Diss. ETH No. 19248

Hardware Architectures for Point-Based Graphics

A dissertation submitted to
ETH Zurich

for the Degree of
Doctor of Sciences

presented by
Simon Heinzle
Dipl. Informatik-Ing., ETH Zurich, Switzerland
born 12 October 1981
citizen of Austria

accepted on the recommendation of
Prof. Dr. Markus Gross, examiner
Prof. Dr. Andreas Peter Burg, co-examiner
Prof. Dr. Philipp Slusallek, co-examiner

2010

Abstract

Point-based geometries have emerged as interesting and valuable alternative to triangles and polygons. Points as graphics primitives are conceptually more simple and provide superior flexibility, especially for objects with high geometric and appearance detail. However, triangles are still the most prominent primitive in computer graphics. One reason is the limited support for point-based graphics in modern graphics accelerators, which have been optimized for triangles as rendering primitives.

In this thesis, we analyze the fundamental differences between triangle-based graphics and point-based graphics for hardware acceleration and identify the limitations and bottlenecks of current graphics processors (GPUs). We then develop new algorithms and hardware architectures for point-based graphics to augment and extend todays graphics processors.

More specifically, we develop a novel point rendering algorithm based on elliptical weighted average (EWA) splatting. We extend the theoretical basis of EWA splatting into the time dimension to incorporate motion-blur for point-based rendering. We present a GPU implementation and analyze the differences and bottlenecks of current hardware architectures. Based on the previous analysis, we present a streamlined EWA splatting algorithm for static scenes and develop a novel hardware architecture for the acceleration of point rendering. The novel architecture fits seamlessly into current, triangle based rendering architectures and is designed to complement triangle rendering. Striving towards a more general point acceleration architecture, we then develop a hardware architecture dedicated to the general and efficient processing of point-sampled geometry. It focuses on the fundamental and computationally most expensive operations on point sets and supports a wide range of point graphics algorithms as well as other graphics algorithms.

We present hardware implementations for both architectures and provide a detailed analysis of hardware costs, performances, and bottlenecks, and we compare the results to the respective GPU implementations. The developed architectures could be integrated into existing triangle-based architectures to complement triangles at modest additional hardware cost in order to accelerate point-graphics efficiently on today's GPUs.

Zusammenfassung

Punktbasierte Geometrie hat sich als interessante und wertvolle Alternative zu Dreiecken und Polygonen erwiesen. Punkte als Grafikprimitive sind konzeptionell einfacher und auch wesentlich flexibler, vor allem für Objekte mit detaillierter Geometrie. Trotzdem sind Dreiecke immer noch die am meisten verbreiteten Primitive in der Computergrafik. Ein Grund dafür ist die limitierte Unterstützung von punktbasierter Grafik in modernen Grafikbeschleunigern, die für Dreiecke als Renderingprimitive optimiert wurden.

Diese Dissertation untersucht die fundamentalen Unterschiede zwischen dreiecks- und punktbasierter Grafik im Bezug auf Hardwarebeschleunigung. In einem ersten Schritt werden die Einschränkungen und Engpässe bestehender Grafikprozessoren (GPU) analysiert. Basierend auf dieser Analyse werden dann neue Algorithmen und Hardware-Architekturen für punktbasierte Grafik vorgestellt, mit denen bestehende Grafikprozessoren ergänzt und erweitert werden können.

Im Detail wird ein neuartiger Algorithmus zum Zeichnen von Punkten basierend auf Elliptical Weighted Average (EWA) Splatting entwickelt. Die theoretische Grundlage von EWA Splatting wird in die zeitliche Dimension erweitert um sogenannte Bewegungsunschärfe zu modellieren. Der Algorithmus wird in auf einem Grafikprozessor implementiert um die Unterschiede und Engpässe bestehender Grafikarchitekturen besser zu verstehen. Basierend auf dieser Analyse wird eine neuartige Hardwarearchitektur vorgestellt die für EWA Splatting von statischen Objekten optimiert ist. Die neuartige Architektur fügt sich nahtlos in bestehende Grafikprozessoren ein die für Dreiecke optimiert sind. Im nächsten Schritt wird eine neuartige Architektur für die effiziente und verallgemeinerte Verarbeitung von Punktgeometrie eingeführt, mit Fokus auf den grundlegenden und rechnerisch teuersten Operationen auf Punktmengen. Diese neue Architektur unterstützt dann eine breite Palette von Punktgrafikalgorithmen sowie allgemeingültigere Grafikalgorithmen.

Beide Architekturen wurden mittels Hardwareimplementierungen verifiziert und bezüglich Hardwarekosten, Performance und Engpässen analysiert. Die entwickelten Architekturen können in bestehende Hardwarearchitekturen für Dreiecke integriert werden um diese mit einer effizienten Unterstützung für Punkte zu ergänzen.

Acknowledgments

My sincere thanks go to my advisor Prof. Markus Gross. His enthusiasm for point-based graphics during my undergraduate lectures motivated me to work towards the first dedicated hardware architectures for points. His continuing support for my work, his expert advice, and his outstanding scientific intuition finally made it possible to achieve my eager goals. Furthermore, I would like to thank Prof. Philipp Slusallek and Prof. Andreas Peter Burg for their ongoing interest in my work, the insightful discussions, and for agreeing to co-examine this thesis.

I am deeply thankful to Prof. Tim Weyrich who taught me everything about software engineering, scientific writing, and how to conduct computer graphics research. My further thanks go to Stephan Oetiker – he taught me how to design and implement such big VLSI systems. Without the help of those two mentors, I would have had a much harder time in the past years.

Many thanks go to all my additional collaborators that contributed in various ways to this thesis: Timo Aila, Sebastian Axmann, Mario Botsch, Diego Browarnik, Flavio Carbognani, Daniel B. Fasnacht, Norbert Felber, Cyril Flaig, Gaël Guennebaud, Hubert Kaeslin, Yoshihiro Kanamori, Peter Luethi, Sebastian Mall, Tomoyuki Nishita, Kaspar Rohrer, Olivier Saurer, Andreas Schmidt, and Johanna Wolf.

Special thanks go to all the past and present members of the Computer Graphics Laboratory, the Applied Geometry Group, and the Scientific Visualization Group at ETH as well as to the folks at Disney Research for the work- and non-work related discussion throughout the years, and more importantly, for being great friends and colleagues.

I would like to thank my friends and family, especially my parents which supported me in my education – without them I would not be where I am right now. Last but not least, I would like to thank Julie for her love, understanding, and her indispensable support during the stressful deadline times.

Contents

CHAPTER 1

Introduction

Triangles and polygons have been the dominant primitive in computer graphics for the last four decades. Many applications such as numerical simulations, surface scanning, or procedural modeling generate millions of tiny triangle primitives – a trend that has been even more accelerated by the rapid evolution of computing machinery. Due to this explosion of the geometric complexity, however, polygonal meshes become difficult to maintain, and constitute a significant overhead in terms of storage demand and computational complexity for maintaining its connectivity. Furthermore, the rendering of tiny triangles can become inefficient, and can even lead to visual errors – the so called aliasing artifacts.

Point-based graphics has evolved into an interesting and valuable alternative due to the conceptual simplicity and superior flexibility of points as graphics primitives. Levoy and Whitted first suggested points [LW85] as representation for objects with high geometric and appearance detail. Since then, researchers have developed a variety of powerful algorithms and pipelines for the efficient representation, processing, manipulation and rendering of point-sampled geometry during the last decade [GP07]. However, triangles are still the most prominent graphics primitive, mainly due to the excellent support for polygonal meshes on current graphics processors. These processors have become ubiquitous in today's computers, but their support of rendering and processing of unstructured points is still only limited.

Motivated by the limited support for point-based graphics in today's graphics processors, this thesis investigates the following two research questions: What are the fundamental differences between points and triangles with respect to hardware architectures? And how can existing architectures be improved to offer dedicated support for point primitives in an efficient way? To understand the problems in detail, this thesis analyzes and develops novel hardware architectures for the rendering and geometry processing of point sets.

The following two sections will present a more detailed overview on point rendering in Section 1.1 and point processing in Section 1.2. The main contributions of this thesis are given in Section 1.3 before an outline of the rest of the thesis is given in Section 1.4.

1.1 Rendering point sampled surfaces

Point rendering is particularly interesting for highly complex models whose triangle representations require millions of tiny primitives which in turn then project to only a few pixels. Well-established among point rendering methods is the technique of elliptical weighted average (EWA) surface splatting [ZPBG01]. EWA splatting allows to render high-quality anti-aliased images of geometric objects that are given by a sufficiently dense sets of sample points. The idea is to approximate local regions of the surface by planar elliptical Gaussian reconstruction kernels in object space – the so-called surface splats. The final surface is then rendered by blending these splats in screen space. Before the final blending step, the splats are combined with a low-pass filter to avoid aliasing in the case of minification. This aliasing sampling artifact can be especially problematic in the case of rendering micropolygons. A detailed introduction into EWA surface splatting is given in Chapter 3.

In its original formulation, the EWA surface splatting framework does not contain a notion of time and generates still frames depicting a perfect instant in time. It therefore lacks realism due to the absence of motion blur. This thesis analyzes the original framework, and extends EWA surface splatting into the time domain to directly and efficiently support motion blur for a new sensation of dynamics in Chapter 4.

Although surface splatting can be implemented on state-of-the-art programmable GPUs, these implementations usually require multiple passes. The main reason is that GPUs are optimized and designed for polygonal

rendering and therefore fundamentally different to point rendering. Unfortunately, the differences manifest themselves in a performance gap of an order of magnitude compared to triangle rendering. This thesis presents a hardware architecture that extends traditional polygon-based architectures to support surface splatting more efficiently in Chapter 5, and then shows that point-based primitives are amenable to efficient hardware implementations.

1.2 Processing of unstructured point sets

A significant amount of research has been devoted to understand meshless surface representations better. It turns out that many point processing methods can be similarly decomposed into two distinct computational steps: the first step constitutes the computation of a neighborhood around a given spatial position, whereas the second step is usually an operator or computational procedure that processes this neighborhood. Examples for fundamental point-graphics operators are weighted averages or covariance analysis, examples for higher-level operators include normal estimation or moving least squares (MLS) approximations [ABCO+01]. Very often, the spatial queries to collect adjacent points constitute the computationally most expensive part of the processing.

However, efficient hardware acceleration for both computational steps – the computation of the neighborhood as well as the stream processing of that neighborhood – is virtually unavailable, as no architecture is able to support both computations simultaneously very well. The main reason for this is the fundamental difference in these two algorithms. While the single instruction, multiple data (SIMD) paradigm of current graphics processing units (GPUs) is very well suited to efficiently implement most stream operators on a neighborhood of points, a variety of limitations including the SIMD processing pattern leave GPUs less suited for efficient neighborhood queries. Conversely, general-purpose processors (such as central processing units, CPUs) feature a relatively small number of floating point units and therefore are less suited for stream operators, but perform better in the recursive traversals of the spatial search.

We investigate a more general hardware architecture for point processing and rendering in Chapter 6. The new architecture alleviates most of the aforementioned problems, while still providing lightweight hardware architecture.

1.3 Principal contributions

In the scope of this thesis we developed new algorithms and novel architectures for point based graphics. Our three main contributions are:

1. **Extension of the theoretical basis of the EWA splatting framework in the time dimension to incorporate motion-blur for point-based rendering.** The conceptual elegance of the approach lies in replacing the 2D Gaussian kernels by 3D Gaussian kernels which unify both the spatial and temporal component. The derived result naturally fits into the EWA splatting algorithm such that the final image can be computed as a weighted sum of warped and bandlimited kernels. Its rendering algorithm shows strong parallels to the original EWA rendering. In addition to the correct rendering approach, this thesis provides an approximative implementation by the description of an entire point rendering pipeline using vertex, geometry and fragment program capabilities of current GPUs. The results of this research are presented in Chapter 4.

2. **A hardware architecture for accelerated rendering of point primitives** using an optimized and streamlined version of EWA surface splatting, based on our analysis and extensions of the original EWA framework. A central feature of the design is the seamless integration of the architecture into a conventional, OpenGL-like graphics pipeline to complement triangle rendering. Some of the novel design concepts include a ternary depth test and the usage of an on-chip pipelined heap data structure for making the memory accesses more coherent. Furthermore, we developed a computationally stable evaluation scheme for perspectively correct splats. As a proof of concept, different versions of the pipeline have been implemented both on reconfigurable FPGA boards and as ASIC prototypes, and have been integrated into an OpenGL-like software implementation. The results of this research are presented in Chapter 5.

3. **A hardware architecture dedicated to the general and efficient processing of point-sampled geometry.** The new architecture was developed as a more general point graphics pipeline and focuses on the fundamental and computationally most expensive operations on point sets. More specifically, the architecture supports neighborhood searches as well as stream algorithms such as moving least squares approximations. It comprises of a configurable *k*d-tree based neighbor search module and a programmable processing module. The spatial search module supports *k*-nearest neighbor queries and range

queries, and it features a novel caching mechanism to exploit the spatial coherence inherent in a variety of point processing algorithms. The architecture was implemented as an FPGA prototype and proves to be lean and lightweight. Therefore, it could be integrated with existing hardware platforms, or combined with a rendering architecture for EWA surface splatting to provide a general and versatile point-processing and rendering platform. The results of this research are presented in Chapter 6.

1.4 Thesis outline

The thesis is organized as follows. Chapter 2 revises previous work in the fields of point-based surface definitions, rendering of point sets and hardware architectures for computer graphics. Work directly related to more specific concepts will be presented in their respective chapters. Chapter 3 will review the concept of EWA surface splatting which is essential to understand the rest of the thesis. Chapter 4 introduces the extension of the EWA framework in the time dimension for motion blur. Chapter 5 then presents a hardware architecture for EWA surface splatting that could be integrated into traditional triangle rendering pipelines. Chapter 6 will then present our hardware architecture for general processing of point sets. Finally, the conclusion of the this thesis is given in Chapter 7.

A list of the notation used in all the paragraphs can be found in Appendix A, a glossary of frequently used terms and abbreviations can be found in Appendix B.

1.5 Publications

In the context of this thesis, following publications have been accepted.

S. HEINZLE, J. WOLF, Y. KANAMORI, T. WEYRICH, T. NISHITA, and M. GROSS. Motion Blur for EWA Surface Splatting. In *Computer Graphics Forum (Proceedings of Eurographics 2010)*, Norrköping, Sweden, May 2010.

This paper extends the theoretical basis of the EWA splatting framework in the time dimension to incorporate motion-blur for point-based rendering.

S. HEINZLE, G. GUENNEBAUD, M. BOTSCH, and M. GROSS. A Hardware Processing Unit for Point Sets. In *Proceedings of the 23rd SIGGRAPH/Eurographics Conference on Graphics Hardware*, Sarajevo, Bosnia and Herzegovina, June 2008.

This paper presents a hardware architecture dedicated to the general and efficient processing of point-sampled geometry. *This paper received with the Best Paper Award.*

S. HEINZLE, O. SAURER, S. AXMANN, D. BROWARNIK, A. SCHMIDT, F. CARBOGNANI, P. LUETHI, N. FELBER, and M. GROSS. A Transform, Lighting and Setup ASIC for Surface Splatting. In *Proceedings of International Symposium on Circuits and Systems (ISCAS)*, Seattle, USA, May 2008.

This paper presents an ASIC implementation of the transform and lighting, and setup stages of surface splatting.

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 In *Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, San Diego, August 2007.

This paper presents a hardware architecture for surface splatting and presents two prototype implementations (ASIC and FPGA).

CHAPTER 2

Related work

Point-based graphics has experienced a considerable amount of research in the areas of modeling, processing, and rendering. In this chapter, we revisit previous work on point rendering (Section 2.1) and point-based surface definitions (Section 2.2). Furthermore, we then revisit graphics hardware architectures (Section 2.3). For an excellent overview of point-based graphics and a compilation of different works in this area we would like to refer the reader to the book "Point-Based Graphics" [GP07].

2.1 Point-based rendering

The birth of point-based graphics. The possibility of using points as rendering primitives to display curved surfaces was first suggested in the pioneering report of Levoy and Whitted [LW85]: a grid of area-less points is transformed to screen space and its density is estimated. Then, a radially symmetric Gaussian filter is applied at all pixel positions and the contribution of each source point is computed. Although this method allows for anti-aliased rendering of points, the radius of the filter is dependent both on the source density and the display sample density. As a drawback, the method requires knowledge about neighborhoods of the source points.

Due to the rapidly increasing complexity of geometric models, the idea of using points gained more interest a decade later. The work by Grossman and Dally [GD98] proposed to render point samples without connectivity. Their method uses dense point clouds with pre-filtered textures, and performs forward mapping of area-less points. The resulting holes due to the forward projection are then filled with a push-pull algorithm performed on the final pixel values. The QSplat multiresolution point rendering system [RL00] by Rusinkiewicz and Levoy introduced a hierarchical data structure and an associated rendering algorithm to interactively display huge point clouds. A hierarchy of bounding spheres is used to represent an object, each level in the hierarchy represents a refined version of its parent level object. The rendering algorithm then traverses the data structure until its a sphere projects to a given size, and finally renders the resulting pixel. The work by Pfister et al. [PZvBG00] introduced surfels as surface elements, similar to pixels as pixel elements. A surfel is represented as a tangent disk of the object, and is associated with a normal, a radius and sample values. The associated rendering algorithm uses a hierarchical representation of the object similar to [RL00].

High-quality elliptical weighted average splatting for rendering. None of the work so far addressed the problem of aliasing in image space. Zwicker et al. [ZPBG02] introduced elliptical weighted average (EWA) surface splatting, a high-quality rendering algorithm for surfaces represented by irregularly spaced point samples. EWA point samples are defined by elliptical disks spanning a reconstruction filter – so-called splats – which overlap in space and therefore are able to effectively avoid holes in the resulting image. The authors introduced a rigorous mathematical framework with anisotropic texture filtering based on a screen space formulation of texture filtering approach by Heckbert [Hec89]. By the use of a texture pre-filter in image space aliasing can be prevented in the case of minification, i.e. when points fall between the pixel sampling points. As EWA surface splatting will be used in this thesis, we present the algorithm in more detail on Chapter 3.

EWA surface splatting has since been extended and reformulated extensively. Ren et al. [RPZ02] introduced an object space reformulation of EWA splatting and implemented the pipeline using conventional GPUs. Their method employs semi-transparent quads with an elliptical texture to approximate the point cloud, and uses multiple passes to approximate the visibility of splats. Zwicker et al. [ZRB+04] showed that the original formulation of the EWA splatting algorithm can lead to artifacts in extreme perspective views due to the linear approximation of the projection on the framework. They proposed a

new technique to express the splat shape in image space using homogeneous coordinates and is able to compute EWA splatting more accurately. The GPU implementation is performed using multiple passes for visibility splatting, accumulation of the visible splats, and a final normalization step.

Botsch et al. [BSK04] introduced a perspectively correct scan conversion of circular splat primitives by locally intersecting the viewing rays with the splat primitive. The reconstruction filter is then evaluated perspectively correct in object space, however the image space pre-filter needs to be approximated with this method. In addition to the new scan conversion, the authors introduced per-pixel lighting by accumulating the normal information, and by evaluating the lighting equation after the surface has been reconstructed. The method was implemented using a multiple passes on the GPU similar to the perspective accurate splatting approach [ZRB+04]. Botsch et al. [BHZK05] then extended the method to handle elliptical splats and deferred shading using EWA surface splatting, and presented an optimized rendering algorithm again using multiple GPU passes. Guennebaud et al. [GBP06] introduced another perspectively correct approach to rasterize elliptical splats, however using fewer operations for the rasterization as Botsch et al. [BHZK05]. The algorithm also allows for incremental updates in the rasterization and the authors introduce a new approximation to the screen space filter approximation. Furthermore, the authors show how to blend triangle meshes and splats and how to render transparent point-based models. Zhang and Pajarola [ZP06] presented the first GPU implementation of EWA surface splatting that can be implemented in a single pass. The algorithm furthermore supports transparent point surfaces by introducing additional rendering passes. The idea of is to introduce deferred blending to delay the final visibility test to a image post-processing pass: a given point set is partitioned into multiple groups of splats that don't overlap in image space. However, the method requires a pre-sorting of the static model data and cannot be applied to dynamic point sets.

2.2 Point-based surface definitions

A central problem in point-based graphics techniques is the definition of a meshless surface representation that approximates or interpolates a set of input points continuously. While various different representations have been devised [GP07], the most important and successful class of such meshless representations is the class of point set surfaces. This class encompasses functional smooth surface approximations of irregular data using moving least squares (MLS) [She68]. Pioneered for the reconstruction of manifolds by

Levin [Lev01, Lev03], it was first introduced to computer graphics by Alexa et al. [ABCO+01, ABCO+03]. In their work, a point set surface is defined as the set of stationary points of an iterative projection operator. At each step of the projection, a planar parametrization is estimated around a local neighborhood, and then a bivariate polynomial is fit over this reference plane.

However, this polynomial approach was relatively expensive to compute and its alternatives relatively unexplored. Significant effort has been devoted to better understand and analyze the properties and limitations of point set surfaces [AK04a, AK04b, AA09] and to develop more efficient computational schemes. By omitting the polynomial fitting step, Amenta at Kil [AK04a] showed that the same surface can be defined and computed by weighted centroids and a smooth gradient field. This definition avoids the planar parametrization issues in the case of sparse sampling, and greatly simplifies the representation.

Adamson and Alexa [AA04] introduced a simple projection scheme based on iterative plane fits. In each iteration step a local planar tangent frame of a the surface is estimated around a small neighborhood and used for the projection. While yielding the same surface as the original definition [ABCO+03], such a plane fit becomes unstable especially for low sampling rates. To overcome this problem, Guennebaud and Gross [GG07] proposed to fit higher order algebraic surface such as spheres instead and showed its stability under low sampling densities. Using a spherical fit with appropriate normal constraints, this approach yields an efficient closed form solution of the underlying algebraic point set surface APSS [GGG08] and has been shown to resemble to the planar solution [AA04] when omitting the higher order terms.

However, all MLS based techniques presented so far can only reconstruct smooth surfaces. Various approaches have been proposed to overcome this limitation. One such class relies of an explicit representation of sharp features by using cell complexes [AA06] or tagged point clouds [GG07] to separate the input samples into different components. A more challenging task is to automatically detect or enhance features present in the input point cloud. A relatively new approach to preserve sharp features has been presented by Öztireli et al. [OGG09]. Their approach is inspired by robust statistics which naturally preserves any kind of high frequency features, from sharp edges to fine details, without any special handling or segmentation. The idea is based on statistic kernel regression, a popular method to estimate the conditional expectation of a random variable [TMF+07], and it can be implemented as a projection procedure with a projection procedure similar to [AA04]. Öztireli shows that their method is very robust over a variety of undersampled objects or noisy objects while still preserving sharp features.

Rendering using point-based surface definitions. Adams and Alexa [AA03] ray-trace point set surfaces [ABCO+01] based on MLS. Their method builds an enclosing sphere structure on top of the point samples in which the surface is contained in a pre-process step and uses this structure to intersect the ray near the surface to guess a good starting point. Then, the point on the viewing ray is iteratively projected onto the surface and the viewing ray is intersected with the local polynomial approximation of the surface. In a similar spirit, Adams et al. [AKP+05] also construct a tight bounding sphere hierarchy, which is then subsequently updated for dynamic data. Similar to the previous work, the authors first intersect with the bounding sphere hierarchy and repeatedly project the point on the ray with the surface until convergence. A different approach using point set surfaces is to iteratively upsample the point set surface to render the surface. Guennebaud et al. [GGG08] developed a adaptive, view-dependent upsampling scheme for the algebraic point set surface definition [GG07], and simply generate small splats which are subsequently rendered using a standard splatting algorithm such as [BHZK05]. Unfortunately, none of the methods for point set surfaces has been extended to support textured objects.

2.3 Graphics hardware

Conventional graphics hardware. Virtually all commercially available graphics hardware is based on triangle rasterization. In these architectures the object vertices are first transformed to the screen space coordinate system. Then the lighting equation is evaluated on each vertex before the vertices are assembled to triangles. The triangles are then possibly clipped to the view frustum, and rasterization converts the triangles to individual fragments or pixels. Finally the fragments are shaded and various tests such as scissor, alpha, stencil and depth test are performed before the frame buffer blending is performed.

Clark [Cla82] presented a first VLSI architecture dedicated to computer graphics performing floating point matrix transformations, clipping, perspective and orthographic projections, and viewport transformations. It was fully implemented in floating point and already exhibited a very high performance compared to the commercially available floating point co-processors in that era.

Fuchs et al. [FGH+85] developed the pixel-planes system for general rasterization of polygons. Their idea was to push logic into a so-called smart memory: for each pixel address (x, y) the memory is able to evaluate a configurable

function $f(x,y) = Ax + By + C$ for each pixel to determine whether to store the value or not. The smart framebuffer then can be used to perform scan conversion of polygons with z-Buffer visibility computation and shading. The Pixel-Planes 5 project by Fuchs et al. [FPE+89] extended on the original Pixel-Planes project by subdividing the framebuffer into tiles, with rasterizers directly assigned to tiles. Each of the rasterizer chips contained the "smart-memory" as presented already in their first version. While their architecture proved to be very scalable, commodity memory chips became more and more dense and did not leave no room for logic in their highly integrated memory arrays.

With the increasing size of screen resolutions and associated memory sizes, Whitton [Whi84] suggested to tile the framebuffers for more parallelism. Pineda [Pin88] presented an algorithm for polygon rasterization suitable for such parallel hardware implementations. Every directed edge of a polygon is represented by a linear edge function that separates the image space into left and right points. A pixel is then inside the polygon if it is on the same side of each directed edge. As the values of the edge functions can be interpolated similar to color and depth values, the algorithm is suited well for high-performance hardware implementations and has been subsequently used in many graphics processors [FPE+89, Ake93, MBDM97, MMG+98, SCS+08]. Olano and Greer [OG97] presented a new triangle scan conversion algorithm using half-edge functions [Pin88], performed entirely in homogeneous coordinates. By using homogeneous coordinates, the algorithm avoids the costly clipping tests and is more amenable to hardware implementation as it allows for heavy pipelining of the processing.

The RealityEngine graphics system [Ake93] marked a new generation of multi-board graphics systems. It featured multiple off-the-shelf programmable floating point processors designated to geometry processing, which were operating in multiple-instruction multiple-data (MIMD) fashion, whereas the rasterization processors were fixed function ASICs. It was able to render lighted, smooth shaded, depth buffered, texture mapped, and anti-aliased triangles. Texture filtering was performed using mip-mapping. The system featured multiple boards such as geometry boards for the transform and lighting computations, and raster memory boards containing the rasterization units and framebuffer memory. The InfiniteReality engine [MBDM97] extended on its predecessor, the RealityEngine. It improved on the RealityEngine by having a geometry distributor, a scheduler to better load balance between the different geometry engine. Furthermore, the off-the-shelf geometry processors were substituted with a custom design, where each processor was basically a SIMD floating point engine. Again, all geometry processor chips were coupled in MIMD fashion. As further optimization the frame

buffer was tiled in vertical strips for better load-balancing of the fragment stages.

McCormack et al. [MMG+98] presented the first single-chip, unified memory 3D graphics accelerator for fast rasterizing of triangles. Their system used the half-plane edge functions and optimized memory bandwidth by batching of fragments and chunking fragment generation to allow to prefetch memory and by employing texture caching. The system supported z-buffered, Gouraud shaded rasterization of triangles and lines with trilinear perspectively correct texture mapping. The next-generation consumer GPUs [Mor00] supported early culling by the use of a hierarchical z-buffer to avoid the execution of fragments that would be discarded later. Further optimizations included tile-based data compression for reduced memory bandwidth, stencil buffers for faster per-tile tests, and fast-clear bits to avoid buffer clears.

At this time, the continuous improvement of semi-conductor technologies fueled the explosive increase in computational power in GPUs. Lindholm et al. [LKM01] presented the first programmable, single-chip mass-market GPU featuring user-programmable vertex and pixel engines. It was implemented as NVIDIA's GeForce3 GPU. The processor was the first coarse grained SIMD processor allowing for high floating point throughput at modest chip area. The authors furthermore presented a power programming interface and made GPU programming accessible for a wide audience. The ATI Xenos architecture[Dog05] introduced a unified shader architecture where vertex and pixel shaders are executed on the same shader cores. The architecture allowed for improved intra-chip load balancing by allocating its programmable shader units dynamically to vertex and fragment processing.

Hasselgren and Akenine-Möller [HAM07] presented a programmable culling unit (PCU) placed right before the pixel shader to make the graphics pipeline even more programmable and efficient. Although pixel shaders do support a 'kill'-operation to cull fragments, the use of the operation seldom makes the execution faster on current GPUs due to the SIMD processing. The PCU executes a cull program before pixel shading starts and then decides conservatively whether to invoke pixel shading for a given tile or not.

Intel entered the 3D graphics market with the Larrabee architecture [SCS+08]. Instead of using a coarse grained SIMD approach, the architecture was based on the x86 Pentium architecture featuring multiple cores. The cores are operated in MIMD fashion, and are interconnected with a ring network. The Pentium cores itself have been augmented with a SIMD vector processor supporting fine grained SIMD within the cores. Only texture filtering is implemented as fixed function logic. All other parts of the rendering pipeline including rasterization and z-buffer handling are implemented entirely in

software. Inherent to the x86 architecture, Larrabee also features multiple levels of caches transparent to the user. The software rendering pipelining used binning of the screen space primitives according to its destination tiles, and assigned them dynamically to the rasterization processes. Due to the flexible software design dynamic load balancing can be implemented depending on the graphics application.

With the increasing geometric complexity in computer graphics triangles have become very small, which result in an overhead for the rasterization setup that assumes triangles to cover a large amount of pixels. Fatahalian et al. [FLB+09] showed a new strategy to rasterize so-called micropolygons by simply testing the micropolygons against randomized screen-samples. Furthermore, they showed how their rasterization algorithm can be extended to support motion blur and defocus as well by sampling the micropolygons in space and in time. Unfortunately, the impact and speed gains against current rasterizers remains unclear in the paper. More importantly, their approach does not resolve aliasing artifacts which can occur during minification.

Very recently commercially available graphics hardware converged towards more and more programmable computing machines, and can be considered programmable stream-processors. The CUDA architecture and software development kit [NVI07] exposed direct access to the GPU for general purpose computing and enabled software developers to take full advantage of the parallelism on such graphic board. Driven by this advancement towards general purpose programmable GPU, the state-of-the art commercial hardware supporting DirectX 11 [Mic10] took these concepts and integrated them into the graphics pipeline for even more configurable support in rendering. The most remarkable advances include geometry shaders to generate new geometry on the fly, programmable tessellation units for subdivision of polygons before rasterization, and DirectCompute shaders that enable programmers to write and read to independent memory locations with atomic operations for more flexibility during rendering. An illustration on the evolution of graphics hardware can be found in Figure 2.1.

