error can also be less using a small number of higher order primitives. Figure 1.5 shows the advantages with using non-linear splats over using large number of linear ones. Quadratic splats can approximate a sphere even with a few tens of primitives.



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

Figure 1.5: A sphere rendered using 3200 linear (1.5a) and quadratic (1.5b) splats. A sphere rendered using 52 linear (1.5c) and quadratic (1.5d) splats. Non-linear splats can approximate the shape better with fewer points.

In this thesis, we consider splats that have an algebraic form. Specifically, we use primitives defined by polynomial functions in the local neighborhood. We call these primitives *algebraic splats*. We represent a given point set with a user-specified small number of algebraic splats with optimal rendering quality. This is done by decimating the point set and jointly approximating each using a local algebraic surfaces based on the MLS procedure. Our rendering provides smooth surfaces with normals everywhere. We can render polynomials directly on today's GPUs using ray-tracing because of the semi-implicit nature of the splats in the local reference domain. The major contributions in this thesis are: (i) A lightweight representation for point-based geometry for a given visual quality. (ii) A multi-resolution representation for continuous LOD using algebraic splats and (iii) Adaptive anti-aliasing for algebraic splats rendering. The David 2mm model can be represented using about 30K (or 0.8% of original) algebraic splats with little or low reduction in visual quality (Figure 1.6) at 220 fps. We are able to ray-cast the David model with adaptive antialiasing (3×3 grid) at 130-150 fps and 80-90 fps with shadowing.

Metaballs: We present a method to ray-cast a large number of dynamic metaballs at interactive rates on a commodity GPU. We rebuild the representation in each frame and render the balls interactively. We use perspective grids as our acceleration structure due to their fast building and high coherence for ray casting. We modify the adaptive marching points method to iteratively evaluate the implicit iso-surface equation [48]. We achieve interactive frame rates for one million metaballs. Our method handles primary rays at least an order of magnitude faster than the best reported so far. Our method can be used to visualize simulations live as a result.

1.5 Organization of Thesis

Chapter 2 discuss the evolution of point based graphics and moving least squares approximation and ray tracing on graphics hardware. We also discuss highly related work in surface representations, direct

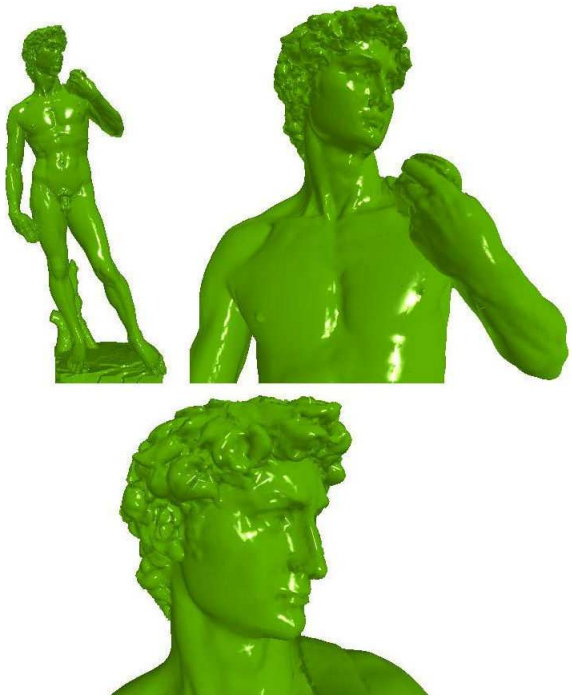


Figure 1.6: David model rendered using 30K algebraic splats at 220 fps on nVidia's GTX 280.

rendering of implicit surfaces and metaball rendering. Chapter 3 and Chapter 4 discuss the representation and rendering of algebraic splat representation. We present a multi-resolution representation and adaptive anti-aliasing for algebraic splat representation. Chapter 5 is devoted to ray-casting metaballs on the GPU.

Chapter 2

Background and Related Work

2.1 Rise of Points

Computer graphics systems traditionally used triangles as rendering primitives. Points were proposed as universal rendering primitives by Levoy and Whitted in 1985 [31]. Instead of deriving a rendering algorithm for each geometry representation, they proposed to subdivide each representation into a sufficiently dense set of sample points. Reyes architecture [13] which used a very similar idea by dicing surface into micro-polygons became very popular during that time and points as rendering primitives did not receive attention until beginning of the last decade.

By the time point-based rendering was resurrected, the graphics landscape had changed considerably. The number of pixels covered by a typical polygon had been shrinking for a decade, and hardware antialiasing was becoming common place. These developments made connectivity less important than it was in 80s, and they made antialiased points a more attractive primitive. At the same time, display screen resolution was rising slowly, which made accurate antialiasing less critical than it was 15 years earlier. In addition, the 1990s saw the development of several new ways to create points, including 3D scanning and particle-based physics simulations. In some cases these pointsets included connectivity information, but in other cases they did not, leading to point-cloud data. Finally, new techniques had been invented for discretely approximating the operators of differential geometry, enriching the set of operations that could be applied to point-based representations of surfaces.

2.1.1 Chronological Overview

In 1985, when Levoy and Whitted [31] proposed points as universal rendering meta-primitives, they argued that for complex objects the coherence and therefore the efficiency of scanline rendering is lost and that points are simple yet powerful enough to model any kind of object. The conceptual idea was to have a unique rendering algorithm (that renders points) and to convert any object to point representation before rendering. This would obviate the need to write a specialized rendering algorithm for every graphics primitive. Very similar idea was used in the Reyes architecture [13] by dicing into micropoly-

gons. Levoy and Whitted used splatting with circular splats to render points, they limited the minimum splat size to a pixel size to avoid aliasing. They used accumulated weights for coverage estimation and an a-buffer [9] for visibility and composition in the same way as Zwicker et al. [65] 16 years later.

In 1992, Szeliski and Tonneson [51] used oriented particles for surface modeling and interactive editing. The oriented particles were points with local coordinate frame, interacting with each other by long-range repulsion forces and short-range attraction forces. However, they did not use point-based rendering for visualization of oriented particles. They visualized the primitives with ellipses and for final production they constructed a surface triangulation. They considered the oriented particles as surface elements, "surfels".

In 1994, Witkin and Heckbert [60] used oriented particles for sampling and interactive editing of implicit surfaces. The particles were visualized in the same way as in [51]. In 1998, point rendering was revisited by Grossman and Dally [18]. The aim was to develop an output sensitive rendering algorithm for complex objects that would support dynamic lighting. The work was mainly inspired by advances in image based rendering.

A real boom of point-based rendering started in 2000 with surfels of Pfister et al. [41] and QSplat of Rusinkiewicz and Levoy [46]. A new motivations for developing point-based techniques were (and still are) advances in scanning technologies and rapidly growing complexity of geometric objects. Pfister et al. [41] extended the work of Grossman and Dally with a hierarchical LOD control and hierarchical visibility culling, they developed a better technique to resolve visibility, the visibility splatting, and they paid a lot of attention to antialiasing. The resulting system was still similar to image based-rendering. A survey of techniques that led to the development of surfels is given in [63].

Rusinkiewicz and Levoy [46], on the other hand, devised a brand new data structure for hierarchical LOD and culling and they used splatting for surface reconstruction. They receded from sampling the objects on regular lattice, therefore they left the influence of image-based rendering. Their aim was to interactively display massive meshes, with rendering quality adapting to the power of the computer used for rendering. Year 2001 brought the EWA surface splatting [65], dynamic object sampling during rendering [58, 49] , hybrid polygon-point rendering systems, differential points [25], spectral processing of point sampled surfaces [40], multiresolution point-based modeling [32], and the MLS surfaces [3].

EWA surface splatting of Zwicker et al. [65] combines the ideas of Levoy and Whitted [31] with Heckberts resampling framework [27] to produce a high quality splatting technique that features anisotropic texture filtering, edge antialiasing and order independent transparency. Hybrid polygon-point rendering systems definitely leave the idea of points as an universal rendering primitive. They build on the observation that points are more efficient only if they project to a small screen space area, otherwise polygons perform better.

Methods for dynamic object sampling produce point samples of rendered geometry in a view dependent fashion as they are needed for rendering (i.e. during rendering, not as a preprocess). Wand et al. [58] used randomized sampling of a triangle set to interactively display scenes consisting of up to 1014 triangles. Stamminger and Drettakis [49] used deterministic sampling pattern to dynamically sample

complex and procedural objects. They also presented a randomized LOD technique which is extremely useful for hardware accelerated point rendering.

Kalaiah and Varshneys differential points [25] extend points with local differential surface properties to better model their neighborhood. The differential points are sampled from NURBS surfaces, but they can smaple points from any kind of differentiable surface. The initial point set is then reduced by eliminating redundant points. More on this in later sections. Pauly and Gross [40] presented a framework for applying Fourier transform and spectral techniques to point sampled surfaces. In order to get the planar supports for the geometry (to be able to apply the FT) they decompose the surface into small patches defined over planar domains. The spectral techniques (i.e. filtering) are then applied to each patch separately and in the end the patches are blended together to get the result of the filtering operation.

Linsen [32] proposed a multiresolution modeling framework for point sampled surfaces. He developed up- and down-sampling operators, surface smoothing and multiresolution decomposition. He also applied CSG operations to point sampled surfaces. Rusinkiewicz and Levoy [46] presented an extended version of QSplat capable of streaming geometry over intermediate speed networks. The focus was mainly on efficient implementation of existing point rendering algorithms (mainly EWA splatting), point-based modeling and on applications of point-based rendering. Pointshop 3D of Zwicker et al. [64] is an editing system that allows to work with point-based surface as with images. Except from all possible effects used in conventional image editors, they could perform altering of surface geometry (carving, sculpting, filtering). Alexa et al. [3] proposed a point-based surface definition that builds on fitting a local polynomial approximation to the point set using moving least squares (MLS). The result of the MLS-fitting is a smooth, 2-manifold surface for any point set. They used the MLS-surfaces to up- and down-sample the point set and to dynamically up-sample the point set during rendering to obtain high quality smooth surface. As an approximation scheme, moving least-squares is insensitive to noise and can be approximated locally [30]. Many formulations of PSS are used for surface reconstruction [5, 15, 16], with linear rendering primitives. The Algebraic Point Set Surfaces (APSS) [20] locally approximate the data using spheres instead of points, which significantly improved stability of projection under low sampling conditions. APSS gives good approximation of the shape at sharp edges. Sphere fitting mechanism uses algebraic distances between points instead of geometric distances. The planar MLS can be obtained as special case of the APSS projection.

2.2 Most Relevant Work

Recent advances in 3D model-acquisition technologies, such as laser scanning, have led us to a stage where we can now scan more accurately (at sub-micron levels) as well as at great distances (even entire cities, in some cases). This has led to the emergence of massive and highly detailed 3D point-cloud data. Such large point-cloud datasets have inspired new research direction in their representation and rendering. Point based representation should be as small as possible while conveying the shape well.

Many technical issued related to point-based graphics boil down to reconstruction and resampling. This section reviews some of the previous work done for efficient representation of points.

Differential Points: Kalaiah and Varshney introduced Differential points [25] which take advantage of the surface curvature information for local illumination and shading. Differential points store local differential surface properties with each point: the principal directions (directions of minimum and maximum curvatures) and the principal curvatures (the normal curvatures in the principal directions). The curvature information is used to derive a local surface geometry at each DP. This surface approximation is used to derive the local normal distribution at each point which is in turn used for shading. Unlike conventional splats representation that use one normal per splat, differential point is a large rectangle with a normal-map.

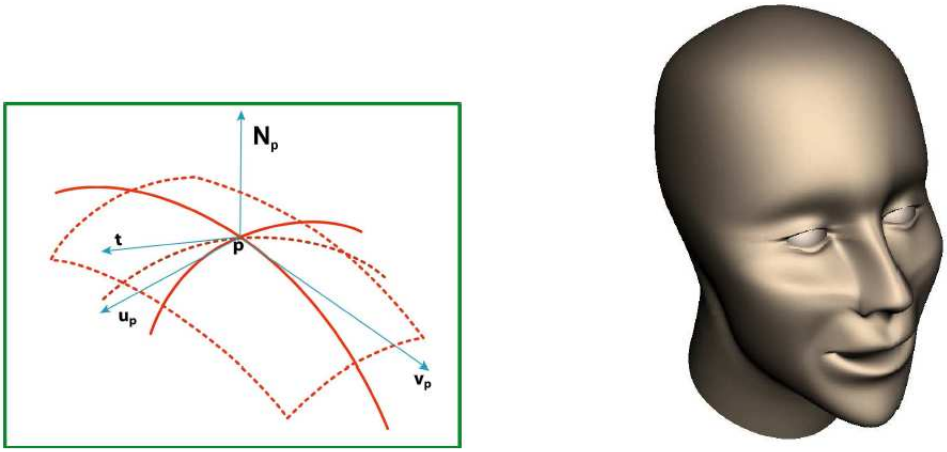


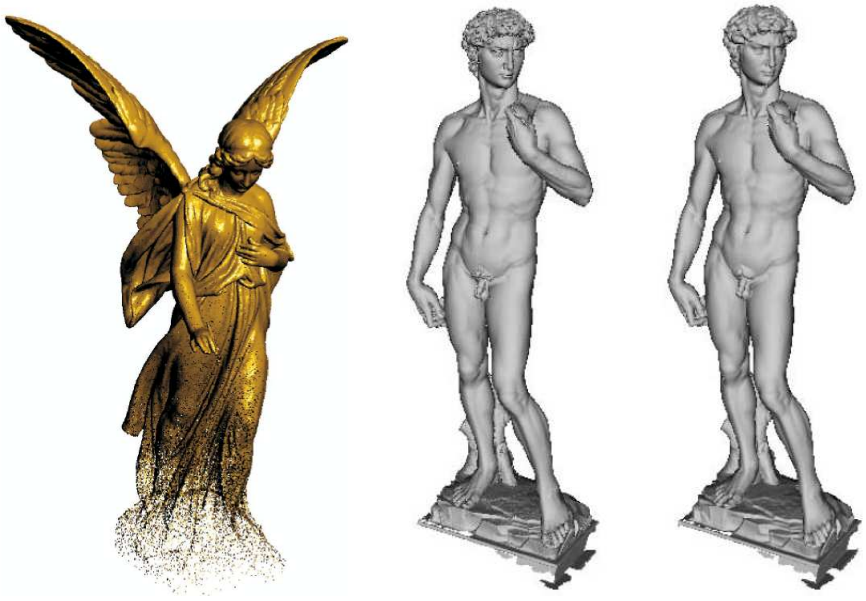
Figure 2.1: Neighborhood of a Differential Point and a rendering of the human head model using Differential Points [25].

The differential points are categorized into 256 varieties based on the relative combination of the principal curvature values. As a preprocess, the normal distribution of each of these quantized DPs is computed and stored as a texture map. Then a rectangle is placed on the tangent plane of each DP, with its width and height being proportional to the principal curvatures. At runtime, the surface approximation at each DP is rendered as a normal mapped rectangle. Thus the differential properties are used for shading. Differential point rendering is excellent at rendering smooth objects. The drawback of their rendering technique is the lack of interpolation between points and is suitable only for smooth surfaces.

The number of primitives being equal, DPs produce a much better quality of rendering than a pure splat-based approach. Visual appearances being similar, DPs are about two times faster and require about 75% less disk space in comparison to splatting primitives. Although DP opened up new possibilities, the differential properties of local surface used here are more rigorously represented by Moving Least Squares(MLS) approximation. MLS are more versatile in representing and rendering the surface

properties. In Chapter 4, we will show that our method is faster by a factor of 10 on most models and require about 90% less diskspace in comparision to linear splats.