Experimental graphics hardware. A few architectures consist of multiple rasterization nodes and create the final image by using composition [FPE+89, MEP92, TK96]. While the scalability can be particularly good in these architectures, certain newer features such as occlusion queries are difficult to implement efficiently. The Talisman architecture [TK96] tried to exploit both spatial and temporal coherence in computer animation to reduce the complexity and cost of hardware architectures. Their idea was to render each object into an independent image layer, and then re-use those images

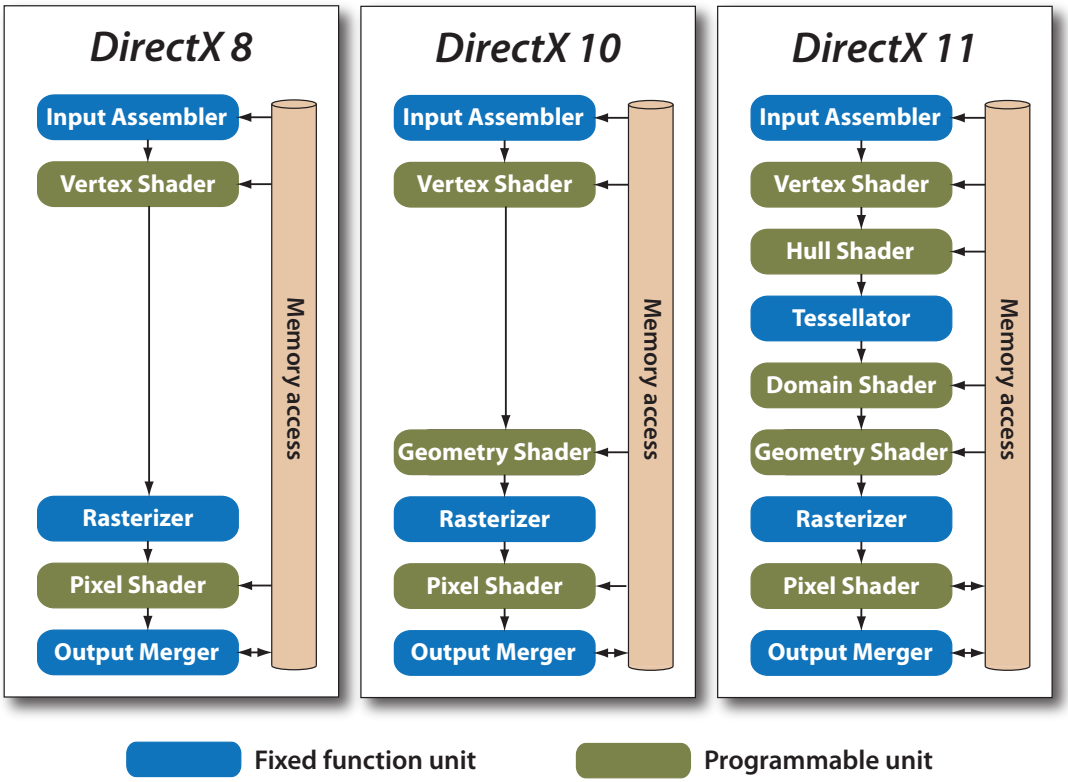


Figure 2.1: *Evolution of programmable graphics hardware.*

directly. For changing objects those layers are updated frequently whereas for stationary objects the images could be reused directly. For fast updates, an affine transformation can be additionally applied to the individual image layers in order to approximate real rendering updates. After all updates have been performed, the individual layers are composited to a final image. Molnar et al. [MEP92] presented their PixelFlow architecture that similar to traditional pipelines rasterizes a polygonal primitive. However, instead of assigning the individual rasterizers to individual portions of the screen, each rasterizer processes a portion of the object. A compositor network finally receives the generated pixels and combines them to the resulting screen image. The authors used a logic enhanced memory compositor similar to pixel planes, and supported supersampling by anti-aliasing as well as deferred shading.

The SaarCOR architecture [SWS02] used a fixed function architecture for ray casting instead of rasterization to display triangle meshes. A more flexible, programmable ray processing unit [WSS05] built on this work and was implemented as a coarse grained SIMD architecture, similar to commercial GPUs. The architecture featured novel hardware concepts for ray-tracing, such as

a dedicated fixed function ray traversal unit that implemented a *k*d-tree acceleration structure. The SIMD engine can be used to implement advanced effects, such as soft shadows, global illumination techniques, as well as the "spawning" of new threads by generating additional secondary rays using a special trace instruction. To alleviate the bandwidth bottleneck of ray-tracing, the architecture furthermore used chunking of spatially similar rays for increased memory locality. The same authors extended the architecture to support dynamic scenes using a B-KD tree [WMS] as spatial index structure, and furthermore estimated the area requirements and performance characteristics of an ASIC implementation of the mentioned architecture [WBS06].

A few experimental hardware architectures have been proposed for the rendering of non-polygonal primitives. The WarpEngine [PEL+00] resembles to an image-based rendering chip using real world images augmented with per-pixel depth as its rendering primitive. The pixels in the input images are treated as connected samples, warped to screen, and additional samples are interpolated bilinearly based on the original ones in order to avoid holes in the image reconstruction. Herout and Zemcik [HZ05] describe a prototype architecture that uses circular constant-colored splats as rendering primitives. The splats are directly visualized on the screen without any blending, and therefore image quality and anti-aliasing are not addressed with this architecture. The same authors [AH04] presented an approach how the forward splatted ellipses can be rasterized incrementally. Unfortunately, such an approach cannot be used for parallel rasterizers. The follow up work [ZP09] presented an improved version of their previous system [HZ05] by making multiple units of the pipeline parallel, without addressing the problem of better image quality. Majer et al. [MWA+08] presented a hardware architecture for point rendering using multiple FPGAs. The architecture projects area-less points that are shaded, transformed and projected to screen, and directly visualized on the screen. However, holes appear when the resolution of the mesh does not meet the sampling requirements of the screen, and aliasing may occur.

Whitted and Kajiya [WK05] proposed making the graphics pipeline fully programmable by replacing polygonal primitives with fully programmable procedural primitives which remain procedural throughout the pipeline. A single processor array would then handle geometric as well as shading elements in a unified way. A so-called programmable sampling controller then replaces the standard rasterizer by generating point samples of the surface without triangles as intermediate representation. Ideally, the sampling density would be controlled adaptively in order to guarantee hole-free reconstruction and avoid superfluous sampling, and their method coarsely samples the object in its parameter space and projects those samples to image

to determine the sampling radius in screen space in a first pass. In the second pass, they refine the sampling based on the sparse sampling. However, a non-negligible amount of oversampling cannot be avoided with their method. As expected, the input bandwidth is tiny however at the cost of very high internal computation.

Stewart et al. [SBM04] describe a triangle rasterization-based architecture that maintains a view-independent rendering of the scene. The idea is to generate a wide set of views simultaneously by replacing the 2D framebuffer with a 4D framebuffer which exhibits strong similarity with light-fields [LH96]. Input primitives are very finely subsampled into point samples associated with irradiance maps which are then casted into the 4D framebuffer. The output images for potentially large number of view points can then be reconstructed from this view-independent representation. Meinds and Barenbrug [MB02] explain a novel texture mapping architecture that uses a forward splatting instead of the traditional reverse texture fetch. In their approach, texture samples are splatted to screen space and then reconstructed in screen space with a pixel pre-filter to calculate the coverage. The authors show high-quality anisotropic and anti-aliased texture filtering can be achieved with this approach at modest cost.

CHAPTER 3

EWA surface splatting

The elliptical weighted average (EWA) surface splatting framework by Zwicker et al. [ZPBG01, ZPBG02] describes a high quality method for antialiased rendering of point sampled surfaces. In this chapter, we will review the framework which is essential to understand following chapters.

EWA surface splatting assumes so-called *splats* as input data. Splats can be considered as point samples that have been extended in space along the tangential plane of the sampled surface. Intuitively, splats are represented as ellipses in object space that mutually overlap with its neighbors. Each splat is associated with an elliptical Gaussian reconstruction kernel which is used to blend neighboring splats to achieve high visual quality.

Similarly to Heckbert's texture filtering approach [Hec89], the splats and its reconstruction kernels are projected to screen space, and additionally convolved with a band-limiting image space pre-filter. The pre-filter guarantees that aliasing artifacts due to minification can effectively be avoided. Then, the screen space reconstruction kernels are rasterized and accumulated to form the final rendered surface.

Zwicker et al. [ZPBG01] formulated the point rendering process as a resampling problem. Section 3.1 establishes the theoretical basis for EWA surface splatting used in the resampling process. Section 3.2 then shows how splats defined in object space can be resampled to the screen space. Section 3.3

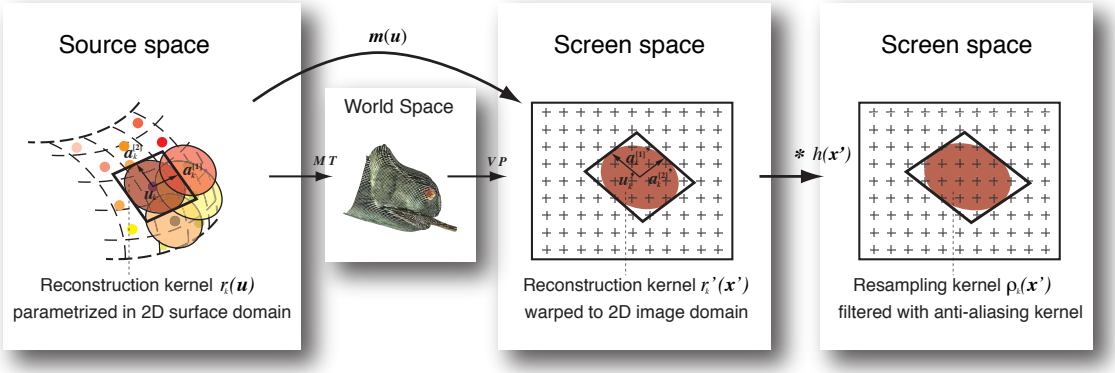


Figure 3.1: *Overview of EWA surface splatting. An object surface is represented by a set of sample points $\mathcal{P}_k$ defined by a center $\mathbf{u}_k$ and two tangential vectors $\mathbf{a}_k^{[1]}, \mathbf{a}_k^{[2]}$. The tangential vectors span a 2-dimensional frame in which the surface is defined, the so called source space. In particular, they span elliptical reconstruction kernels on the surface. These reconstruction kernels are then projected into the 2-dimensional image space yielding the projected reconstruction kernel. Finally, the kernels are convolved with an anti-aliasing pre-filter in image space to avoid aliasing due to minification. The intermediate step for the world space was just included for illustration purposes.*

finally presents practical rendering algorithms that implement the EWA surface splatting.

3.1 EWA framework

In the EWA splatting framework, a surface is represented as a set of irregularly spaced samples $\mathcal{P}_k$ – also called *splats*. Each splat $\mathcal{P}_k$ is associated with a position $\mathbf{u}_k$ and two tangent axes $\mathbf{a}_k^{[1]}, \mathbf{a}_k^{[2]}$ which span an ellipsoidal reconstruction kernel $r_k(\mathbf{u})$. Additionally, an attribute sample w_k describes the appearance of the surface[1]. See Figure 3.1 for an illustration.

The continuous surface attribute function $f(\mathbf{u})$ is then defined over a local surface parametrization $\mathbf{u}$:

$$f(\mathbf{u}) = \sum_{k \in \mathbb{N}} w_k r_k(\mathbf{u}) \,, \tag{3.1}$$

i.e. it is expressed as a weighted sum over all attribute samples. The domain of $f(\mathbf{u})$ will be denoted as *source space* in the following text.

[1]The attribute sample w_k is defined as a scalar unit without loss of generality for the derivation. In practice the sample could also be a vector unit, e.g. the diffuse color in RGB color space

The point rendering process can now be formulated as resampling process from source space to screen space. First, Equation (3.1) is projected to screen space. Then, the screen space kernels are low-pass filtered to avoid aliasing due to minification. The resulting filtered kernels are finally resampled to the pixel grid to render the point-sampled surface. The following three subsection will present more details on the resampling process.

1. Projection from source to screen space

The continuous attribute function (Eq. 3.1) is projected from source space to screen space using the projective mapping

$$\mathbf{m}(\mathbf{u}) : \mathbb{R}^2 \rightarrow \mathbb{R}^2. \tag{3.2}$$

The mapping is linear in its arguments, it is locally invertible, and it assigns a surface point $\mathbf{u}$ to its corresponding screen position $\mathbf{x}'$. Using this mapping the screen space signal can be formulated as

$$g(\mathbf{x}') = f(\mathbf{m}^{-1}(\mathbf{x}')). \tag{3.3}$$

By combining Equations (3.1) and (3.3) the screen space signal reformulates to

$$g(\mathbf{x}') = \sum_{k \in \mathbb{N}} w_k r'_k(\mathbf{x}') \,, \tag{3.4}$$

where

$$r'_k(\mathbf{x}') = r_k(m^{-1}(\mathbf{x}')) \tag{3.5}$$

represents one reconstruction kernel projected to screen space. Therefore, all reconstruction kernels can be projected to screen before their sum is computed.

2. Bandlimitation of the screen space signal

The screen space signal (Eq. 3.4) is then bandlimited using an anti-aliasing pre-filter $h(\mathbf{x}')$:

$$\begin{aligned} g'(\mathbf{x}') &= g(\mathbf{x}') * h(\mathbf{x}') \\ &= \int_{\mathbb{R}^2} g(\xi) h(x - \xi) d\xi \\ &= \sum_{k \in \mathbb{N}} w_k \rho_k(\mathbf{x}') \,. \end{aligned} \tag{3.6}$$

The filtered resampling kernels $\rho_k(\mathbf{x})$ are given as

$$\rho_k(\mathbf{x}') = \int_{\mathbb{R}^2} r'_k(\xi) h(\mathbf{x}' - \xi) d\xi. \tag{3.7}$$

The bandlimitation guarantees that no reconstruction filter falls between the pixel sampling grid, and effectively avoids aliasing due to minification.

3. Sampling of the continuous output function

Exploiting the linearity of the projection operator, Equation (3.6) then states that each reconstruction kernel r_k can be projected and filtered individually to derive the resampling kernels ρ_k. Finally, the contributions of these kernels can be summed up in screen space and sampled along the pixel grid to arrive at the final output image.

3.2 Surface resampling

This section presents the resampling process based on the EWA framework: First, the object space reconstruction filters are introduced. Then we show how the screen space warp to calculate the filtered resampling kernels in screen space can be derived. In contrast to Zwicker et al. [ZPBG02], we use homogeneous coordinates for this derivation.

3.2.1 Object space EWA resampling filter

EWA surface splatting uses elliptical Gaussians as reconstruction kernels and low-pass filters since they provide two features that are crucial for EWA splatting: Gaussians are closed under affine mappings and convolution. A two-dimensional elliptical Gaussian $G^2_{\mathbf{Q}}(\mathbf{x})$ with conic matrix $\mathbf{Q}$ is defined as:

$$G^2_{\mathbf{Q}}(\mathbf{x}) = \frac{|\mathbf{Q}_{2x2}|}{2\pi} e^{-\frac{1}{2}x^T \mathbf{Q} x}. \tag{3.8}$$

The conic matrix is defined as:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_{2x2} & \begin{matrix} * \\ * \end{matrix} \\ \begin{matrix} * & * \end{matrix} & * \end{bmatrix} = \mathbf{T}^{-T} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{T}^{-1}, \tag{3.9}$$

where $\mathbf{Q}_{2x2}$ denotes the upper 2x2 submatrix, and $\mathbf{Q}$ is constructed with

$$\mathbf{T} = \begin{bmatrix} \mathbf{a}^{[1]} & \mathbf{a}^{[2]} & \mathbf{u} \\ 0 & 0 & 1 \end{bmatrix} . \tag{3.10}$$

In this definition the elliptical Gaussian is centered at position $\mathbf{u} \in \mathbb{R}^2$, and oriented and scaled along the two main axes $\mathbf{a}^{[1]}, \mathbf{a}^{[2]} \in \mathbb{R}^2$.

Any arbitrary affine transformation can easily be applied: let $\mathbf{u} = \mathbf{M}\mathbf{x}$ be the affine transformation. The resulting new Gaussian then transforms to

$$G^2_{\mathbf{Q}}(\mathbf{M}\mathbf{x}) = |\mathbf{M}_{2x2}| G^2_{\mathbf{M}^{-T}\mathbf{Q}\mathbf{M}^{-1}}(\mathbf{x}) . \tag{3.11}$$

Convolving of two Gaussians with variance matrices $\mathbf{V}$ and $\mathbf{Y}$ can simply achieved by adding the respective conic matrices:

$$(G^2_{\mathbf{V}} * G^2_{\mathbf{Y}})(\mathbf{x}) = G^2_{\mathbf{V}+\mathbf{Y}}(\mathbf{x}) . \tag{3.12}$$

These two properties are used in the following resampling step to simplify the computations.

3.2.2 Screen space EWA resampling filter

The object space resampling filter is first mapped from source space to camera space, perspectively projected to screen, and bandlimited to avoid aliasing. The resulting screen space EWA resampling filters can then be rasterized along the pixel grid.

Mapping from source space to object space. The points $\mathbf{u} = [u_0, u_1, 1]$ are mapped from the local surface parametrization to camera space $\hat{\mathbf{u}}_k = [\hat{u}_0, \hat{u}_1, \hat{u}_2, 1]$. This mapping is defined as

$$\mathbf{x}_k = \mathbf{M}\mathbf{T}_k\mathbf{u}, \tag{3.13}$$

where $\mathbf{M} \in \mathbb{R}^4 \times \mathbb{R}^4$ is the model-view matrix that transforms points from object space to the camera system, and $\mathbf{T}_k \in \mathbb{R}^4 \times \mathbb{R}^3$ defines the mapping from the local surface parametrization to object space:

$$\mathbf{T}_k = \begin{bmatrix} \mathbf{a}^{[1]}_k & \mathbf{a}^{[2]}_k & \mathbf{u}_k \\ 0 & 0 & 1 \end{bmatrix} . \tag{3.14}$$

The two axes $\mathbf{a}^{[1]}_k$ and $\mathbf{a}^{[2]}_k$ define the shape and planar orientation of the Gaussian ellipsoid, the point $\mathbf{u}_k$ defines the center of the ellipsoid in space.

Perspective projection. The camera space points $\mathbf{x}$ are projected using the projection matrix $\mathbf{P}$ combined with the viewport transformation $\mathbf{V}$. The projection is finally achieved by dividing by the depth coordinate. Unfortunately, the perspective transformation does not constitute an affine transformation. Zwicker et al. [ZPBG02] approximated the projection a Taylor expansion $\mathbf{J}_k$ at point $\mathbf{x}_k$:

$$\mathbf{J}_k = \frac{\partial}{\partial \mathbf{x}}(\mathbf{V}\mathbf{P}\mathbf{x}_k). \tag{3.15}$$

The Taylor expansion $\mathbf{J}_k \in \mathbb{R}^3 \times \mathbb{R}^4$ then constitutes an affine approximation of the projection.

Screen space reconstruction kernel. The full projection step can now be described as a affine transformation given by

$$\mathbf{x}' = \mathbf{m}_k(\mathbf{u}) = (\mathbf{J}_k\mathbf{M}\mathbf{T}_k)\mathbf{u}\,. \tag{3.16}$$

Conveniently, the reconstruction kernel in screen space can be expressed as a Gaussian with variance matrix $(\mathbf{J}_k\mathbf{M}\mathbf{T}_k)$, as Gaussians are closed under affine transformations (Eq. 3.11):

$$r'_k(\mathbf{x}') = |\mathbf{J}_k\mathbf{M}\mathbf{T}_k| G_{\mathbf{V}'_k}(\mathbf{x}'), \tag{3.17}$$

with the new conic matrix

$$\mathbf{V}'_k = (\mathbf{J}_k\mathbf{M}\mathbf{T}_k)^{-T}\mathbf{Q}(\mathbf{J}_k\mathbf{M}\mathbf{T}_k)^{-1}\,. \tag{3.18}$$

Bandlimitation. As a last step, a Gaussian low-pass filter $h = G_{\mathbf{V}^h}$ is applied. By using the property that a convolution of two Gauss kernels can be expressed by the addition of its variance matrices (Eq. 3.12), the final reconstruction filter is expressed as:

$$\begin{aligned}\rho_k(\mathbf{x}) &= r'_k * h \\ &= |\mathbf{V}_k| G_{\mathbf{V}'_k + \mathbf{V}^H}(\mathbf{x}').\end{aligned} \tag{3.19}$$

The lowpass-filter conic matrix is usually chosen to be $\mathbf{V}^H = \mathrm{diag}(1,1,0)$. The following section will now give practical rendering algorithms for this screen space mapping.

3.3 Rendering algorithms

The resampling procedure presented in the previous section has been implemented mainly using two different approaches, backward and forward

mapping algorithms. Forward mapping algorithms transform all input splats to screen space and rasterize the resulting screen space reconstruction kernels, following the procedure defined in Section 3.2 very closely. Backward mapping algorithms in contrast use a ray casting approach to intersect the viewing rays with the reconstruction kernels in object space.

The original formulation of EWA splatting [ZPBG01] uses a forward mapping approach based on the affine approximation of the projection (see previous Section 3.2.2). We will present this approach in the following Section 3.3.1 and discuss the implications of this approximation.

To overcome some of the limitations of the forward mapping, various approaches using a backward mapping approach have been proposed. We will discuss these options in the following Section 3.3.2 and show how this approach can be used within a rasterization framework.

3.3.1 Forward mapping algorithm

In general, this rendering algorithm closely follows the strategy of Section 3.2.2, with some practical changes and optimizations.

The Gaussian reconstruction filter decays very fast and will have negligible contribution on points far away from its center. Therefore, the Gaussians are usually cut off at a given (low) function value, and therefore the splat's extent in space can be limited drastically.

Furthermore, the definition of the local surface parametrization omits the problem of visibility, and therefore requires a mechanism to detect the visible and hidden surfaces. While different strategies including front to back ordering are possible, Zwicker et al. [ZPBG01] use a z-buffer approach: a new depth value is compared to the already stored depth value in the z-buffer. If the difference between the two depths is smaller than a given threshold, the contribution is added to the current pixel. In Chapter 5 we will present a ternary depth test similar to this definition.

The algorithm is performed as follows:

1. For each input splat $\mathcal{P}_k$:
    a) Construct the screen space reconstruction filter with conic matrix $\mathbf{V}'_k$ (Eq. 3.18) by using an affine approximation $\mathbf{J}_k$ of the projection.
    b) Construct the filtered resampling filter ρ_k.

c) Determine position and extent of the bounding box of the surface splat based on the cutoff value. For each pixel inside the bounding box:

   i. Evaluate the filtered Gaussian resampling filter $\rho_k(\mathbf{x}')$.

   ii. Determine visibility of splat at current pixel.

   iii. If the splat belongs to the visible surface, accumulate with current pixel value.

2. Normalize final pixel values by accumulated weights.

One important property of EWA surface splatting can be observed in this algorithm. The Gaussian filters do usually not form a partition of unity, and therefore the accumulated weights for the visible surface do not sum up to one. As a result, the final pixel values have to be normalized as soon as all splats belonging to a surface have been splatted. Only after the normalization step, the surface is completely reconstructed.

Unfortunately, the Jacobian is only a very coarse approximation of the projection. This leads to strong perspective distortions often occurring around object edges, which in turn lead to computationally very unstable results. Whereas a solution for this problem has been proposed [ZRB+04] by using more accurate perspective approximations, this solution comes at increase the computational burden by an order of magnitude.

Another solution to this problem will be presented in the next section: the projection can be evaluated exactly at the cost of approximating the screen space filter instead.

3.3.2 Backward mapping algorithm

To overcome the errors introduced by the affine approximation of the projection, Botsch et al. [BSK04, BHZK05] proposed a simple backward algorithm. Their algorithm processes each splat independently, estimates its screen space bounding box and subsequently casts viewing rays to the tangential frame of the splat. In the tangential frame the value of the Gaussian can be evaluated perspectively correct. However, the backward mapping algorithm cannot evaluate the screen space anti-aliasing pre-filter $h(\mathbf{x}')$ correctly anymore, which can be approximated instead.

The algorithm can be described as follows:

1. For each input splat $\mathcal{P}_k$:

a) Determine/estimate position and extent of bounding box of surface splat based on the cutoff value.

b) For each pixel inside the bounding box:

   i. Intersect viewing ray with the tangent frame of $\mathcal{P}_k$.

   ii. Evaluate the object space Gaussian reconstruction filter $r(\mathbf{u})$.

   iii. Approximate screen space filter in object space.

   iv. Determine visibility of splat at current pixel.

   v. If the splat belongs to the visible surface, accumulate with current pixel value.

2. Normalize final pixel values by accumulated weights.

This approach leads to computationally much more stable results, and the approximation of the anti-aliasing pre-filter does not introduce any visible artifacts. For more information on the approximation of the pre-filter please refer to [BSK04, BHZK05].

Due to the superiority in terms of stability we will use and extend this backward mapping in the following chapters.

3.4 Conclusion

This chapter revisited the EWA surface splatting framework, and we derived the resampling procedure using affine coordinates. Rendering algorithms based on the resampling procedure can be classified into direct forward mapping algorithms, and backward mapping algorithms. Backward algorithms are computationally more correct and stable, and variants of backward algorithms will be used in the following chapters.

In the next chapter, the original framework will be extended in the time domain to support the generation of motion-blurred images with point-sampled geometry directly.

CHAPTER 4

Motion blur for EWA surface splatting

The EWA surface splatting framework presented in the previous chapter generates still frames, which depict a perfect instant in time, and therefore lacks realism and the sensation of dynamics due to the absence of motion blur. The term motion blur denotes the visual effect that appears in still images or film sequences when objects moving with rapid velocity are captured. The image is perceived as smeared or blurred along the direction of the relative motion to the camera. The reason for this is that the image taken by a camera in fact is an integration of the incoming light over the period of exposure. This appears natural to us because the human eye behaves in a similar way. To achieve this effect in computer graphics the most correct approaches are either temporal supersampling, that is, producing frames as a composite of many time instants sampled above the Nyquist frequency, or – which is theoretically more justified – bandlimiting the incoming signal before sampling to guarantee that its Nyquist frequency is met.

In this chapter we propose a new method for applying motion blur to EWA surface splatting. Section 4.1 will give an overview on related work for motion blur. In Section 4.2 we then present a consistent extension of the theoretical basis of the EWA splatting framework into the time dimension. The novel extension includes a temporal visibility to mathematically represent motion-blurred images with point-sampled geometry. The conceptual elegance of our approach lies in replacing the 2D Gaussian kernels which continuously

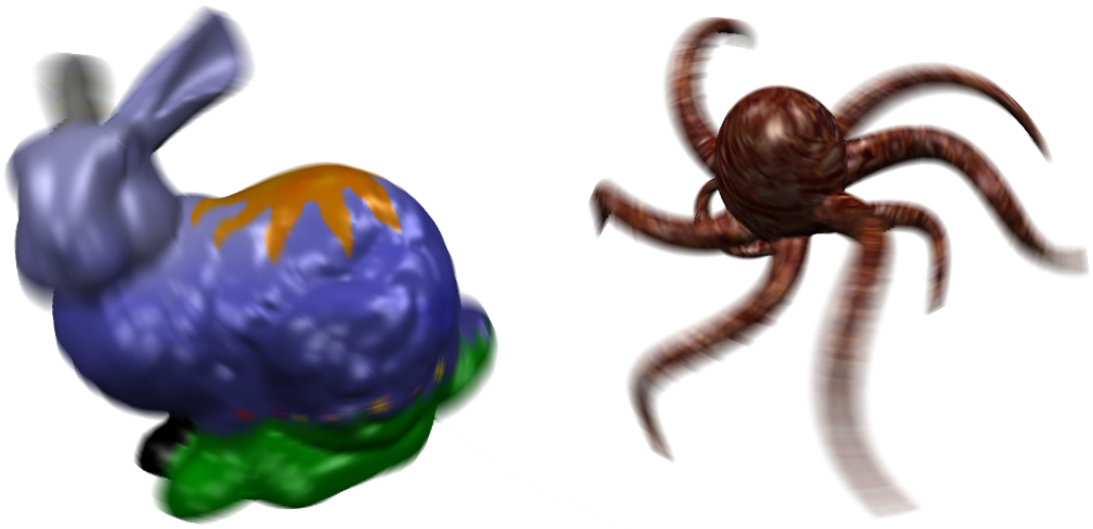


Figure 4.1: *EWA motion blur examples rendered with our GPU implementation.*

reconstruct the point-sampled surface by 3D Gaussian kernels which unify a spatial and temporal component. By use of these kernels the scene can be reconstructed continuously in space as well as time. Additionally, the incoming signal can be bandlimited before sampling to guarantee that its Nyquist frequency is met. The derived result naturally fits into the EWA splatting algorithm such that the final image can be computed as a weighted sum of warped and bandlimited kernels.

Based on the developed mathematical framework, we introduce a rendering algorithm with strong parallels to the original EWA surface splatting in Section 4.3. This algorithm applies ellipsoids with spatial and temporal dimensionality as new rendering primitives. Specifically, the surface splats are extended by a temporal dimension along the instantaneous velocity vector. The emerging ellipsoids automatically adapt to the local length of the piecewise linearized motion trajectory. We then present how temporal visibility can be solved by using an adapted A-Buffer [Car84] algorithm.

We provide an approximation of the rendering algorithm by the description of an entire point rendering pipeline using vertex, geometry and fragment program capability of current GPUs in Section 4.4. Finally we conclude with a discussion of the introduced approximations and their effect on image quality and rendering performance in Section 4.5. In addition, we compare results of the software implementation using A-Buffers and the GPU approximation with ground truth images generated by temporally highly supersampled traditional EWA surface splatting. Finally we will conclude this chapter with an outlook for future work in Section 4.6.

4.1 Motion blur in computer graphics

A well-argued discussion on motion blur is provided by the work of Sung et al. [SPW02]. There the authors define the problem of motion blur based on the rendering equation and categorize previous work according to the respective approach to approximate this equation:

$$i(\omega,t) = \sum_l \int_\Omega \int_T \underbrace{r(\omega,t)}_{\text{Shutter}} \underbrace{g_l(\omega,t)}_{\text{Visibility}} \underbrace{L_l(\omega,t)}_{\text{Luminance}} \, dt d\omega \,. \tag{4.1}$$

The function $i(\omega,t)$ represents the incoming luminance from a solid angle ω at time t. The sum iterates through all l objects in the scene and integrates the luminance $L_l(\omega,t)$ over the total solid angle Ω from the environment, in the exposure time T. The term $r(\omega,t)$ describes a shutter reconstruction filter, and $g_l(\omega,t)$ describes the visibility function.

Monte Carlo integration methods such as Distributed Raytracing [CPC84, Coo86] try to approximate the integral directly by stochastic supersampling. However, due to the randomly distributed oversampling, a large number of samples is usually required to avoid excessive noise and such to achieve visually plausible results. In a similar context, the frameless rendering approach [BFMZ94] presents an alternative to traditional frame-based rendering and simulates motion blur directly via immediate randomized pixel updates. In their approach, pixels are updated in random order with the latest available input parameters, and displayed immediately on the screen. Therefore, an image on the screen is a mixture of past and present images and can thus be considered a highly undersampled version of Distributed Raytracing.

Haeberli and Akeley [HA90] achieved motion blur by supersampling multiple scenes rasterized at different time instants and therefore approximate Equation (4.1) directly. Nevertheless, to avoid banding artifacts the complete scene must be rendered at a frequency higher than the Nyquist frequency of the element with the fastest motion. In the case of a nearly static scene with a small number of fast moving objects, their approach produces an unnecessary high constant number of samples. Recent work by Egan et al. [ETH+09] observed that motion blur is caused by a shear in the space-time signal as well as in the frequency domain. Based on the frequency analysis an adaptive sampling scheme can be derived. Combining this sampling with a sheared reconstruction filter then produces high-quality results with lower sampling rates as compared to previous supersampling methods. Fatahalian et al. [FLB+09] presented an approach to render micropolygons with motion blur by stochastically sampling the volumes swept by moving polygons.

Their approach unfortunately comes at very high computational cost, and is still inefficient in terms of sample test efficiency.