Point-Set Surfaces: Point Set Surface(PSS) [3] is an efficient smooth surface representation for point sets, which is constructed using Levins moving least squares projection operator [30]. The projection is an iterative procedure where at each step the point is projected onto a polynomial approximation of the neighboring data, and the polynomial approximation is fitted from a local reference plane computed by a non-linear optimization. The approximation is dynamically sampled at render-time to produce the required sampling density which assures a hole-free reconstruction. To compute the local polynomial approximations, they use their MLS fitting. The result of the MLS-fitting is a smooth, 2-manifold surface for any point set. The basic building blocks of MLS-surfaces are projection operators that can project any point near the surface onto the surface (while the surface is represented solely by the point samples) and computation of local polynomial approximation to the surface at any point. The primitives of rendering for point set surfaces are still flat shaded quads.



(a) (b)

Figure 2.2: (2.2a) **PSS:** A point set representing a statue of an angel. The density of points and, thus, the accuracy of the shape representation are changing (intentionally) along the vertical direction [3]. (2.2b) **SLIM:** Control of nodes in LOD rendering of David model. Left: leaf nodes (932K). Right: LOD nodes (126K) [27].

Recently, Guennebaud et al [20] proposed an algebraic point set surfaces framework to locally approximate the data using algebraic spheres. Compared to MLS approximations [3], this strategy exhibits high tolerance with respect to low sampling densities while retaining a tight approximation of the surface. Guennebaud et al. [20] define the surface by means of sphere fitting which significantly improves

Representation	Primitives	Rendering	Noise	Size
Differential Points	Normal Mapped Rectangles	2× speedup	Not Robust to Noise	25% of original
Point Set Surfaces	Points	Not Realtime	Robust to Noise	NA
SLIM	Low order polynomials	5× speedup	Not Robust to Noise	NA
Algebraic Splats	Low order MLS surfaces	10× speedup	Robust to Noise	10% of original

Table 2.1: Comparision of Various Closely Related Works.

the robustness against low sampling density. For PSS [3], even though fitting polynomials allows achieving tighter approximations, the approach fails when the data cannot be locally represented as a height field. However, moving least squares of fitting of algebraic spheres performs better at the regions of high curvature.

Adamson and Alexa [1] developed an intersection test based on Alexas MLS-surfaces. The idea of the intersection test is to iteratively project a point on the ray onto the surface to converge to the intersection point. The intersection test between a ray and a point sampled surface is not a solved problem. This method shares the advantages of the MLS-surfaces: it is possible to define a minimum feature size and to filter out every feature smaller than this size, the surface is smooth and manifold. These advantages are spoiled by long execution times (hours to compute a single image)

The MLS method associates a local approximation with each location in the parameter domain. For the evaluation of MLS surface one has to compute the polynomial coefficients for each location. Our approach is to reduce the amount of computations is to divide the local parameter space into larger surface elements and use only one approximating function in that neighbourhood. To achieve an overall continuous approximation, we make the cells slightly overlapping and blend the local approximations in the areas of overlap. In Chapter 3 we discuss methods for doing this.

SLIM Another class of surface approximation algorithms are based on Multi-level Passion of Unity (MPU) [38]. Their idea of representation is a blend of locally fitted implicit surfaces. Sparse Low-degree IMplicits (SLIM) approximate the geometry using bivariate polynomials. Efficient rendering is done by blending the primitives in screen space. However this approach still suffers from polynomial fitting limitations and does not properly define a smooth surface due its view dependent nature. These surfaces are parametrized over an ϵ-ball neighborhood of points, which is not suitable for irregularly sampled or noisy models. In Chapter 4 we will show that our rendering scheme is performs at twice the speed of SLIM for a given number of splats, but requires less number of splats for a given visual quality.

2.2.1 Locally Optimial Projection

Lipman *et al* [34] proposed a parametrization-free projection for geometry reconstruction using the Locally optimal projection (LOP) operator. We use the LOP operator for decimation of the point set as it can adapt to the local structure well. The LOP operator projects a set of points $X = \{x_i\} \subset \Re^3$ to an

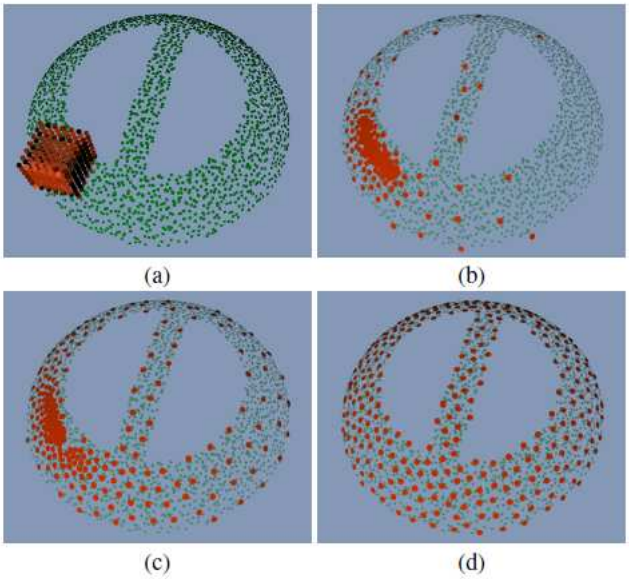


Figure 2.3: This example depicts the distribution of point by LOP operator. (a): Starting from a crude initial guess (red points projected onto the green point-set), the operator iteratively (b - d) distribute the points regularly while respecting the geometry faithfully [34].

input set of points $P = \{p_j\} \subset \Re^3$. Points in X move in each iteration of LOP so as to reduce the sum of weighted distances to P. After several iterations, the points in X are regularly distributed into the input-point cloud. If we start with M arbitrary points as the set X, an optimal approximation of P using M points can be obtained on convergence. The rationale is that the input data can be regarded as consisting of multiple observation of the same data (e.g., multiple registered scans), and LOP operator generates a concise set of points that represents well the input data. In a sense, it operates like a multivariate median, which defines a representative to a set of samples. Then, to up-sample the initial projection, we enriched the above projected set and performed few (3 or 4) iterations of LOP. See Figure 2.3 where an initial crude guess results in a fair and faithful approximation.

2.3 Moving Least Squares Approximation

Suppose that the discrete point set $P = p_i \in R^D, D = 3, i \in 1, 2, ..., n$, p_i is the positional information, is sampled from an unknown surface S. The goal of the MLS surfaces is to find a computational method, which directly defines a reconstructed surface from P. Assuming that the surface function $f(x)$ is defined in arbitrary parameter domain Ω, which approximates the given scalar values f_i for a moving

point $xinR^D$ in the MLS sense f is taken from Π_r^D, the space of polynomials with total degree r in D spatial dimensions. The idea is to start with a weighted least squares (WLS) formulation $f'(x)$ for an arbitrary fixed point in R^D, and then move this point over the entire parameter domain, where a WLS fit is evaluated for each point individually. Then, the fitting function $f(x)$ is obtained from a set of local approximation functions $f'(x)$.

$$f(x) = \underset{f'(x)\in\Pi_r^D}{\arg\min}\ \Sigma_{i=1}^{n} w_i(\|xp_i\|)\|f'(p_i)f_i\| \tag{2.1}$$

$$f'(x) = g^T(x)c(x) = \Sigma_{j\in[1,m]} g_j(x)c_j(x) \tag{2.2}$$

then, $f(x)$ can be expressed as

$$f(x) = \min \Sigma_i w_i((\|xp_i\|)\|g^T(p_i)c(x)f_i\| \tag{2.3}$$

where, $g(x) = [g_1(x), g_2(x), .., g_m(x)]^T$ is the polynomial basis vector and $c(x) = [c_1(x), c_2(x), .., c_m(x)]^T$ is the vector of unknown coefficients, which we wish to resolve to satisfy Equation 2.3. The number m of elements in $g(x)$ and $c(x)$ is $m = (D+r)!/(D!r!)$. $w_i((\|xp_i\|)$ is the weighting function by distance to x, and has the following characteristics: compact support, non-negative, and monotone decreasing. Many choices for the weighting function have been proposed, such as the Gaussian

$$w_i(d) = e^{d^2/h^2} \tag{2.4}$$

where $h$is a spacing scalar parameter, which can be used to smooth out small features in the data, and can be also called as the radius of supporting region or bandwidth.

2.3.1 MLS Surface

Moving Least Square approximation has two stages. First, a local reference domain, H_i, at a point r_i is established by fitting a weighted least square plane to the points in the neighborhood. The normal (n, D) is estimated by minimizing

$$\sum_{j=1}^{N} \left(\langle n, p_j\rangle - D\right)^2 \theta\left(||p_j - q_i||\right), \tag{2.5}$$

where q_i is the origin of the plane and p_j is a point in neighborhood of r_i. The projection of r_i onto the plane is used as the origin. In the second phase, a local bi-variate polynomial approximation g_i of surface, S, is computed in the local reference domain (u_i, v_i, n_i) by optimizing

$$\sum_{j=1}^{N} \left(g_i\left(u_j, v_j\right) - f_j\right)^2 \theta\left(||p_j - r_i||\right), \tag{2.6}$$

where (u_j, v_j) is the projection of p_j onto H and $f_j = \langle n, p_j - r_i\rangle$ is the height of p_j over H. The weighting function θ is a smooth, positive, monotone decreasing function θ, such as the Gaussian given by $\theta\left(d\right) = e^{(\frac{-d^2}{h^2})}$. In practice k nearest neighbors of r_i only are used for the computation.

2.4 Metaballs

A fast visualization method that can handle a large number of metaballs at high quality is highly desirable. State of the art methods are able to ray-cast 128K metaballs at 2-3 fps. Every metaball has a density function. A set of these can represent smooth deformable, free-form shapes represented as the iso-surface of a scalar field. The isosurface can be visualized by tessellating the surface [36] and then rendering the corresponding polygonal mesh (for example, using marching cubes algorithm [36]). The major drawback of such techniques is the quality of resulting mesh depends on the resolution of the grid. Polygonaization can also miss objects smaller than gird resolution. On the other hand, with a high-resolution grid, polygonization can keep the finite details and generate high quality surfaces. This needs high-computation and memory and is not suited for interactive applications.

Loop and Blinn [35] and Singh and Narayanan [48] rendered very small number of blobbies interactively (See Figure 2.4). Both methods are not scalable to large number of metaballs. In [35], Loop and Blinn proposed a GPU-based ray casting method to render algebraic surfaces defined by Bzier tetrahedrons with up to quartic polynomials. For ray-isosurface intersection tests, the coefficients of the equations are efficiently computed by linearly interpolating the vertex attributes of each tetrahedron. The computation of vertex attributes must take the neighboring metaballs into account, which can be a considerable performance hit for a large number of moving metaballs.

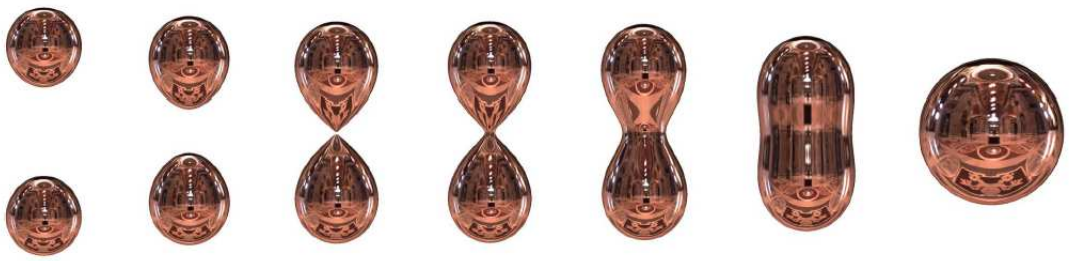


Figure 2.4: Several states of an animated fourth order algebraic surface rendered in real-time using [35]

Iwasaki et al. [24] used metaballs for visualizing the results of a particle-based fluid simulation. They divided the space into a grid, and accelerated the evaluation of the density function at each grid point using the GPU. Similarly to polygonization, however, their method can miss small splashes. Recently, van Kooten et al. [53] visualized metaballs on the GPU, using on-surface particles distributed by repulsive forces. Their method can exploit the temporal coherence of animations, however, as they reported, it suffers from small objects due to the limited resolution of the buffer for repulsion computations. Since our method is based on ray casting, it can display small objects.

Recently, Kanamori et al. [28] efficiently ray cast a large number of metaballs on GPU using depth peeling and Bezier clipping [37], but since they rely on rasterization from the image, their method is limited to primary rays, thus restricting their rendering effects to the ones achievable in screen space (for instance, shadow maps or screen space ambient occlusion), contrary to our method which performs

true ray tracing. More recently Gourmel et al. [17] presented fitted-BVH structure for metaball rendering. Their method has a high data structure building time and doesn't scale beyond 100K metaballs at interactive rates.. Dynamic metaballs necessitate fast structure building, which is difficulty for kd-trees, octrees, and BVH [62]. Uniform grids can be rebuilt fast on the GPUs using efficient data parallel primitives [39, 26]. Perspective grids provide fast building and high coherence for ray casting [39].

2.5 Graphics Hardware

Commodity based graphics hardware involving 3D functions were first developed by Matrox, Creative, S3 and ATI in 1995. Graphics hardware production for consumer PC market grew with the introduction of Voodoo Graphics PCI by 3Dfx in 1996. 3dfx also provided a graphics API called GLIDE for their VooDoo cards, which was kepy at a much lower level to the hardware. This was difficult for game programmers by they were able to harness the full power of voodoo cards. The first vooodoo card was capable of only rasterization and pixel processing. Later Nvidia developed the first card named GeForce 256(SDR) in Oct 1999. This card was capable of fixed function hardware based transformation of vertices and lighting with the Direct3D7 and OpenGL 1.2 API.

The GeForce 3, was the first programmable GPU, allowing game developers to do much more (2001). This breakthrough came as Direct3D8 introduced the first widely-used family of shaders: Shader Model 1.1. ATI came up with Radeon 8500 supporting Shader Model 1.4. The shader model at this stage involved a few of assembly instructions in the shader programs. Also branching and looping were not supported at all. The next version, Shader Model 2.0 (Direct3D9), involved improvement in-terms of shader code length. GeForceFX series and Radeon 9700 brought it to the consumers in 2003.

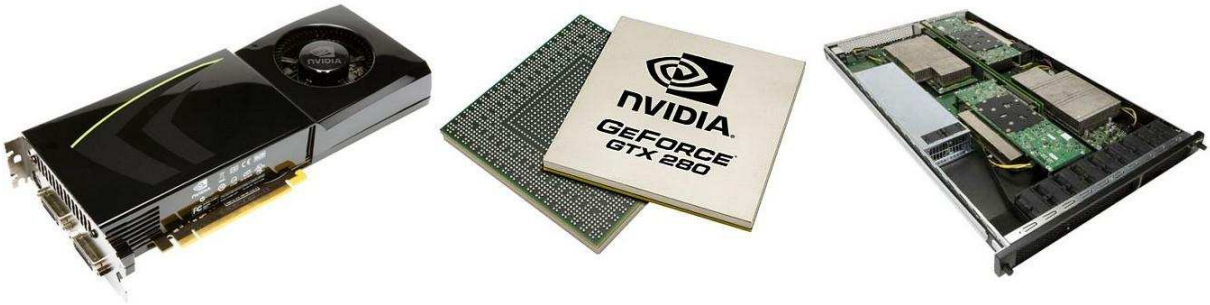


Figure 2.5: Nvidia GTX 280, Nvidia GPU Chip for GTX 200 Series and Nvidia Tesla 4 GPU System shown above. Images obtained from freely available documents provided by Nvidia.

In 2005, we had the GeForce 6 and 7 series, which support Direct3D9c and Shader Model 3.0. X800 supported Direct3D9c but only Shader Model 2.0b. Shader Model 3.0 also saw the dawn of high level shading languages. Direct3D9c named its shading language HLSL and OpenGL2.0 called it GLSL1.1. Shader Model 3.0 contained several improvements: improved floating point precision, more code length, support dynamic branching but with a performance hit, unrolled looping and vertex texture

lookup. Due to such enhancements, Shader Model 3.0 remained the most used by game developers, graphics researchers and GPGPU programmers. NVIDIA released its own shading language Cg in 2003, which was closer to HLSL and was apparently used widely.

By the end of year 2006. NVIDIA released the first card, 8800GTX of their G80 family. This supported SHader Model 4.0 introduced by Direct3D10 System. OpenGL 2.1 whith GLSL1.20 got introduced in November 2006 which fully supported the new features of ShaderModel 4.0 AMD is expected to release its R600 series card, Radeon 2900GTX, in May 2007. The improvements in shader model 4.0 are significant and tend to solve most issues game developers raise.

CUDA (Compute Unified Device Architecture) which is a fundamentally new computing architecture, provides an access to the tremendous processing power of NVIDIA GPUs through a revolutionary new programming interface. The core technology of GPUs which is parallel data processing is now exploited with GPUs entering the space of massively parallel processing. The GPUs now act as a co-processor to the CPU, offloading major part of processing. The GPUs are getting more powerful and more programmable with every generation. While they speed up the rendering of conventional geometry, their impact can be felt more in rendering higher level primitives that are slow to render.

2.6 Ray Tracing

The ray casting algorithm for rendering was first presented by Arthur et al. [6]. Ray casting renders images by tracing a ray from the eye, one per pixel, into the environment. The next important research breakthrough was proposed by Rubin and Whitted [45] in 1979. They extended the idea of ray casting by generating three types of rays: reflection, refraction and shadow, when a ray hits a surface. These techniques have been attempted over years, across architectures like, CPUs, multi-cores, clusters, CellBE, FPGAs etc. Beam Tracing [21] was introduced to exploit the spatial coherence of polygonal environments. Rather than working with high number of rays per image, beam tracing sweeps areas of the scene to form beams. Spatial data structures like kd-trees, octrees and grids have been used for efficient traversal of large models for ray tracing. Grid data structure was used for one of the first ray tracing implementation on the GPU [42]. The data structure was built once on the CPU and stored as 3-D texture on the GPU for traversal.

Before the introduction of the GPU, ray tracing was performed on CPU or on a cluster of CPUs. A single CPU works sequentially on all the rays and finds closest intersections. With the increase in CPU cores and multi threaded architectures, ray tracing could be performed efficiently on a set of processors. MLRTA [44] performs fast ray tracing by allowing a group of rays to start traversing the tree data structure from a node deep inside the tree, saving unnecessary operations. RLOD [61] uses a LOD based scheme which integrates simplified LODs of the model into kd-tree, performing efficient ray-triangle intersections. Wald et al. [54] ray trace deformable objects on the CPU using a bounding volume hierarchy (BVH). They exploit the fact that the topology of the BVH is not changed over time so that only the bounding volumes need be re-fit per frame. In another work, they [56, 55] ray trace

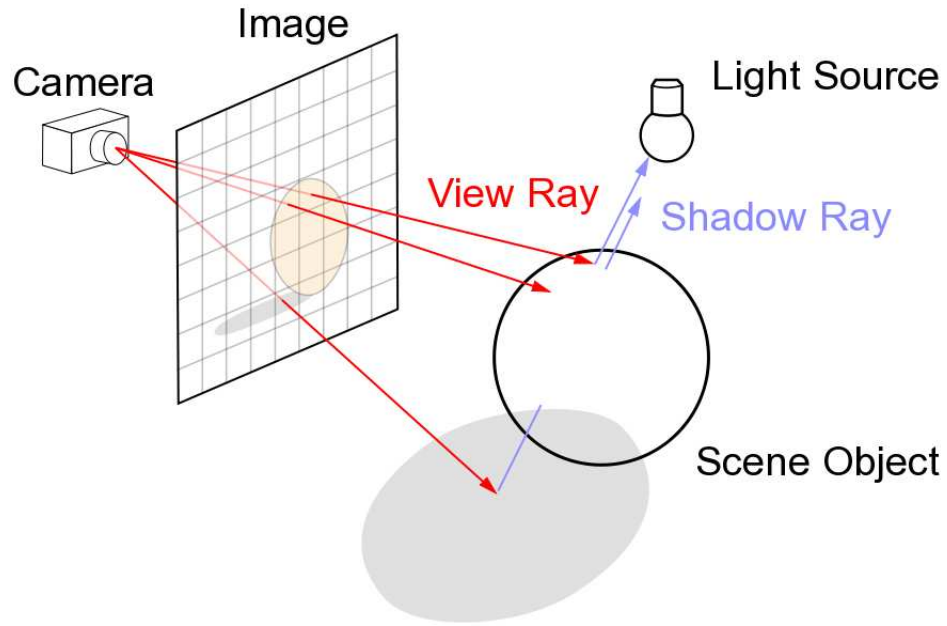


Figure 2.6: The ray tracing algorithm builds an image by extending rays into a scene

animated scenes by rebuilding the grid data structure per frame. They use a new traversal scheme for grid-based acceleration structure that allows for traversing and intersecting packets of coherent ray using an MLRTA-inspired frustum-traversal scheme. Ray tracing has also been performed on non-triangulated models like implicit surfaces [29, 48] and geometry images [19, 11].

Programmable GPUs can perform limited ray tracing due to the constrained programming model [10, 42]. Ray tracing was performed as a multi-pass operation due to insufficient capability of the fragment shaders. With the growth in programmability of the GPU, more efficient methods have emerged which use the looping and conditional operations. Most of the work for ray tracing on GPU uses pre-built data structures, given that the cost of building parallel data structures may be high [23].

Several ray-tracing techniques for point set surfaces on CPU have been proposed [2, 57], but are slow. An architecture for hardware-accelerated rendering of point primitives is presented by Tim Weyrich et al [59], their pipeline implements a refined version of EWA splatting, a high quality method for antialiased rendering of point sampled representations. Recently Linsen et al [33] proposed a splat based ray tracing of point clouds on CPU. GPU rendering is more popular these days and high-level primitives for GPUs are being proposed for ray tracing [43].

Several methods for GPU-based ray tracing of implicit surfaces have been proposed [14, 29, 48]. Sigg et al [47] rendered molecular models directly using spheres, cylinders and ellipses using ray-casting. Stoll *et al* [50] proposed an incremental ray casting method for quadratic surfaces on the GPU. Loop and Blinn proposed a GPU based algorithm for rendering up to fourth order algebraic surfaces defined by tri-variate Bezier tetrahedra [35].

2.7 Rendering High-level Primitives

We used ray-tracing on the GPU to render algebraic splat representation and iso-surface defined by million metaballs. Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The fundamental operation used by our rendering method is the intersection of a ray with an object, see Figure 2.6. The intersection is computed in the pixel shader for each pixel, given the parameters of the object being rendered. This step is essentially the conventional ray casting implemented on the pixel shader. The equation of a ray is $R_f = c_i + t\vec{d_f}$, where c_i is origin of rays, t is ray parameter and $\vec{d_f}$ is direction of ray for fragment f. Our algorithm for rendering a general implicit object on the GPU is given below. The object is given by the implicit form $f(P) = 0$. The algorithm is designed for an Nvidia G80 architecture which implements the Shader Model 4.0. It has parts for the CPU and the vertex and fragment processors on the GPU. Bulk of the work is done on the fragment processors.

Algorithm 1 Ray-tracing using Shader Model 4.0 (f)

CPU: An OpenGL program performs the following

1: Pass the parameters of $f(P)$ to the graphics pipeline as graphics bindings such as texture coordinates, color and position. A texture can be used if more data needs to be sent.
2: Draw an OpenGL primitive such that the screen-space area of the object is covered by it. This ensures that all pixels will be drawn and the corresponding shaders will be executed. The primitive used could be a dummy one with the right number of vertices.

Vertex Shader: A vertex shader performs the following

1: Pass the parameters from the CPU to the pixel shader.
2: Transform the OpenGL primitive drawn by the CPU to cover the screen-space area of the object using the object parameters.
3: Perform other pixel independent calculations required for the pixel shader and passes on the results.

Fragment Shader: A pixel shader performs the following.

1: Receive the parameters of the object and own pixel coordinates (i, j) from the pipeline.
2: Perform an acceptance test for (i, j) based on the parameters of $f(P)$. This involves computing exactly if the pixel will be on the projected region of the object. This may require the parameters of $f()$, the Modelview, Projection, Viewport matrices, etc. The acceptability can be computed in a 2D texture space in some cases.
3: Compute the ray-object intersection for accepted pixels. This involves solving an equation in t that is based on $f()$.
4: Compute the 3D point corresponding to the smallest t among the intersecting points. Also compute the depth and normal at that point using $f()$
5: Shade the pixel using the lighting, material, normal, and viewing information that is available to the shader. The reflected ray at the intersection point can be pursued to apply environment mapping, refraction etc

It is important to setup the screen-space bounding area as compactly as possible as it affects the computation time. A compact bounding polygon is a good option. The CPU and the vertex shader set

this up in combination. Every pixel in the bounding area need to check if it is part of the actual object. The ray-object intersection will give imaginary results for pixels that are outside the object.

Ray-surface intersection test may vary depending on the kind of geometry we are using. We render algebraic splats on the GPU using ray-casting. Algebraic splats are quadratic, cubic and quartic in nature. Each algebraic splat is of the form $g(u, v, n) = 0$, where u, v and n define local coordinate system. The equation of a ray is $R_f = c_i + t\vec{d_f}$, where c_i is origin of rays for i^{th} algebraic splat, t is ray parameter and $\vec{d_f}$ is direction of ray for fragment f. To find ray-surface intersection, the splat equation is transformed into the ray parameter space t, $F(t)$. Solving $F(t)$ for roots gives the ray-surface intersections. The smallest positive root gives the Each fragment or ray has a different equation, which can be solved independently.

Quadratic surfaces are defined by $p^T Qp = 0$ where Q is a 4×4 matrix. Substituting $p = O + tD_f$, the $F_f(t) = 0$ reduces to $At^2 + Bt + C = 0$. Each fragment or ray has a different equation, which can be solved independently. We solve the quadratic equation $At^2 + Bt + C = 0$ using discriminant $D = B^2 - 4AC$ to get the roots as $\frac{-B\pm\sqrt{D}}{2A}$. The smaller positive of these two roots gives the rays intersection with surface. All quadrics can be ray-casted by solving a quadratic equation. Analytic solutions to the ray dependent equation exists only for simple forms of $F_f()$, such as polynomials of order less than five. The cubic and quartic are solved using the following methods.

Solving cubic equations: We solve the cubic equation using the method given by Blinn [7]. For a cubic equation $Ax^3 + 3Bx^2w + 3Cxw^2 + Dw^3 = 0$, compute $\delta_1 = AC - B^2$, $\delta_2 = AD - BC$, and $\delta_3 = BD - C^2$. The discriminant is defined as $\Delta = 4\delta_1\delta_3 - \delta_2^2$. The sign of the discriminant and the values of δ_i^s determine if it has one triple root, one double and a single real root, three distinct real roots, or one real root and one complex conjugate pair as roots.

Solving quartic equations: We use the Ferrari method described by Herbison-Evans for the fourth order polynomials [22]. The equation is first depressed by removing the cubic term to the form $t^4+pt^2+qt+r = 0$. If r is zero, the roots are 0 and the roots of the cubic equation. If r is non-zero, the equation can be written as a product of two quartic equations. This is done by rewriting it as $(t^2+p)^2+qt+r = pt^2 + p^2$. This is followed by a substitution y such that the right hand side (RHS) becomes a perfect square. The equation then transforms to $(t^2 + p + y)^2 = (p + 2y)t^2 - qt + (y^2 + 2yp + p^2 - r)$. Now, for the RHS to be a perfect square its discriminant must be zero which leads to a cubic equation in y whose root is found as described before.

In case of metaball rendering, the surface is defined using non-algebraic implicit equation and a procedural rendering is adapted. It involves on-the-fly creation of arbitrarily accurate shape from compact descriptions, in the form of implicit equations. The graphics display pipeline cannot render procedural geometry directly. Procedural objects are converted to piecewise linear models using polygons and lines before rendering. This results in a loss in resolution and incurs computational overhead. Ray-tracing methods can handle procedural geometry to produce high-quality renderings. Since the surface is non-algebraic, ray-sruface intersection is found using ray marching algorithm. Marching points algorithm

marches in equal stepsizes until the root is found which is detected by a sign change in the function. Marching points wastes computation by computing the function values at many points. Adaptive marching points algorithm marches adaptively to find the root [48]. Only fourth or lower order surfaces can be rendered using analytical roots, adaptive marching points algorithm can ray-trace arbitrary implicit surfaces exactly, by sampling the ray at selected points till a root is found. Adapting the sampling step size based on a proximity measure and a horizon measure delivers high speed. Chapter 5 provides more details about ray-surface intersection test used in metaball rendering algorithms.

Chapter 3

Algebraic Splats

Large number of linear splats are needed to represent the shape of most smooth models. Since the number of linear primitives needed is large, the rendering is slow. Non- linear patches can approximate the shape of the model well in large neighborhoods. Thus, piecewise non-linear patches can represent the model using fewer number of primitives with same or better quality than linear primitives. In this work, we consider splats that have an algebraic form. Specifically, we use primitives defined by polynomial functions in the local neighborhood. We call these primitives *algebraic splats*. We represent a given point set with a user-specified small number of algebraic splats with optimal rendering quality. This is done by decimating the point set and jointly approximating each using a local algebraic surfaces based on the MLS procedure. Our rendering provides smooth surfaces with normals everywhere. We can render polynomials directly on today's GPUs using ray-tracing because of the semi-implicit nature of the splats in the local reference domain.

There are many ways to approximate the surface from an unstructured point cloud. Moving Least Squares (MLS) surfaces are attractive as they can be constructed using local computations. The result of the MLS-fitting is a smooth, 2-manifold surface for any point set. The MLS procedure approximates the local neighborhood using a polynomial in a local reference domain. It is possible to define a minimum feature size and to filter out every feature smaller than this size. It is also well suited to filter noisy input data. We create algebraic splats from the MLS polynomials by bounding each to a disc in the local parameter domain.

3.1 Moving Least Square Surfaces

Moving Least Square approximation has two stages. First, a local reference domain, H_i, at a point r_i is established by fitting a weighted least square plane to the points in the neighborhood. The normal (n, D) is estimated by minimizing

$$\sum_{j=1}^{N} \left(\langle n, p_j \rangle - D\right)^2 \theta \left(||p_j - q_i||\right), \tag{3.1}$$

where q_i is the origin of the plane and p_j is a point in neighborhood of r_i. The projection of r_i onto the plane is used as the origin. In the second phase, a local bi-variate polynomial approximation g_i of surface, S, is computed in the local reference domain (u_i, v_i, n_i) by optimizing

$$\sum_{j=1}^{N} \left(g_i\left(u_j, v_j\right) - f_j\right)^2 \theta\left(||p_j - r_i||\right), \tag{3.2}$$

where (u_j, v_j) is the projection of p_j onto H and $f_j = \langle n, p_j - r_i \rangle$ is the height of p_j over H. The weighting function θ is a smooth, positive, monotone decreasing function θ, such as the Gaussian given by $\theta\left(d\right) = e^{\left(\frac{-d^2}{h^2}\right)}$. In practice k nearest neighbors of r_i only are used for the computation.

There are many variations and extensions [12] to the original MLS surface approach [30]. In this work, we have used original MLS surface approach [30]. Our method is independent of MLS surface approximation method used. Other variations and extensions which generated algebraic MLS surfaces be can be easily fit into this formulation. For example, the MLS surfaces can be generated by Robust MLS[16], which preserves the sharp features.