Most other works reduce the complexity of the problem by making assumptions on the scene behavior and/or employing further simplifications. The work of Korein and Badler [KB83] presents an approach that computes the exact per-pixel visible intervals for each geometry element based on their continuous movement, assuming constant shading and a temporal box filter. They validate their method by applying it to screen space circular disks with constant shading. The small set of the disk's properties are then interpolated over time to determine the exact intervals which are subsequently used to calculate to final motion-blurred image. The method we propose uses a similar approach for EWA surface splats to resolve the temporal visibility. Grant [Gra85] proposes a stationary 4D representation for 3D polyhedra in linear motion to compute temporally continuous visible polyhedra in the 3D image plane. This approach relies on a temporal box filter and assume constant shading over time. However, linear movements of polygonal surfaces do not result in planar surfaces in every case and subsequently cannot be represented by 4D polyhedra correctly.

Further methods are based on a geometric morphing of the objects or the introduction of new geometry. In the case of particle systems, [Ree83] suggests rendering particle points without area as line segments. [Cat84] uses a circular filter at every pixel to accumulate the contributions of a geometric primitive. The author shows that motion blur can be achieved by morphing the filter or, as he proposes, by morphing the objects in screen space. This morphing is performed per pixel, and converts the temporal contribution of the primitive to a spatial contribution for motion blurred images. Unfortunately, shading and texturing cannot be applied for his method. [WZ96] assumes that a human viewer cannot distinguish correct motion blur from approximations for interactive frame rates. The authors therefore approximate motion blur by constructing new semi-transparent geometry based on the motion vector. However, their method cannot handle the visibility function and inter-object relations correctly.

Another field is constituted of various post-processing techniques which operate on the synthesized images and disregard the actual geometry. Potmesil and Chakravarty [PC83] produce motion blur by a convolution of the rendered still image with a point spread function derived from the motion of the objects. The two-and-a-half-D motion blur algorithm [ML85] handles multiple scene objects by convolving them individually, followed by a composition in a back-to-front manner. In comparison, similar to other post-processing techniques [Max90, Shi93, CW93], these approaches cannot adapt to local

properties of the geometry and cannot address the situation where moving objects cannot be separated into non-overlapping layers in depth.

The work of [MMS+98] proposes a splatting approach for volumetric rendering and shows how motion blur can be achieved by constructing new footprint functions based on circular splats and their respective motion vector in the image plane. However, the visibility problem cannot be solved using their method. The method of [GM04] extends this approach for EWA surface splatting by focusing on strong motion hints instead of photo-realistic motion blur. Their method combines a static, sharp image of the scene with blurred motion hint ellipses constructed from the original object using its motion vectors. Their approach uses a simplified version of EWA surface splatting as described by Zwicker et al. [ZPBG01], and – similar to our work – constructs three-dimensional reconstruction kernels by convolving the two-dimensional object space ellipses with a low-pass filter along the motion direction. However, their method directly projects the kernels to an affine screen space approximation, and neglects the temporal visibility. It is therefore not able to reproduce photo-realistic motion blur. A more detailed discussion about the differences to our approach is presented in Section 4.3.5.

In comparison to the previous work presented here our method solves the motion blur equation by supersampling the objects in world space and by combining the samples analytically using a reconstruction filter. The following section will present this approach in more detail.

4.2 Extended EWA surface splatting

To formulate the problem of motion blur we interpret an image as a 2D signal in screen space. For an instantaneous image at time t, the intensity at screen space position $\mathbf{x}$ is given by the continuous spatio-temporal screen space signal $g(\mathbf{x},t)$. A motion-blurred image which captures the scene over the exposure period T is represented by $G_T(\mathbf{x})$. The intensity value at position $\mathbf{x}$ is generated by a weighted integration of incoming intensities over the exposure time:

$$G_T(\mathbf{x}) = \int_T a(t) g(\mathbf{x},t) dt \,, \tag{4.2}$$

where $a(t)$ denotes a time-dependent weighting function used to model the influence of the camera shutter and the medium which captures the scene. The process of generating $G_T(\mathbf{x})$ can be considered as a resampling problem of $g(\mathbf{x},t)$.

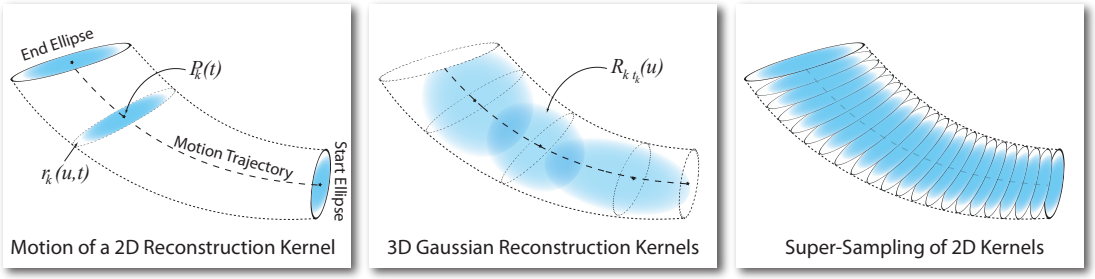


Figure 4.2: *A single 2D splat moving along its motion trajectory (left). We place volumetric kernels that comprise the spatial and temporal domain of the moving 2D splat (center) along the motion trajectory. In comparison, an approach consisting of a pure accumulation of temporal supersamples would require a high number of sampled 2D splats (right).*

In the following subsections we extend the original EWA framework [Hec89, ZPBG02] presented in Chapter 3 by a time dimensionality and introduce a temporal visibility function to determine occluded surface parts. We then introduce three-dimensional reconstruction kernels representing a local, linear approximation of the points' motion trajectories, very much like the two-dimensional reconstruction kernels of EWA splatting do in the spatial domain. The algorithm presented in Section 4.3 finally renders these kernels to the screen.

4.2.1 The continuous spatio-temporal screen space signal

We extend the continuous surface function defined in Equation (3.1) by a time dimensionality to $f(\mathbf{u},t)$. Similar to Section 3.1 the surface is defined over a local surface parametrization $\mathbf{u}$, also called the source space.

The projective mapping from Equation (3.2) is then redefined as

$$\mathbf{m}(\mathbf{u},t)\colon \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2 \times \mathbb{R} \tag{4.3}$$

which maps the continuous surface function from source space to screen space. It is locally invertible for a fixed time instant t and assigns a surface point $\mathbf{u}$ to its corresponding screen position $\mathbf{x}$. Using this mapping the screen space signal (Eq. 3.3) can be reformulated as spatio-temporal signal

$$g(\mathbf{x},t) = f(\mathbf{m}^{-1}(\mathbf{x},t),t)\,. \tag{4.4}$$

The continuous surface function $f(\mathbf{u},t)$ itself is represented by the set $\{P_k(t)\}$ of time-dependent, irregularly spaced point samples. Each point $P_k(t)$ is

associated with an ellipsoidal reconstruction kernel $r_k(\mathbf{u},t)$ which is centered at a position $\mathbf{u}_k(t)$ at time t. The continuous surface function $f(\mathbf{u},t)$ is reconstructed similarly to Equation (3.1) by the weighted sum

$$f(\mathbf{u},t) = \sum_{k\in\mathbb{N}} w_k(t) r_k(\mathbf{u},t) \,, \tag{4.5}$$

where $w_k(t)$ denotes the attribute value of the k-th point sample at time t.

We define the projective mapping $\mathbf{m}(\mathbf{u},t)$ individually for each reconstruction kernel as $\mathbf{m}_k(\mathbf{u},t)$ to simplify the following derivations.

4.2.2 Time-varying EWA surface splatting

To extend EWA surface splatting correctly into the time dimension the visibility function $v_k(\mathbf{x},t)$ needs to be introduced. This function defines the visibility of any surface point $\mathbf{u}(t) = \mathbf{m}_k^{-1}(\mathbf{x},t)$ in the local parametrization plane of point sample $P_k(t)$ at time t from the camera viewpoint. By combining Equations (4.4) and (4.5) the spatio-temporal screen space signal reformulates to

$$g(\mathbf{x},t) = \sum_{k\in\mathbb{N}} w_k(t) v_k(\mathbf{x},t) r'_k(\mathbf{x},t) \,, \tag{4.6}$$

where $r'_k(\mathbf{x},t) = r_k(\mathbf{m}_k^{-1}(\mathbf{x},t),t)$ represents a reconstruction kernel projected to screen space. Similar to Zwicker et al. [ZPBG02], we bandlimit the spatio-temporal screen space signal with a spatio-temporal anti-aliasing filter $h(\mathbf{x},t)$:

$$\begin{aligned} g'(\mathbf{x},t) &= g(\mathbf{x},t) * h(\mathbf{x},t) \\ &= \int_T \int_{\mathbb{R}^2} g(\xi,\tau) h(x-\xi, t-\tau) d\xi d\tau \\ &= \sum_{k\in\mathbb{N}} w_k(t) \rho_k(\mathbf{x},t) \end{aligned} \tag{4.7}$$

where the filtered resampling kernels $\rho_k(\mathbf{x},t)$ are given as

$$\rho_k(\mathbf{x},t) = \int_T \int_{\mathbb{R}^2} v_k(\xi,\tau) r'_k(\xi,\tau) h(\mathbf{x}-\xi, t-\tau) d\xi d\tau \,. \tag{4.8}$$

The visibility is dependent on the reconstruction kernels, and $\rho_k(\mathbf{x},t)$ can be evaluated as follows. In the first step, all reconstruction kernels are filtered using $h(\mathbf{x},t)$. The visibility is then determined based on the filtered reconstruction kernels leading to the filtered resampling kernels.

The above equations state that we can first project and filter each reconstruction kernel $r_k(\mathbf{u},t)$ individually to derive the resampling kernels $\rho_k(\mathbf{x},t)$. As a

consequence the contributions of these kernels can be accumulated in screen space where occluded parts are masked out by the filtered visibility functions $v'_k(\mathbf{x}, t)$. This separation simplifies the computation, as the attribute function can be arbitrary, while the visibility function is well defined by the geometry and its associated motion.

The equations presented so far are very similar to the original formulation in Section 3.1. All functions and filters have been extended by a time domain, and a visibility function has been introduced to handle occlusions. In the next section the extensions will be used for the temporal reconstruction of point sampled surfaces.

4.2.3 Temporal reconstruction

In analogy to a spatial reconstruction of $f(\mathbf{u}, t)$ we sample the motion trajectories of $P_k(t)$ in time and subsequently build a reconstruction which is also continuous over time. To achieve this we employ ellipsoidal 3D reconstruction kernels $R_{kt_k}(\mathbf{u})$ which define linearized trajectory patches of the moving point samples, similarly to the 2D reconstruction kernels that define linearized patches of the surface in the original EWA splatting algorithm. These new reconstruction kernels are centered at the point-sample positions $\mathbf{u}_{kt_k} = \mathbf{u}_k(t_k)$ and are constructed by convolving the elliptical 2D Gaussian kernels $r_k(\mathbf{x}, t_k)$ with a 1D Gaussian along the instantaneous velocity vector. The surface function is now continuously reconstructed as follows:

$$f(\mathbf{u}, t) = \sum_{k \in \mathbb{N}} \sum_{t_k} w_k(t) R_{kt_k}(\mathbf{u}) \,. \tag{4.9}$$

Combining this result with Equation (4.4) leads to the following equation for the continuous spatio-temporal screen space signal $g(\mathbf{x}, t)$:

$$g(\mathbf{x}, t) = \sum_{k \in \mathbb{N}} \sum_{t_k} w_k(t) v_{kt_k}(\mathbf{x}, t) R'_{kt_k}(\mathbf{x}) \,, \tag{4.10}$$

where $R'_{kt_k}(\mathbf{x}) = R_{kt_k}(\mathbf{m}_k^{-1}(\mathbf{x}, t_k))$ are the reconstruction kernels projected to screen space. The time index t_k reflects that the sampling times are chosen adaptively for the respective point-sample trajectories depending on the motion and shading changes over time, see Figure 4.2 for an illustration. The actual sampling we used in our implementation is described in Section 4.4.5.

An explicit expression for the bandlimited spatio-temporal screen space signal of Equation (4.7) can then be expressed as:

$$\begin{aligned} g'(\mathbf{x},t) &= g(\mathbf{x},t) * h(\mathbf{x},t) \\ &= \sum_{k\in\mathbb{N}} \sum_{t_k} w_k(t)\hat{\rho}_{kt_k}(\mathbf{x},t) \,, \end{aligned} \tag{4.11}$$

with the filtered resampling kernels $\hat{\rho}_{kt_k}(\mathbf{x},t)$:

$$\hat{\rho}_{kt_k}(\mathbf{x},t) = \int_T \int_{\mathbb{R}^2} v_{kt_k}(\xi,\tau) R'_{kt_k}(\xi,\tau) h(\mathbf{x}-\xi, t_k-\tau) d\xi d\tau \,. \tag{4.12}$$

The resampling kernels $\hat{\rho}_{kt_k}(\mathbf{x},t)$ are again evaluated by first filtering the reconstruction kernels and then determining the visibility based on the filtered reconstruction kernels. We use the A-Buffer approach presented in Section 4.3.4 to resolve the visibility in practice: in a first step, the visibility function is computed by determining which filtered reconstruction kernels contribute to a single pixel. The visibility is then used as temporal opacity contribution for each kernel when evaluating the integral in Equation (4.2).

4.3 Rendering

This section briefly gives an overview of the rendering algorithm before going into detail. First, the reconstruction filters $R'_{kt_k}(\mathbf{x})$ are constructed in a similar way to EWA surface splatting. The two axes $\mathbf{a}^{[1]}, \mathbf{a}^{[2]}$ span the plane of the original 2D surface splat and a third temporal axis $\mathbf{a}^{[3]}$ is constructed based on the motion vector $\mathbf{m}$. The latter is determined as the difference vector between the start and end positions of a point with respect to the time window of the temporal supersample and the temporal axis is constructed as $\mathbf{a}^{[3]} = \frac{\alpha}{2}\mathbf{m}$, where α controls the variance in the time domain.

The reconstruction filter is projected to screen space, sampled at the pixel locations by rasterization and accumulated to the current image. We use a backward mapping algorithm similar to Section 3.3.2 to integrate the filter in a perspectively correct manner along the direction of the viewing ray. To reduce the computational complexity we also limit the extent of the filter to a fixed cut-off value and can therefore limit the extent of the filter to a bounding polygon in screen space.

The correct visibility is computed using an A-Buffer approach to separate distinct portions of the surface, according to the time interval the surface has been visible. After all visible filter integration values have been accumulated

the result needs to be normalized since, in general, the EWA filters do not form a partition of unity in space.

The following subsections will provide details on the construction of the resampling filter, the integration of the resampling filter along the viewing ray, the bounding volume and the visibility A-Buffer.

4.3.1 The 3D spatio-temporal reconstruction filter

In analogy to EWA surface splatting we choose ellipsoidal Gaussians $\mathcal{G}^3_{\mathbf{Q}}(\mathbf{x})$ as 3D reconstruction kernels $R_{kt_k}(\mathbf{x}) = \mathcal{G}^3_{\mathbf{Q}_{kt_k}}(\mathbf{x})$ and as low-pass filters. Additionally to the properties of being closed under affine mappings and convolution, another property is that the line integral of a 3D Gaussian resembles to a 2D Gaussian, as will be seen later in this section. Using these properties we then can analytically compute samples for the rasterization.

We define a 3D ellipsoidal Gaussian $\mathcal{G}^3_{\mathbf{Q}}(\mathbf{x})$ with the 4x4 quadric matrix $\mathbf{Q}$ using homogeneous coordinates $\mathbf{x} = [x\ y\ z\ 1]^{\mathrm{T}}$ similar to Section 3.2.1 as:

$$\mathcal{G}^3_{\mathbf{Q}}(\mathbf{x}) = \sqrt{\frac{\delta^3|\mathbf{Q}|}{\pi^3}}\, e^{-\delta\, \mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x}}\,, \tag{4.13}$$

where the Gaussian is normalized to the unit volume integral. The scaling factor δ controls the variance of the Gaussian. The quadric matrix $\mathbf{Q}$ can be decomposed into $\mathbf{Q} = \mathbf{T}^{-\mathrm{T}}\mathbf{D}\mathbf{T}^{-1}$, where the 4x4 transformation matrix $\mathbf{T}$ is constructed out of the three arbitrary, independent axis vectors $\mathbf{a}^{[1]}$, $\mathbf{a}^{[2]}$, $\mathbf{a}^{[3]}$ spanning the Gaussian centered at point $\mathbf{u}$:

$$\mathbf{T} = \begin{bmatrix} \mathbf{a}^{[1]} & \mathbf{a}^{[2]} & \mathbf{a}^{[3]} & \mathbf{u} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.14}$$

and $\mathbf{D} = \mathrm{diag}(1,1,1,0)$ is a diagonal matrix.

From a geometric viewpoint the three axis vectors span a skewed coordinate system, that is to say, the system of the Gaussian filter, see Figure 4.3. A point in space is transformed into the coordinate system of the Gaussian filter by $\mathbf{T}^{-1}\mathbf{x}$ and the weight of the filter is evaluated based on the distance of this point to the origin.

The correct screen space bandlimiting filter is approximated by evaluating it in object space similar to [GBP06]. We first estimate the spacing of the pixel grid in object space. If the length of the axes, projected onto this estimation,

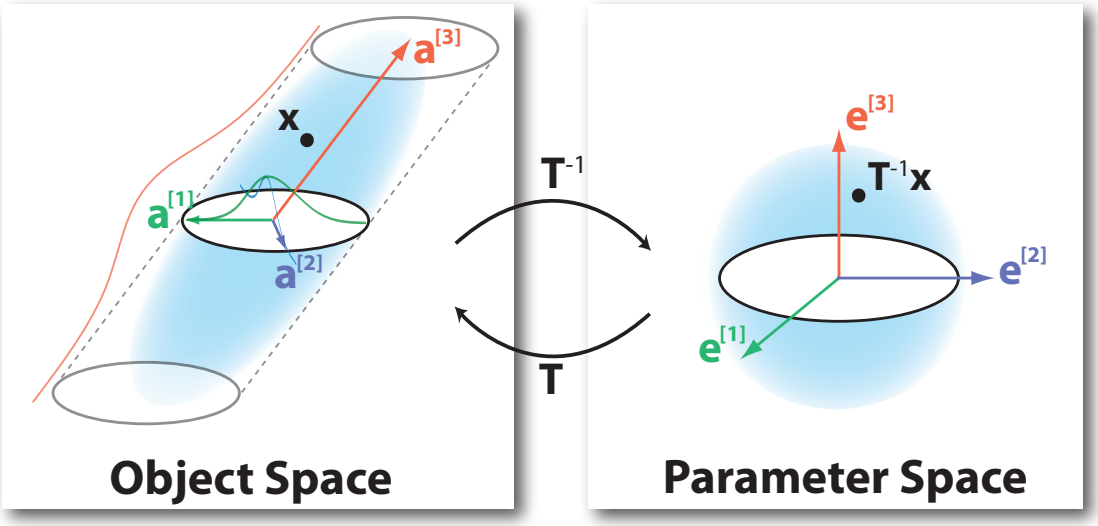


Figure 4.3: *We construct the 3D reconstruction kernels based on the two original splat axes* $\mathbf{a}^{[1]}, \mathbf{a}^{[2]}$ *and the instantaneous motion vector* $\mathbf{m}$*. The matrix* $\mathbf{T}$ *is used to transform points and lines from object space to the respective parameter space. In parameter space an iso-level of the reconstruction kernel can be interpreted as the unit sphere.*

does not match the filter width, we enlarge the axes to the size of the filter radius.

The next section will show how to integrate a reconstruction filter along the viewing ray.

4.3.2 Sampling of the reconstruction filter

To evaluate the contribution of a single 3D Gaussian to a pixel, the Gaussian is integrated along the path of the viewing ray within the integration time which equals the time of exposure. The viewing ray for a pixel can be put in parametrized form as $\mathbf{r}(s) = \mathbf{p} + s\,\mathbf{d}$, where $\mathbf{p} = [x_w\ y_w\ 0\ 1]^{\mathrm{T}}$ denotes the pixel position in window coordinates and vector $\mathbf{d} = [0\ 0\ 1\ 0]^{\mathrm{T}}$ represents the viewing direction of the ray. In a first step the viewing ray is transformed into the parameter system of the ellipsoid:

$$\tilde{\mathbf{r}}(s) = (\mathbf{V} \cdot \mathbf{P} \cdot \mathbf{M} \cdot \mathbf{T})^{-1}\,\mathbf{r}(s) = \tilde{\mathbf{p}} + s\,\tilde{\mathbf{d}}\,, \tag{4.15}$$

where $\mathbf{V}$, $\mathbf{P}$ and $\mathbf{M}$ denote the viewport, projection and model-view matrix, respectively.

In a next step $\tilde{\mathbf{r}}(s)$ is transformed to the integration line $\mathbf{l}(t)$. The integration line parametrizes the position on the ray at integration time t. Let

$\tilde{\mathbf{p}}' = \tilde{\mathbf{p}}_{xyz}/\tilde{\mathbf{p}}_w$ and $\tilde{\mathbf{d}}' = \tilde{\mathbf{d}}_{xyz}$ be the de-homogenized vectors. The integration line is then defined as

$$\mathbf{l}(t) = \mathbf{b} + \mathbf{f}t\,, \tag{4.16}$$

with the transformed support point and direction

$$\mathbf{b} = \tilde{\mathbf{p}}' - \frac{\tilde{\mathbf{p}}'_z}{\tilde{\mathbf{d}}'_z}\tilde{\mathbf{d}}',\ \ \mathbf{f} = -\frac{\tilde{\mathbf{d}}'}{\tilde{\mathbf{d}}'_z}\,, \tag{4.17}$$

where $\mathbf{b}$ is projected along $\tilde{\mathbf{r}}(s)$ to time $z = 0$ and $\mathbf{f}$ is the direction from point a time $z = 0$ to the point at time $z = 1$. Recall that the z-axis in parameter space represents the temporal dimensionality. Figure 4.4 illustrates this conversion. The integral along the 3D Gaussian visible in the time interval $[t_a, t_b]$ becomes

$$\begin{aligned}\int_{t_a}^{t_b} \mathcal{G}^3(\mathbf{l}(t)dt &= \int_{t_a}^{t_b} e^{-\delta\,\mathbf{l}^{\mathrm{T}}(t)\mathbf{l}(t)}dt \\ &= \int_{t_a}^{t_b} e^{-\delta\,(\mathbf{b}^{\mathrm{T}}\mathbf{b}+2\mathbf{b}^{\mathrm{T}}\mathbf{f}\,t+\mathbf{f}^{\mathrm{T}}\mathbf{f}\,t^2)} \\ &= \int_{t_a}^{t_b} \underbrace{e^{-\delta\,\mathbf{l}_{xy}^{\mathrm{T}}(t)\mathbf{l}_{xy}(t)}}_{\text{Spatial}} \cdot \underbrace{e^{-\delta\,t^2}}_{\text{Temporal}}\ dt\,,\end{aligned} \tag{4.18}$$

where the normalization of the Gaussian is performed implicitly by the transformation from the viewing ray to the integration ray. The solution for the finite integral is given as

$$\begin{aligned}\int_{t_a}^{t_b} \mathcal{G}^3(\mathbf{l}(t))dt = &\frac{\sqrt{\pi}e^{-\frac{\delta(\mathbf{b}^{\mathrm{T}}\mathbf{b}\mathbf{f}^{\mathrm{T}}\mathbf{f}-(\mathbf{b}^{\mathrm{T}}\mathbf{f})^2)}{\mathbf{f}^{\mathrm{T}}\mathbf{f}}}{2\sqrt{\delta\mathbf{f}^{\mathrm{T}}\mathbf{f}}} \\ &\cdot\left(\mathrm{erf}(f(t_b)) - \mathrm{erf}(f(t_a))\right),\end{aligned} \tag{4.19}$$

where

$$f(x) = \sqrt{\frac{\delta}{\mathbf{f}^{\mathrm{T}}\mathbf{f}}}\left(\mathbf{f}^{\mathrm{T}}\mathbf{f}\cdot x + \mathbf{b}^{\mathrm{T}}\mathbf{f}\right),\ \ \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt\,. \tag{4.20}$$

The Gauss error function $\mathrm{erf}(x)$ cannot be computed analytically and will be stored as a look-up table for performance reasons.

Note that a closed solution only exists for the infinite integral, and will be given for completeness:

$$\int_{-\infty}^{\infty} \mathcal{G}_I^3(i(t)dt \;=\; \frac{\sqrt{\pi}e^{-\frac{\delta(\mathbf{b}^{\mathrm{T}}\mathbf{b}\mathbf{f}^{\mathrm{T}}\mathbf{f}-(\mathbf{b}^{\mathrm{T}}\mathbf{f})^2)}{\mathbf{f}^{\mathrm{T}}\mathbf{f}}}{\sqrt{\delta\mathbf{f}^{\mathrm{T}}\mathbf{f}}}. \tag{4.21}$$

However, due to the visibility function the bounded integral in Equation (4.19) needs to be evaluated for correct results.

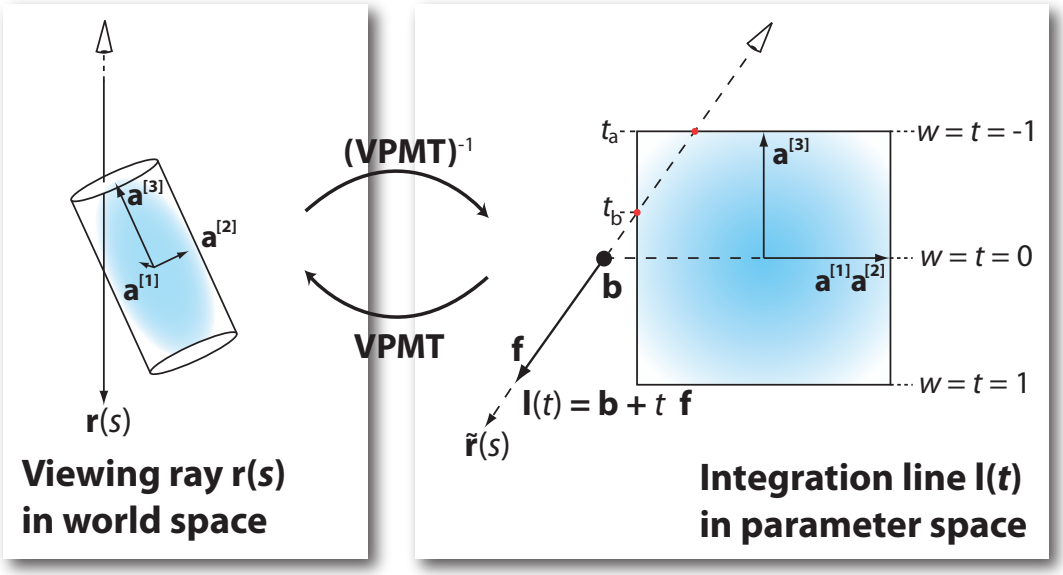


Figure 4.4: *To evaluate the line integral along the viewing ray* $\mathbf{r}(s)$ *it is first transformed to the parameter space viewing ray* $\tilde{\mathbf{r}}(s)$ *and finally normalized to the integration line* $\mathbf{l}(t)$*. The integration line, again, is normalized to the time interval* $[-1,1]$ *which represents the total integration period. The integral is evaluated in the time interval* $[t_a, t_b]$ *in which the kernel is visible.*

4.3.3 Bounding volume restriction

Theoretically the Gaussian reconstruction filters need to be evaluated on the whole image plane. Nevertheless, the Gaussian decays very fast and rays through pixel locations farther than a certain cut-off distance from the projected kernel center have negligible contributions to the image. To simplify the computations for the visibility we therefore bound the evaluation of the line integral by bounding the Gaussian filter with a 3D tube.

The tube is defined as $\mathbf{x}^{\mathrm{T}}\mathbf{T}^{-\mathrm{T}}\mathbf{D}_{xy}\mathbf{T}^{-1}\mathbf{x} = 0$ in object space, where $\mathbf{D}_{xy} = \mathrm{diag}(1,1,0,-1)$ is a diagonal 4x4 matrix and T is identical to the variance matrix of the ellipsoid in Equation (4.14). Intersection points between a ray and the tube are then determined by inserting Equation (4.15) into this relation and solving the quadratic equation

$$\mathbf{r}(s)^{\mathrm{T}}(\mathbf{VPMT})^{-\mathrm{T}}\mathbf{D}_{xy}(\mathbf{VPMT})^{-1}\mathbf{r}(s) = \tilde{\mathbf{r}}_{xy}(s)^{\mathrm{T}}\tilde{\mathbf{r}}_{xy}(s) = 0\,. \tag{4.22}$$

In addition to the quadratic equation we bound the tube by the two ellipses lying on the cut-off planes $\tilde{\mathbf{r}}_z(s) = \pm 1$ and arrive at the bounding cylinder.

In the same way as the ellipsoid volume is constrained in object space, its extent in screen space can be bounded by the projection of the cylinder. The bounding box computation of the latter can be performed similarly to Sigg et al. [SWBG06b]: First, the two axis-aligned bounding boxes of the bounding ellipses are computed. Then the convex hull polygon of these bounding boxes is computed.

The exact derivation for the bounding box computation of the bounding ellipses can be found in Section 5.4.1 of the following chapter. The convex hull polygon of the two bounding boxes can then be determined efficiently by comparing each of the four corresponding corner pairs separately. For each pair the following simple relation holds: If one of the two vertices lies in the inside quadrant the inside vertex is discarded. The inside quadrant is defined by the two lines of the bounding box adjacent to the vertex. In case the vertex is not in the inside quadrant both vertices are connected. This simple algorithm can be organized in a way that it yields a sorted list of points defining the convex hull and can be directly used as an output for a triangle strip by interleaving the vertex order.

The following Listing 4.1 shows the pseudo-code for one of the four quadrant cases, the other three cases can be solved accordingly. By careful ordering of the cases, a point list defining a convex polygon is produced. See Figure 4.5 for an illustration of the algorithm for all four cases. The numerically stable and efficient method to compute the bounding boxes of the bounding ellipses can be found later in Section 5.4.1 when the hardware architecture for surface splatting is presented.