3.2 Algebraic Splat at a Point

The MLS surface fits the surface in its local neighborhood. We generate an algebraic splat for point r_i by bounding its MLS approximation, $g_i\left(u, v\right)$, to a disc of radius R_i around projection of r_i on the plane H_i (see Figure 3.1). The algebraic splat at point r_i is thus defined in coordinate frame (u_i, v_i, n_i) as

$$g_i(u, v) - n = 0, \tag{3.3}$$

subject to

$$||u_i^2 + v_i^2|| \leq R_i^2, \tag{3.4}$$

where R_i is a function of h_i, which is a local feature size explained later. MLS surfaces are infinite surfaces in local reference domain but are correct only in a local neighborhood. Disk bounded MLS surface in local-reference domain is called an *algebraic splat*. Figure 3.1 shows the MLS surface and the algebraic splat defined using it. Algebraic splats are semi-implicit in nature and provide a good compromise between explicit representations and implicit like MPUs and RBFs. MLS makes it easy to compute the intrinsic properties of the surface like normals.

Algebraic splats of any order can be generated by controlling the degree of the polynomial used in the MLS approximation of the surface. We restrict our attention to the second, third and fourth orders as they are powerful enough to approximate local neighborhood. The process to generate the algebraic splat for a point r_i is given below.

1. Compute local feature size h_i for each point r_i.
2. Compute the MLS approximation at point r_i using the entire point-set. Each point is Gaussian weighted based on the distance to r_i and the feature size h_i.

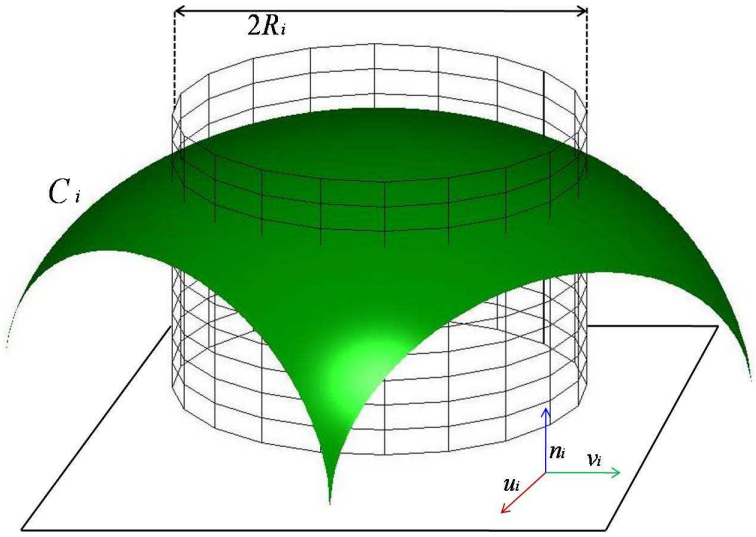


Figure 3.1: Algebraic splats are the MLS surface restricted in a disc at the origin in local reference domain. The cylinder in local reference domain bounds the MLS surface.

3. Generate the algebraic splat a_i as $(o_i, u_i, v_i, h_i, C_i)$.

An algebraic splat is represented by a tuple $(o_i, u_i, v_i, h_i, C_i)$, where o_i is the origin of the local coordinate system. C_i gives the coefficients of the bi-variate polynomial g_i. The number of coefficients depends on the order of the polynomial. We need 6, 10 and 15 coefficients for quadratic, cubic and quartic polynomials, respectively. The disk radius R_i is chosen at run time as a function of h_i.

The local feature size h_i plays a major role in approximating the model. A small value of h_i makes the approximation more local and large values of h_i smooths out the sharp features. We define h_i as the average local distance between the points. This is calculated as the average distance of k nearest neighbors in the local parametric domain for each point.

3.3 Algebraic Splat Generation

Approximating the shape of the point set using a small number of algebraic splats has two aspects. First, the point set is decimated or approximated. Second, a non-linear splat is computed for each point in the reduced set with reference to the original point set. The decimated point sets can have a fixed number of points or can satisfy a quality criterion. A variety of point sampling algorithms were proposed earlier for linear primitives. We need an algorithm that goes well with algebraic splat generation. We present two algorithms which decimate the point-set for algebraic splat representation.

3.3.1 Quality Based Decimation

A splat is assumed to approximate the shape in some neighborhood of radius sh_i, where s is a factor that controls the quality. We discard points within the distance sh_i in the parametric domain. At $s = 1$, the radius of splat is equal to the local feature size. If $s < 1$ then more points are included in the final hole-free representation of model. Number of points in the final set S depends on the *inverse quality factor* s. By changing it, we can control the detail of the model. This helps us design multi-resolution representation for algebraic splats (Section 3.4). The decimation procedure is described below.

1. Remove a point p_i (possibly at random) from input point-set, P and add it to the set S of selected points.

2. Discard all points from P that are inside a circle of radius sh_i, the cut-off distance in the parametric domain.

3. Continue steps 1 and 2 until all points are exhausted from P.

4. Output algebraic splats for each point in S.

The quality based decimated set S has fewer than 10% of the points for most models at $s = 1$, in practice. Figure 3.2 shows rendering of Dragon and Happy Buddha models at quality $s = 1$. The selection procedude in step 1 can be improved to get better or optimal representation (in terms of placement and final number of splats) at higher computational cost. Our experience suggests that random selection produces good quality representations.

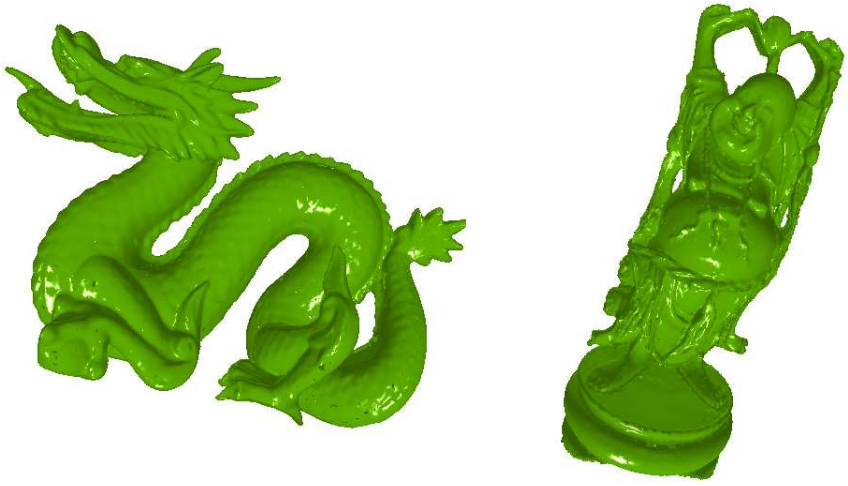


Figure 3.2: Rendering of Dragon with 42K (9.6% of original) and Happy Buddha with 24K (4.4% of original) algebraic splats using quality based decimation scheme at $s = 1$.

3.3.2 Fixed-size Decimation

The model can be approximated by a user-selected number, m, of points. Lipman *et al* [34] proposed a parametrization-free projection for geometry reconstruction using the Locally Optimal Projection (LOP) operator. We use the LOP operator to decimate the point set to a given target number of points. We combine decimation and algebraic splat generation as follows.

The LOP operator projects a set of points $X = \{x_i\} \subset \Re^3$ to an input set of points $P = \{p_j\} \subset \Re^3$. Points in X move in each iteration so as to reduce the sum of weighted distances to P. After several iterations, the points in X are regularly distributed into the input-point cloud. If we start with m arbitrary points as the set X, an optimal approximation of P using m points can be obtained on convergence. This gives an optimal representation of the original shape with m points. LOP is a slow process, however the number of iterations needed for convergence will be less if the set is close to input-point cloud. We combine decimation and algebraic splat generation as follows.

1. Select m random points from the input cloud.

2. Apply LOP operator between these points and the input point cloud P till convergence, resulting in the decimated set D.

3. For each point in D, compute its MLS surface using the set P.

4. Output the algebraic splat for point in D. Calculate radius using the local feature size of the decimated set D.

The decimated point set D is regularly distributed into the shape of the model and gives an optimal approximation of model [34]. This process generates algebraic splat representation with a user specified number of points. Figure 3.3 shows the rendering of fixed-points decimation scheme.

Figure 3.3: Rendering of Igea model with fixed size sampling at 2%, 5% and 10% of original number of points using quadratic algebraic splats.

3.3.3 Approximation Error

We can estimate the error in approximation, as its related to the MLS approximation error estimated by Alexa *et. al.* [4]. The error of approximation at point p_i in the original point cloud is due the combined error of the nearby overlapping splats. The error at a point p_i is given by $\delta_i = \sum_{j \in J} \epsilon_{ij} \theta \left(||o_j - p_i|| \right)$, where ϵ_{ij} is the MLS approximation error at point p_i due to splat a_j, J a set of overlapping splats at point p_i and o_j the center of the splat a_j. The total approximation error, Φ, of the algebraic splat representation is the sum of deviations of the point set from the surface defined by algebraic splats and can be given by $\Phi = \sum_{i=1}^{N} \delta_i$. The surface approximation error ϵ depends on the local feature size h, but is bounded [3, 4]. This ensures total weighted approximation error of algebraic splats is also bounded. If f is the blended surface represented by non-conforming algebraic splats and g is the surface represented by point set then upper bound on the error of approximation is defined as $||g - f|| \leq L \cdot h^{m+1}$, where m is the degree of polynomial and L is the constant that involves $(m+1)^{th}$ derivative of f, i.e., $M \in O\left(f^{m+1}\right)$. If h is high, the error increases, and the surface is smoothed.

3.4 Multi-resolution Representation

Point clouds are amenable to level-of-detail approaches. We introduce a multi-resolution representation for algebraic splats. In Section 3.3, we discussed a quality based decimation scheme. The inverse-quality factor s controls the number of points in the final set. The multi-resolution representation is computed in two stages. In the first stage, we calculate the algebraic splat representations for a minimum (P_m) and a maximum quality (P_M) defined by the user for a particular multi-resolution representation. These correspond to the inverse quality factor of s_0 and s_k respectively, with $s_0 > s_k$ Now P_d, a difference set of P_m and P_M is computed consisting of splats in P_M but not in p_m. For representations between the maximum and minimum resolutions, selectively add splats from P_d to P_m.

In the second stage, P_d is partitioned into K sets by dividing the interval $[s_0, s_k]$ into K steps. The j^{th} partition of the difference set $P_d[j]$ is given by

$$P_d[j] = ASR(s_j) - ASR(s_{j-1}) \tag{3.5}$$

where $ASR(s_j)$, represents the points in the representation for inverse quality factor s_j, computed using the procedure given in Section 3.3.1. Small values of K gives a few discrete LODs while a large K approximates a continuous LOD scheme. A sufficiently large K can ensure smooth LOD transitions without abrupt popping effects due to the introduction of new points. This avoids the need for expensive LOD blending needed when using a few discrete LODs.

In practice, we use 25-50 steps between s_01 and s_k, depending on the model and divide the interval linearly. The LOD is defined at each leaf-node of the octree, which is discussed later. The points in each leaf node are sorted from lower to higher LOD sets. All points till the current LOD level l are used for rendering. If i^{th} node n_i is in l^{th} LOD level then the following set of points are rendered.

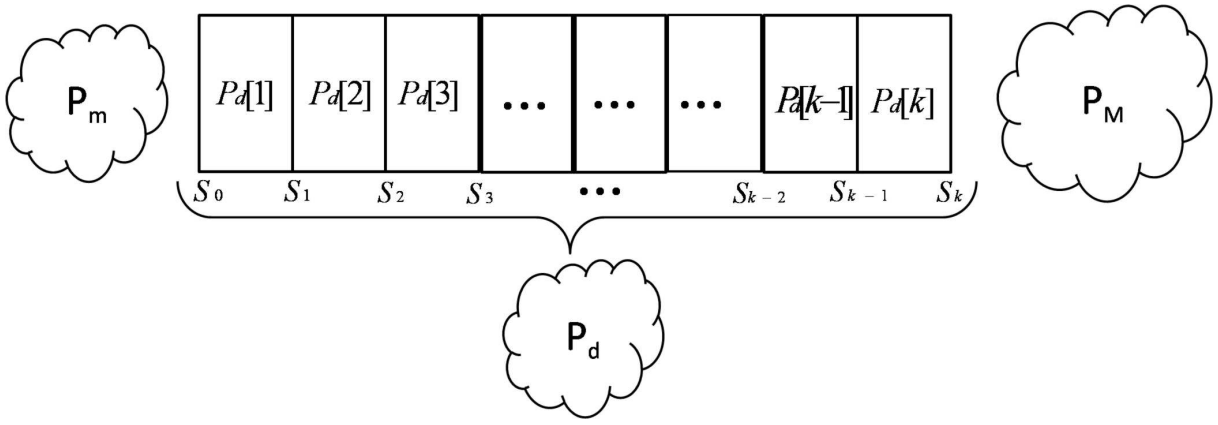


Figure 3.4: Points in the leaf node are arranged in the order of quality based LODs to enable continuous LOD.

$$P_l = P_m \cup P_d\left[j\right], \forall 1 \leq j \leq l \tag{3.6}$$

3.5 Final Data structure

Hierarchical octree data structures are one of the most common choices to handle large point sets (e.g., for interactive rendering). Octree is built by subdivision of space and organize the data in a hierarchical data structure, with the leaf nodes being the actual data. It is advisable to avoid excessive recursive subdivision downto a single data element per leaf node, and instead strive to have a bucket of up to k points per leaf of the hierarchy. This octree is a axis-aligned subdivision at an mean point inside the cell. This strategy has a simpler hierarchy structure and may require less information to be maintained per node. Despite the fact that we are dealing with point-sampled surface objects given by a point set P, these splats are not zero-dimensional elements but do have a spatial, planar extent as outlined earlier. Thus, each splat has an associated disk. In general, objects with spatial extent require referencing from multiple cells

We build an octree structure over the algebraic point set by recursively dividing the bounding box of points until each cell has fewer than a predetermined number of points. Building octree is a one time, pre-rendering operation. The octree is used for view frustum culling and LOD management. Each leaf node consists of a sorted array of points according to partitions defined in Section 3.4. See Figure 3.5 for multiresolution rendering of David.

The final representation consists of an algebraic splat for each point in the decimated set, each needing 16-25 floating point numbers depending on the degree of the MLS surface. These numbers are packed into a 2D texture and stored on the GPU memory. Each splat occupies a 2D section to take advantage of texture caching. The information is stored node-wise. All the data from a node are grouped together in increasing LOD level as described above.