Listing 4.1: *Bounding polygon computation example for quadrant I.*

```
Vector2 ul1; // Upper left corner of bounding box 1
Vector2 ul2; // Upper left corner of bounding box 2

float d = (ul1 .x− ul2.x) ∗ (ul1.y − ul2.y)
float min = MIN(ul1, ul2); // Component wise minimum
float max = MAX(ul1, ul2); // Component wise maximum

if (d < 0) { // Outside quadrant case
    return Vertex2D(min.x, max.y) and Vertex2D(max.y, min.x);
} else { // Inside quadrant case
    return Vertex2D(min.x, min.y);
}
```

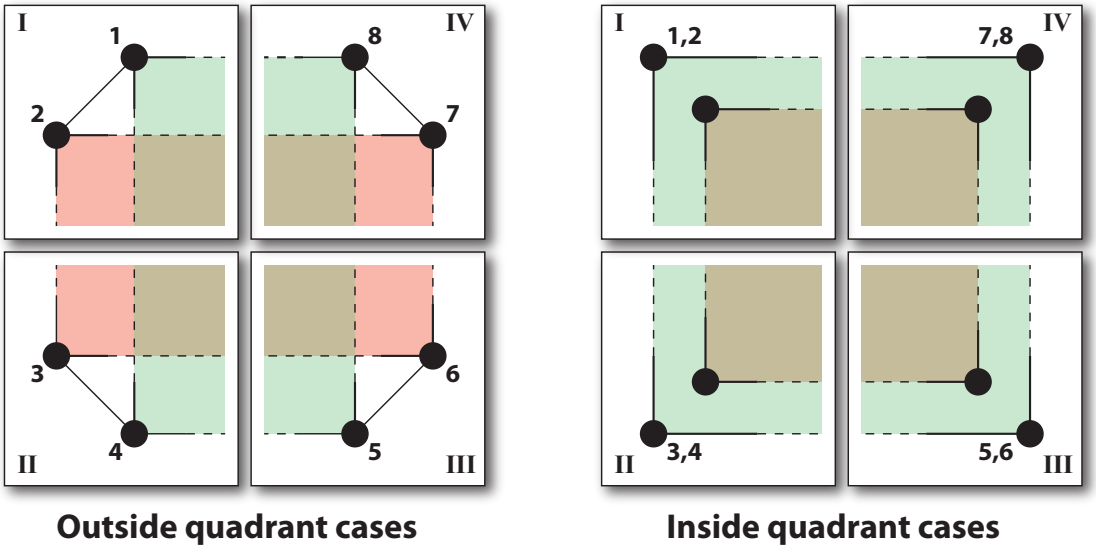


Figure 4.5: *Convex hull of two axis-aligned screen space bounding boxes. The algorithm compares the corresponding four corners of the boxes in counter-clockwise order. If none of the two corresponding corners lie within the inside quadrant of the other corner, both corners belong to the convex hull (left). Otherwise the corner located in the inside quadrant is discarded (right).*

4.3.4 Visibility

The visibility of a surface point is a discontinuous, non-bandlimited function. The optimal strategy to solve the visibility problem for motion-blurred EWA surface splatting in terms of visual quality is similar to the approach proposed by [KB83]. As a result of the integration of all kernels along the viewing rays, the time intervals in which each single kernel is visible at a pixel are known. These integration intervals $[t_a^n, t_b^n]$ are furthermore associated with a depth interval $[d_a^n, d_b^n]$, resembling the sorted lists of the A-Buffer method [Car84].

Our adopted A-Buffer algorithm iterates over the intervals starting with the earliest time interval. All subsequent time intervals are compared for temporal and depth overlap using a ternary depth test commonly used in EWA surface splatting. In case both a temporal and depth overlap is detected, the intervals belong to reconstruction kernels associated with the same surface. Therefore, they are weighted and accumulated to the current pixel value and finally normalized. If there is only a temporal overlap present, the depth interval of the later time interval is adjusted accordingly by removing the occluded interval from it. After all intervals have been processed, the visibility functions $v_k(\mathbf{x}, t)$ of all reconstruction kernels contributing to the pixel are known. The final pixel value is calculated based on Equation (4.2)

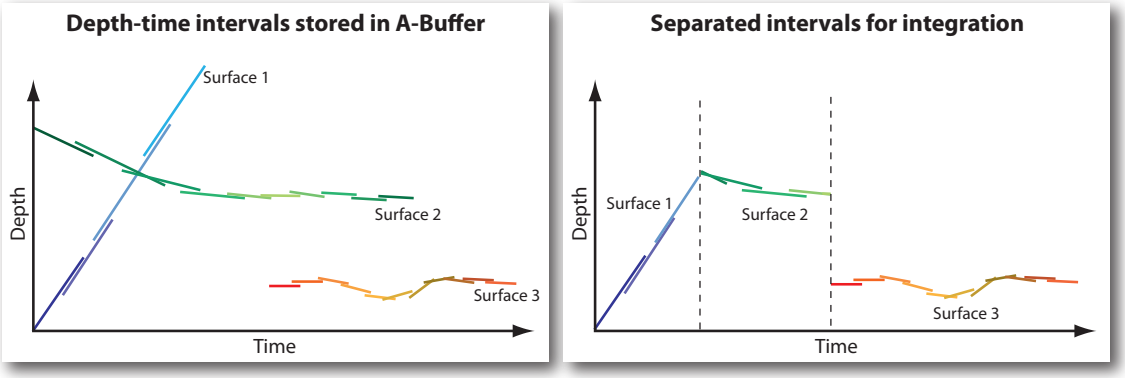


Figure 4.6: *The adopted A-Buffer algorithm for visibility culling. The A-Buffer contains the time-depth intervals of all splats passing through a pixel in the integration period, as shown for one pixel in the left graph. Our visibility culling algorithm determines whether splats overlap in depth and in time, and subsequently clips hidden splats that do not overlap in depth for a given time interval, as shown on the right graph. The resulting three intervals are integrated and normalized separately before the final pixel value can be calculated based on Equation (4.2).*

by summing up all contributions weighted with the time-dependent shutter function $a(t)$. Figure 4.6 shows an illustration of this algorithm for one pixel.

The visibility algorithm as described above could be implemented efficiently on a ray-tracer which naturally traverses the objects in a roughly sorted depth order. Section 4.4 presents an approximative strategy to solve the visibility problem on modern GPUs.

4.3.5 Discussion

This rendering framework is able to produce realistic motion blur by extending the EWA surface splatting framework in its theoretical basis. Some results are shown in Figures 4.9a and 4.10a. Our method determines the temporal visibility of the ellipsoidal reconstruction kernels and is able to render motion-blurred scenes using accurate surface contributions dependent on their visibility.

Similar to our work, the method of [GM04] also constructs three-dimensional reconstruction kernels by convolving the two-dimensional object space ellipses with a low-pass filter along the motion direction. In a subsequent step, the reconstruction kernels are projected to screen space ellipses and used for rendering.

However, our framework and its rendering algorithm differ substantially from [GM04]. Their work combines a static image of the scene combined with blurred motion hint ellipses. The static scene is generated using the original EWA surface splatting algorithm, whereas the blurred motion hint ellipses are rendered using a modified technique. Furthermore, their work neglects the temporal visibility and cannot separate the spatial contributions of the motion ellipses and consequently their opacity is not accurate. The results of this limitation are the striping artifacts and disturbing over-blending in areas with high overlap of motion ellipses. Therefore it cannot reproduce photo-realistic motion blur as our approach.

4.4 GPU implementation

The GPU implementation approximates the rendering framework using multiple render passes as depicted in Figure 4.7:

1. The first pass renders elliptical splats at the start time of the integration period into the depth buffer only. The result is used as an approximation for the subsequent depth tests.
2. The second pass renders elliptical splats at the end time of the integration period, similar to step 1.
3. The third pass approximates the visibility against static objects and the background by determining the earliest instant of time in which geometry is visible for every pixel.
4. The fourth pass determines the latest instant of time in which geometry is visible for every pixel, similar to step 3.
5. The fifth pass blends all visible reconstruction kernels into the accumulation buffer.
6. The sixth pass performs the normalization of the blended kernels.

The following sections provide details on these steps.

4.4.1 Visibility passes 1 and 2

The first two passes render the scene to the depth buffer only by making use of elliptical discs as rendering primitives. The first pass renders the scene at the start of the integration period, whereas the second pass renders the same scene transformed to the end of the integration period. Similar to [BHZK05],

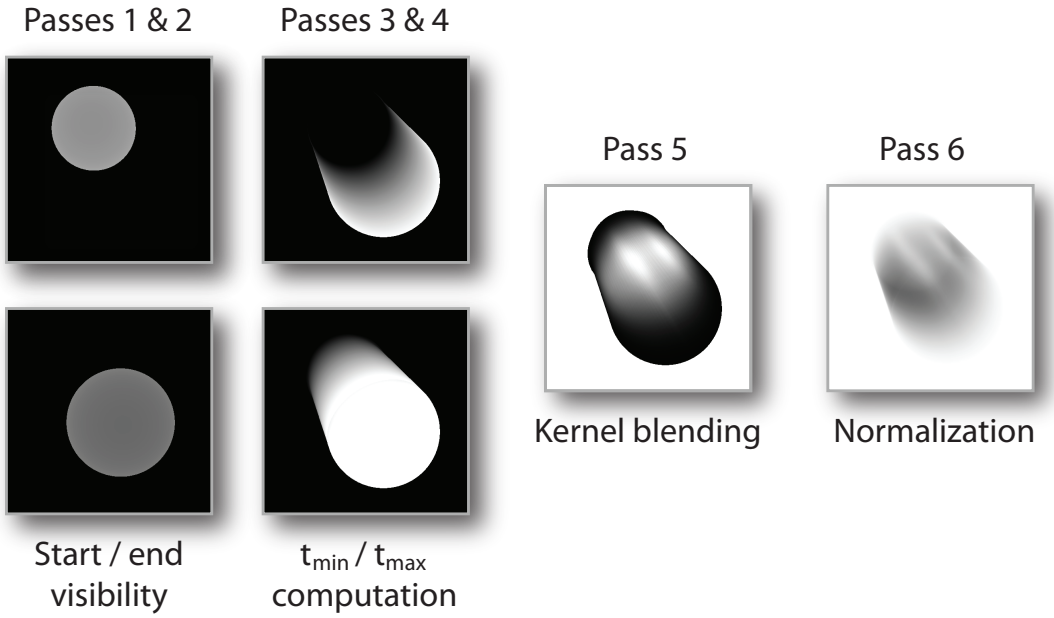


Figure 4.7: *The GPU implementation uses six passes to synthesize motion-blurred images. The first two passes approximate the visibility of the surface function at the start and end of the exposure time (left). The subsequent passes three and four approximate the visibility of the object in relation to the background by computing the per-pixel time intervals during which geometry is visible (middle left). After all reconstruction kernels have been blended (middle right), the accumulated values are normalized and the rendered object is composed with the background (right).*

the elliptical discs are rendered without blending to determine the visible surface at a single time instant for each pixel.

The resulting two depth buffers are then applied during the subsequent steps to approximate the kernel visibilities. The depth test works as follows. For every ellipsoid its minimum depth at the current fragment is determined and compared to the corresponding entries of both depth buffers. If the ellipsoid is visible in either one of the depth buffers the processing is continued, otherwise the rest of the computations for this fragment can be discarded.

4.4.2 Background visibility passes 3 and 4

In a next step we approximate the solution for the visibility problem with respect to the background or static objects by estimating the total fraction

of time in which any of the kernels is contributing to a pixel. The fifth pass reuses the same vertex and geometry shader as described here.

The vertex shader constructs the velocity vector based on the transformation matrices of the start and end frame. The normals of the 2D splats at the start and end frame are computed to perform back-face culling in case both normals are facing away from the camera. In a following step the screen space bandlimiting filter is approximated by means of ensuring that the projection of the axes equals at least the size of the filter radius. Based on the outcome of the filter the three axis vectors $\mathbf{a}^{[1]}, \mathbf{a}^{[2]}, \mathbf{a}^{[3]}$ are constructed. To avoid numerical issues during the transformation of the viewing ray we enforce a minimum length of the velocity axis $\mathbf{a}^{[3]}$ as well as a minimum angle between the $\mathbf{a}^{[1]}, \mathbf{a}^{[2]}$ plane and $\mathbf{a}^{[3]}$. Then, based on the axis vectors $\mathbf{a}^{[1]}, \mathbf{a}^{[2]}, \mathbf{a}^{[3]}$ and the kernel center $\mathbf{u}$, the transformation matrix $(\mathbf{PMT})^{-1}$ is constructed. Similar to [SWBG06b] we compute the two axis-aligned bounding boxes for the start and end frame ellipses. Instead of simply taking the combined axis-aligned bounding box, we send both bounding boxes to the geometry shader to construct a tighter bounding primitive.

The geometry shader combines the two axis-aligned bounding boxes to a convex octagon with the method described earlier in Section 4.3.3. The vertices of the octagon are computed in counter-clockwise order and some of the neighboring vertex pairs may consist of the same vertex. Accordingly to simplify the computations in the geometry shader we always output eight vertices for a triangle strip resulting in up to six triangles.

The fragment shader estimates the time intervals in which the ellipsoids are visible. The idea is to determine the earliest time instance t_{min} and the latest time instance t_{max} at which any reconstruction kernel is covering the pixel. The viewing ray is first transformed from camera space to parameter space and intersected with the cylinder. The z-component of the intersection points can directly be used as the integration interval $[t'_{min}, t'_{max}]$ of a single kernel. The depth test ensures that only the smaller or greater values, respectively, are written to the depth buffer. At the end of the rendering cycle the two depth buffers contain the earliest and the latest times per pixel when any of the kernels becomes visible.

Pass 6 later uses $[t_{min}, t_{max}]$ as an approximation of the correct time intervals.

4.4.3 Blending pass 5

In this pass the vertex and geometry program of the passes 3 and 4 are reused. The fragment shader transforms the viewing ray to the integration

ray and performs the integration. To evaluate $\exp(\mathbf{x})$ and $\mathrm{erf}(x)$ we utilize look-up tables which are stored in textures. As an approximation, all visible reconstruction kernels are blended using the integration weight of their whole integration period. To perform an accurate visibility test partial occlusions during the kernel integration period would have to be resolved as will be discussed later.

4.4.4 Normalization pass 6

In a final step the rendered image is normalized by the sum of the kernel weights. The resulting image is combined with the background where the blending weight is given by the integral $\int_{t_{\min}}^{t_{\max}} e^{-t^2} dt$ reusing the time interval information acquired in the fourth and fifth pass.

4.4.5 Sampling of the motion path

To support fast, non-linear movements we resample the motion path in the case of such movements. The motion path is split into equidistant time frames and all 6 passes are performed on the sampled sub-sets individually. The results of each iteration are accumulated into an accumulation texture, and each sample is weighted using the integral $\int_{t_{\mathrm{subframe}_i}}^{t_{\mathrm{subframe}_i+1}} e^{-t^2} dt$. Furthermore, a higher sampling of the motion path helps to reduce visible artifacts due to the approximation of the visibility function. This approach allows us therefore to trade rendering speed for correctness of the result. Note that this is similar to supersampling, however, a much lower amount of samples is generally needed.

4.5 Results and limitations

The GPU implementation is able to achieve high-quality pictures at competitive frame rates compared to the pure temporal supersampling approach. Figures 4.1 and 4.8 show some sample renderings and comparisons. Table 4.1 lists the corresponding temporal supersampling rates and performance figures for all examples in this paper. All results have been computed on an NVIDIA GeForce 280 GPU.

The crucial benefit of our approach is that the volumetric kernels adapt automatically to the speed of the motion. In cases where an object moves at varying speeds a pure temporal supersampling would usually sample the

Table 4.1: *Performance comparisons for the depicted examples, measured using an NVIDIA GTX 280. The supersampling factors 'x' have been chosen for similar visual quality.*

Figure	3D Kernels		Super sampling		Point
	x	Points/s	x	Points/s	count
4.1.a	4	1.80M	30	1.44M	350k
4.1.b	2	4.63M	25	2.91M	466k
4.8.a/4.8.e	4	1.91M	30	1.78M	554k
4.8.b/4.8.f	12	0.42M	40	0.46M	303k
4.8.c/4.8.g	12	0.63M	80	0.65M	310k
4.8.d/4.8.h	1	1.65M	60	0.50M	825k

complete animation at a constant rate nevertheless, whereas the volumetric kernels adapt implicitly.

As a limitation inter-object relationships cannot be handled correctly on the GPU because the visibility cannot be computed exactly, see Figure 4.9. The proposed approximation using the depth images is only able to exhibit a binary visibility function based on the visibility of the ellipsoids in the start and end frame. Additionally, artifacts may occur due to the approximated back-face culling where kernels may get falsely discarded or accepted.

The visibility passes for determining $t_{\min}$ and $t_{\max}$ try to approximate the total time during which any geometry is visible at a fragment. This information is used to blend the blurred image with the background. As only the earliest and latest time instants are determined, problems arise if, for example, geometry is only visible shortly at the beginning and the end of the time interval, see Figure 4.10. However, our approximation would classify the whole interval as being covered by kernels.

While objects moving at similar speed are blended in a plausible way, fast moving objects cannot be blended plausibly with slow moving objects due to the above mentioned approximations. Although our framework has been designed for generality, the visibility approximation limits the applicability of the GPU algorithm to scenes where different objects do not exhibit simultaneous overlaps in time and depth intervals. However, the artifacts which arise in this case can be alleviated by increasing the temporal supersampling rate along the motion trajectory which sacrifices rendering performance in favor of quality.

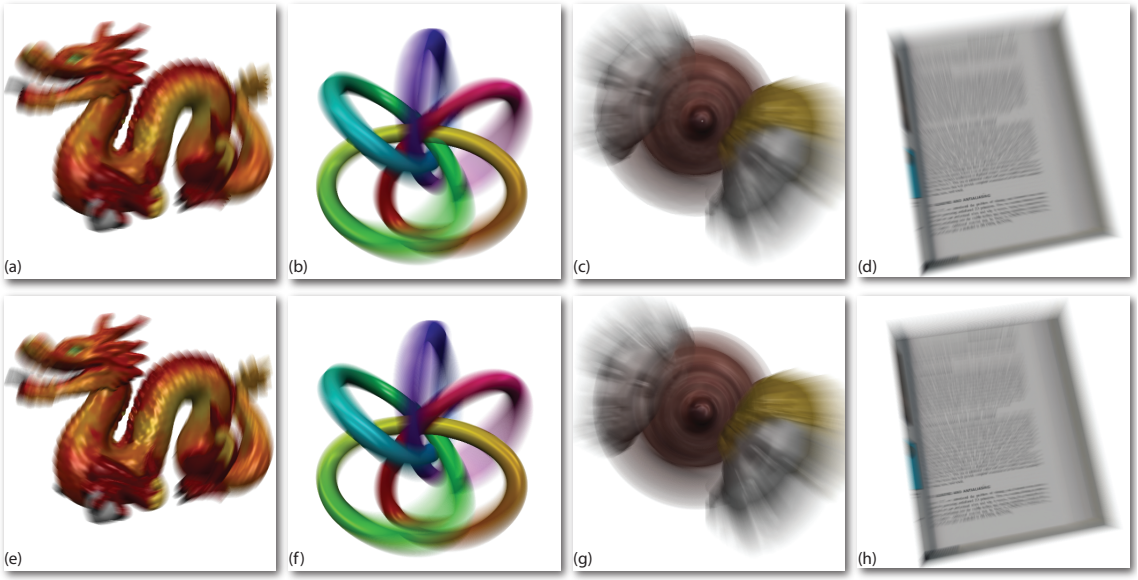


Figure 4.8: *Comparison of the result images rendered with our GPU implementation (a-d) and images rendered with supersampling of conventional EWA surface splatting framework (e-h), performance figures are given in Table 4.1. Figures (a,b): dragon is rotating along the depth axis. Figures (b,f): colored knot rotating around the up and depth axis. Figures (c,g): fast rotating face and Igea heads, the heads moving along depth axes and overlap the face. Figures (d,h): book moving towards the camera.*

4.6 Conclusion

In this chapter we have introduced a new method for rendering dynamic point-based geometry with motion blur effects based on EWA surface splatting. We have extended the theoretical basis of the EWA framework in a conceptually elegant and consistent way and consequently provided a mathematical representation of motion-blurred images with point-sampled geometry. Our method replaces the 2D surface splats of the original EWA surface splatting framework by 3D kernels which unify a spatial and temporal component. This allows for a continuous reconstruction of the scene in space as well as time and the motion-blurred image is computed as a weighted sum of warped and bandlimited kernels. Accordingly, we have described an approximation of the theoretical algorithm by means of a six-pass GPU rendering algorithm. Our point rendering pipeline applies ellipsoids with spatial and temporal dimensionality as new rendering primitives and exploits vertex, geometry and fragment program capability of current GPUs. Our framework for point splatting maintains generality by avoiding any assumptions or constraints on the nature of the captured scene such as motion or

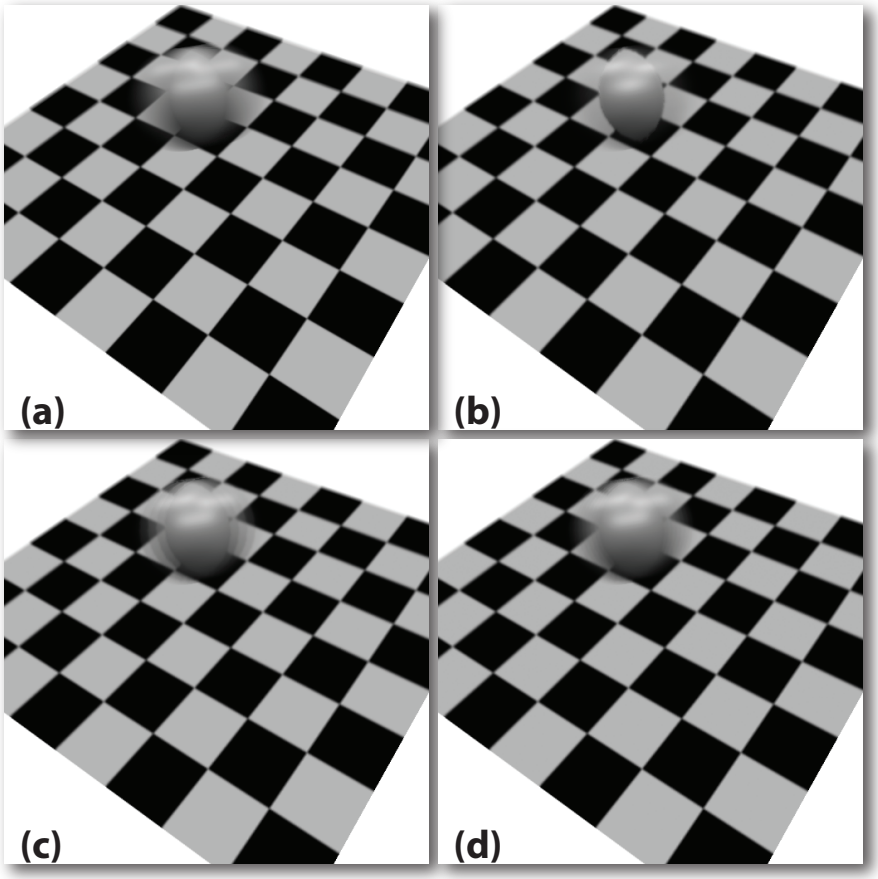


Figure 4.9: *Inter-object relations can only be handled using a higher sampling due to the visibility approximation on the GPU. Figure (a) A-Buffer software implementation. GPU implementation: (b) 1 sample, (c) 4 samples, (d) 8 samples.*

lighting.

As result, we identified several shortcomings of todays graphics architectures. First, a tertiary depth test deciding whether surface splats occlude, overlap or are hidden can only be approximated using the binary depth test of current GPUs. Therefore, multiple passes including a normalization pass are needed to render even static point-sampled scenes. Second, the absence of a more configurable framebuffer to support A-buffer implementations to solve for visibility requires additional multiple passes and has a significant impact on the quality vs. speed tradeoff. Only a very recent paper by Yang et al. [YHGT10] showed how linked lists can be constructed in real-time on newest, state-of-the-art DirectX 11 GPUs. Their approach seems very promising to support our A-buffer data structure and to achieve higher visual quality at faster rendering speeds for EWA motion blur.

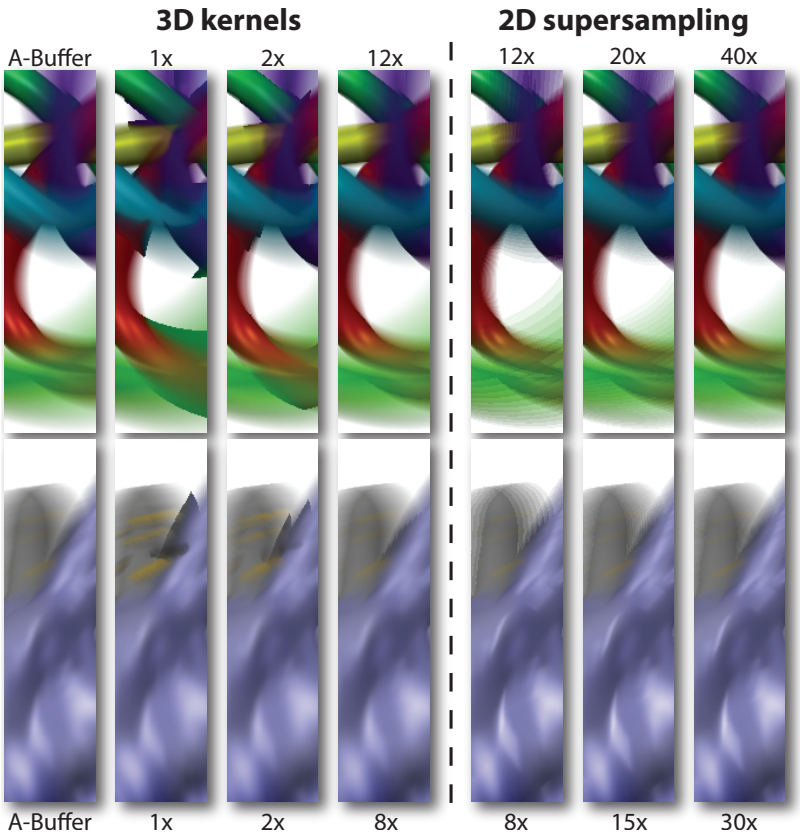


Figure 4.10: *Left: 3D kernels rendered with the A-Buffer software implementation and with different sampling rates for the GPU implementation. Right: 2D kernel supersampling with varying sampling rate. The visibility approximation for the GPU produces artifacts when geometry has been visible only shortly once at the beginning and once at the end of the integration interval. As a solution, the number of temporal samples can be increased. The A-Buffer implementation does not suffer from those artifacts.*

In the following chapter we will revisit point-based rendering and its differences to traditional triangle-based rendering again, with focus on hardware architectures. Based on the analysis in this chapter we will present a novel hardware architecture for point-based rendering that complements triangle rendering instead of replacing it.

CHAPTER 5

Hardware architecture for surface splatting

In this chapter, we present a novel architecture for hardware-accelerated rendering of point primitives. The architecture combines the advantages of triangles and point-based representations. Our pipeline focuses on a refined version of the elliptical weighted average (EWA) surface splatting for static scenes, and alleviates some of the bottlenecks analyzed and presented in the previous chapter. A central feature of our design is the seamless integration of the architecture into conventional, OpenGL-like graphics pipelines so as to complement triangle-based rendering.

The specific properties of the EWA splatting algorithm required a variety of novel design concepts including a ternary depth test and using an on-chip pipelined heap data structure for making the memory accesses of splat primitives more coherent. In addition, we develop a computationally stable evaluation scheme for perspectively corrected splats. We implement our architecture both on reconfigurable FPGA boards as well as on ASIC prototypes, and we integrate it into an OpenGL-like software implementation. Finally, we present a detailed performance analysis using scenes of various complexity to show that surface splatting is suited for hardware implementations.

5.1 Overview

The rendering pipeline used for our hardware architecture is based on a stream-lined variant of EWA surface splatting for static scenes which has been introduced in Chapter 3. The design philosophy of the presented hardware architecture is motivated by the desire to seamlessly integrate high quality point rendering into modern graphics architectures to complement their strengths and features. The specific properties of EWA rendering required a variety of novel concepts and make our design different from conventional graphics architectures. For instance, multiple splats belonging to one surface have to be accumulated all before the final surface can be displayed. Also, splats belonging to different surfaces have to be separated from each other. Due to these differences, previous implementations of EWA surface splatting required multiple GPU passes and could not keep up with the performance of triangle based rendering, which is especially true for our extended version of EWA splatting for motion blur presented in Chapter 4.

We will show that our architecture addresses the issues for stationary scenes and renders splats efficiently in single pass. Furthermore, using deferred shading of the splat surfaces allows us to apply virtually any fragment shading program to the reconstructed surface. Some of the central novel features of our design include a constant-throughput pipelined heap data structure on chip for presorting of the splat primitives and to obtain better cache coherence. In addition, we have developed a ternary depth test unit for surface separation. We also present a novel, perspectively corrected splat setup which is faster than previous methods and has significantly improved numerical stability.

We implemented two prototype versions of our architecture on Field-Programmable Gate Arrays (FPGA) and Application-Specific Integrated Circuit (ASIC), and we also designed a graphics board for the latter, see Figure 5.1. Furthermore, we demonstrate the integration into existing pipelines in Mesa 3D, an OpenGL-like software implementation [Mes]. The analysis of our research prototypes shows that the proposed architecture is well suited for integration into conventional GPUs, providing high efficiency, scalability, and generality, and we achieved a theoretical peak performance of over 17.5 million splats/sec while operating at a comparably slow clock frequency of 70 MHz on the FPGA implementation.

The rest of this chapter is organized as follows. We will briefly revisit the performance issues regarding implementations of the EWA surface splatting framework on GPUs to motivate our design in Section 5.2, and give a comparison to traditional triangle rendering. A design overview is given

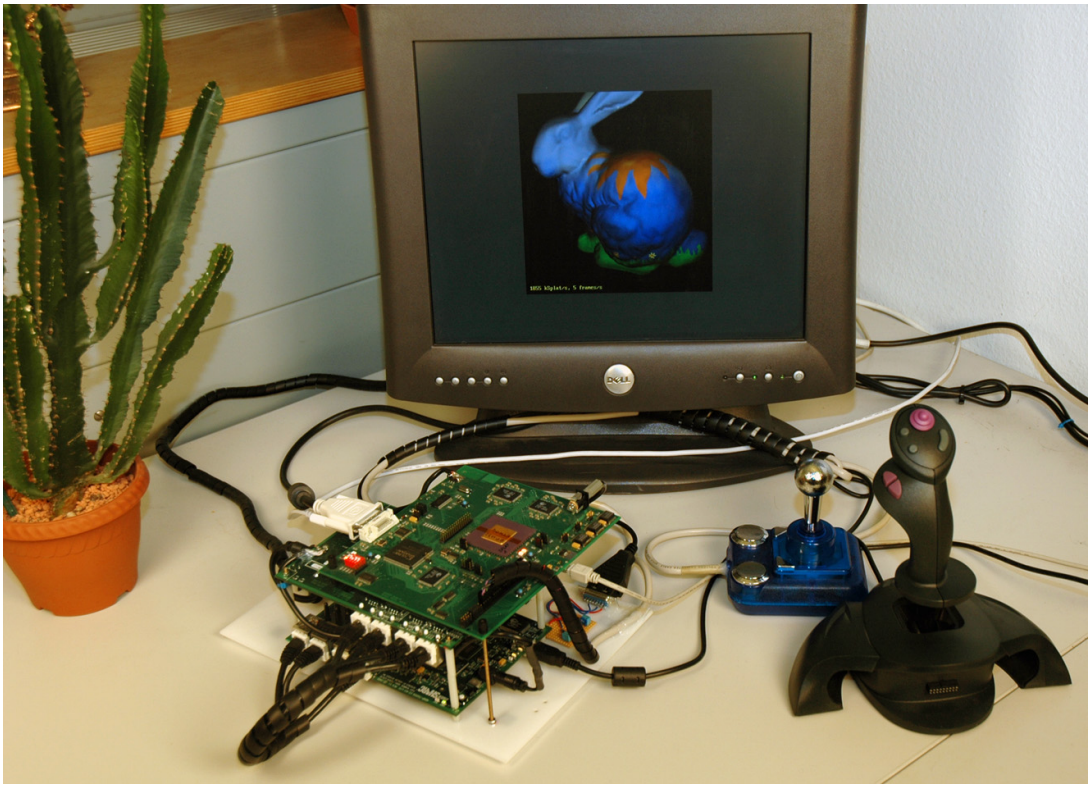


Figure 5.1: *Point rendering system featuring an ASIC implementation of our point rendering architecture. The system runs stand-alone, independent from a host PC.*

in Section 5.3. The rendering pipeline used for our hardware architecture is then outlined in Section 5.4. Our algorithms and hardware architecture are described in Section 5.5. The characteristics of our FPGA, ASIC, and Mesa implementations are outlined (Section 5.6), and the seamless integration of the new functionality into existing APIs is covered in Section 5.6.3. Finally, Section 5.7 provides test results, and Section 5.8 discusses the limitations and weaknesses of the current design and lists potential future work.

5.2 Performance of EWA surface splatting on current GPUs

While the EWA splatting framework (Chapter 3) can be implemented on modern programmable GPUs, the result is a relatively inefficient multi-pass algorithm. Based on the analysis presented in Chapter 4, the following performance issues can be identified in particular:

- There is no dedicated rasterizer unit, which would efficiently traverse the bounding rectangle of a splat and identify the pixels the splat

overlaps. As a result, this stage has to be implemented as a fragment shader program.

- Accurate accumulation and normalization of attributes cannot be done in a single pass due to the lack of necessary blending modes.
- The depth values must be accumulated and normalized exactly like other attributes, and thus a normalizing depth test hardware unit would be required for single-pass rendering. Such a unit has to support the ternary depth test (pass, fail, accumulate) of EWA splatting.
- The attribute accumulation imposes a heavy burden on frame buffer caches due to the overlap of splat kernels, and current caches may not be optimal for the task.

We will address these issues in the following sections and present our novel hardware architecture for EWA splatting. The design focuses on stationary two-dimensional splats for the ease of implementation. Most of the introduced concepts, however, are also applicable to the extension of the EWA surface splatting algorithm for motion blur.

5.3 Design overview

Our hardware architecture aims to complement the existing triangle rendering functionality with EWA splats, and make maximum re-use of existing hardware units of current GPUs.

The new algorithms used by our splat rasterization unit are described in Sections 5.4.1 and 5.4.2. In order to provide maximum performance and flexibility, we designed the pipeline to render EWA splats in a single pass. For that, a ternary depth test (Section 5.4.3) and extended blending functionality (Sections 5.4.4 and 5.4.5) are needed.

In terms of integration into existing GPUs, a particular challenge is that the execution model of splatting is different from triangle rasterization. While individual triangles represent pieces of surfaces, in splatting a part of the surface is properly reconstructed only after all the contributing splats have been accumulated and the sum has been normalized (this also concerns depth values). We achieve this by routing the splats through a frame buffer-sized *surface reconstruction buffer* before proceeding with fragment shading, tests, and frame buffer blending using the existing units. In effect, this architectural detail implements deferred shading [DWS+88, BHZK05] for splats, and as a result any fragment shading program can be used.

Figure 5.2 illustrates the overall design, which will be detailed in the following sections.

We chose to implement a custom reconstruction buffer due to performance reasons. An efficient implementation needs double buffering, fast clears, tracking of dirty pixels, high-performance accumulation of pixels, and direct feeding of normalized dirty fragments to the shader. We omitted support for A-buffer like structures for simplicity and without loss of generality. Note that if our pipeline was embedded into a GPU that already supported this functionality for off-screen surfaces and A-buffer structures, those resources could be reused.

In order to significantly improve the caching efficiency over typical frame buffer caches, we propose a novel reordering stage in Section 5.5.2. This is a crucial improvement because the accesses to the surface reconstruction buffer may require a substantial memory bandwidth if the on-chip caches are not working well.

5.4 Rendering pipeline

In this section, we will outline our modified EWA surface splatting pipeline optimized for hardware implementations.

A splat is defined by its center $\mathbf{u}$, and two tangential vectors $\mathbf{a}^{[1]}$ and $\mathbf{a}^{[2]}$, as illustrated in Figure 3.1. The tangential vectors span the splat's local coordinate system that carries a Gaussian reconstruction kernel[1]. $\mathbf{a}^{[1]}$ and $\mathbf{a}^{[2]}$ may be skewed to allow for the direct deformation of splat geometry [PKKG03]. In addition, each splat has a variable-length attribute vector $\mathbf{s}$ that contains surface attributes to be used by vertex, tessellation, geometry and fragment shading units.

The splat transform and lighting computations are similar to vertex processing, and thus the existing vertex shader units can be re-used. Table 5.1 lists the computations required in various stages of our EWA surface splatting pipeline. The corresponding triangle pipeline[2] operations are shown for comparison purposes only, and may not exactly match any particular GPU.

[1]Gaussians are widely used because they lead to simple closed form formulas. Alternatives such as low-degree B-spline filters might produce sharper images while being somewhat more costly to evaluate.

[2]This triangle rasterization code uses edge functions to obtain affine barycentric coordinates that are used for a pixel-inside-triangle test and depth interpolation. The barycentric coordinates are then perspective corrected, after which any vertex parameter can be interpolated to the current pixel by using them as weights.

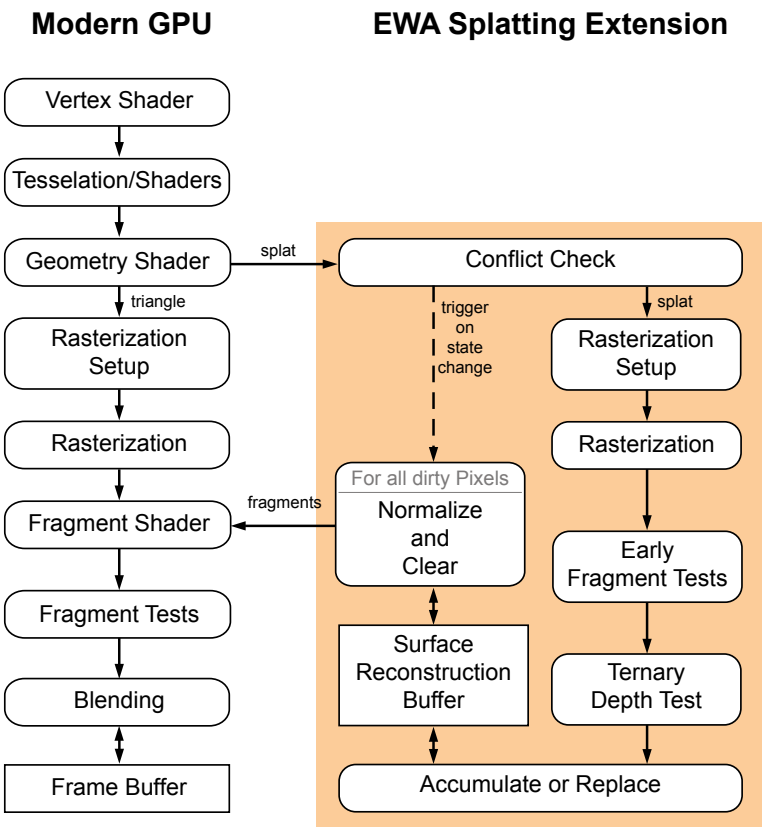


Figure 5.2: *The integration of EWA surface splatting into a conventional graphics pipeline. For splat primitives the triangle rasterization is bypassed, all other existing resources are reused. The key element is the surface reconstruction buffer. On state changes, readily reconstructed fragments are fed back into the traditional graphics pipeline.*

Both the triangle and splat codes could be somewhat simplified, but are shown in this form for notational convenience.

The remainder of this section describes the rest of the proposed EWA splatting pipeline in detail.

5.4.1 Rasterization setup

The rasterization setup computes per-splat variables for the subsequent rasterization unit, which then in turn identifies the pixels overlapped by a projected splat. The triangle pipeline includes specialized units that perform the same tasks for triangles, although the exact computations are quite different. For

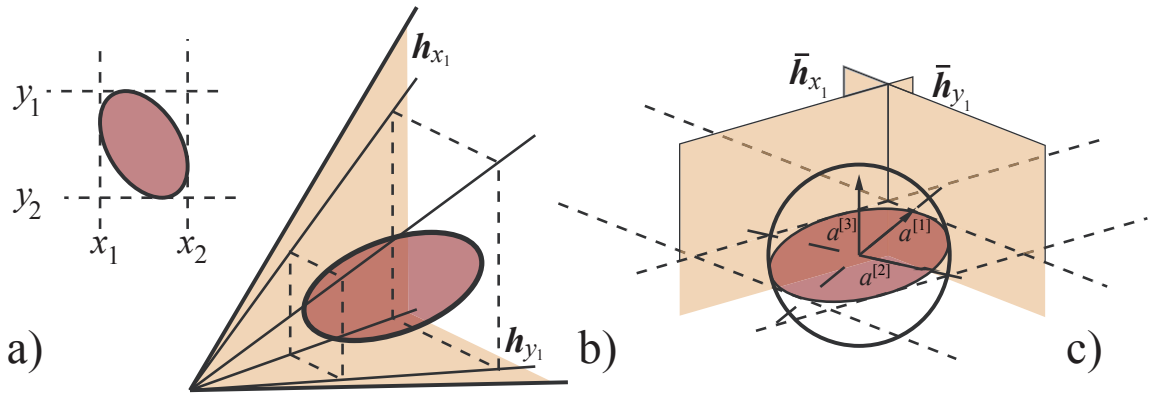


Figure 5.3: *Splat bounding box computation. The screen space bounding rectangle (a) corresponds to the splat's bounding frustum in camera space (b). Transformation into local splat coordinates maps the frustum planes to tangential planes of the unit sphere (c).*

the splat rasterization to be efficient, the setup unit needs to provide a tight axis-aligned bounding rectangle for the projected splat. Additionally, our ternary depth test needs to know the splat's depth range.

Previous bounding rectangle computations for perspective accurate splats have used either a centralized conic representation of the splat [ZRB+04] or relied on fairly conservative approximations. According to our simulations, the method of Botsch et al. [BSK04] overestimates the bounding rectangle by an average factor of four, as it is limited to square bounding boxes leading to inefficient rasterization. Gumhold's [Gum03] computation does not provide axis-aligned bounding boxes, which would result in setup and rasterization costs similar to triangle rasterization. The conic-based approach suffers from severe numerical instabilities near object silhouettes, and its setup cost is also rather high. Especially when the rendering primitives are tiny, it is beneficial to reduce the setup cost at the expense of increased per-fragment cost [MBDM97].

We developed a new algorithm for computing tight bounds without the need for matrix inversions and a conic centralization, i.e., without the operations that make the conic-based approach unstable. Moreover, unlike some of the previous approaches, our technique remains stable also for near-degenerate splats.

Our approach works on $\mathbf{u}'$, $\mathbf{a}^{[1]\prime}$, and $\mathbf{a}^{[2]\prime}$ in camera space and computes the axis-aligned bounding rectangle under a projective transform P. The key idea is to treat the splat as a degenerate ellipsoid: points on the splat correspond to the set of points within the unit sphere in the local splat coordinate system

spanned by $\mathbf{a}^{[1]\prime}$, $\mathbf{a}^{[2]\prime}$, and $\mathbf{a}^{[3]\prime}$, where $\mathbf{a}^{[3]\prime}$ is reduced to zero length. The splat in clip space is built by this unit sphere under the projective mapping PS with

$$S = \begin{pmatrix} \mathbf{a}^{[1]\prime} & \mathbf{a}^{[2]\prime} & \mathbf{0} & \mathbf{u}' \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{5.1}$$

where S maps point within the system of the unit sphere into the world coordinate system.

Finding the bounding rectangle coordinates x_1, x_2, y_1, and y_2 in clip space (see Figure 5.3a), corresponds to finding the planes that are adjacent to the splat in clip space (Figure 5.3b):

$$\mathbf{p}^\top \cdot \mathbf{h}_x = \mathbf{p}^\top \cdot (-1,0,0,x)^\top = 0 \tag{5.2}$$

$$\mathbf{p}^\top \cdot \mathbf{h}_y = \mathbf{p}^\top \cdot (0,-1,0,y)^\top = 0\,, \tag{5.3}$$

with $\mathbf{p} \in \mathbb{R}^4$ denoting a point on the plane. Mapping these planes to the local splat coordinate system by the inverse transformation $(PS)^{-1}$ then yields

$$\bar{\mathbf{h}}_x = (PS)^\top \mathbf{h}_x \tag{5.4}$$

$$\bar{\mathbf{h}}_y = (PS)^\top \mathbf{h}_y\,. \tag{5.5}$$

The transformed planes then define the tangential planes to the unit sphere in $(\mathbf{a}^{[1]\prime}, \mathbf{a}^{[2]\prime}, \mathbf{a}^{[3]\prime})$ (see Figure 5.3c). Hence, the splat's bounding rectangle can be determined by solving for so that $\bar{\mathbf{h}}_x$ and $\bar{\mathbf{h}}_y$ have unit-distances to the origin. For homogeneous coordinates, unit-distance to the origin is given when following condition is fulfilled:

$$\bar{\mathbf{h}}_x^\top D\, \bar{\mathbf{h}}_x = 0 \tag{5.6}$$

$$\bar{\mathbf{h}}_y^\top D\, \bar{\mathbf{h}}_y = 0\,, \tag{5.7}$$

with $D = \mathrm{diag}(1,1,0,-1)$.

The resulting quadratic equations can then be solved for x_1, x_2 and for y_1, y_2. Analogously, the depth extent of the splat can be computed starting from $\mathbf{h}_z = (0,0,-1,z)^\top$.

Note that the transformation of the homogeneous normals needs to be performed with the inverse transpose of the desired transformation $(PS)^{-1}$, and therefore avoids explicit inversion of S (and P). This is the reason for stability in degenerate cases. The computation can be considered a special case of the quadric bounding box computation by Sigg et al. [SWBG06a]. This bounding box computation is also used in Section 4.3.3 to compute the bounding ellipses for the three-dimensional reconstruction kernels.

Table 5.1 shows the resulting computation for the rasterizer setup, providing $x_1, x_2 = p_1 \pm h_1$, and $y_1, y_2 = p_2 \pm h_2$. The third, degenerate column of S has been removed as an optimization.

5.4.2 Rasterization

Our rasterization unit traverses all sample positions within the bounding box and computes for each sample the intersection point $\mathbf{s}$ of the corresponding viewing ray and the object space plane defined by the splat. The sample lies inside the splat if the distance ρ from $\mathbf{s}$ to the splat's center is within a specified cutoff radius. For samples inside the splat, ρ is used to compute the fragment weight $w_f = k(\rho^2)$, with k denoting the reconstruction kernel function.

A conic-based rasterization [ZRB+04] was not used due to its high setup cost and numerical instabilities. Botsch et al. [BSK04] project each pixel to the splat's tangential frame. In order to be fast, the splats are defined by orthogonal tangential vectors that are scaled inversely to the splat extent. However, the reciprocal scaling leads to instabilities for near-degenerate splats, and orthogonality is a pre-condition that cannot be guaranteed in our case.

Section 4.3.2 presented a method to compute the contribution of three-dimensional reconstruction kernels analytically. Table 5.1 shows how the ray-splat intersection can be computed very efficiently in the case of two-dimensional reconstruction kernels. The viewing ray through the sample position (x, y) is represented as an intersection of two planes $(-1, 0, 0, x)^\top$ and $(0, -1, 0, y)^\top$ in clip space. The planes are transformed into the splat's coordinate system using the matrix T, as computed by the rasterization setup. The point $\mathbf{s}$ follows by intersecting the projected planes $\mathbf{k}$ and $\mathbf{l}$ with the splat plane. Due to the degeneracy of the splat's coordinate system, the computation of $\mathbf{s}$ and the distance to splat's origin ρ^2 requires very few operations.

We use the EWA screen space filter approximation by Botsch et al. [BHZK05]. This filter can easily be applied by clamping ρ^2 to $\min\{\rho^2, \|(x, y) - (c''_x, c''_y)\|^2\}$, and by limiting the bounding box extents h_i to a lower bound of one.

5.4.3 Ternary depth test

EWA surface splatting requires a ternary z-test during surface reconstruction. Rather than the two events *z-pass* and *z-fail*, it has to support three conditions

$$\begin{cases} \textit{z-fail}, & z_d < z_s - \varepsilon \\ \textit{z-pass}, & z_d > z_s + \varepsilon \\ \textit{z-blend}, & \text{otherwise} \end{cases} . \tag{5.8}$$

The additional case *z-blend* triggers the accumulation of overlapping splats. An ε-band around incoming splat fragments defines whether a fragment it belongs to the same surface patch and needs to be blended. The size of the ε is computed from the splat's depth range h_z as

$$\varepsilon = h_z \cdot \varepsilon_{\Delta z} + \varepsilon_{\text{bias}} , \tag{5.9}$$

in which the following two parameters offer further flexibility and control:

$\varepsilon_{\Delta z}$ Scales the depth extents of the splat, usually set to 1.0.

$\varepsilon_{\text{bias}}$ Accounts for numerical inaccuracies, comparable to glPolygonOffset's parameter *unit*.

The ternary depth test is used only for surface reconstruction and does not make the regular depth test obsolete. The integration into a triangle pipeline still requires the traditional depth test.

It remains to be mentioned that the software implementations described by Pfister et al. [PZvBG00] and Räsänen [RÖ2] use deep depth buffers to store additional information for the blending criterion, similar to the A-Buffer used in Chapter 4. However, in our tests the benefits did not justify the resulting big impact on additional storage costs and complexity of a deep depth buffer for EWA splatting without motion blur.

5.4.4 Attribute accumulation

Our pipeline supports two kinds of splat attributes: typical *continuous* attributes need to be blended, whereas *cardinal* ones (e.g., material identifiers) must not.

Unless the ternary depth test fails, all continuous attributes in $\mathbf{s}_s$ are weighted by the local kernel value w_f. If the test resulted in *z-pass*, weight w_d and attributes $\mathbf{s}_d$ currently stored in the reconstruction buffer are replaced by

$$(w_d, \mathbf{s}_d) \quad := \quad \begin{cases} (w_f, w_f \cdot \mathbf{s}_s), & \text{for continuous attributes} \\ (w_f, \mathbf{s}_s), & \text{for cardinal attributes} \end{cases} . \tag{5.10}$$

On *z-blend*, continuous attributes are blended to the reconstruction buffer:

$$(w_d, \mathbf{s}_d) \quad := \quad (w_d + w_f, \mathbf{s}_d + w_f \cdot \mathbf{s}_s)\,. \tag{5.11}$$

Cardinal attributes cannot be blended. Instead, the destination value depends on whether the incoming or stored weight is greater:

$$(w_d, \mathbf{s}_d) \quad := \quad \begin{cases} (w_f, \mathbf{s}_s), & w_d < w_f \\ (w_d, \mathbf{s}_d), & w_d \geq w_f \end{cases} \quad . \tag{5.12}$$

As a result of the maximum function, a splat's cardinal attributes are written to its screen space Voronoi cell with respect to the surrounding splats. (See also Hoff III et al. [HKL+99].)

Please note that the depth is accumulated in the same manner as the continuous attributes. As a consequence, the depth values of the accumulation buffer need to be normalized by the accumulated weight before the ternary depth test is evaluated.

5.4.5 Normalization

Once the accumulation is complete, all continuous attributes must be normalized with the accumulated weight before sending the fragments to the fragment shader units.

$$\mathbf{s}_f = \mathbf{s}_d / w_d. \tag{5.13}$$

A crucial point in our design is deciding when the accumulation is complete, and the fragment has to be released from the surface reconstruction buffer.

This is trivially the case when triangle data follows splats. However, detecting the separation between two splat surfaces is more involved. The accumulated splats define a surface only after *all* affecting splats have been processed, and thus the completion of a surface cannot be reliably deduced from individual splats or the accumulated values. Therefore our design needs to utilize higher-level signals. We currently consider the end of glDrawElements() and glDrawArrays(), and the glEnd() after rendering individual splats to indicate the completion of a surface. Other scenarios include changing a fragment shader program, texture pointers, render target, or the set of splat attributes to accumulate, i.e., whenever the subsequent fragment shading would become ambiguous.

Symbols	
$\mathbf{u}, \mathbf{n}, \mathbf{a}^{[1]}, \mathbf{a}^{[2]}$	Center, normal, and two tangent vectors
$\mathbf{u}', \mathbf{n}', \mathbf{a}^{[1]\prime}, \mathbf{a}^{[2]\prime}$	Transformed to camera space
$\mathbf{u}''$	Center projected to screen space
$\mathbf{s}_{s,v,f}$	Splat, vertex, and fragment attribute vectors
M, P	Model-view matrix, projection matrix
T	Transformation from clip space to canonical splat
$\mathbf{t}_i$	i-th column of T
$^\top, ^{-\top}$	Transpose, inverse transpose
$k(), w_f$	Reconstruction kernel function, $k()$ at a fragment
ρ	Distance from splat's origin in object space
A	Area of a triangle
$\langle\rangle, \times, \star$	Dot, cross, and component-wise product
$\mathrm{diag}(a,b,c,d)$	Diagonal matrix with entries a,b,c,d

Transform and lighting (using vertex shader)	
EWA Surface Splat	*Triangle (for each vertex)*
$\mathbf{u}'' = PM\mathbf{u}$	$\mathbf{u}'' = PM\mathbf{u}$
$\mathbf{u}' = M\mathbf{u}$	$\mathbf{u}' = M\mathbf{u}$
$\mathbf{a}^{[1]\prime} = M\mathbf{a}^{[1]}$, $\mathbf{a}^{[2]\prime} = M\mathbf{a}^{[2]}$	
$\tilde{\mathbf{n}}' = \mathbf{a}^{[1]\prime} \times \mathbf{a}^{[2]\prime}$	$\tilde{\mathbf{n}}' = M^{-\top}\mathbf{n}$
$\langle \mathbf{u}', \tilde{\mathbf{n}}' \rangle > 0\,? \Rightarrow \mathrm{back-face, kill}$	
$\mathbf{n}' = \tilde{\mathbf{n}}' / \lVert\tilde{\mathbf{n}}'\rVert$	$\mathbf{n}' = \tilde{\mathbf{n}}' / \lVert\tilde{\mathbf{n}}'\rVert$
$\mathbf{s}_s = \mathrm{lighting}(\mathbf{s}, \mathbf{u}', \mathbf{n}')$	$\mathbf{s}_v = \mathrm{lighting}(\mathbf{s}, \mathbf{u}', \mathbf{n}')$

Table 5.1: Top: *Symbols used for this derivation* Bottom left: *Our splat rendering pipeline.* Bottom right: *The corresponding triangle pipeline operations.*

5.4.6 Fragment shading and tests

Once the normalized fragments emerge from the reconstruction buffer, they are fed to fragment shader units. Basically this implements deferred shading for splat surfaces, and fragment shading programs can be executed exactly like they would be for fragments resulting from triangles. As a result the performance, applicability, and generality of splatting are significantly improved.

As a final step, the fragment tests and frame buffer blending are executed using existing units. As an optimization, similarly to current GPUs (see Section 2.3), some of the fragment tests can be executed already at the time of splat rasterization for early fragment culling.

Rasterization setup (fixed function)	
EWA Surface Splat, $i \in [1 \ldots 3]$	*Triangle*, $i \in [1 \ldots 3]$
$T = (P \begin{pmatrix} \mathbf{a}^{[1]\prime} & \mathbf{a}^{[2]\prime} & \mathbf{0} & \mathbf{u}' \\ 0 & 0 & 0 & 1 \end{pmatrix})^\top$ $D = \mathrm{diag}(1,1,0,-1)$ $d = \langle \mathbf{t}_4, D\mathbf{t}_4 \rangle$ Center of projected splat: $p_i = \frac{1}{d}\langle \mathbf{t}_i, D\mathbf{t}_4 \rangle$ Half extents of bounds: $h_i = \sqrt{p_i^2 - \frac{1}{d}\langle \mathbf{t}_i, D\mathbf{t}_i \rangle}$	$\mathbf{p}_i = (c''_{i_x}, c''_{i_y})$ $A = [(\mathbf{p}_3 - \mathbf{p}_1) \times (\mathbf{p}_2 - \mathbf{p}_1)]_z$ $A < 0\,? \Rightarrow \mathrm{back-face, kill}$ Edge functions: $\mathbf{N}_1 = (\mathbf{p}_{1_y} - \mathbf{p}_{2_y}, \mathbf{p}_{2_x} - \mathbf{p}_{1_x})$ $\mathbf{N}_2 = (\mathbf{p}_{2_y} - \mathbf{p}_{3_y}, \mathbf{p}_{3_x} - \mathbf{p}_{2_x})$ $\mathbf{N}_3 = (\mathbf{p}_{3_y} - \mathbf{p}_{1_y}, \mathbf{p}_{1_x} - \mathbf{p}_{3_x})$ $\mathbf{e}_i = (\mathbf{N}_i, -\langle \mathbf{N}_i, \mathbf{p}_1 \rangle) \in \mathbb{R}^{1\times 3}$ Bounding rectangle: $\mathbf{br}_{\min} = \min(\mathbf{p}_i)$ $\mathbf{br}_{\max} = \max(\mathbf{p}_i)$

Rasterization (fixed function)	
EWA Surface Splat $\forall (x,y) \in [c''_x \pm h_x, c''_y \pm h_y]$	*Triangle* $\forall (x,y) \in \mathbf{br}, i \in [1 \ldots 3]$
$\mathbf{k} = -\mathbf{t}_1 + x\mathbf{t}_4$ $\mathbf{l} = -\mathbf{t}_2 + y\mathbf{t}_4$ $\mathbf{s} = l_4(k_1,k_2) - k_4(l_1,l_2)$ Distance to splat's origin: $\rho^2 = \langle \mathbf{s}, \mathbf{s} \rangle / (k_1 l_2 - k_2 l_1)^2$ $\rho^2 > \mathrm{cutoff}^2\,? \Rightarrow \mathrm{outside, kill}$ $w_f = k(\rho^2)$	Barycentric coordinates: $b_i = \langle \mathbf{e}_i, (x,y,1) \rangle / A$ $b_i < 0\,? \Rightarrow \mathrm{outside, kill}$ Perspective correction: $c_i = b_i w_1 w_2 w_3 / w_i$ $r = \sum c_i$ $b'_i = c_i / r$ $\mathbf{s}_f = \sum b'_i \mathbf{s}_{v_i}$

Table 5.2: Left: *Our splat rendering pipeline.* Right: *The corresponding triangle pipeline operations are shown for comparison purposes. The computation of per-fragment attributes* $\mathbf{s}_f$ *for EWA splats is described in Sections 5.4.3–5.4.5.*

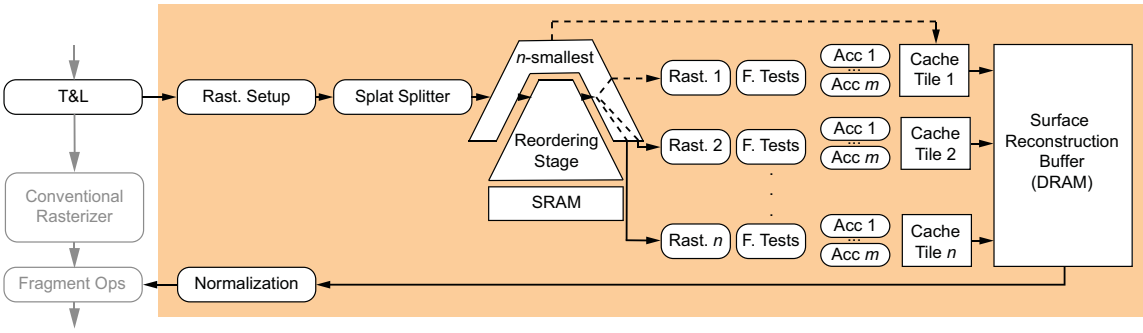


Figure 5.4: *Hardware architecture of the EWA surface splatting extension.*

5.5 Hardware architecture

This section describes the impact of the proposed pipeline on the hardware architecture. While parts of the operations, such as the transform and lighting stage, can be mapped to existing hardware components of a GPU, some of the concepts introduced require additional hardware units. Figure 5.4 provides an overview over the proposed hardware extension. The remainder discusses the extensions in more detail.

5.5.1 Rasterization setup and splat splitting

The rasterization setup was implemented as a fully pipelined fixed-function design, which computes the bounding rectangle of a splat along with the depth extents as well as the necessary matrix composition.

In order to simplify cache management, we further subdivide the bounding rectangle of a splat according to 8×8 pixel tile boundaries. This decision is consistent with modern frame buffer caches that work using $M \times M$ pixel tiles [Mor00], and thus allows us to maintain a direct mapping of rasterizers to cached tiles. Our tile-based reconstruction buffer cache shares the infrastructure of existing frame buffer caches.

The splat splitter unit clips the bounding rectangles against the boundaries of the 8×8 pixel tiles of the reconstruction buffer before reordering the for better cache coherence. This results in potentially multiple copies of a splat so that the copies differ only in their bounding rectangle parameters. Due to the small size of splats, the number of additionally generated splat copies is generally not significant.

5.5.2 Splat reordering

Motivation The small size of EWA splats along with the relatively high overdraw during accumulation put an exceptionally high burden on the reconstruction buffer cache. A careful ordering of the splats during a preprocess helps only to certain extent, and does not apply to procedural geometry or other dynamically constructed or dynamically transformed splats. When investigating the reconstruction buffer cache hit rates, we concluded that significant improvements can be achieved by introducing a new pre-rasterization cache that stores splats in spatially ordered manner. The splats are then drawn from the cache so that all splats inside a particular tile are rasterized consecutively. We refer to this process as "splat reordering". In our tests the reordering reduced the reconstruction buffer bandwidth requirements considerably (Section 5.7), especially for small cache sizes.

What we want to maintain is a list of splats for each reconstruction buffer tile, and then select one tile at a time and rasterize all the splats inside that tile as a batch. A seemingly straightforward method for achieving this is to use N linked lists on chip. However, in order to support large output resolutions, there cannot be a separate list for each reconstruction buffer tile. As a result, the approach is highly vulnerable when more than N concurrent lists would be needed. Furthermore, linked lists are not ideally suited for hardware implementation, although some implementations have been proposed [JLBM05]. As alternative to the overhead of N concurrent lists, hash-based solutions have been proposed for collision detection [THM+03]. Unfortunately, hash collisions occur frequently for small sized hash tables and can trigger a substantial amount of cache misses.

After initial experiments with these data structures, we switched to a another solution.

Processing order of tiles Ideally the tiles should be processed in an order that maximizes the hit rate of the reconstruction buffer cache. According to our tests, one of the most efficient orders is the simplest one: visiting all non-empty tiles in a scan-line order (and rasterizing all the splats inside the current tile). This round robin strategy is competitive against more involved methods that are based on either timestamps or the number of cached splats per tile. One reason for this behavior is that in the scan-line order each tile is left a reasonable amount of time to collect incoming splats. The other strategies are more likely to process a tile prematurely, and therefore suffer in cases when more splats follow after the tile has already been evicted from the reconstruction buffer cache.

We decided to implement the scan-line order using a priority queue that sorts the incoming splats by their tile index. As will be shown, our implementation executes several operations in parallel and provides constant-time insertions and deletions to the queue. To ensure a cyclic processing order of tiles, the typically used *less than* ($<$) ordering criterion of the priority queue has to be modified. Assuming that the currently processed tile index (the front element of the queue) is I_c, we define the new ordering criterion of the priority queue $i \prec_{I_c} j$ as

$$\begin{cases} \textit{false}, & i < I_c \leq j \\ \textit{true}, & j < I_c \leq i \\ i < j, & \text{otherwise} \end{cases} \tag{5.14}$$

It can be shown that introducing this variable relation into any sorting data structure, its internal structure stays consistent as long as I_c is adjusted to I'_c only if there is no remaining element i with $(i = I_c)$ and $(I_c \prec_{I'_c} i)$ and $(i \prec_{I_c} I'_c)$. Note that these three conditions correspond to the inequality $I_c \leq i < I'_c$ interpreted cyclically.

Pipelined heap A heap is a completely balanced binary tree that is linearly stored in breadth-first order, and can be used to implement a priority queue. It is well suited for hardware implementations, as the child and parent nodes of a given node index can be found using simple shift and increment operations on the indices. This means that no pointers are required, and the data structure has no storage overhead. The data structure must always fulfill the heap condition, i.e., that a parent node key is less than or equal to the keys in its two child nodes. As a consequence, the smallest element of a heap is always stored at the root node, which is the core property of a priority queue.