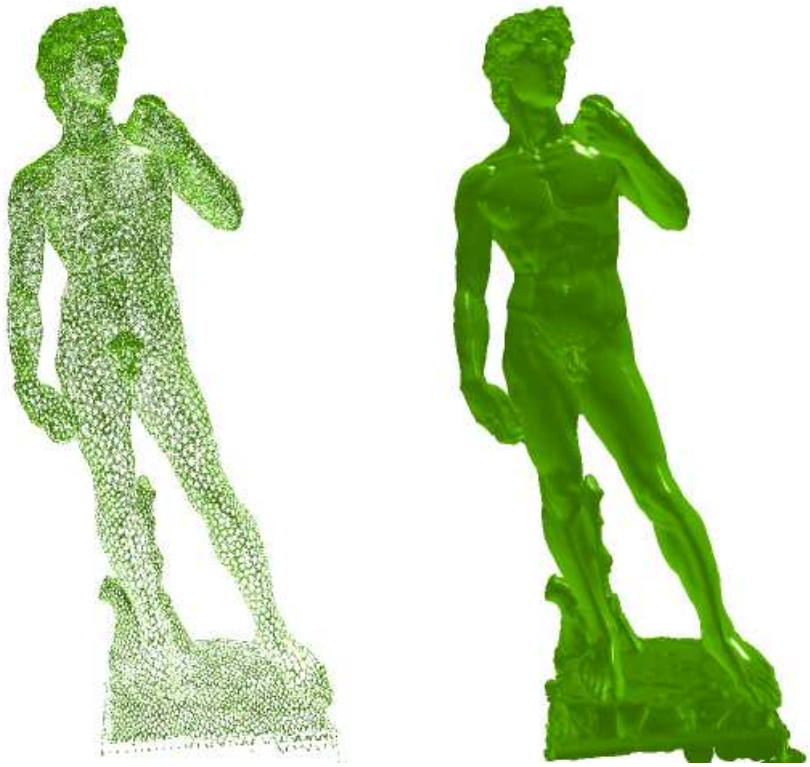


Figure 3.5: Left: A continuous resolution representation of David model for a viewpoint above his head looking down. Notice the point density decreases with distance. Right: Rendering of those points using algebraic splats from a frontal position at 160fps. The quality decreases from head to toe.

Chapter 4

Algebraic Splat Rendering

We render algebraic splats on the GPUs using ray tracing. The independence and parallelism offered by the GPU makes them a good match for ray tracing applications. Each ray-surface intersection is independent of others and the equations can be solved in parallel [35, 48]. For each splat we need to ray trace only a small region on the screen. The computation depends on the total area we are ray tracing and not on the window size. Rendering splats directly can create discontinuities at intersections. The surfaces need to be blended to have a visually smooth approximation (see Figure 4.1). Each splat is ray-traced as described in Section 4.1. A 2-pass rendering algorithm is used to blend close by overlapping surfaces [65]. Octree nodes outside the view frustum are culled away. Our rendering method supports frustum culling, adaptive anti-aliasing and shadows with high speed and quality.

4.1 Rendering a Single Splat

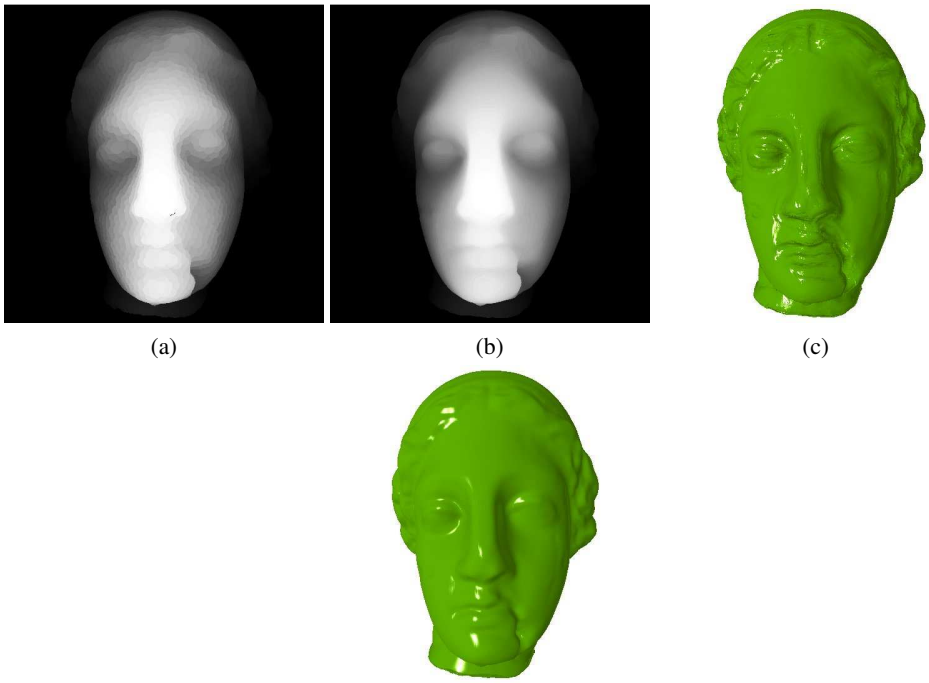
An algebraic splat is ray traced only inside a screen space bounding box. To do so, we display the same region by a rectangle. This generates the fragments necessary for finding ray-surface intersection. The information required to ray-trace i^{th} splat is sent from CPU through the pipeline. Since a splat has its own coordinate system, the camera center and the ray are transformed to local coordinate system for ray tracing.

The equation of a ray is $R_f = c_i + t\vec{d_f}$, where c_i is origin of rays for i^{th} algebraic splat, t is ray parameter and $\vec{d_f}$ is direction of ray for fragment f. Equation 4.1 gives the splat equation in ray parameter space t, $F_f(t)$.

$$F_f(t) = g_i(c_x + td_x, c_y + td_y) - (c_z + td_z) = 0, \tag{4.1}$$

where $F_f(t)$ is a polynomial in t. The smallest positive real root of $F_f(t)$ inside the disc gives the closest point of ray-algebraic splat intersection. u_i, v_i of desired root should satisfy $\|u_i^2 + v_i^2\| \leq R_i^2$. Figure 4.2 shows various types of ray surface intersection possible.

If algebraic splat is at most fourth order, Equation 4.1 can be solved analytically. For quadratic splats, F_f, can be solved trivially. We use the simple and robust method proposed by Blinn to solve



(a) (b) (c)

(d)

Figure 4.1: Figures (4.1a) and (4.1b) show depthmap after first and second passes. Figures (4.1c) and (4.1d) show the rendering after the first and second passes. Zoom in to see details.

cubic splats [7] and the Ferrari method to solve quartic splats [22]. The normal of the surfaces is given by the surface gradient $\vec{\nabla}\{g_i(u,v) - n\}$ at the point of intersection. The normals are transformed to the camera coordinates and are used in lighting calculations. The rendering time of a splat depends on the area of the screen-space bounding box. Table 4.1 shows the time taken for a single splat in some models for quadratic, cubic and quartic.

4.2 Blending Multiple Splats

The final representation consists of an algebraic splat for each point in D, each needing 16-25 floating point numbers depending on the degree. These numbers are packed into a 2D texture and stored on the GPU memory. Each splat occupies a 2D section to take advantage of texture caching. Each splat can be accessed by vertex shader and rendered as described above. The rendering of these splats directly can create discontinuities at intersection. The surfaces need to be blended to have a visually smooth approximation.

Multiple splats are rendered using two passes: the visibility pass and the blending pass. In the first pass, each splat is rendered independently as mentioned in Section 4.1. The closest surface intersections are stored in the depth buffer. The depth values are shifted away from camera by a small factor δ, to

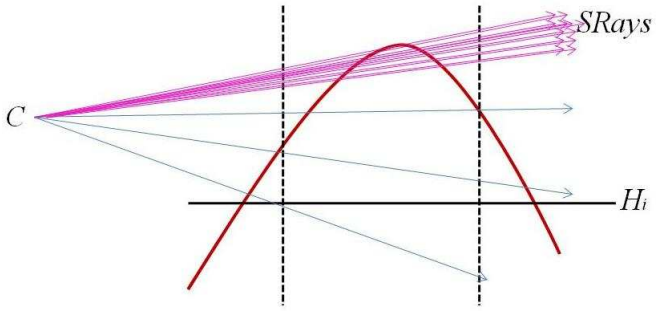


Figure 4.2: Ray surface intersections for an algebraic splat. The pink colour rays denote the silhouette rays for anti-aliasing.

ensure that all surfaces within a small distance of the closest surface will succeed the depth test in the second pass and can be blended [50].

Splat Size	Quadratic (μ sec.)	Cubic (μ sec.)
512×512	0.8	1.2
256×256	0.3	0.7
128×128	0.09	0.015

Table 4.1: Average timing for rendering a single splat for different window sizes on a GTX 280.

In the second pass, the colors and depths of the ray-surface intersections that are δ distance from the closest surface are blended along the ray. Early z-culling will ensure that the farther splats are quickly discarded. The surfaces at each point are blended using Gaussian weights similar to EWA splatting [65, 8]. The output of the color buffer after second pass is $(\sum \alpha_i C_i, \sum \alpha_i)$, where α_i is the weight given to the point and $C_i = (r_i, g_i, b_i)$ is the color after shading. The weight α_i decreases exponentially with the distance from o_i in the local reference domain. The net effect is similar to EWA splatting used with linear splats [65]. Final step normalizes the texture per-pixel and writes it to the color buffer. The color written is $(\sum \alpha_i C_i / \sum \alpha_i, 1.0)$. This ensures that the splat points within the δ-distance of the closest splat are interpolated.

Model	#splats	Quadratic	Cubic
David	29769	165	81
Lucy	9123	211	104
Dragon	17,653	128	73
Bunny	1315	286	146

Table 4.2: Rendering speed in frames per second on some models using different algebraic splats

4.3 Adaptive Anti-aliasing

Ray tracing is a point sampling process; the rays used to assess light intensities are infinitely thin. It treats a scene as a set of unrelated intensity values. Improving the sampling will reduce aliasing effects. Super-sampling involves casting more than one regularly spaced sample per pixel and using the average of the results for the pixel intensity. This method is often slow as the ray tracer needs to render high resolution images. A 2× full screen super sampling decreased rendering performance by 75%.

Super-sampling is needed only for the silhouette pixels in-practice. We perform an adaptive sampling of the silhouette pixels by first computing the derivative of the algebraic splats. The pixel is a silhouette pixel if the magnitude of the derivative is small. If a pixel is marked as silhouette, it is split into a grid of sub-pixels as shown in Figure 4.3. The black spots are the places where the ray intersects the pixel. These grid of rays are used to calculate intensities at those pixel (top image in Figure 4.3). The intensity value of the pixel is the average of all the sub-pixels. Adaptive sampling works well on boundary edges. In-practice the additional cost is significantly less compare to super-sampling the whole scene (see Table 4.5). Probabilistic/stochastic sampling of rays can be used instead of the fixed supersampling with equal ease in our method. Figure 4.4 shows the results of adaptive anti-aliasing on Igea model.

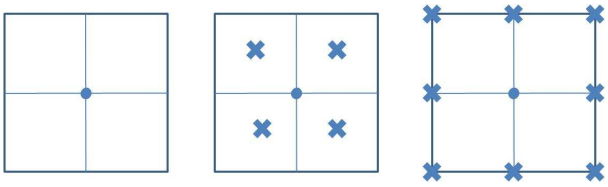


Figure 4.3: Supersamplig silhouette pixels using 2×2 and 3×3 schemes. Left: No supersampling. Middle: 2×2 supersampling. Right:3×3 supersampling. Crosses show the supersampled rays.

4.3.1 View Frustum Culling

View frustum culling is the process of removing objects that lie completely outside the viewing frustum from the rendering process. Culling is key for getting higher rendering speed for closeup views. This makes the rendering faster, if most of the geometry is outside frustum. Splats which are not present in the frustum are culled. Checking the culling condition for each splat is time consuming and it effects rendering speeds. We use octree to optimize this process. A node is checked for culling condition instead of each splat. If a node is outside the view frustum, then all its children will be outside view frustum.

Testing a box (a node) against view-frustum is a little bit trickier and requires more signed distance computations. But testing a sphere requires requires only a distance computation against more complex calculations for the box case. A bounding sphere for each node is tested against viewfurstum for culling. If the sphere is totally inside or outside of the frustum then the box is also totally inside or outside of the frustum respectively. When the sphere intersects the frustum walls test the box itself.

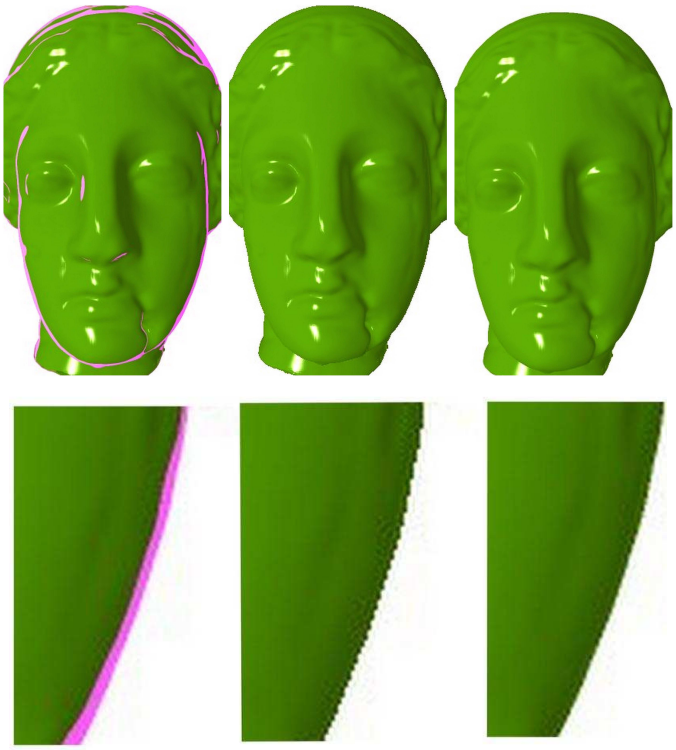


Figure 4.4: TOP: Igea with and without anti-aliasing. Left image shows the silhouette pixels in pink. Middle and right images show normal and anti-aliased views of Igea. BOTTOM: Close up of the Igea's left cheek.

4.3.2 Shadow Mapping

Shadows increases the realism in rendering. Shadow mapping is an image-based shadowing technique and is just one of many different ways of generating shadows. The main advantages of using shadow mapping is it requires no knowledge or processing of the scene geometry, since shadow mapping is an image space technique, working automatically with objects created or altered on the GPU. The trouble with rendering high quality shadows is that they require a visibility test for each light source at each rasterized fragment. The clever insight of shadow mapping is that the depth buffer generated by rendering the scene from the lights point of view is a pre-computed light visibility test over the lights view volume. The lights depth buffer (or the shadow map) partitions the view volume of the light into two regions: the shadowed region and the unshadowed region. The visibility test is of the form

$$p_z \leq shadowMap(p_x, p_y) \tag{4.2}$$

where p is a point in the lights image space.

With shadow-mapping our algorithm need four rendering passes. Two rendering passes from point of view of light for generation of shadowmap and two rendering passes from camera point of view. In generation shadowmap passes (first two passes), no colour computations are done. Only the blended

Model	Quadratic (% compressed)	Cubic (% compressed.)
David	91	93
Igea	95	94
Dragon	90	93

Table 4.3: Reduction in disk space using algebraic splats.

depth buffer of the algebraic splat representation is computed. These two passes are similar to rendering discussed in section 4.2 but from light's perspective without any colour computations. This shadow map essentially a 2D function indicating the depth of the closest pixels to the light. This shadowmap is used for depth comparison in the second of the two rendering passes (fourth pass). If d_{sm} and d_c are depth from shadow map and relative to camera respectively. We will perform the following test at each fragment while rendering from the camera point of view. If $d_{sm} \geq d_c$ then this fragment is blocked by something from the light. So the fragment is shadowed. If $d_{sm} \cong d_c$ then the fragment is lit. The results of shadowmapping on the algebraic splats are shown in figure 4.8.