Unfortunately, a heap is not ideal when high throughput is required. Insertion and removal of heap elements require $O(\log n)$ operations, where n is the number of elements in the heap. Figure 5.5 (a) and (b) show that the two basic heap operations, SIFT-UP and SIFT-DOWN, traverse the tree levels in opposite directions. However, if it were possible to reverse the traversal direction for one of the two operations, they could be parallelized to propagate through the levels without collision [RK88].

This can be achieved by adding auxiliary buffers between the heap levels that can hold one element each, see Figure 5.5 (c). For SIFT-DOWN*, the path that the operation takes through the heap tree is known in advance as it is the same fixed path as for the SIFT-UP: the propagation is going to traverse all heap levels before finally an element is stored at the first free leaf, that is,

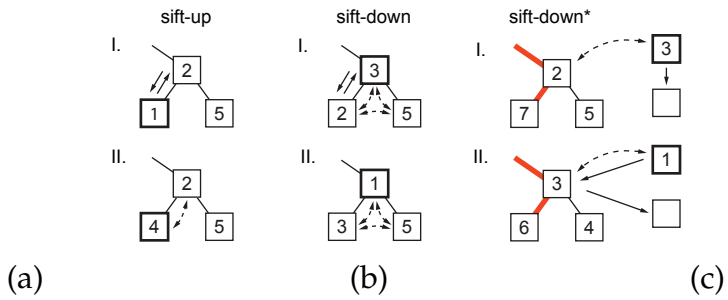


Figure 5.5: *Heap operations. Roman numbers refer to two independent examples of each operation. Dashed arrows denote required comparisons, solid arrows move operations, and bold boxes the new element. (a) In* SIFT-UP ON INSERTION *the new element is inserted to the last leaf node ascending in the hierarchy until the local heap condition is met. (b) In* SIFT-DOWN ON DELETION *the root node is replaced with the last leaf, descending until the heap condition is re-established. The different propagation directions of (a) and (b) make parallel execution of the* $O(\log n)$ *operations impossible. (c) In modified* SIFT-DOWN* ON INSERTION, *the introduction of auxiliary registers between heap levels allow to propagate inserted elements downward. The path for this operation is known in advance (in red).*

at the end of the linearized heap. Consequently, this target address is passed to SIFT-DOWN* with every element that is inserted, tying the execution to that path. Figure 5.6 shows the resulting comparison operations along the propagation path of an inserted element. Figure 5.7 shows an example of a full deletion operation using SIFT-DOWN ON DELETION, Figure 5.8 shows an example of a full insertion operation using SIFT-UP ON INSERTION, and Figure 5.9 shows an example of a full insertion operation with auxiliary buffers using the SIFT-DOWN* ON INSERTION operation.

SIFT-DOWN ON DELETION usually requires the last element of the heap to be

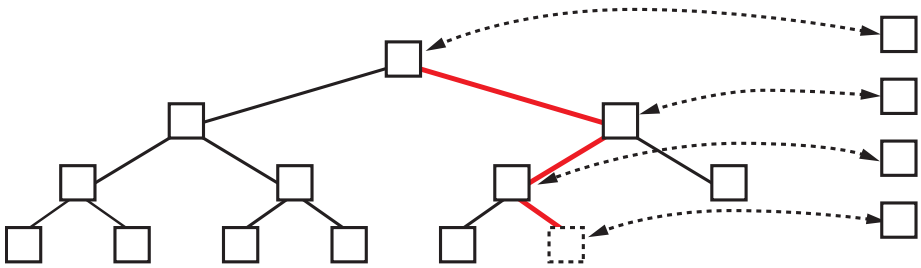


Figure 5.6: *The path an insertion process takes on* SIFT-DOWN* *is known in advance. Elements are propagated using a chain of auxiliary buffers. Dashed arrows denote element comparisons.*

moved to the root node before it sifts down the hierarchy. This procedure has to be adapted in case non-empty auxiliary buffers exist. In this case, the element with the highest target address is removed from its auxiliary buffer and put at the root node instead. It is then sifting down again until the heap condition is met.

Our implementation resembles the design Ioannou and Katevenis [IK01] proposed for network switches. Each heap level is accompanied by a separate controlling hardware unit that holds the auxiliary buffer. The controller receives insertion and deletion requests from the preceding level, updates a memory location in the current level, and propagates the operation to the successive heap level. Each heap level is stored in a separate memory bank and is exclusively accessed by the separate controller unit. The controller units work independently and share no global state. This makes the design easily scalable. The only global structure is a chain that connects all auxiliary buffers to remove an advancing element for root replacement.

Using this design, insertion and removal have constant throughput (two cycles in our FPGA implementation), independent of the heap size. This makes it ideal for hardware implementation of the splat reordering strategy that we propose. The improvement offered by the reordering stage over conventional caches will be investigated in Section 5.7.

5.5.3 Rasterization and early tests

The rasterization was implemented as a fully pipelined fixed-function design. Each splat sample is evaluated independently, allowing for a moderate degree of parallelization within the rasterizer. However, as splats are expected to cover very few pixels, a more efficient parallelization can be obtained by using multiple rasterizers that work on different splats in parallel.

As described in Section 5.5.1, it makes sense to map each rasterizer exclusively to a single cache tile. Let n rasterizers process tile indices $I_1 \dots I_n$. The reordering stage then needs to issue entries of these n keys in parallel. This can be achieved by wrapping the reordering stage by a unit that maintains n FIFOs that feed the rasterizers. The new stage receives splats from the reordering stage until an $(n+1)$st key $I_{n+1} \succ \{I_1 \dots I_n\}$ occurs. This key is held until one of the rasterizer FIFOs is empty and ready to process key I_{n+1}. A useful property is that I_{n+1} is a look-ahead on the next tile to be processed. Consequently, it can be used to trigger a reconstruction buffer cache prefetch to reduce the wait cycles introduced when exchanging the loaded cache tile with new data. In case a rasterizer unit is already working on a splat to

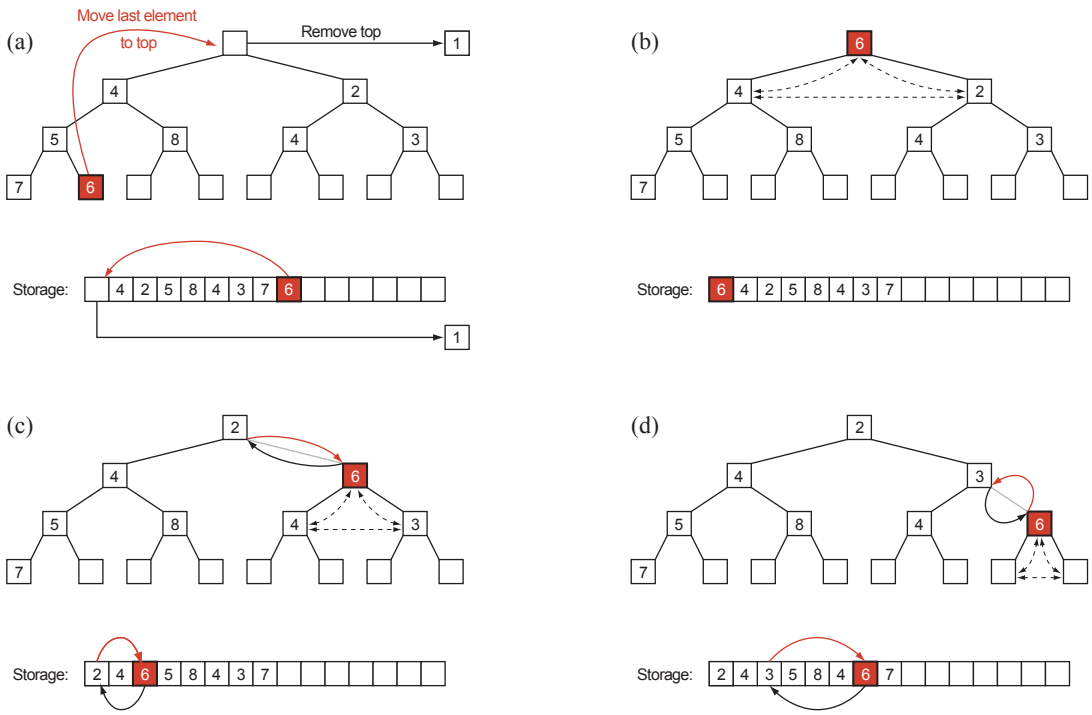


Figure 5.7: *Example for a* SIFT-DOWN ON DELETION *operation. Dashed arrows represent the performed comparisons. Solid arrows denote the performed move operations. The bold red box is highlighting the element that is moved in the current step. First, the top element is removed and the last element is inserted at the beginning of the heap (a). The newly inserted element is then successively compared with both child elements (b). The smallest of the three compared elements is moved to the parent node, and subsequently used for the next comparison (c). Once both child elements are bigger or a leaf node is reached, the* SIFT DOWN *operation stops (d).*

be inserted into the reordering unit, the FIFO wrapper intercepts such keys $I_1 \dots I_n$ and by-passes them directly to the respective FIFOs.

After the rasterization, side-effect free, early fragment tests are performed to eliminate unnecessary fragments early.

5.5.4 Accumulation and reconstruction buffer

The splat fragments are dispatched to the accumulation units in a round robin fashion, and each unit receives a full attribute vector for accumulation into the reconstruction buffer. The accumulators were implemented as scalar units in order to ensure good hardware utilization for any configurable number

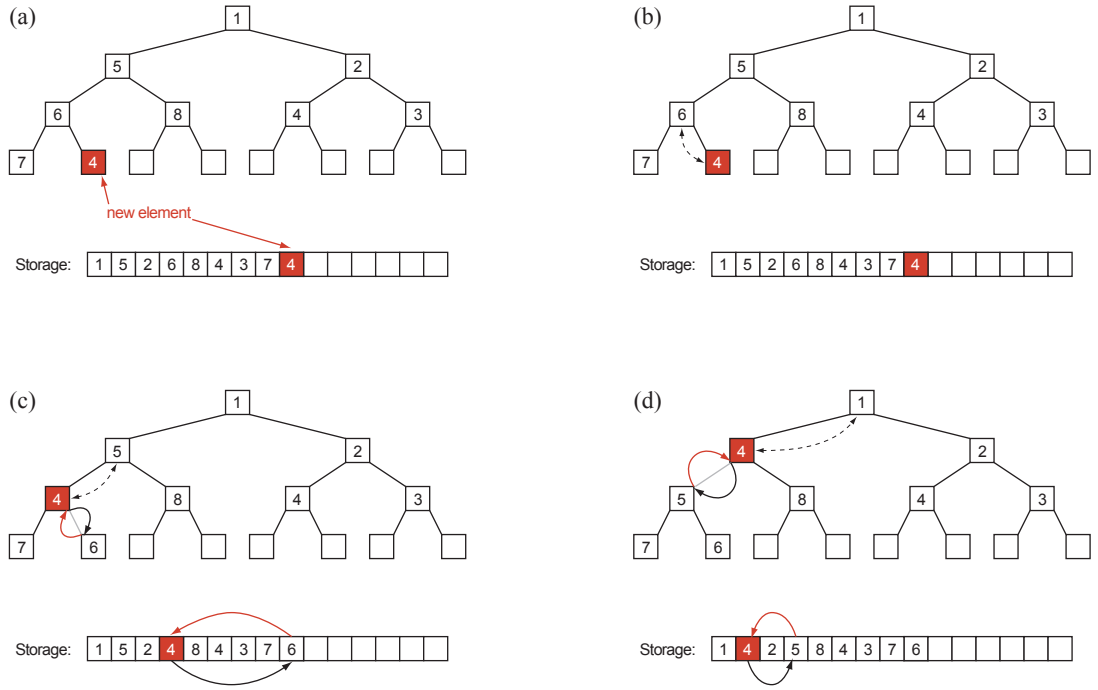


Figure 5.8: *Example for a* SIFT-UP ON INSERTION *operation. Dashed arrows represent the performed comparisons. Solid arrows denote the performed move operations. The bold red box is highlighting the element that is moved in the current step. A new element is inserted at the end of the heap (a). The new element is then successively compared with its parent node (b). If the new element is smaller than the parent node it is moved up, and subsequently used for the next comparison (c). Once the parent node is smaller or the root node has been reached, the* SIFT-UP *operation stops.*

of attributes. As the accumulators can work on different splats at a time, the design remains efficient even for very small splats with a high number of attributes.

Due to the accumulation of multiple splats per pixel, the surface reconstruction buffer needs to provide space for all attributes at a precision that exceeds 8 bits per channel, ideally using a floating point format. Hence, it usually requires more space than the frame buffer, and it needs to be allocated in external memory. Its caching requirements are similar to those of the frame buffer, and therefore the caching architecture can be shared with the frame buffer.

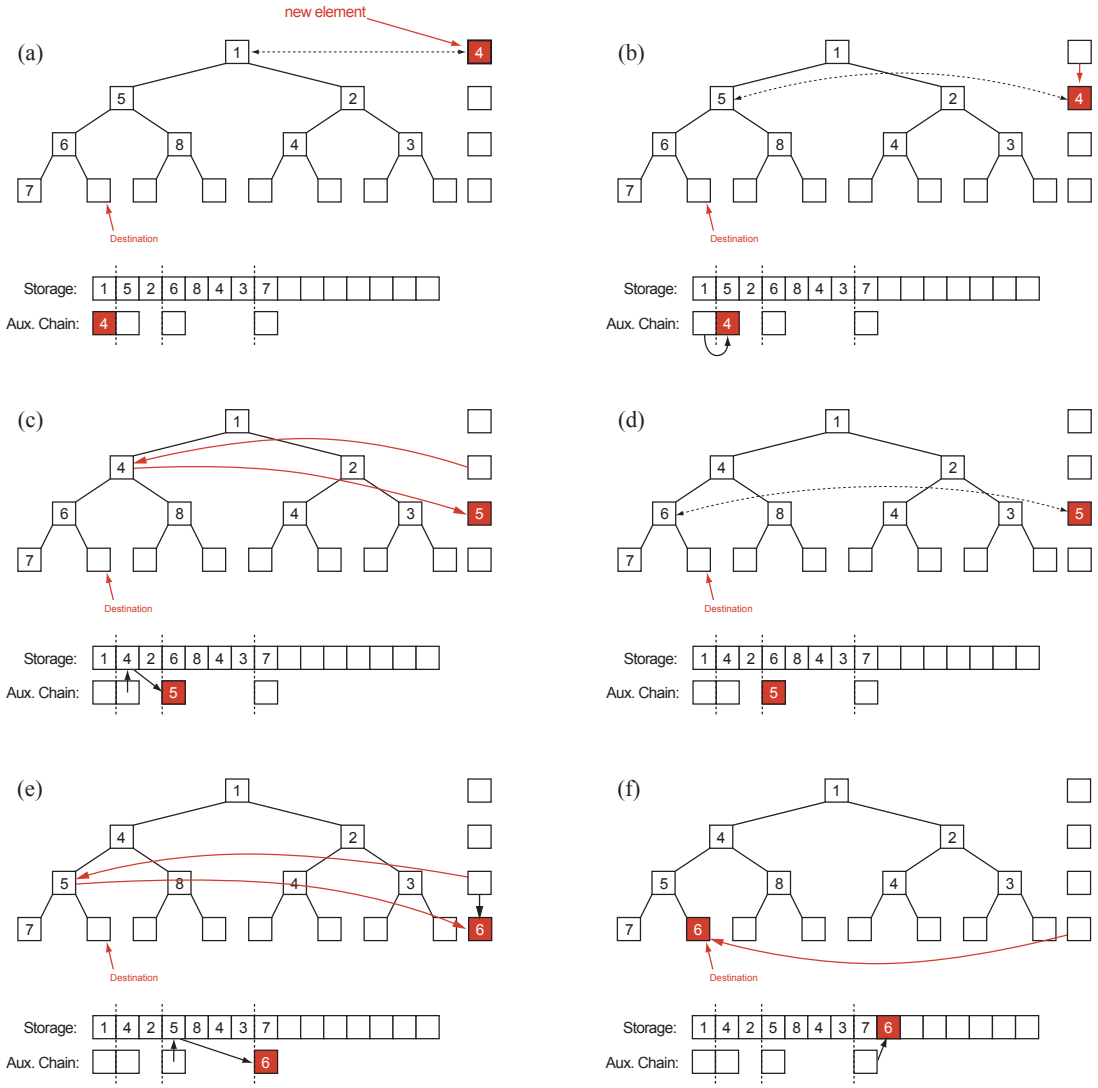


Figure 5.9: *Example for a* SIFT-UP DOWN* INSERTION *operation, using the same data as for the* SIFT-UP ON INSERTION *example from Figure 5.8. Dashed arrows represent the performed comparisons. Solid arrows denote the performed move operations. The bold red box is highlighting the element that is moved in the current step. A new element is inserted in the auxiliary buffer of the first level and compared to the first element of the heap (a). If the new element is bigger than the top element, the new element is passed to the auxiliary buffer of the next level. In the next level the new element is subsequently compared to the element lying on the path to the destination (b). If the new element is smaller than the existing element, the new element is moved to its place and the element is passed to the auxiliary buffer of the next level (c). The element in the auxiliary buffer is then subsequently compared with the element lying on the path (d) and exchanged in case the element is smaller (e). Once the destination leaf is reached, the item from the auxiliary buffer is written to that position (f).*

5.6 Implementations

We evaluated the proposed rendering architecture using three prototype implementations, the early VLSI prototype, an advanced FPGA prototype and an implementation of an OpenGL extension for surface splatting.

5.6.1 VLSI prototype

An early experiment uses an Application-Specific Integrated Circuit (ASIC) implementation of parts of our architecture to show that the heap data structure efficiently reduces bandwidth requirements and allows high-quality EWA surface splatting at moderate memory bandwidth.

We built a custom chip that contains a splat splitter, a reordering stage that holds up to 1024 surface splats, a rasterizer, and a blending stage that supports three color channels, depth, and weight. All computations and the reconstruction buffer use fixed-point representations. A small on-chip cache provides enough space to store a single reconstruction buffer tile. Due to limited die area, transform and lighting and rasterization setup are implemented on two parallel DSP boards, each of them featuring two DSPs that transmit the rasterizer input to the memory-mapped ASIC. This implementation still uses a splat setup according to [ZRB+04].

The ASIC has been manufactured in a 0.25 μm process using 25 mm^2 die area, and operates at a maximum frequency of 196 MHz. A custom-built printed circuit board (PCB) inter-connects the DSP and the ASIC, holds the reconstruction buffer memory, and provides a USB2.0 communication channel to a PC. The PCB finally displays the normalized reconstruction buffer via DVI output. Two joysticks directly attached to the DSP boards allow to optionally control the configuration independent from a PC. Figure 5.10 shows the final system.

To further evaluate the resource requirements of a full splatting pipeline, we designed a second custom ASIC implementing the transform, lighting and rasterization setup stages for EWA surface splatting. The pipeline was implemented with full IEEE 754 single precision floating point arithmetic and exploits much less resources in terms of chip surface and power dissipation than the DSP units, while achieving similar performance. Due to the restricted chip size we employed resource sharing[3] of arithmetic units at a high level: for

[3]Resource sharing (also referred to as multiplexing or time sharing) is often employed in VLSI design when the cost of fully parallel processing is unaffordable. The idea is to re-use certain building blocks in order to trade throughput against chip area.

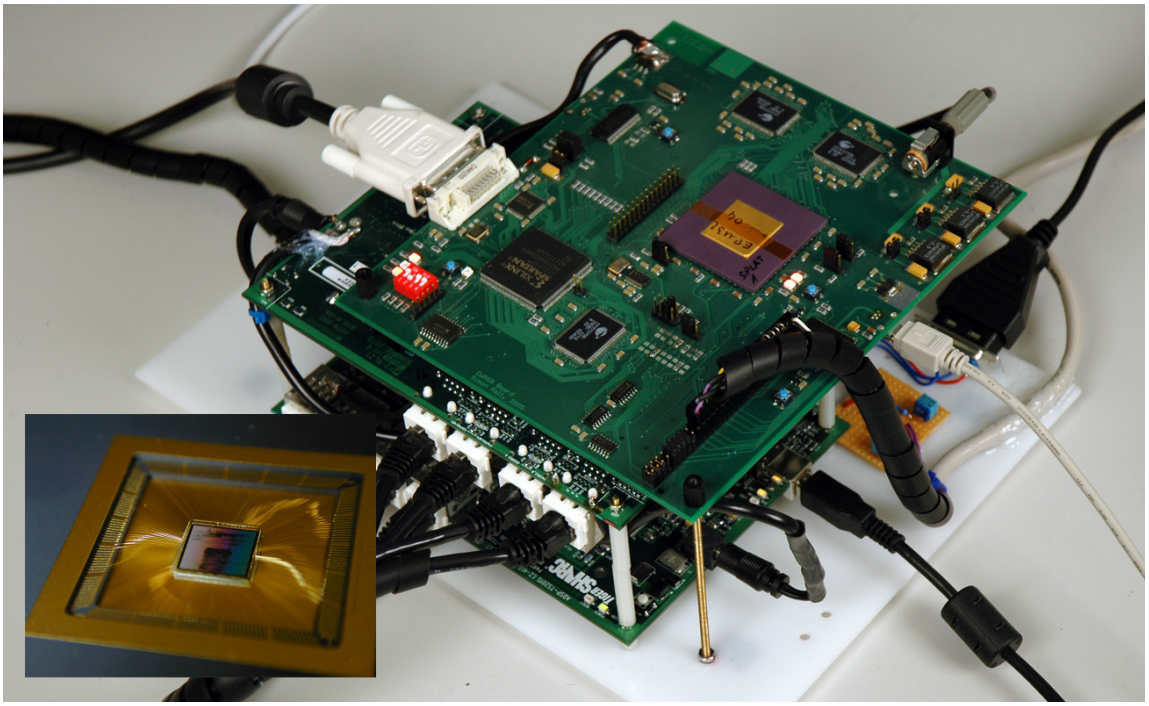


Figure 5.10: *Our prototype board with an ASIC implementation of the EWA splatting rasterizer.*

example, the transformation from world space to clip space performs similar computations on the position and the two tangent axes. The bounding box computation performs very similar computations on the separate dimensions. By applying high-level resource sharing, the number of floating point units used in the design was reduced by about 50% to 46 adders, 70 multipliers, 5 inverse and 2 inverse square roots.

The ASIC has been manufactured in a 0.18 μm process using 8.2 mm^2 die area, and operates at frequencies up to 147 MHz with 300 mW power dissipation.

Figure 5.11 shows the floor plans of both setup and rasterizer ASICs.

5.6.2 FPGA implementation

In order to further investigate our architecture, we aimed at a complete and improved implementation of the proposed architecture, based on Field-Programmable Gate Arrays (FPGA). We again partitioned the design into two major blocks, Setup and Rasterization, distributed over two FPGAs. T&L and the rasterization setup are performed on a Virtex 2 Pro 2VP70-5 board with 128 MB DDR DRAM and 8 MB DDR2 SRAM. A Virtex 2 Pro VP100-6 board with 256 MB DDR DRAM and 16 MB DDR2 SRAM holds the reordering stage

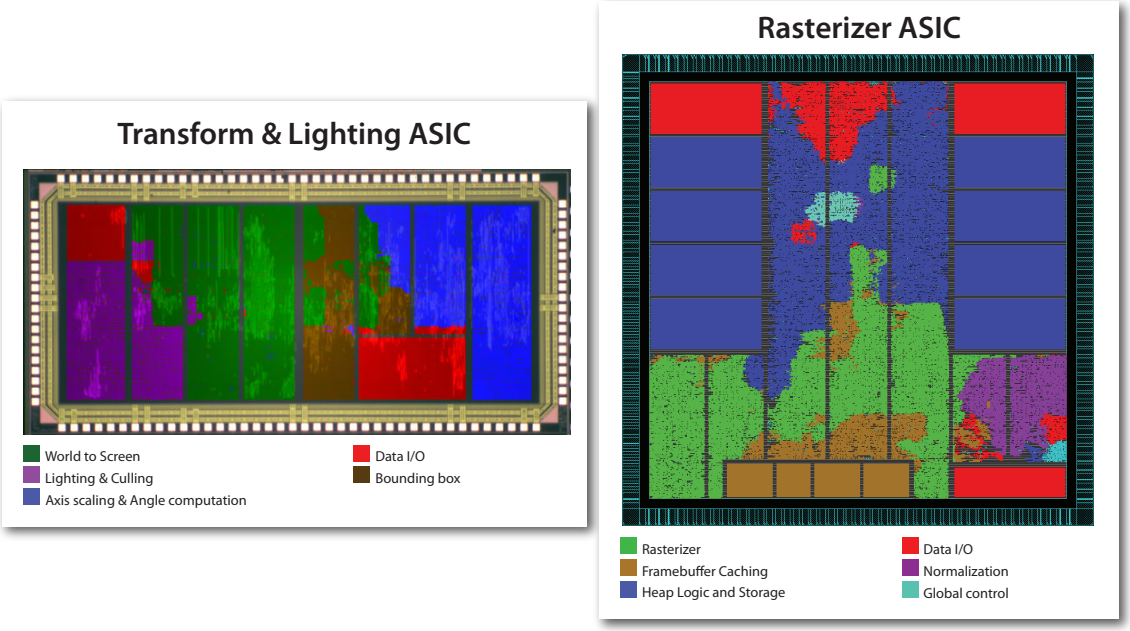


Figure 5.11: *ASIC floorplans for the transform and lighting ASIC (left) and the rasterizer ASIC (right). The chip dimensions have been normalized according to the manufacturing process and both floorplans are drawn in scale for comparison.*

and the rasterization pipeline. The two boards are inter-connected using a 2 GB/s LVDS communication. See Figure 5.12 for a schematic.

Our design runs at an internal clock of 70 MHz. The DRAM transfers up to 1.1 GB/s, while the SRAM, running at 125 MHz, provides a bandwidth of 4 GB/s.

Surface splats are uploaded to the first FPGA over the PCI bus. It is possible to define simple display lists to render scenes from within the FPGA's local memory. After the setup stage, the rasterization input is transferred to the second FPGA. There the reordering stage can store tile indices and 16-bit pointers for 4095 splats in 12 heap levels on chip. All the attributes and other per-splat data are stored temporarily in external SRAM, and have to be retrieved when the respective splat exits the reordering stage again. This requires a small memory management unit. The attributes are represented and manipulated as 16-bit floating-point numbers in the same format as the half-precision floating point format as defined for OpenGL as extension [NVI].

Our prototype features two parallel rasterizers that process the same splat at a time, using pixel-interleaving. Each rasterizer is accompanied by four accumulation units. For splats with up to 4 surface attributes, this allows

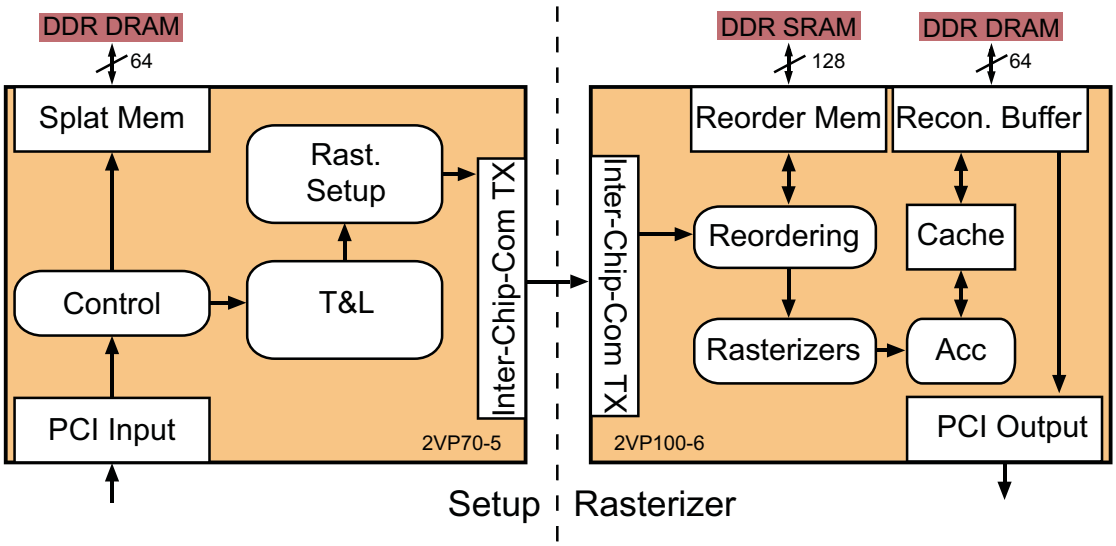


Figure 5.12: *Two FPGAs, coupled to implement a fixed-function Transform&Lighting stage followed by the proposed rasterization architecture.*

blending two fragments every cycle, leading to a theoretical peak performance of 140 M blended fragments/s. Larger attribute vectors lead to a decreased pixel output. Our design currently supports up to 14 attributes per splat. Additionally, FLIPQUAD sampling [AMS03] is supported for improved silhouette anti-aliasing.

In order to simulate the embedding of our extension into an existing GPU, we pass the normalized output from the second FPGA to the GPU as one or multiple textures. Rendering a screen-sized quad with this texture effectively passes the normalized fragments from our reconstruction buffer to the fragment stage of the graphics card. This allows to apply additional operations, such as a fragment program.

5.6.3 OpenGL integration

Although the proposed extension is not specific to a particular graphics API, we demonstrate its integration into OpenGL. The integration was tested by embedding an additional rasterization module into Mesa [Mes], an OpenGL-like software rendering library. The module contains a software simulation of the splat splitter, a reordering stage, and the splat rasterization pipeline. Apart from this, the only required extensions to Mesa were the API additions and a hook to detect state changes that trigger a release of surface fragments. All relevant parameters, such as the current attribute selection could naturally be retrieved from the OpenGL context.

On the API side, EWA surface splats were seamlessly integrated with other rendering primitives, hiding most of the specifics of the new primitive from the user. For instance, the newly introduced reconstruction buffer is hidden from the user. All API extensions were built as a regular OpenGL extension, which enables the rendering of individual as well as arrays of splats. For example, one can render splats similarly to GL_POINTS:

```
glBegin(GL_SPLATS_EXT);
  glColor3f(1, 1, 1);
  glTangentOne3fEXT(1, 0, 0);
  glTangentTwo3fEXT(0, 1, 0);
  glVertex3f(0, 0, 0);
glEnd();
```

Analogously, vertex arrays can be used as with any other rendering primitive:

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_FIRST_TANGENT_ARRAY_EXT);
glEnableClientState(GL_SECOND_TANGENT_ARRAY_EXT);

glVertexPointer(3, GL_FLOAT, 0, vertices);
glColorPointer (3, GL_FLOAT, 0, colors);
glTangentOnePointerEXT(GL_FLOAT, 0, us);
glTangentTwoPointerEXT(GL_FLOAT, 0, vs);

glDrawArrays(GL_SPLATS_EXT, 0, count);
```

We evaluated the use of EWA surface splats within an OpenGL application using this extension. As an example of a fairly complex OpenGL application, we chose the game Quake 2. We replaced some models in the game by point-sampled objects. Apart from the respective calls to the splat extension, no further changes to the code were required. Figure 5.13 shows two screenshots.

5.7 Results

Table 5.3 shows statistics of three representative scenes that cover a range of typical splatting scenarios. The scenes vary in screen coverage, average splat size, and average overdraw. Scene 1 features mostly splat radii that are larger than a pixel, while Scene 2 shows a minification situation. See the magnified inset in Scene 2 to judge the anti-aliasing quality. Scene 3 combines a wide range of splat sizes in a single view, see Figure 5.14. The

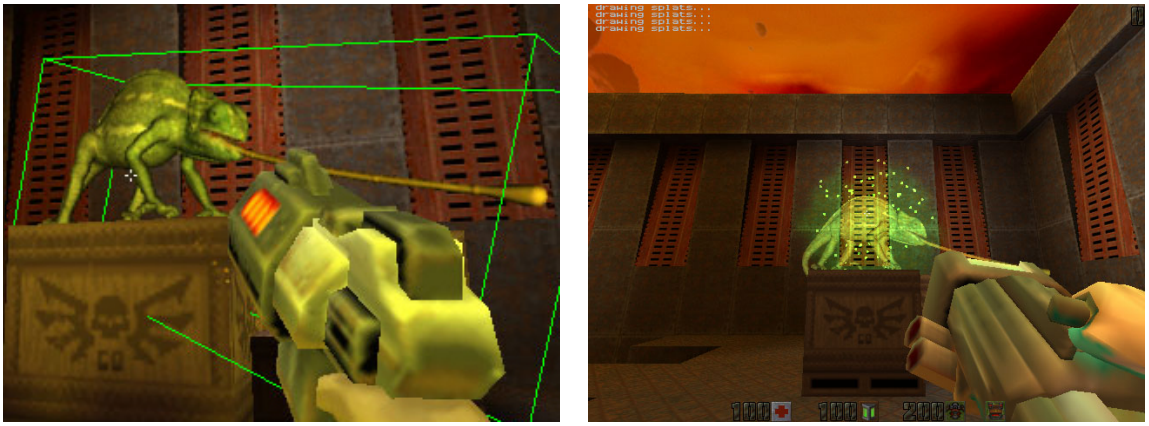


Figure 5.13: *The integration of EWA splat objects into this existing, fairly complex OpenGL application did not require any changes of the surrounding GL code.*

displayed statistics were obtained in 512 × 512 resolution with FLIPQUAD supersampling enabled. The screen-shots partly show deferred shading as described in Section 5.6.2.

5.7.1 Performance measurements

The theoretical peak performance of our rasterizer ASIC prototype is 200 M fragments per second. Artificial test vectors were capable of reproducing this value, and measurements at a 512 × 512 resolution resulted in a peak performance of 3 M splats per second (10 M splats/s at 128 × 128). With realistic scenes, the design's throughput is limited by the DSP performance. The four parallel DSPs process 2.5 M splats/s, of which typically 1 to 1.25 M splats/s reach the ASIC after back-face culling, driving the ASIC close to its maximum fill rate. The T&L and splat setup ASIC was designed to meet the input rate of the rasterizer and is able to process 2.94 M splats per second.

The theoretical peak performance of our FPGA is 17.5 M splats and 140 M fragments per second. The setup chip is capable of processing 26.7 M splats/s. The pipelined heap, which operates at 35 MHz, allows for a maximum throughput of 17.5 M splats/s. The inter-FPGA communication provides a constant throughput of 30 M splat/s and is therefore no bottleneck in the prototype implementations. Rasterizers and accumulators run at 70 MHz, issuing two fragments per cycle for up to four attributes per fragment. With data from our representative test scenes, each of the FPGA's units reaches its nominal throughput. As expected, the fragment throughput scales linearly with the number of attributes exceeding four, which means that the attribute

			
	Scene 1	**Scene 2**	**Scene 3**
Splat count	101 685	101 685	465 878
Coverage	31.2%	1.3%	43.5%
Overdraw	6.8	27.7	17.6
Samples / splat	19.36	3.5	12.6
Shading	deferred	T&L	deferred
Attributes	14	3	6

Table 5.3: *Test scenes used for evaluation, rendered using the FPGA.*

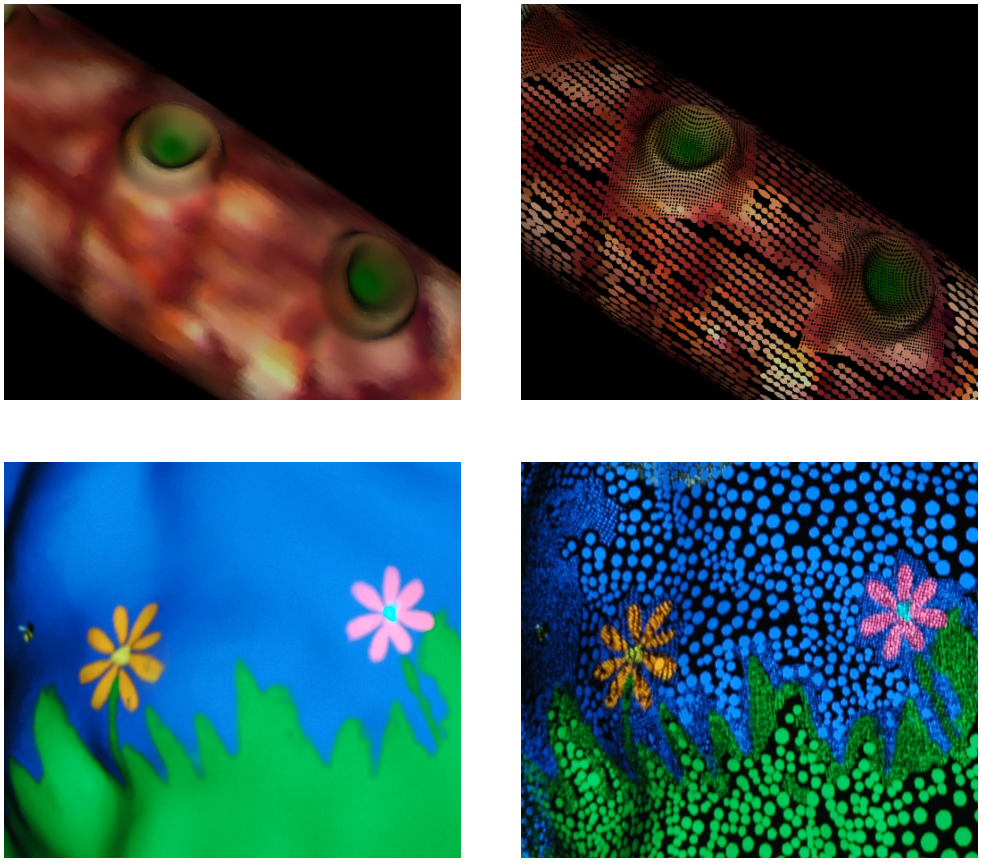


Figure 5.14: *ASIC rendering of an irregularly sampled surface. On the right, the splats have been artificially shrunk to illustrate the sample distribution. Images show photographs taken off a screen.*

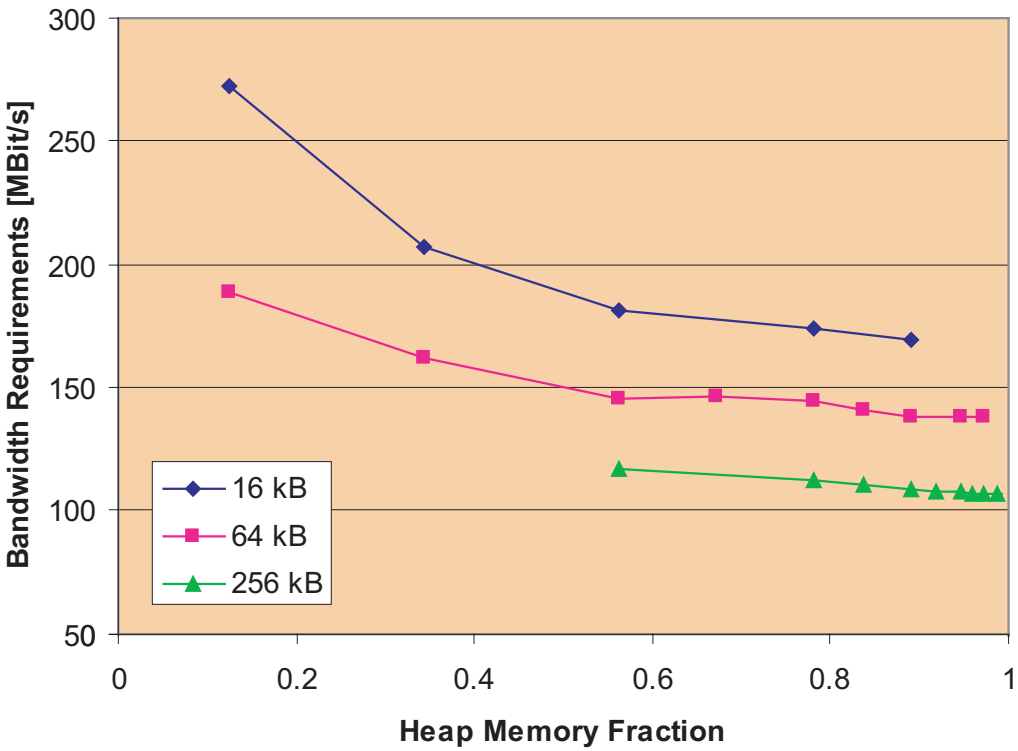
accumulators are kept busy at all times. Due to space restrictions, the rasterizer FPGA contains cache logic for only a single cache tile. This becomes manifest in reduced performance when measuring the fully integrated graphics pipeline. For average scenes, the splat throughput ranges from only 0.7 to 2 M splats/s, with measured peak performances of up to 4 M splats/s. The single-tile reconstruction buffer cache disallows prefetching, which stalls the pipeline whenever a new cache tile is being accessed. Due to the relatively high current logic utilization of the rasterizer FPGA is only 82% (64% flip-flops, 82% LUTs), routing and timing restrictions make it very difficult to add the required additional cache logic. Hence, investigations with a bigger, next-generation FPGA would be necessary for higher performance.

In order to evaluate the efficiency of the proposed reordering architecture, we simulate its performance in comparison to a classical caching architecture. Our measurements assume, rather optimistically, that the classical tile cache is fully associative and uses a least-recently used (LRU) policy. The tile size is set to 8×8 pixels. Assuming that a certain die area allowing for a fixed amount of on-chip memory is given, we simulate different partitionings between the pipelined heap and the LRU cache. Beginning with a pure LRU cache, we continually reduce the number of cache tiles, down to two remaining tiles, filling the remaining on-chip memory with a pipelined heap. Graph 1 shows the resulting bandwidth requirements for different memory fractions occupied by the heap, assuming a budget of 16, 64, and 256 kB on-chip memory, respectively. As can be seen, trading cache memory for a reordering stage improves the external memory bandwidth requirements.

5.7.2 Scalability

Compared to our FPGA's theoretical peak performance of 17.5 M splats per second at a rather low clock frequency, the NVIDIA GeForce GT 280 is able to render between 40 M and 70 M splats per second at much higher core clock frequency of 602 MHz. The GT 280 performs shader computations even at 1296 MHz with an external memory bandwidth of up to 141.7 GB/s. Judging from our ASIC design that runs 196 MHz on a fairly low-end manufacturing process (0.25μ), scaling to $\geq$500 MHz seems plausible using a modern ($\leq 0.065\mu$) process. That transition alone would give $\sim$200 M splats per second.

Most of the functional units (splat splitter, rasterization, fragment tests, accumulation, and normalization) in Figure 5.4 are pipelined fixed function designs that can process one splat or fragment per cycle. Further parallelization of these units can be achieved by duplication, and the relative



Graph 1: Bandwidth simulation of Scene 1, showing the effect for varying heap / cache memory ratio, for a total of 16, 64, and 256 kB on-chip memory each. Shown: bandwidth requirements to reconstruction buffer with respect to the fraction of heap memory.

number of rasterizers or accumulators can be trivially modified. The current implementation of the reordering stage has a throughput of two cycles per splat. The throughput is limited by the heap deletion operation, but it should be possible to optimize that into a single cycle by further engineering. Further parallelism could also be achieved by duplication and then assigning a subset of frame buffer tiles to each unit.

Both ASIC evaluations can give a hint on the impact of scaling the design to a full sized ASIC. The GT200 GPU chip from NVIDIA was manufactured with 575 mm^2 with a 65nm process[4]. Normalizing the areas of our designs to a 65nm process, the areas resemble to 1.62 mm^2 for the rasterizer, and 1.06 mm^2 for the transform and lighting and setup ASIC. The total area combined therefore only represents 0.5% of the die area of the GT200 GPU.

In terms of gate count our design is very small compared to modern GPUs, and thus clock speed improvements as well as duplication of the necessary

[4]At the time of writing, NVIDIA released their GT 400 chip. The chip was manufactured at 40nm with a die size of 529 mm^2, which is equivalent to a chip with 1397 mm^2 in a 65nm process

units seem possible. We therefore estimate that for a full scale design it would be realistic to achieve the impressive performance above 500 million splats per second.

5.8 Conclusions and future work

We have described and analyzed our EWA splatting hardware design, and listed the limitations and open issues on the way to a full-scale EWA splatting system. We have also proposed a method for embedding the splatting functionality into OpenGL. We believe that efficient support for splatting would be a valuable addition, also because splats have other uses beside surface representation, e.g., visualization of indirect illumination [GKBP05] that has been decoupled from the (possibly polygonal) scene geometry.

We designed the entire splatting pipeline as dedicated hardware units in order to freely study what kind of architecture would be most suitable for the operations required by splatting. Some of the deviations from current GPU architectures could in fact be beneficial for triangle rendering as well. For example, it would be possible to use the pipelined heap for improving the coherence of texture fetches and frame buffer accesses in triangle rendering.

Cache size and the amount of parallelism in our current research prototypes are still modest due to the limited capacity of our FPGAs. The latest generation FPGAs would offer much higher capacity. This would allow to exploit the parallelism further, and to carry out more detailed performance and scalability analysis.

As the proposed EWA splatting pipeline does not utilize deep buffers or A-Buffer like implementations, it can reconstruct only one layer (e.g. the closest one) of a splat surface for each pixel and does not support the motion-blur EWA sampling approach directly. An additional depth test would have to be included to the splatting pipeline in order to avoid reconstructing occluded surfaces and to support, for example, depth peeling for order-independent transparency [Eve01]. However, the most recent DirectX 11 GPUs have been shown to support novel data-structures such as linked lists [YHGT10] for the framebuffer, and could be used to support these deep buffers without the need for additional hardware for surface splatting. Shadow map lookups, on the other hand, can be implemented in the fragment shader and require no additional units.

While triangle rasterization architectures have been studied for decades, splat rasterization is a relatively unexplored field. It might be possible to simplify the splat setup and rasterization algorithms further. Also, it would

be worthwhile to study the numerical accuracy required in the fixed-function units, as the savings in die area could be significant.

In the same spirit as polygons can be tessellated in modern GPUs, point clouds can be supersampled from a control point surface. The rendering architecture presented in this chapter, however, does not support such upsampling. While the most recent DirectX 11 GPUs [Mic10] added programmable tessellation units that allow for a wide variety of tessellation procedures on polygonal meshes, point sampled surfaces cannot be upsampled directly using these new units due to the the lack of explicit connectivity information. The following chapter will therefore investigate a more general hardware architecture for point processing that computes this connectivity information on the fly. Based on the found neighborhood, the architecture is then able to compute point upsampling as well as a wide range of other point processing algorithms efficiently.

CHAPTER 6

Processing unit for point sets

This chapter introduces a more general hardware architecture for geometry processing of point sampled data. The new architecture is designed to complement the rendering architecture presented in the previous chapter. Our design is focuses on fundamental and computationally expensive operations on point sets including k-nearest neighbors search, moving least squares approximation, and others. Our architecture includes a configurable processing module allowing users to implement custom operators and to run them directly on the chip. A key component of our design is the spatial search unit based on a kd-tree performing both kNN and εN searches. It utilizes stack recursions and features a novel advanced caching mechanism allowing direct reuse of previously computed neighborhoods for spatially coherent queries. In our FPGA prototype, both modules are multi-threaded, exploit full hardware parallelism, and utilize a fixed-function data path and control logic for maximum throughput and minimum chip surface. A detailed analysis demonstrates the performance and versatility of our design.

6.1 Overview

Point sampled geometries have proven to be a flexible and robust representation for upsampling and deformation of surfaces [GP07]. While the

architecture presented in the previous Chapter 5 focused on the rendering of point primitives, it does not support the manipulation of the input point mesh for applications such as animation or tessellation. In this chapter, we will therefore present a more general architecture to support the processing of point-sampled data.

A main characteristic of point-based representations is the lack of connectivity. It turns out that many point processing methods can be decomposed into two distinct computational steps. The first one includes the computation of some neighborhood of a given spatial position, while the second one is an operator or computational procedure that processes the selected neighbors. Such operators include fundamental, atomic ones like weighted averages or covariance analysis, as well as higher-level operators, such as normal estimation or moving least squares (MLS) approximations [Lev01, ABCO+01]. Very often, the spatial queries to collect adjacent points constitute the computationally most expensive part of the processing. In this paper, we present a custom hardware architecture to accelerate both spatial search and generic local operations on point sets in a versatile and resource-efficient fashion.

Spatial search algorithms and data structures are very well investigated [Sam06] and are utilized in many different applications. The most commonly used computations include the well known k-nearest neighbors (kNN) and the Euclidean neighbors (εN) defined as the set of neighbors within a given radius. kNN search is of central importance for point processing since it automatically adapts to the local point sampling rates.

Among the variety of data structures to accelerate spatial search, kd-trees [Ben85] are the most commonly employed ones in point processing, as they balance time and space efficiency very well. Unfortunately, hardware acceleration for kd-trees is non-trivial. While the SIMD design of current GPUs is very well suited to efficiently implement most point processing operators, a variety of architectural limitations leave GPUs less suited for efficient exact kd-tree implementations. For instance, recursive calls are not supported due to the lack of managed stacks. In addition, dynamic data structures, like priority queues, cannot be handled efficiently. They produce incoherent branching and either consume a lot of local resources or suffer from the lack of flexible memory caches. Conversely, current general-purpose CPUs feature a relatively small number of floating point units combined with a limited ability of their generic caches to support the particular memory access patterns generated by the recursive traversals in spatial search. The resulting inefficiency of kNN implementations on GPUs and CPUs is a central motivation for our work.

In this chapter, we present a novel hardware architecture dedicated to the efficient processing of unstructured point sets. Its core comprises a configurable, kd-tree based, neighbor search module (implementing both kNN and εN searches) as well as a programmable processing module. Our spatial search module features a novel advanced caching mechanism that specifically exploits the spatial coherence inherent in our queries. The new caching system allows to save up to 90% of the kd-tree traversals depending on the application. The design includes a fixed function data path and control logic for maximum throughput and lightweight chip area. Our architecture takes maximum advantage of hardware parallelism and involves various levels of multi-threading and pipelining. The programmability of the processing module is achieved through the configurability of FPGA devices and a custom compiler.

Our lean, lightweight design can be seamlessly integrated into existing massively multi-core architectures like GPUs. Such an integration of the kNN search unit could be done in a similar manner as the dedicated texturing units, where neighborhood queries would be directly issued from running kernels (e.g., from vertex/fragment shaders or CUDA programs). The programmable processing module together with the arithmetic hardware compiler could be used for embedded devices [Vah07, Tan06], or for co-processors to a CPU using front side bus FPGA modules [Int07].

The prototype is implemented on FPGAs and provides a driver to invoke the core operations conveniently and transparently from high level programming languages. Operating at a rather low frequency of 75 MHz, its performance competes with CPU reference implementations. When scaling the results to frequencies realistic for ASICs, we are able to beat CPU and GPU implementations by an order of magnitude while consuming very modest hardware resources.

Our architecture is geared toward efficient, generic point processing by supporting two fundamental operators: Cached kd- tree-based neighborhood searches and generic meshless operators, such as MLS projections. These concepts are widely used in computer graphics, making our architecture applicable to as diverse research fields as point-based graphics, computational geometry, global illumination and meshless simulations.

The rest of this chapter is organized as follows. We will first revisit the family of moving least squares surfaces as an illustrative example for point-graphics operators in Section 6.2. We then review related work on data structures for meshless operations in Section 6.3. We will present the fundamentals on spatial searching and our new coherent cache in Section 6.4. In Section 6.5 we introduce our hardware architecture for generic point processing and describe

the neighbor search module and the processing module and its involved algorithms. We then give more details on the actual FPGA implementation of the presented pipeline, and discuss the resource requirements in terms of FPGA resources in Section 6.6. To evaluate our design and to demonstrate the versatility of the architecture, we will show some implemented examples in Section 6.7 and give detailed performance analysis for the architecture and its submodules. Section 6.8 then concludes this chapter and gives an outlook on future work.

6.2 Moving Least Squares surfaces

To understand the needs of point processing operators better, we will review the family of moving least squares (MLS) surfaces in this section. Point set surfaces [ABCO+01] are implicitly defined by a given control point set. A projection procedure then projects any point near the surface onto the surface. The MLS surface [Lev01] is then defined as the points projecting onto themselves.

One instructive example is the projection operator defined by Alexa and Adamson [AA04]. It is an efficient MLS operator using planes for the implicit definition of the surface. The approximating surface of a point set $P = \{\mathbf{p}_i \in \mathbb{R}^3\}$ is implicitly defined as the zero set of the function:

$$\hat{S} = \{\mathbf{x} \in \mathbb{R}^3 | f(\mathbf{x}) = 0\} \,. \tag{6.1}$$

The function $f(\mathbf{x})$ is defined based on local planar approximations of the surface:

$$f(\mathbf{x}) = \mathbf{n}(\mathbf{x})^T(\mathbf{a}(\mathbf{x}) - \mathbf{x}) \,. \tag{6.2}$$

The two functions $\mathbf{a}$ and $\mathbf{n}$ are then defined over a neighborhood $N(\mathbf{x})$ of points near a query point $\mathbf{x}$. The function $\mathbf{a}$ defines a weighted average of the neighborhood and can be described as

$$\mathbf{a}(\mathbf{x}) = \frac{\sum_{N(\mathbf{x})} \theta(\|\mathbf{x} - \mathbf{p}_i\|)\mathbf{p}_i}{\sum_{N(\mathbf{x})} \theta(\|\mathbf{x} - \mathbf{p}_i\|)}, \tag{6.3}$$

where $\theta(\|\mathbf{x} - \mathbf{p}_i\|)$ is a weighting function. The function $\mathbf{n}$ denotes the weighted average of all the normals within the neighborhood and is computed very similarly as

$$\mathbf{n}(\mathbf{x}) = \frac{\sum_{N(\mathbf{x})} \theta(\|\mathbf{x} - \mathbf{p}_i\|)\mathbf{n}_i}{\sum_{N(\mathbf{x})} \theta(\|\mathbf{x} - \mathbf{p}_i\|)}. \tag{6.4}$$

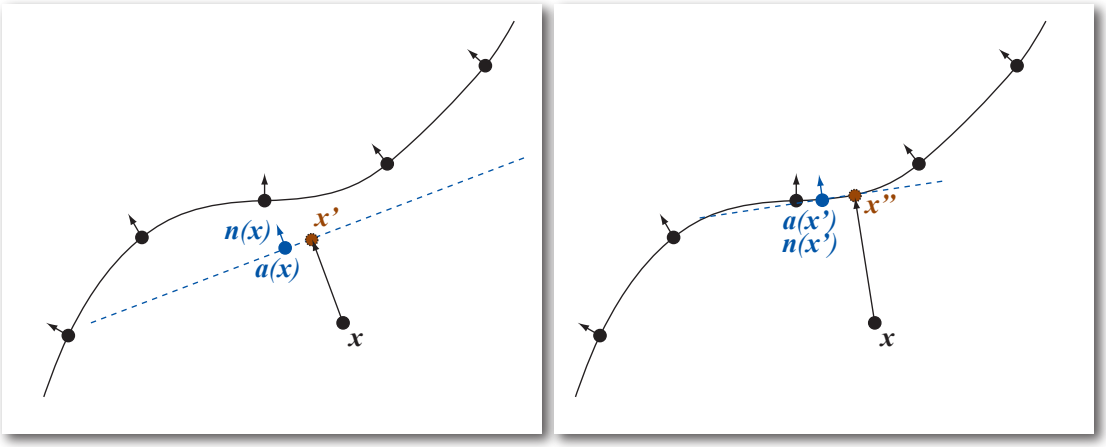


Figure 6.1: *Two steps of the almost orthogonal projection procedure based on plane fits. The algorithm projects a query point* $\mathbf{x}$ *onto a surface defined by the input point cloud, where each point is associated with a normal. In a each projection step,* $\mathbf{n}(\mathbf{x})$ *and* $\mathbf{a}(\mathbf{x})$ *define the current orthogonal frame onto which the query point* $\mathbf{x}$ *is projected iteratively.*

Using this definition, Alexa and Adamson [AA04] define a simple iterative projection procedure that projects points onto the surface:

1. Set $\mathbf{x}' = \mathbf{x}$ as start point.
2. Calculate the k nearest neighbors to $\mathbf{x}'$ to gather the neighborhood $N(\mathbf{x}')$.
3. Compute the averaged neighbor $\mathbf{a}(\mathbf{x}')$ and normal $\mathbf{n}(\mathbf{x}')$.
4. Project $\mathbf{x}$ onto the current tangent plane defined by $\mathbf{a}(\mathbf{x}')$ and $\mathbf{n}(\mathbf{x}')$: $\mathbf{x}' = \mathbf{x} - \mathbf{n}(\mathbf{x}')^T(\mathbf{a}(\mathbf{x}') - \mathbf{x})\mathbf{n}(\mathbf{x}')$.
5. Evaluate stop criterion: if $\|\mathbf{n}(\mathbf{x}')^T(\mathbf{a}(\mathbf{x}') - \mathbf{x}')\| > \epsilon$, go back to 2.

Two steps of this projection operator are illustrated in Figure 6.1.

In fact, most meshless operators are performed in a similar strategy: in a first step, the neighborhood around a point is computed. Then, a projection or refinement is calculated based on this neighboring point cloud and the whole loop is evaluated again until a final stop criterion is met.

This concludes the brief introduction on MLS based surface definitions, and we will now present related work on data structures for meshless operators and details on spatial searching and our caching strategy used for our architecture.

6.3 Data structures for meshless operators

A key feature of meshless approaches is the lack of explicit neighborhood information which typically has to be evaluated on the fly. The large variety of spatial data structures for point sets [Sam06] evidences the importance of efficient access to neighbors in point clouds. A popular and simple approach is to use a fixed-size grid in which points are sorted. However, this method does not prune the empty space and is thus inefficient in terms of memory storage. The grid file [NHS84] uses a directory structure to divide the space into multiple blocks. All grid cells and contained points are then grouped and indexed from the directory structure. Similarly, locality-preserving hashing schemes [IMRV97] provide better use of space by mapping multi-dimensional grid coordinates to a hash-table of lower dimensionality. However, due to the mapping to some lower dimensional space the calculation of the hash-index may produce collisions and then deteriorate performance. In both cases the respective grid sizes have to be carefully aligned to the query range to achieve optimal performance.

In contrast to grid-like structures, the quadtree [FB74] imposes a hierarchical access structure onto a regular grid using a d-dimensional d-ary search tree. The tree is constructed by splitting the space into 2^d regular subspaces, and empty space can be discarded efficiently by pruning the corresponding tree branches. However, the footprint of the tree might be relatively big and traversing the tree might become expensive due to the d number of indirections per node. The kd-tree [Ben85], one of the most popular spatial data structures, splits the space successively into two half-spaces along one dimension. It thus combines efficient space pruning with small memory footprint. Very often, the kd-tree is used for k-nearest neighbors search on point data of moderate dimensionality because of its optimal expected-time complexity of $O(\log(n) + k)$ [FBF77, Fil79], where n is the number of points in the tree. Extensions of the initial concept include the kd-B-tree [Rob81], a bucket variant of a kd-tree, where the partition planes do not need to pass through the data points. Instead of storing data points in the internal leaves, points are stored linearly in bins or buckets. In the following, we will use the term kd-tree to describe this class of spatial search structures.

Approximate kNN queries on the GPU have been presented by Ma et al. [MM02] for photon mapping, where a locality-preserving hashing scheme similar to the grid file was applied for sorting and indexing point buckets. In the work of Purcell et al. [PDC+03], a uniform grid constructed on the GPU was used to find the nearest photons, however this access structure performs only well on similarly sized search radii. Zhou et al. [ZHWG08] presented

an algorithm for constructing *k*d-trees on the GPU for raytracing and photon mapping. The paper uses an approximated *k*-nearest neighbors search on the GPU: instead of finding the true *k*-nearest neighbors, the algorithm iteratively performs a range search. In each iteration, the new search range is corrected towards the true range defined by the *k*-nearest neighbors. The paper shows how the GPU search can be applied to point cloud modeling and we will give a comparison to its method later in this chapter.

In the context of ray tracing, various hardware implementations of *k*d-tree ray traversal have been proposed. These include dedicated units [WSS05, WMS06] and GPU implementations based either on a stack-less [FS05, PGSS07] or, more recently, a stack-based approach [GPSS07]. Most of these algorithms accelerate the *k*d-tree traversal by exploiting spatial coherence using packets of multiple rays [WBWS01] and are targeted for ray-triangle intersection. Unfortunately, the more generic access pattern of *k*NN queries is different to ray traversal, and the *k* nearest neighbor search usually requires a sorted priority list of the currently found neighbors.

In order to take advantage of spatial coherence in nearest neighbor queries, we introduce a coherence neighbor cache system, which allows us to directly reuse previously computed neighborhoods. This caching system, as well as the *k*NN search on *k*d-tree, are presented in detail in the next section.

6.4 Spatial search and coherent cache

In this section we will first briefly review the *k*d-tree based neighbor search and then present how to take advantage of the spatial coherence of the queries using our novel coherent neighbor cache algorithm.

6.4.1 Neighbor search using *k*d-trees

The *k*d-tree [Ben85] is a multidimensional search tree for point data. It recursively splits the space along a splitting plane that is perpendicular to one of the coordinate axes and hence can be considered a special case of binary space partitioning trees [FKN80].

In its original version every node of the tree stores a point, and the splitting plane has to pass through that point. A more commonly used approach is to store points, or buckets of points, in the leaf nodes only. Figure 6.2 shows an example of a balanced 2-dimensional *k*d-tree which stores the points in the leaves. Note that balanced *k*d-trees are always superior in terms of storage

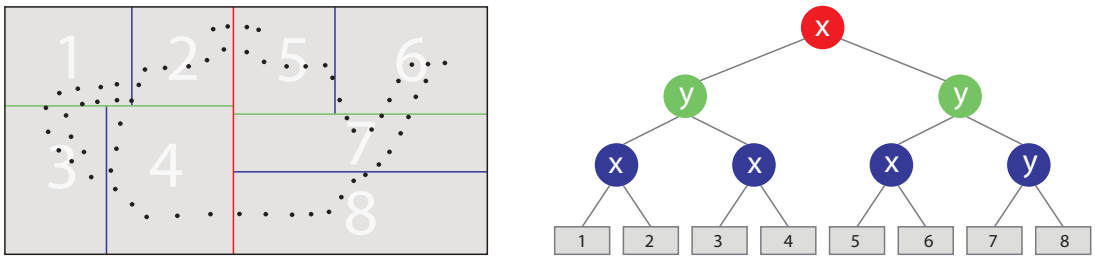


Figure 6.2: *The kd-tree data structure: The left image shows a point-sampled object in 2D, and the respective spatial subdivision computed by a kd-tree. The right image displays the kd-tree, points are stored in the leaf nodes.*

overhead, and also exhibit good properties for nearest neighborhood searching. Balanced *k*d-trees furthermore can always be constructed in $O(n \log_2 n)$ for n points [OvL80].

εN search

An εN search, also called ball or range query, aims to find all the neighbors around a query point $\mathbf{q}_i$ within a given radius r_i. It is performed using the *k*d-tree data structure as follows (Listing 6.1): the algorithm traverses the tree recursively down to the half space in which the query point is contained until a leaf node is encountered. At the leaf node, all points that lie within the query radius r_i are added to the result list. Then, the algorithm enters the backtracking stage and recursively ascends and descends into the other half spaces if the distance from the query point to the half space is smaller than r_i.

Listing 6.1: *Recursive search of the εN in a kd-tree.*