4.4 Results

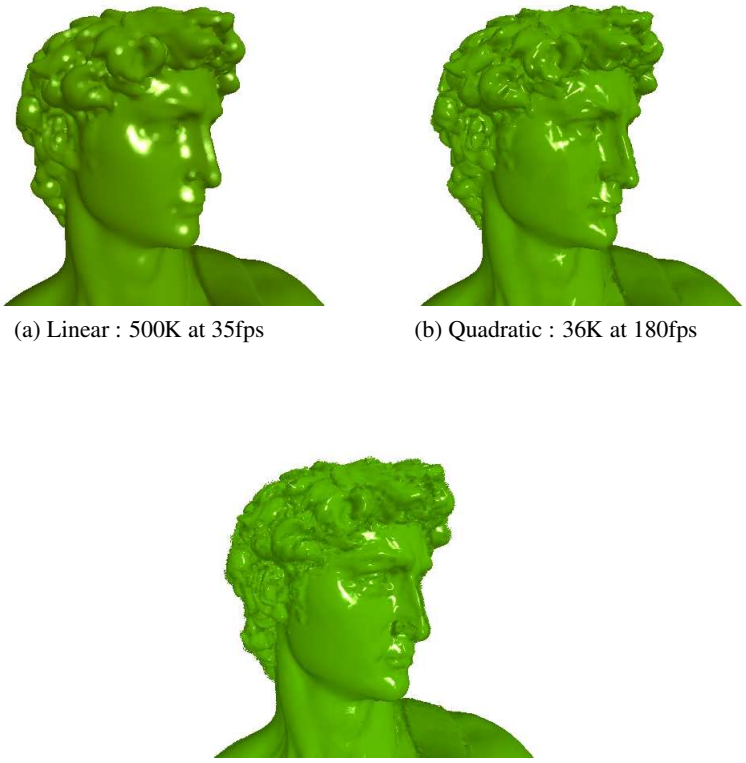
We demonstrate our representation and rendering algorithms on point based models ranging from 32K to 3.6M points. All these models are approximated using algebraic splats of second and third order. The quartic splats were equivalent to the cubic ones in quality (the 4^{th} order coefficients were close to 0) but were much slower to render. We will present the details of advantages algebraic splats has to offer. All fps numbers are taken for 512×512 rendering window using nVidia GTX 280.

In Table 4.7, "# splats" column shows the number of primitives required for algebraic splats representation (at $s = 1$). With algebraic splats, all the models require less that 10% of total number of linear primitives. This reduces the CPU-GPU memory bus activity and the computation required compared to linear splats. Although quadratic and cubic splats store more data per splat compared to linear, the overall disk-space requirement is less compared to linear splats and differential points (See Table 4.3).

As the order of algebraic splats increases, similar quality of rendering is achieved with fewer number of primitives. Number of primitives needed decreases for higher order splats with the quadratic and cubic splats being the fastest for rendering. Figure 4.5 shows the pictures with same visual quality for splats of different order.

In Figure 4.6, 75K surfels are used to represent Igea. Bottom row shows the closeup of eye and we can see the increase in detail. While cubic splats add more detail to the models, they are slower to render compared to quadratic splats.

Our multi-resolution representation is designed to render geometry using continuous LOD. For a moving camera, the number of splats rendered varies from frame to frame without effecting the rendering speed drastically. To demonstrate this, we defined a path around each model to involve the least LOD to best LOD for rendering. The rendered images for David from such a path can be seen in the



(a) Linear : 500K at 35fps

(b) Quadratic : 36K at 180fps

(c) Cubic : 29K at 135 fps

Figure 4.5: Comparison of head of David for different algebraic splat representations with similar visual quality.

accompanying video. The graph (Figure 4.7) shows the rendering speed of David model at selected viewpoints. The rendering speed is almost constant throughout the path. In Figure 4.10 the level of detail rendering on David model at selected points are shown. The number of splats used for rendering increases as the camera move closer to object. The rendering speed is constant because fewer splats are in the frustum at higher resolutions. A shadow mapping technique was used for more realistic rendering with shadows (Figure 4.8). Table 4.4 shows the rendering speed with shadow mapping.

Our adaptive anti-aliasing is fast compared to full screen anti-aliasing (FSAA) offered by hardware. Our method decreases the frame rate about 20-30% on most models, while FSAA results in a 75% of drop for $2\times$ rendering (Table 4.5). In Figure 4.4, silhouette pixels are marked in pink. Results of anti-aliasing then shown on Lucy and David.

Model	No Shadows	Shadows
David	165	89
Lucy	211	98
Dragon	128	64
Bunny	286	159

Table 4.4: Rendering speed with and without shadows, using a shadow mapping technique.

Model	No-AA	2x-FSAA	2×2-AAA	3×3-AAA
David	220	54	165	144
Lucy	290	49	211	169
Dragon	198	46	128	94
Bunny	401	110	286	207

Table 4.5: Rendering speed comparison with and with-out Adaptive Anti-aliasing. FSAA: Full Screen Anti-Aliasing. AAA: Adaptive Anti-Aliasing

4.5 Discussion

The major focus of our thesis is to provide high-order primitives which represents the shape the better. The motivation to use non-linear surface elements is to reduce the number of primitives. This reduces the rendering computation and related memory bus activity. While differential points [25] reduce the disk space by 75%, our representation saves more than 90% diskspace (Table 4.3). The advantages provided by these are in terms of optimal representation, fast rendering.

Table 4.7 shows speed achieved with second and third order algebraic splats on some of the models at $s = 1$. Adamson and Alexa [1] developed an intersection test based on PSS projection procedure but the advantages of this method are spoiled by long execution powers (hours to compute a single image) for small geometry compared to David. Unlike their method [1], our method is fast and renders 5 million algebraic splats per second. In contrast, we didn't have true secondary rays for computing shadows.

Like differential points, algebraic splats require less space. However our method renders true geometry (unlike normal mapped rectangles) and are more robust in handling noise (more useful in case of scanned models). Using DPs[25] increased the speed by two times, but algebraic splats increased the rendering speed by $10\times$.

The quality based decimation gives comparable quality with SLIM but requires less number of primitives. This may be due to inability of SLIM to approximate over large neighborhoods. With algebraic splats, we can represent the point based models with less number of primitives with little or no drop in visual quality. Table a 4.6 gives a comparison of rendering speeds of SLIM and algebraic splats. At a given number of splats, the algebraic splats performs better even on comparable GPUs.

Limitations and Future Work One of the limitation of this method is smoothing of sharp edges. This can be solved using other MLS formulations such as Robust MLS (RLMS)[16]. The quality based

Model	#Splats	Algebraic splats			SLIM[27]
		G1	G2	G3	G3
Lucy	130K	92	49	11	7
Armadillo	24K	168	94	39	23
Dino	6K	308	180	87	44
David	932K	18	10	4	3
David	126K	108	69	14	7

Table 4.6: Comparison of the FPS of algebraic splat rendering on G1 (nVidia GTX280), G2 (nVidia 8800 GTX), G3 (nVidia 7900 GTX) with SLIM surface rendering on G3 for equal number of primitives. The Hole-free representation with $s = 1$ is comparable in quality with SLIM and achieves much better rendering speeds (see Table 4.7).

decimated representation may not give optimal representation or placement of splats. Getting optimal representation is computationally expensive. We can look at GPU based solutions for preprocessing, which make our method online. The emphasis of our representation is high quality representation and fast rendering. The lighter models we use may not be good to approximate the underlying geometry of the object.

Texture mapping may not be trivial in case of points and algebraic splats. Each splat is a implicit surface. Each splat can be texture mapped independently and a blending or interpolation need to be done across splats. Otherway is to adapt a texture mapping to the point cloud obtained at the end of Blending Pass. Texturemapping for this represention is largely an unexplored problem.

Model	Quality based decimation		
	#splats (% of total)	FPS	
		Quadric	Cubic
David	29769 (0.8)	165	95
Angel	7270 (3.0)	285	150
Armadillo	6567 (3.8)	240	133
Bone	5324 (3.9)	194	99
Bunny	1315 (3.6)	190	113
Dino	2292 (4.0)	290	145
Dragon	17,653 (4.0)	130	77
H.Buddha	14,897 (2.7)	165	89
Horse	2983 (6.1)	350	178
Igea	4819 (3.6)	177	92
Lucy	9123 (3.5)	220	113
Santa	3922 (5.1)	311	171
Sphere	112 (3.5)	480	251

Table 4.7: Comparison of rendering speed of models with 2×2 adaptive anti aliasing and viewfrustum culling using algebraic splats for Quality based decimated representation of model on nVidia's GTX280.

(a) Linear

(b) Quadratic

(c) Cubic

(d) Close up of Linear, Quadratic and Cubic

Figure 4.6: Rendering of Igea with 75K splats of different types.

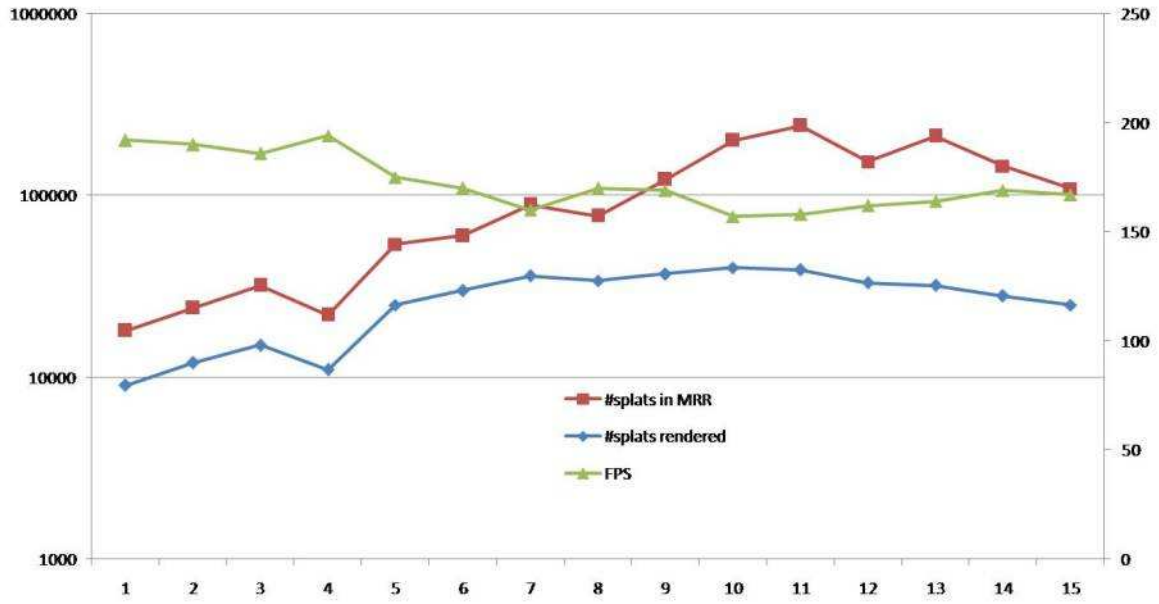


Figure 4.7: Statistics for the David model with 3.6 million points at different points along a pre-defined walk-through path. The red and blue lines show the number of points in the multiresolution representation and the number of rendered points. The points are shown in logscale on the left in thousands. The green like shows the corresponding frame-rates on the right

Figure 4.8: Igea with shadows using a shadow mapping rendered rendered at 95 fps.

Figure 4.9: Lucy and David anti-aliased. The first image in top and bottom rows shows the silhouette pixels in pink. These are the pixels marked for anti-aliasing. The second and third images in both rows show the original and anti-aliased rendering.

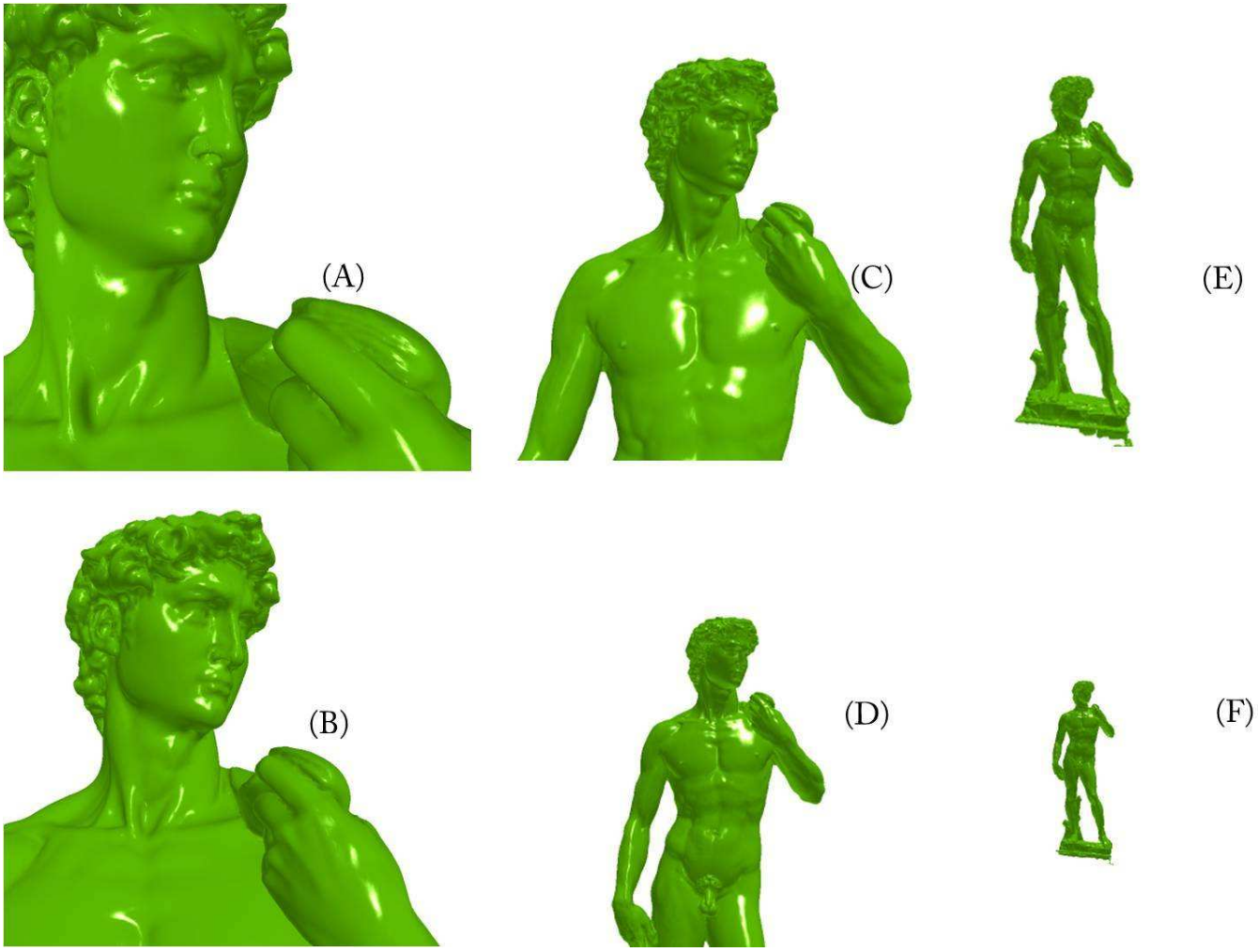


Figure 4.10: Rendering of the David model at 6 LOD levels. The model has 450K, 180K, 121K, 40K, 24K and 14K splats respectively from A to F

Chapter 5

Metaballs

Metaballs, also known as blobby objects, are popular particle-based implicit surface representations. We can think of a metaball as a particle surrounded by a density field, where the density attributed to the particle (its influence) decreases with distance from the article location. A surface is implied by taking an isosurface through this density field - the higher the isosurface value, the nearer it will be to the particle. The powerful aspect of metaballs is the way they can be combined. By simply summing the influences of each metaball on a given point, we can get very smooth blending of the spherical influence fields. They have been used widely to visualize the results of particle based simulations, fluids, and other dynamic objects. Such simulations use a very large number of particles, usually in the order of millions. A fast method to render a large number of metaballs at high quality is thus desirable. The current state of the art can ray cast 128K metaballs at 2-3 fps [17].

5.1 Metaballs

A metaball i is a sphere, with a potential field centered at its center p_i. The density field of i is defined by a density function f_i, which monotonically decreases with the distance r from p_i. f_i has a finite support R_i and reduces to zero beyond it (Figure). The isosurface of the density field is defined using a threshold T.

The isosurface with more than one metaball is defined by the following equation

$$f(x) = \sum_{i=0}^{N} f_i(x) - T = 0, \tag{5.1}$$

for a threshold T. The bounding sphere of a metaball is a sphere centered at p_i with radius R_i. The typical field function used is $f_i(x) = (1 - (\frac{R_i}{r_i})^2)^2,\ \ r_i \leq R_i$ and $f_i(x) = 0$ otherwise. The density function has finite support and allows skipping metaballs that are out of range.

A Metaball is a sphere with a density field. The density field of metaball i is defined by a density function f_i, which monotonically decreases when r, the distance from the center p_i, increases. If f_i has a finite support R_i, f_i equals to zero when r is larger than R_i. With some threshold T, we can get the isosurface of the density field. The iso-surface for N metaballs is derived from the following equation.

$$f(x) = \sum_{i=0}^{N-1} q_i f_i(||x - p_i||) - T = 0, \tag{5.2}$$

where f_i is the density function and q_i is the density coefficient.

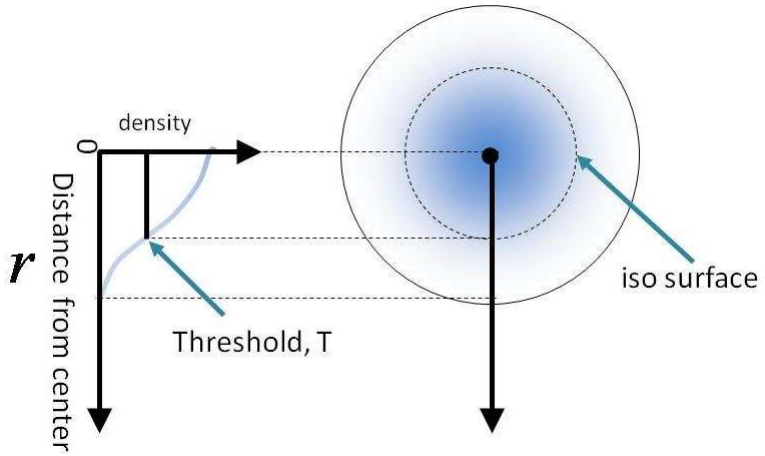


Figure 5.1: A Single Metaball

Every metaball has a density function. A set of them can represent smooth deformable, free-form shapes represented as the iso-surface of a scalar field. The isosurface can be visualized by tessellating the surface using marching cubes algorithm [36] and rendering the resulting polygonal mesh [52]. The quality of resulting mesh depends on the resolution of the grid and can miss objects smaller than the grid resolution. With a high-resolution grid, polygonization can generate high quality surfaces, but at high computation and memory costs.

Another method of visualizing metaballs uses ray-casting [28, 17, 37]. This method renders the surface precisely. However, there are no analytical solutions for ray and iso-surface intersections for most interesting scalar fields. The intersection has to be computed iteratively. The computational cost involved in evaluating the potential function of the surface increases with the number of metaballs.

In this work, we present a method to ray-cast a large number of dynamic metaballs at interactive rates on a commodity GPU. We rebuild the representation in each frame and render the balls interactively. We adapt perspective grids as our acceleration structure due to their fast building and high coherence for ray casting. We modify the adaptive marching points method to iteratively evaluate the implicit iso-surface equation [48]. We achieve interactive frame rates for one million metaballs. Our method handles primary rays at least an order of magnitude faster than the best reported so far. Our method can be used to visualize simulations live as a result. Secondary rays do not achieve high performance, but are less important in live visualization.

Loop and Blinn [35] and Singh and Narayanan [48] rendered very small number of blobbies interactively. Dynamic metaballs necessitate fast structure building, which is difficulty for kd-trees, octrees, and BVH[62]. Uniform grids can be rebuilt fast on the GPUs using efficient data parallel primitives

[39, 26]. Perspective grids provide fast building and high coherence for ray casting [39]. Jagmohan and Narayanan presented the marching points method that uses function evaluation and search to find roots along the ray [48]. This extracts very high performance from today's GPUs and are well suited for metaball rendering.

Adaptive Marching Points Algorithm: The ray-isosurface itersection is computed using a modified adaptive marching points algorithm [48]. The *marching points* algorithm that samples each ray uniformly in t to find solutions of the equation $S(x, y, z) = S(p(t)) = 0$. This marches in equal stepsizes until the root is found which is detected by a sign change in the function. Marching points wastes computation by computing the function values at many points. the *adaptive marching points* that samples each ray non-uniformly based on the distance to the surface and the closeness to a silhouette. Though only fourth or lower order surfaces can be rendered using analytical roots, *adaptive marching points* algorithm can ray-trace arbitrary implicit surfaces exactly, by sampling the ray at selected points till a root is found. Adapting the sampling step size based on a proximity measure and a horizon measure delivers high speed. The horizon measure helps in silhouette adaptation and provides good quality silhouettes. We use adaptive algorithm for each voxel while marching through the perspective datastructure.

5.2 Acceleration Structure

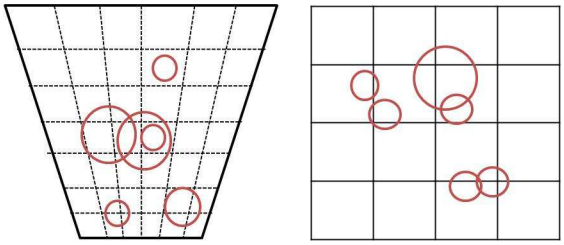


Figure 5.2: Perspective grid. The red-circles represent the projected metaballs. Left: Top view Right: Tile subdivision in image-space.

Ray-casting a surface defined by dynamic metaballs needs an acceleration structure that is fast to build and efficient to traverse. We adapted the perspective grid structure, previously used for dynamic polygonal geometry [39]. We divide the view-frustum into several perspective voxels. The image (or the front plane of the frustum) is divided into equal sized tiles, each of which extends in z to small frustums. These are divided using planes spaced uniformly in the Z direction. Thus, the voxel grid is divided into slabs along Z and tiles along XY directions (Figure 5.2).

Each blob can inserted into one or more voxels of the grid. The influence of a density function is non-zero until the radius R_i. It needs to be inserted into each cell within that region if the cells have to be processed independently. For each voxel, a list of metaballs which may influence density field inside it needs to be known. The data structure is built in two passes. Algorithm 2 gives more details. First, we

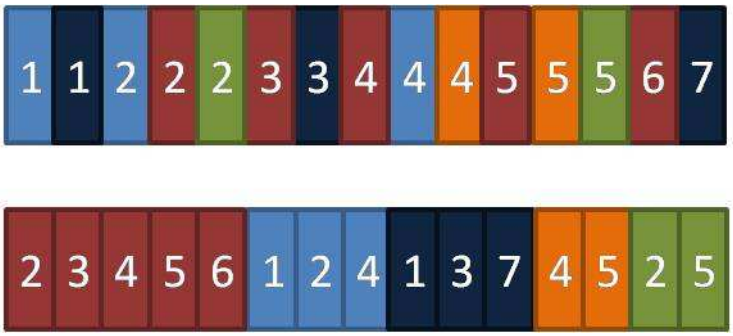


Figure 5.3: Data Structure Representation. Voxels are colour coded. The numbers represent the $ballID$. Top: represents the voxel's intersected each metaball. Bottom: The final data-structure with a list of ball indices for each voxel grouped together.

Algorithm 2 Building Perspective Grid for Metaballs

1: **for** each metaball or blob in parallel **do**
2: Project it to the camera. Keep track of bounds in z and in image space of each.
3: Count the number tiles in XY and number of slabs in Z overlapped by the metaball.
4: The maximum number of voxels overlapped is the product of the two. Record this in a global memory array for each metaball.
5: **end for**
6: Scan this array to get max number of blob-voxel combinations. Each metaball also knows its starting position in the final list of such pairs.
7: **for** each blob in dataset in parallel **do**
8: Project each to the camera
9: **for** each voxel overlapped as calculated before **do**
10: Write *(voxelId, blobId)* to the global memory.
11: **end for**
12: **end for**
13: Sort the *(voxelId, blobId)* list with *voxelID* as the key.

calculate the total number of potential voxel-ball overlaps. This information is used to allocate memory for data structure. The second pass creates the *(voxelId, blobId)* list by going over each metaball. This list is then split with the *voxelID* as the key. We use the SplitSort algorithm for this step as it is scalable in the keysize [39]. This creates a list of blobs that can influence each voxel (Figure 5.3). This structure is built every frame from scratch. The sorting step is the most time-consuming step. The data structure building takes 11ms (on average) for 200K blob primitives.

5.3 Coherent Ray-Casting

Ray casting processes rays of each rectangular tile together. The data structure brings all blobs impacting each cell together. Since tiles and voxels are aligned, the rays can step through voxels that

map to the same tile completely coherently in the front to back order. We map one CUDA thread to a ray and the threads processing a tile to a block. The list of blobs of the cell are loaded into the shared memory of the block and processed together by all threads. The blobs may be loaded in batches if all can't be held in the shared memory.

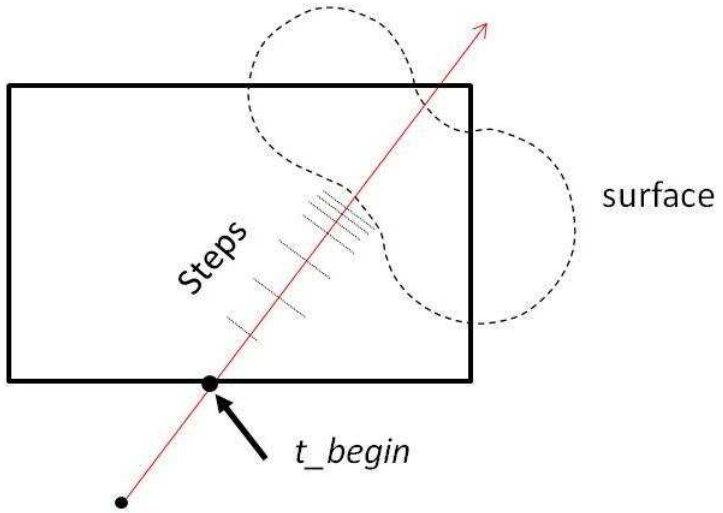


Figure 5.4: Adaptive ray marching algorithm. The stepsize decreases as the ray approaches surface.

Each ray looks for isosurface intersection using a modified adaptive marching points algorithm [48]. For each voxel in the tile, the entering and exiting t values are computed for each ray. The empty voxels are skipped without checking for isosurface. We march along the ray, evaluating the $f(x)$ given by Equation 5.1 at discrete steps, looking for a change in sign. The step size is adjusted so as to take bigger steps in free space, based on the $f(x)$ value. When the ray is close to a silhouette, the gradient of f will be small. We take smaller steps based when the gradient is small. The interval is then bisected a fixed number of times using the Newton's bisection method to reach close to the root. We find that 10 bisections produce good results in practice. The intersection point and the normal are stored along with the voxelId. This information is later used for shading. Algorithm 3 lists the process briefly.

Secondary rays can be generated from the intersection points and the process can be repeated. Secondary rays are tricky to process on GPU as they are not coherent. We introduce coherence by bringing all rays that are passing trough a voxel together and process them. For each ray, we generate the cells it passes through using a 3D DDA algorithm, creating a list of *(voxelId, rayId)*. This list is sorted with the *voxelID* as the key to gather all rays that are passing through a voxel. All voxels that are included in the list are then processed in parallel on the GPU. The closest point of intersection for each ray is kept track of for shading. For shadow rays, it sufficient to know if any intersection exists. Secondary rays are important to generate good stills and are relatively unimportant for visualizing dynamic, simulation results.

Algorithm 3 Coherent Ray-Casting

```
1: for each ray in parallel do
2:   for each voxel starting withe front one do
3:     if voxel is empty then
4:       Skip voxel
5:     end if
6:     Initialize t_enter and t_exit for the ray and voxel. Set step size δ for ray marching to base_step.
7:     Load blobs of the voxel into shared memory
8:     Synchronize threads of the block
9:     March along the ray from t_enter. Evaluate f(x) at t and t + δ, where δ is step-size.
10:    if f(x) changes sign then
11:      Bisect the interval 10 times to get exact root.
12:      Store the point of intersection, the normal, and the voxelId for shading.
13:      Mark the ray as having done.
14:    else
15:      Adjust step size δ based on proximity and derivative [48].
16:    end if
17:  end for
18: end for
```

5.4 Results

We tested our algorithms on dynamic metaball datasets ranging from 10K to 1M million blobs, on an Nvidia GTX280. All results are given for a 1024×1024 window. Table 5.4 shows the average time taken to build the perspective grids and to render each frame for different models.

In [17], the fitted BVH structure was built on the CPU which took about 260ms for 128K primitives. For an equivalent number of metaballs, our data structure building time is less than 8 ms. The average build time for 1 million primitives is about 38 milliseconds, making our method suitable for live visualization of particle based simulations. The rendering time for 128K primitives is 400 ms per frame for [17] compared to 35 ms for our method. We can render a million metaballs in about 94 milliseconds. Table 5.4 gives more detailed run times for our method. Performance of Secondary rays is not fast due to the lack of coherence, with a rendering time of 150 ms for 1K primitives.

The uniform grid in perspective space is brings coherence to primary rays and is very fast for them. The performance deteriorates if the density of primitives varies greatly across cells due to some rays or CUDA threads having to handle a large number of primitives. This can be alleviated by balancing the load better across blocks, but at some drop in performance. The performance of our algorithm does depend on the step size chosen for marching the points. In practice, we find that a base step size of 10% of the minimum R_i radius in a cell gives accuracy as well as speed. We also reduce the step size by a factor of 2 or 4 based on proximity to the surface or a silhouette as explained earlier.

Dataset	Size	DS Build (ms)	Raycast Time (ms)
Scattered	150K	8ms	40ms
Scattered	1M	35ms	94ms
Tornado	450K	23ms	86ms

Table 5.1: Average time taken for data-structure building and rendering metaballs.

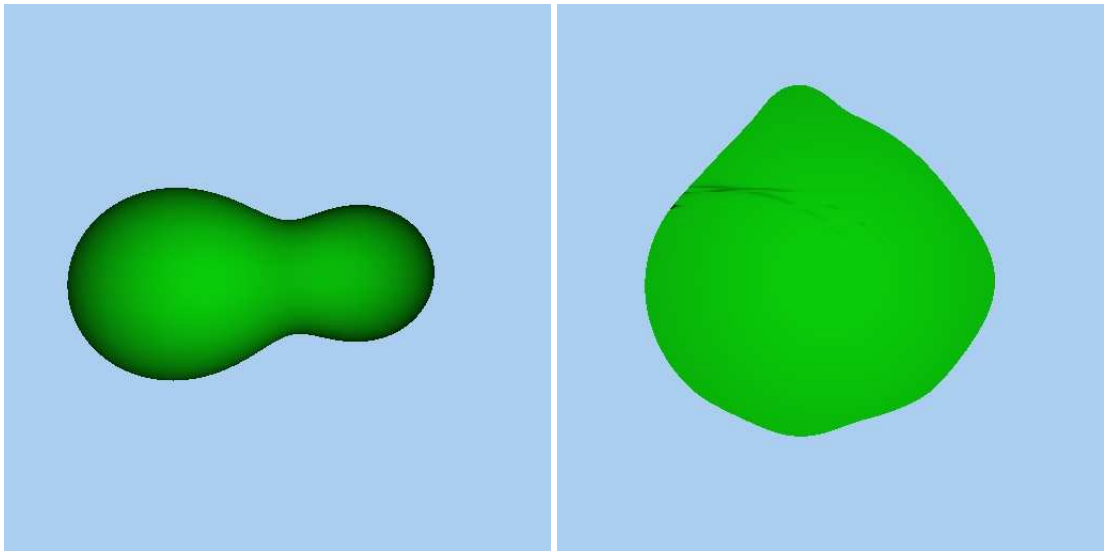


Figure 5.5: Raycasing metaballs. Left: The iso-surface defined by 2 metaballs. Right:Surface defined by a very dense set of metaballs (100K).

Our method does not work well, when the density of metaballs is too high. The number of primitives involved in computation of ray-surface intersection is more resulting in the slow render speeds. If the stepsize if not selected properly in ray marching step, it may miss the iso-surface.

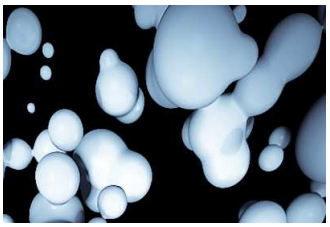


Figure 5.6: Rendering of secondary rays with 1K metaballs at 150ms per frame.

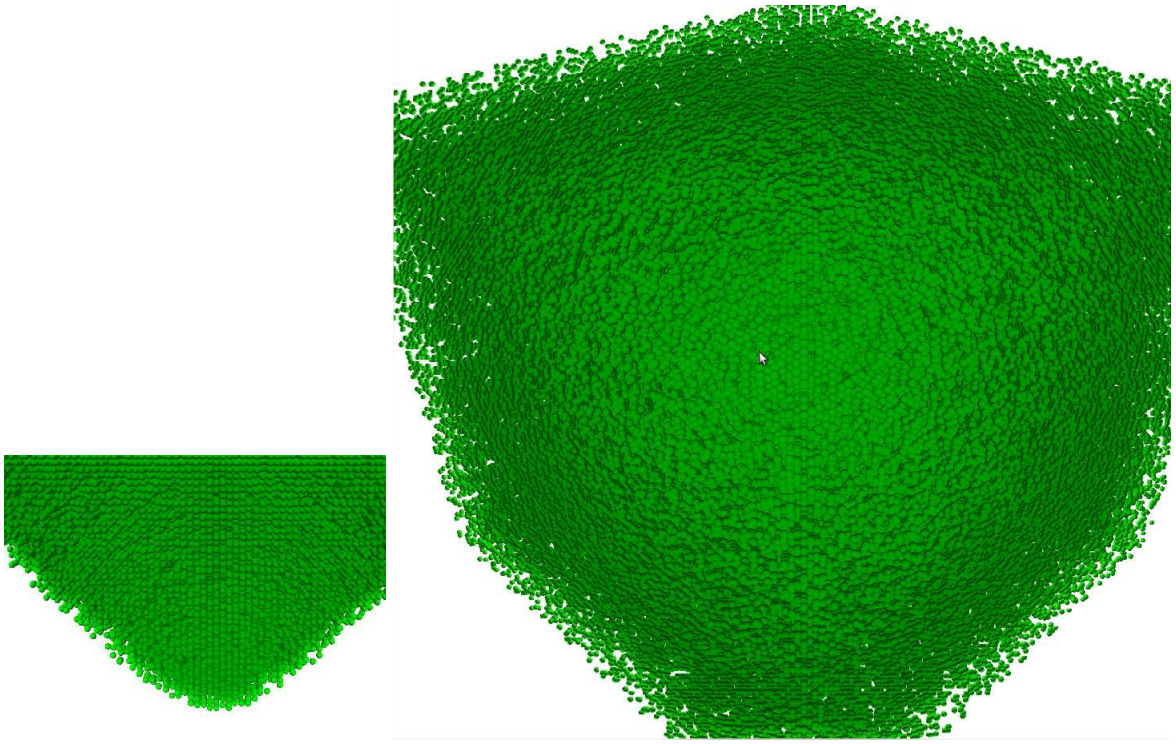


Figure 5.7: Rendering of one million metaballs. Top: Tornadeo model using scattered metaballs. Bottom: iso-surface defined my one million metaballs.

Chapter 6

Conclusions

In this thesis, we explored a new high order primitive for representing geometry. In the first part of this thesis, we presented a compact representation for point based models using non-linear surface elements. In the second part, we presented a method for fast ray casting of a dynamic surface defined by large number of attributed points called Metaballs.

We introduced non-linear rendering primitives for point based representation called algebraic splats, which have the potential to represent large point-based models using few primitives with little or no drop in visual quality. Using this representation large models can be rendered at interactive speeds on commodity GPUs. High quality rendering of complex models with anti-aliasing, continuous LOD and shadows at interactive rates make this representation a good fit for ray tracing based games and designing miniature models for mobile gaming.

Quadratic and cubic splats as basic rendering primitive are able to approximate point-based models using 10% or fewer points than linear splats, for equivalent visual quality. Though rendering a non-linear patch is slower compared to a linear splat, the overall speed of rendering of an object is faster. The approximation error can also be less using a small number of higher order primitives. We represent a given point set with a user-specified small number of algebraic splats with optimal rendering quality. This is done by decimating the point set and jointly approximating each using a local algebraic surface based on the Moving Least Squares (MLS) approximation.

We presented a method to ray trace a million metaballs at interactive rates on a commodity GPU. We use a persecutive grid structure to exploit maximum coherence and a marching points approach for fast root finding on the GPU. The simplicity of the algorithm makes it easy to implement and fast. The secondary rays can also be handled by this method, but is not fast due to the lack of coherence. Our method brings real-time visualization to particle based simulations involving a million or more particles today. Our approach will gain directly from any improvement in the base performance of future GPUs. It may thus be possible to handle an order of magnitude more complex models on the GPUs that are just being released. In future work, one can improve the performance of the seondary rays.

Related Publications

1. Naveen Kumar Bolla, P. J. Narayanan, **Algebraic Splat Representation For Point Based Models** (Conference Paper), *IEEE Sixth Indian Conference on Computer Vision, Graphics and Image Processing (ICVGIP 2008)*,71 − 78, 16-19 Dec,2008, Bhubaneswar, India.

2. Naveen Kumar Bolla, P. J. Narayanan, **Algebraic Splats for High-Quality Rendering of Points** (Journal Submission) *(Submitted to Graphical Models, Under Review)*

3. Naveen Kumar Bolla, P. J. Narayanan, **Real-time Ray-tracing of Million Metaballs**

Bibliography

[1] A. Adamson and M. Alexa. Approximating and intersecting surfaces from points. In *SGP '03: Proc. of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 230–239, 2003.

[2] A. Adamson and M. Alexa. Ray tracing point set surfaces. In *SMI '03: Proceedings of the Shape Modeling International 2003*, pages 272–279, 2003.

[3] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. In *Proceedings of the IEEE Conference on Visualization VIS*, pages 21–28, 2001.

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

[5] N. Amenta and Y. J. Kil. Defining point-set surfaces. *ACM Trans. Graph.*, 23(3):264–270, 2004.

[6] A. Appel. Some techniques for shading machine renderings of solids. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45, New York, NY, USA, 1968. ACM.

[7] J. F. Blinn. How to solve a cubic equation, part 5: Back to numerics. *IEEE Computer Graphics and Applications*, 27(3):78–89, 2007.

[8] N. K. Bolla and P. J. Narayanan. Algebraic splats representation for point based models. In *ICVGIP*, pages 71–78, 2008.

[9] L. Carpenter. The A-buffer, an antialiased hidden surface method. pages 13–18, 1988.

[10] N. A. Carr, J. D. Hall, and J. C. Hart. The ray engine. In *In Proceedings of Graphics hardware*, 2002.

[11] N. A. Carr, J. Hoberock, K. Crane, and J. C. Hart. Fast gpu ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface*, 2006.

[12] Z.-Q. Cheng, Y.-Z. Wang, B. Li, K. Xu, G. Dang, and S.-Y. Jin. A survey of methods for moving least squares surfaces. In *Symposium on Point-Based Graphics*, 2008.

[13] H. L. Cook, L. Carpenter, and E. Catmull. The reyes image rendering architecture. pages 28–35, 1988.

[14] E. de Groot and B. Wyvill. rayskip: faster ray tracing of implicit surface animations. *GRAPHITE*, 2005.

[15] S. Fleishman, D. Cohen-Or, M. Alexa, and C. T. Silva. Progressive point set surfaces. *ACM Transactions on Graphics*, 22(4):997–1011, 2003.

[16] S. Fleishman, D. Cohen-Or, and C. T. Silva. Robust moving least-squares fitting with sharp features. *ACM Trans. Graph.*, 24(3):544–552, 2005.

[17] O. Gourmel, A. Pajot, M. Paulin, L. Barthe, and P. Poulin. Fitted bvh for fast raytracing of metaballs. *Computer Graphics Forum*, 29(2):281–288, may 2010.

[18] J. P. Grossman and W. J. Dally. Point sample rendering. In *In Rendering Techniques 98*, pages 181–192. Springer, 1998.

[19] X. Gu, S. J. Gortler, and H. Hoppe. Geometry images. *ACM Trans. Graph.*, 21(3), 2002.

[20] G. Guennebaud and M. H. Gross. Algebraic point set surfaces. *ACM Trans. Graph*, 26(3):23, 2007.

[21] P. S. Heckbert and P. Hanrahan. Beam tracing polygonal objects. In *Proceedings of the conference on Computer graphics and interactive techniques*, 1984.

[22] D. Herbison-Evans. Solving quartics and cubics for graphics. In *Graphics Gems V*, pages 3–15, 1995.

[23] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree gpu raytracing. In *In Proceedings of I3D 2007*, 2007.

[24] K. Iwasaki, Y. Dobashi, F. Yoshimoto, and T. Nishita. Real-time rendering of point based water surfaces. In *Computer Graphics International*, pages 102–114, 2006.

[25] A. Kalaiah and A. Varshney. Modeling and rendering of points with local geometry. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):30–42, 2003.

[26] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the Conference on High Performance Graphics*, pages 23–28. ACM, 2009.

[27] T. Kanai, Y. Ohtake, H. Kawata, and K. Kase. Gpu-based rendering of sparse low-degree implicit surfaces. In *GRAPHITE*, pages 165–171, 2006.

[28] Y. Kanamori, Z. Szego, and T. Nishita. GPU-based fast ray casting for a large number of metaballs. *Computer Graphics Forum (Proc. of Eurographics 2008)*, 27(3):351–360, 2008.

[29] A. Knoll, Y. Hijazi, C. Hansen, I. Wald, and H. Hagen. Interactive ray tracing of arbitrary implicits with simd interval arithmetic. *Interactive Ray Tracing, 2007. RT '07*, pages 11–18, Sept. 2007.

[30] D. Levin. Mesh independent surface interpolation. *Geometric Modelling for Scientific Visualization*, 2003.

[31] M. Levoy and T. Whitted. The use of points as a display primitive. *Technical Report 85-022, University of North Carolina at Chapel Hill, January, 1985.*

[32] L. Linsen. Point cloud representation. Technical report, Fakultt fr Informatik, Universitt Karlsruhe, 2001.

[33] L. Linsen, K. Muller, and P. Rosenthal. Splat-based ray tracing of point clouds. In *Journal of WSCG*, volume 15, 2008.

[34] Y. Lipman, D. Cohen-Or, D. Levin, and H. Tal-Ezer. Parameterization-free projection for geometry reconstruction. *ACM Trans. Graph*, 26(3):22, 2007.

[35] C. Loop and J. Blinn. Real-time GPU rendering of piecewise algebraic surfaces. *ACM Trans. Graph.*, 25(3):664–670, 2006.

[36] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4), 1987.

[37] T. Nishita and E. Nakamae. A method for displaying metaballs by using bzier clipping. *Computer Graphics Forum*, 13:271–280, 1994.

[38] Y. Ohtake, A. G. Belyaev, M. Alexa, G. Turk, and H.-P. Seidel. Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3):463–470, 2003.

[39] S. Patidar and P. J. Narayanan. Ray casting deformable models on the gpu. In *Sixth Indian Conference on Computer Vision, Graphics & Image Processing, ICVGIP 2008*, pages 481–488, 2008.

[40] M. Pauly and M. Gross. Spectral processing of point-sampled geometry. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 379–386, 2001.

[41] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: surface elements as rendering primitives. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342. ACM Press/Addison-Wesley Publishing Co., 2000.

[42] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, 2002.

[43] S. M. Ranta, J. M. Singh, and P. J. Narayanan. GPU objects. In *Proc. of Indian Conference on Computer Vision, Graphics and Image Processing, 2006*, volume 4338 of *LNCS*, pages 352–363, 2006.

[44] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3), 2005.

[45] S. M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 110–116, New York, NY, USA, 1980. ACM.

[46] S. Rusinkiewicz and M. Levoy. Streaming QSplat: A viewer for networked visualization of large, dense models. In *Symposium for Interactive 3D Graphics*, 2001.

[47] C. Sigg, T. Weyrich, M. Botsch, and M. Gross. GPU-based ray-casting of quadratic surfaces. In *Symposium on Point-Based Graphics*, pages 59–65. Eurographics Association, 2006.

[48] J. M. Singh and P. J. Narayanan. Real-time ray tracing of implicit surfaces on the gpu. *IEEE Trans. Vis. Comput. Graph.*, 16(2):261–272, 2010.

[49] M. Stamminger and G. Drettakis. Interactive sampling and rendering for complex and procedural geometry. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 151–162. Springer-Verlag, 2001.

[50] C. Stoll, S. Gumhold, and H.-P. Seidel. Incremental raycasting of piecewise quadratic surfaces on the gpu. *IEEE Symposium on Interactive Ray Tracing 2006*, pages 141–150, 2006.

[51] R. Szeliski, R. Szeliski, D. Tonnesen, and D. Tonnesen. Surface modeling with oriented particle systems. In *Computer Graphics*, pages 185–194, 1991.

[52] Y. Uralsky. Practical metaballs and implicit surfaces. *Game Developers Conference*, 2006.

[53] K. van Kooten, G. van den Bergen, and A. Telea. Point-based visualization of metaballs on a gpu. In *GPU Gems 3*, 2007.

[54] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1), 2007.

[55] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph.*, 25(3), 2006.

[56] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics*, 2007.

[57] I. Wald and H.-P. Seidel. Interactive ray tracing of point-based models. In *Proc. of the Eurographics Symposium on Point-Based Graphics*, pages 9–16, 2005.

[58] M. Wand, M. Fischer, I. Peter, F. Meyer auf der Heide, and W. Strasser. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 361–370. ACM, 2001.

[59] T. Weyrich, S. Heinzle, T. Aila, D. B. Fasnacht, S. Oetiker, M. Botsch, C. Flaig, S. Mall, K. Rohrer, N. Felber, H. Kaeslin, and M. Gross. A hardware architecture for surface splatting. *ACM Trans. Graph.*, 26(3):90, 2007.

[60] A. Witkin and P. S. Heckbert. Using particles to sample and control implicit surfaces. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 269–277. ACM, 1994.

[61] S.-E. Yoon, C. Lauterbach, and D. Manocha. R-LODs: Fast LOD-based ray tracing of massive models. *Vis. Comput.*, 22(9), 2006.

[62] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5), 2008.

[63] M. Zwicker, M. Gross, and H. Pfister. A survey and classification of real time rendering methods. *Technical Report 332, Computer Science Department, ETH Zurich, 1999. 2,3*.

[64] M. Zwicker, M. Pauly, O. Knoll, and M. Gross. Pointshop 3D: An interactive system for point-based surface editing. *ACM Trans. Graph.*, 21(3):322–329, 2002.

[65] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, 2001.