```
Point query; // Query point
float radius; // Query radius
List list; // Output list

void find_range(Node node) {
  if (node.is_leaf) {
    // Loop over all points contained by the leaf's bucket
    // and insert in output list if contained within search radius
    for (each point p in node)
      if (distance(p,query) < radius)
        list.insert(p);
```

```
  } else {
    partition_dist = distance(query, node.partition_plane);
    // decide whether going left or right first
    if (partition_dist > 0) {
      // descend to left branch first
      find_range(node.left);
      // recurse: evaluate other half−space only if it is close enough
      if (radius > abs(partition_dist))
        find_range(node.right);
    } else {
      // descend to right branch first
      find_range(node.right);
      // recurse: evaluate other half−space only if it is close enough
      if (radius > abs(partition_dist))
        find_range(node.left);
  }
}
```

kNN search

Very similar to the εN search, the k-nearest neighbors search in a kd-tree is performed as follows (Listing 6.2): the algorithm traverses the tree recursively down the half space in which the query point is contained until a leaf node is encountered. At the leaf node, all points contained in that cell are sorted into a priority queue of length k. In a backtracking stage, the algorithm then recursively ascends and descends into the other half spaces if the distance from the query point to the farthest point in the priority queue is greater than the distance of the query point to the cutting plane. The priority queue is initialized with elements of infinite distance.

Intuitively, this resembles to a modified εN search: the algorithm starts with an infinite search radius and reduces this radius successively as points are sorted into the priority queue. In most applications it is desirable to bound the maximum number of found neighbors also for the εN search, and thus the kNN search can be used to find the k-nearest neighbors where the maximum distance of the selected neighbors is bound by r_i. This behavior can then be trivially achieved by initializing the priority queue with placeholder elements at a distance r_i.

Note that in high-level programming languages, the stack implicitly stores all important context information upon a recursive function call and reconstructs

the context when the function terminates. As we will discuss subsequently, this stack has to be implemented and managed explicitly in a dedicated hardware architecture.

Listing 6.2: *Recursive search of the kNN in a kd-tree.*

```
Point query; // Query Point
PriorityQueue pqueue; // Priority Queue of length k

void find_nearest (Node node) {
  if (node.is_leaf) {
    // Loop over all points contained by the leaf's bucket
    // and sort into priority queue.
    for (each point p in node)
      if (distance(p,query) < pqueue.max())
        pqueue.insert(p);
  } else {
    partition_dist = distance(query, node.partition_plane);
    // decide whether going left or right first
    if (partition_dist > 0) {
      // descend to left branch first
      find_nearest(node.left);
      // recurse: evaluate other half space only if it is close enough
      if (pqueue.max() > abs(partition_dist))
        find_nearest(node.right);
    } else {
      // descend to right branch first
      find_nearest(node.right);
      // recurse: evaluate other half space only if it is close enough
      if (pqueue.max() > abs(partition_dist))
        find_nearest(node.left);
  }
}
```

6.4.2 Coherent neighbor cache

Several applications such as up-sampling or surface reconstruction issue densely sampled queries. In these cases, it is likely that the neighborhoods of multiple query points are the same or very similar. The coherent neighbor cache (CNC) exploits this spatial coherence to avoid multiple computations of

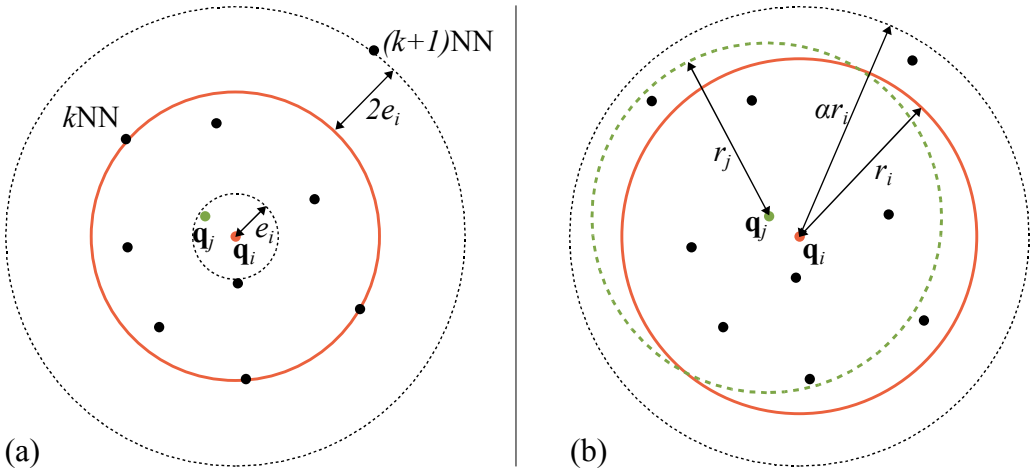


Figure 6.3: *The principle of our coherent neighbor cache algorithm. (a) In the case of kNN search the neighborhood of* $\mathbf{q}_i$ *is valid for any query point* $\mathbf{q}_j$ *within the tolerance distance* e_i*. (b) In the case of εN search, the extended neighborhood of* $\mathbf{q}_i$ *can be reused for any ball query* $(\mathbf{q}_j, r_j)$ *which is inside the extended ball* $(\mathbf{q}_i, \alpha r_i)$*.*

similar neighborhoods. The basic idea is to compute slightly more neighbors than necessary, to cache these extended neighborhood and to use the results for subsequent, spatially close queries.

Assume a query of the kNN of the point $\mathbf{q}_i$ is to be performed (Figure **??**a). Instead of looking for the k nearest neighbors, we compute the $k+1$ nearest neighbors $N_i = \{\mathbf{p}_1, ..., \mathbf{p}_{k+1}\}$. Let e_i be half the difference of the distances between the query point and the two farthest neighbors:

$$e_i = \frac{\|\mathbf{p}_{k+1} - \mathbf{q}_i\| - \|\mathbf{p}_k - \mathbf{q}_i\|}{2} \,. \tag{6.5}$$

Then, e_i defines a tolerance radius around $\mathbf{q}_i$ such that the kNN of any point inside this ball are guaranteed to be equal to $N_i \setminus \{\mathbf{p}_{k+1}\}$ and could therefore re-use the this neighborhood.

Therefore, the cache stores a list of the m most recently used neighborhoods N_i together with their respective query point $\mathbf{q}_i$ and their computed tolerance radius e_i. Given a new query point $\mathbf{q}_j$, we need to determine whether a suitable neighborhood has been found earlier. If the cache contains a neighborhood N_i such that $\|\mathbf{q}_j - \mathbf{q}_i\| < e_i$, then this neighborhood $N_j = N_i$ is reused. In case of a cache miss, a full kNN search is performed and its result stored in the cache.

In order to further reduce the number of cache misses, it would be possible to compute even bigger neighborhoods, i.e., the $k+c$ nearest ones. However,

for $c \neq 1$ the extraction of the true kNN this would then require to sort the set N_i for every cache hit, and would therefore require much bigger cache logic and also prevent the simple sharing of the neighborhood by multiple processing threads.

Moreover, in many applications it is preferable to tolerate some approximation in the neighborhood computation. Given any positive real ε, a data point $\mathbf{p}$ is a $(1+\varepsilon)$-*approximate k-nearest neighbor* (AkNN) of $\mathbf{q}$ if its distance from $\mathbf{q}$ is within a factor of $(1+\varepsilon)$ of the distance to the true k-nearest neighbor. As we show in our results, computing AkNN is sufficient in most applications. This tolerance mechanism is accomplished by computing the value of e_i as follows,

$$e_i = \frac{\|\mathbf{p}_{k+1} - \mathbf{q}_i\| \cdot (1+\varepsilon) - \|\mathbf{p}_k - \mathbf{q}_i\|}{2+\varepsilon} \,. \tag{6.6}$$

The extension of the caching mechanism to ball queries is depicted in Figure 6.3b. Let r_i be the query radius associated with the query point $\mathbf{q}_i$. First, an extended neighborhood of radius αr_i with $\alpha > 1$ is computed. The resulting neighborhood N_i can be reused for any ball query $(\mathbf{q}_j, r_j)$ with $\|\mathbf{q}_j - \mathbf{q}_i\| < \alpha r_i - r_j$. Finally, the subsequent processing operators have to check for each neighbor its distance to the query point in order to remove the wrongly selected neighbors. The value of α is a tradeoff between the cache hit rate and the overhead to compute the extended neighborhood. If an approximate result is sufficient, then a $(1+\varepsilon)$-AkNN like mechanism can be accomplished by reusing N_i if the coherence test $\|\mathbf{q}_j - \mathbf{q}_i\| < (\alpha r_i - r_j) \cdot (1+\varepsilon)$ holds true.

6.5 A hardware architecture for generic point processing

In this section we will describe our hardware architecture supporting the previously introduced algorithms for spatial searching, as well as a reconfigurable point processing module. In particular, we will focus on the design decisions and features underlying our processing architectures, while the implementation details will be described in in Section 6.6.

6.5.1 Overview

Our architecture is designed to provide an optimal compromise between flexibility and performance. Figure 6.4 shows a high-level overview of the architecture. The two main modules, the neighbor search module and the

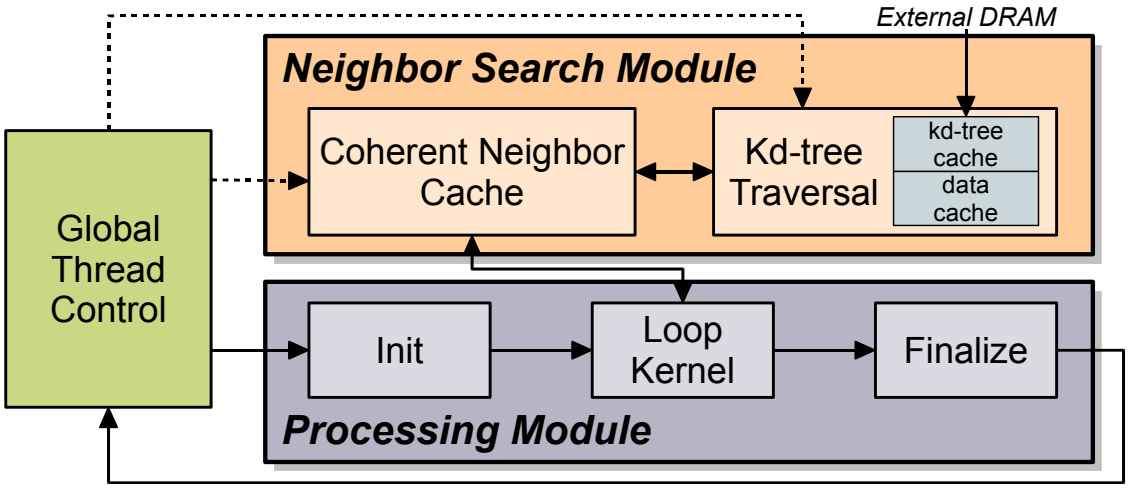


Figure 6.4: *High-level overview of our architecture. The two modules "Neighbor Search" and "Processing" can be operated separately or in tandem.*

processing module, can both be operated separately or in tandem. A global thread control unit manages user input and output requests as well as the module's interface to the driver and subsequently to high level programming languages, such as C++. The thread control could furthermore serve as interface to our point rendering architecture presented in Section 5.3, and would be integrated in conjunction with the tessellation stage after the vertex shader stages in Figure 5.2.

The core of our architecture is the configurable *neighbor search module*, which is composed of a *k*d-tree traversal unit and a coherent neighbor cache unit. We designed this module to support both *k*NN and εN queries with maximal sharing of resources. The module uses fixed function data paths and control logic for maximum throughput and for small chip area consumption. We furthermore designed every functional unit to take maximum advantage of hardware parallelism. Multi-threading and pipelining were applied to hide memory and arithmetic latencies. The fixed function data path also allows for minimal thread-storage overhead. All external memory accesses are handled by a central memory manager and supported by data and *k*d-tree caches.

In order to provide optimal performance on limited hardware, our *processing module* is also implemented using a reconfigurable fixed function data path design. Programmability is achieved through the configurability feature of FPGA devices and by using a custom hardware compiler for arithmetic instructions. The integration of our architecture with existing or future general purpose computing units like GPUs is discussed in section 6.7.2.

A further fundamental design decision is that the *k*d-tree construction is

currently performed by the host CPU and transferred to the subsystem. This decision is justified given that the tree construction can often be accomplished in a preprocess for static point sets, whereas neighbor queries have to be carried out at runtime for most point processing algorithms. More importantly, our experiments have also shown that for moderately sized dynamic data sets, the kd-tree construction times are negligible compared to the query times.

Before going into detail, it is instructive to describe the procedural and data flows of a generic operator applied to some query points: After a request for a given query point is issued, the coherent neighbor cache is checked first. If a cached neighborhood can be reused, a new processing request is generated immediately. If no such neighborhood can be found, a new neighbor search thread is issued. Once a neighbor search thread is terminated, the least recently used neighbor cache entry is replaced with the attributes of the found neighbors and a processing thread is generated. The processing thread loops over the neighbors and writes the results into the output buffer, from where they are eventually read back by the host.

In all subsequent figures, *blue* indicates memory while *green* stands for arithmetic and control logic.

6.5.2 kd-tree traversal unit

The kd-tree traversal unit is designed to receive a query $(\mathbf{q}, r)$ and to return at most the k-nearest neighbors of $\mathbf{q}$ within a radius r. The value of k is assumed to be constant for a batch of queries.

This unit starts a query by initializing the priority queue with empty elements at distance r, and then performs the search following the algorithm of Listing 6.2. Although this algorithm is a highly sequential operation, we can identify three main blocks to be executed in parallel due to their independence in terms of memory access. As depicted in Figure 6.5, these blocks include NODE TRAVERSAL, STACK RECURSION, and LEAF PROCESSING.

The NODE TRAVERSAL subunit traverses the path from the current node down to the leaf cell containing the query point. Memory access patterns include reading of the kd-tree data structure and writing to a dedicated STACK. This stack is explicitly managed by our architecture and contains all traversal information for backtracking. Once a leaf is reached, the LEAF PROCESSING subunit gathers all points contained in that leaf node and inserts the into one of the PRIORITY QUEUES of length k. Memory access patterns include reading point data from external memory and read-write access to the priority queue.

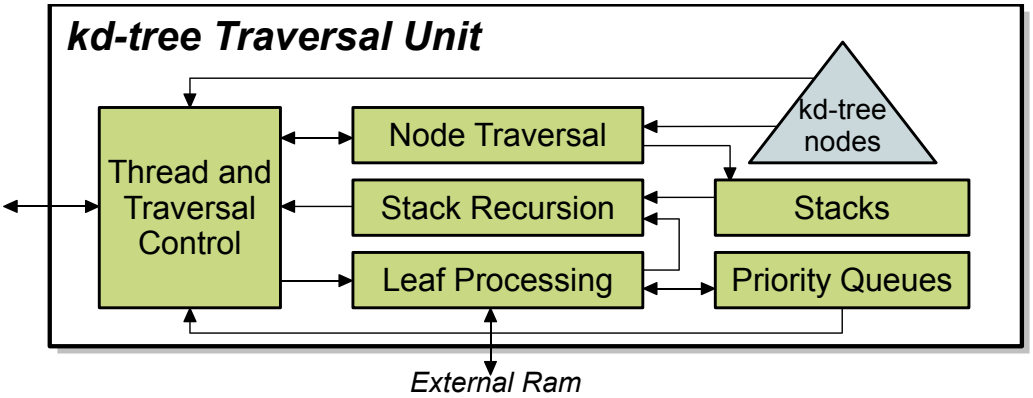


Figure 6.5: *Top level view of the kd-tree traversal unit.*

After a leaf node has been left, backtracking is performed by recurring up the stack until a new downward path is identified in the STACK RECURSION unit. The only memory access is reading the stack.

6.5.3 Coherent neighbor cache unit

The coherent neighbor cache unit (CNC), depicted in Figure 6.6, maintains a list of the m most recently used neighborhoods in a least recently used order (LRU). For each cache entry the list of neighbors N_i, its respective query position $\mathbf{q}_i$, and a generic scalar comparison value c_i, as defined in Table 6.1, are stored. The *coherence check* unit uses the latter two values to determine possible cache hits and issues a full kd-tree search otherwise.

The *neighbor copy* unit updates the neighborhood caches with the results from the kd-tree search and computes the comparison value c_i according to the current search mode. For correct kNN results, the top element corresponding to the $(k+1)$NN needs to be skipped. If ϵN queries have been issued, all elements that are further than the search radius need to be discarded. Note that this module works very similar for both kNN and ϵN, the subtle differences for the execution are summarized in Table 6.1.

6.5.4 Processing module

The processing module, depicted in Figure 6.7, is composed of three customizable blocks: an initialization step, a loop kernel executed sequentially for each neighbor, and a finalization step. The three steps can be globally iterated

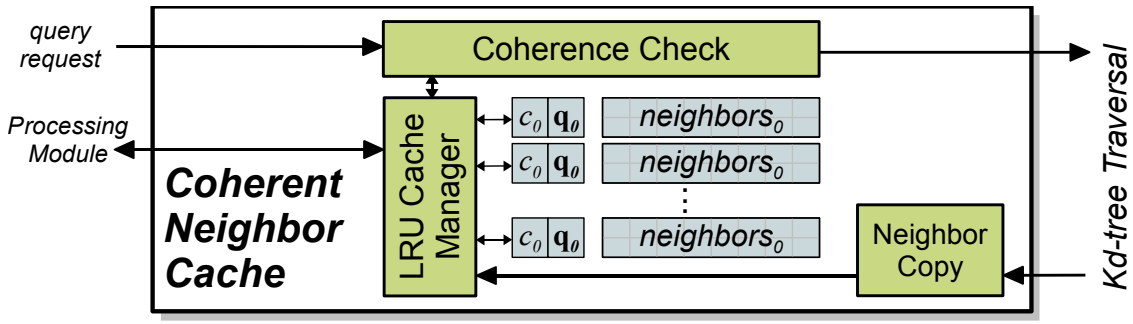


Figure 6.6: *Top level view of coherent neighbor cache unit.*

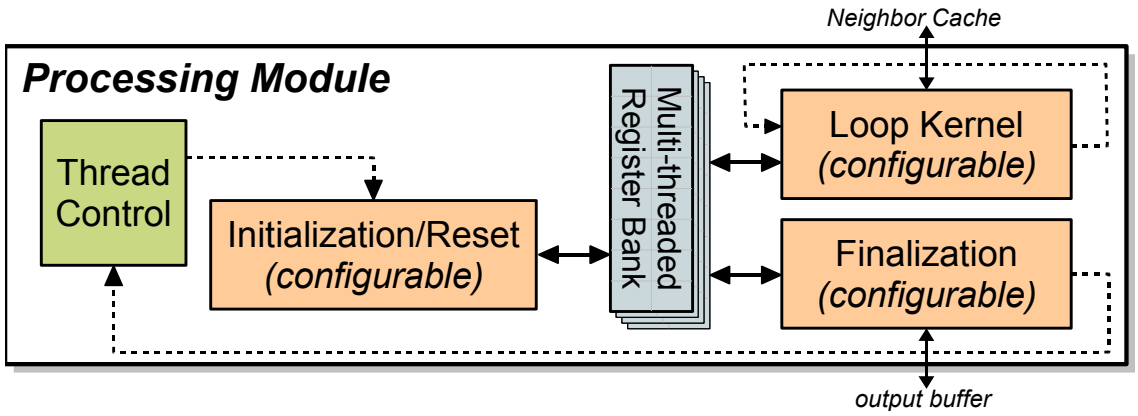


Figure 6.7: *Top level view of our programmable processing module.*

Search mode	kNN	εN
$c_i =$	e_i (Eq. 6.6)	distance of top element
Skip top element:	always	if it is the empty element
Coherence test:	$\lVert\mathbf{q}_j - \mathbf{q}_i\rVert < c_i$	$\lVert\mathbf{q}_j - \mathbf{q}_i\rVert < c_i - r_j$
Generated query:	$(\mathbf{q}_j, \infty)$	$(\mathbf{q}_j, \alpha r_j)$

Table 6.1: *Differences between kNN and εN searches in the coherent neighbor cache unit.*

multiple times, where the finalization step controls the termination of the loop. This outer loop can then be used to implement a big variety of graphics algorithms, e.g., an iterative MLS projection procedure or meshless fluid computations. Listing 6.3 shows an instructive control flow of the processing module.

Listing 6.3: *Top-level view of the type of algorithm that can implemented by the processing module.*

```
Vertex neighbors[k]; // custom type
OutputType result; // custom type
int count = 0;
do {
  init(query_data, neighbors[k], count);
  for (i=1..k)
    kernel(query_data, neighbors[i]);
} while (finalization(query_data, count++, &result));
```

All modules have access to the query data (position, radius, and custom attributes) and exchange data through a shared register bank. The initialization step furthermore has access to the farthest neighbor, which can be especially useful to, e.g., estimate the sampling density. All modules operate concurrently, but on different threads.

The three customizable blocks are specified using a pseudo assembly language which our arithmetic compiler transforms into fixed function data paths and control logic. The compiler supports various kinds of floating point and integer arithmetic operations, comparison operators, and reads and conditional writes to the shared register bank. Fixed size loops can be achieved using loop unrolling, variable loops can be achieved using the full computation loop as described in Listing 6.3. The compiler can either generate optimized, high-performance full throughput data paths, or can employ resource sharing for smaller throughput and smaller resource footprint in case of limited hardware resources.

This arithmetic hardware compiler would be particularly interesting for embedded FPGA devices or FPGA coprocessors [Vah07, Tan06, Int07]. For such systems, software developers can develop arithmetic loops efficiently without knowledge of hardware design using our compiler. When embedded into a GPU as presented in Section 5.3, this module would rather share the unified shader architecture present in such architectures instead of using FPGA reprogrammability.

6.6 Prototype implementation

This section describes the prototype implementation of the presented architecture using Field Programmable Gate Arrays (FPGAs). We will focus on the key issues and non-trivial implementation aspects of our system. At the end

of the section, we will also briefly sketch some possible optimizations of our current prototype, and describe our GPU based reference implementation that will be used for comparisons in the result section.

6.6.1 System setup

The two modules, neighbor search and processing, are mapped onto two different FPGAs. Each FPGA is equipped with a 64 bit DDR DRAM interface and both are integrated into a PCI carrier board, with a designated PCI bridge for host communication. The two FPGAs communicate via dedicated LVDS DDR communication units. The nearest neighbors information is transmitted through a 64 bit channel, the results of the processing operators are sent back using a 16 bit channel. Although the architecture would actually fit onto a single Virtex2 Pro chip, but we strived to cut down the computation times for the mapping, placing, and routing steps in the FPGA synthesis. The communication does not degrade performance and adds only a negligible latency to the overall computation.

6.6.2 *k*d-tree traversal unit

We will now revisit the *k*d-tree traversal unit of the neighbor search module illustrated in Figure 6.5 and discuss its five major units from an implementational perspective.

The following implementation allows up to 16 parallel threads operating in the *k*d-tree traversal unit. A detailed view of the stack, stack recursion, and node traversal units is presented in Figure 6.8. The *node traversal unit* determines the distance of the query point to the half planes, pushes the path to the further half plane onto a shared stack for subsequent backtracking, and continues the traversal in the nearer half plane. The *stack recursion unit* retrieves unprocessed paths from the stack in the backtracking phase of the algorithm, compares the distance of the half-plane to the query point with the current search radius. If the distance is bigger than the radius, the unit continues the backtracking. If the distance is smaller, a new traversal request is issued. Both units are fully pipelined and can accept a new request in every clock cycle.

Each of the parallel threads has its own small lightweight stack, managed by the *stack unit*. Compared to general purpose architectures, however, our stacks are much lighter and are therefore stored on-chip to maximize performance. Only pointers to unprocessed tree paths as well as the distances

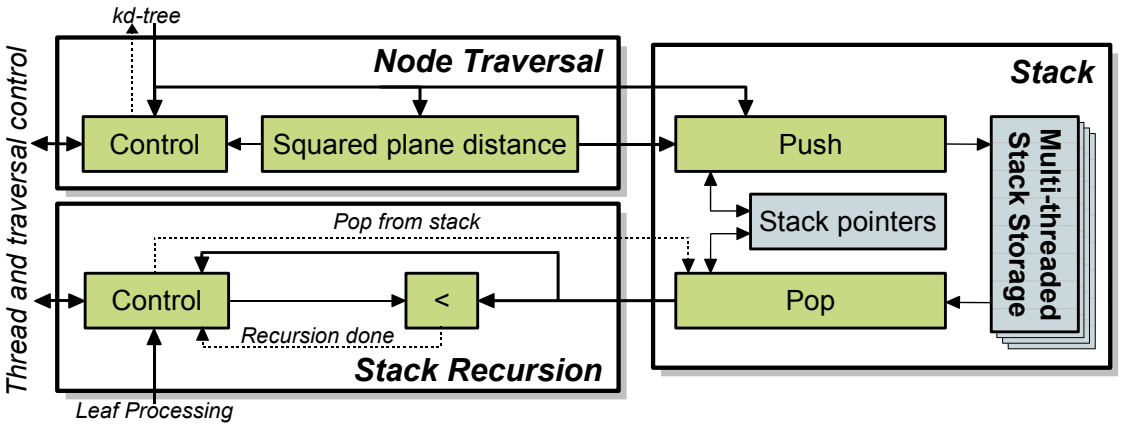


Figure 6.8: *Detailed view of the sub-units and the storage of the node traversal, stack recursion and stack units. These units are responsible of descending down the tree, keeping track of the paths not taken in the stack, and recurring up the tree.*

of the query point to the bounding plane need to be stored. Therefore, each stack entry is composed of 6 bytes only, 2 bytes store the next tree address and 4 bytes store the floating point distance. For practical reasons we bound the tree depth to reasonable maximum, and therefore the stack is bound as well. Our current implementation includes 16 parallel stacks: one for each thread and each stack having a depth of size 16. The stack supports one push operation to any stack and pop operation from any other stack in parallel in the same clock cycle. New stack operations can be issued in every clock cycle.

The leaf processing and priority queue units are presented in greater detail in Figure 6.9. The *leaf processing unit* manages the access to external memory whenever a thread arrives at a leaf. It requests all points contained in the leaf node and calculates the distances to the query point. The leaf points are then inserted into the thread's priority queue. The *priority queue unit* manages the priority queues of all threads. Similar to the stack unit, there are 16 priority queues on-chip. The queues store the distances as well as pointers to the point data cache. Our implementation uses a fully sorted parallel register bank of length 32 and therefore supports up to 32 nearest neighbors. It allows the insertion of one element in every clock cycle. Note that this fully sorted bank works well for small queue sizes because of the associated short propagation paths and small area overhead. For larger k, a constant time priority queue similar to the pipelined heap [WHA+07] could be used. The pipelined heap is presented in Chapter 5.

For ease of implementation, the *kd-tree structure* is currently stored linearly on-chip in the spirit of a heap: Given the address i in memory, the two

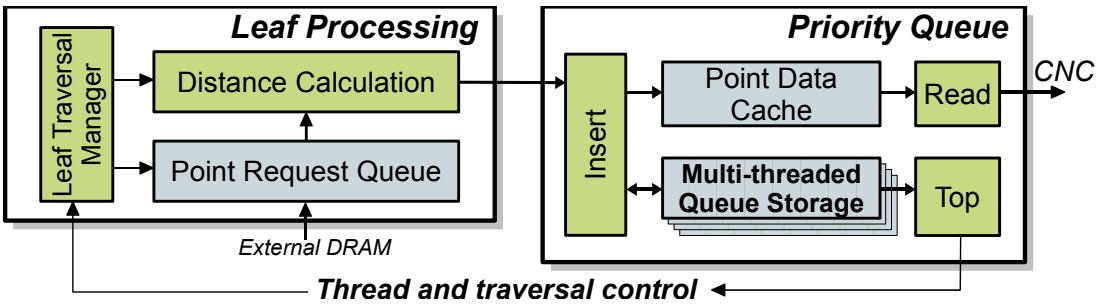


Figure 6.9: *Detailed view of the leaf processing and parallel priority queue units. The sub-units load external data, compute distance and resort the parallel priority queues.*

corresponding children are stored at locations $2i$ and $2i+1$. We bound the maximum tree depth to 14 levels. The internal nodes and the leaves are stored in separate blocks where points are associated with leaf nodes only. The $2^{14}-1$ internal nodes store the splitting planes (32 bit floating point), the dimension (2 bit enumerator) and a flag indicating when an internal nodes is a leaf (1 bit). This additional bit allows to support unbalanced kd-trees as well. The 2^{14} leaf nodes store begin and end pointers to the point buckets in the off-chip DRAM (25 bits each).

The total storage requirement of the full kd-tree is therefore 170 kBytes only. A tree using a pointer representation would require 217 kBytes for a fully balanced tree[1].

In case of an unbalanced leaf, a so-called 'internal leaf' then needs to be mapped to a leaf node. Instead of storing the address, the corresponding leaf point address a_{leaf} for any given internal node address a_{internal} is then found by

$$a_{\text{leaf}} = a_{\text{internal}}\, 2^{d-l(\text{internal})}\,, \tag{6.7}$$

The full tree depth is denoted by d, and $l(\text{internal})$ denotes the tree level at which the internal node is located. Note that this computation essentially shifts the internal address such that the leading 1 in the address is then located in the most significant bit of the tree address.

[1]This calculation has been performed for a storage optimized tree in pointer representation: 2 bit are used to denote the dimension, 1 bit whether the node is internal and external, and 50 bit are combined storage: either the splitting plane and the two pointer to the children can be combined in 50 bit in case of an internal node, otherwise the start and end pointers to the point buckets can be stored

6.6.3 Coherent neighbor cache unit

The coherent neighbor cache (CNC) unit (Figure 6.6) can store eight cached neighborhoods and contains a single *coherence check* sub-unit. This sub-unit tests incoming queries for cache hits by iterating over all eight entries. The *LRU cache manager* unit maintains the list of caches in least recently used (LRU) order and synchronizes queries between the processing module and the *k*d-tree search unit using a multi-reader write lock primitive. These locks can be either acquired for reading or writing and allows multiple threads to read from a cache entry simultaneously, but writing threads must acquire an exclusive ownership. For a higher number of cache entries, the processing time increases linearly due to the iteration step. As a remedy, however, additional coherence test units could be used to partition the workload and hence reduce the number of iterations.

As the neighbor search unit processes multiple queries in parallel, it is important to carefully align the processing order. Usually, subsequent queries are likely to be spatially coherent and would eventually be issued concurrently to the neighbor search unit. Therefore, the neighbor search unit would then search for spatially similar neighborhoods at the same time and would render the CNC useless. To prevent this problem, the queries should be interleaved. In the current system this task is left to the user, which allows to optimally align the interleaving based on the nature of the queries.

6.6.4 Processing module

The processing module (Figure 6.7) is composed of three reconfigurable custom units managed by a *processing controller*. The reconfigurable units represent FPGA blocks that can be reprogrammed by FPGA reconfiguration. The units communicate through a multi-threaded quad-port bank of 16 registers. The repartitioning of the four ports to the three custom units is automatically determined by our hardware compiler.

Depending on the processing operator, our compiler might produce a high number of floating point operations thus leading to significant latency, which is, however, hidden by pipelining and multi-threading. Our implementation allows for a maximum of 128 threads operating in parallel and is able to process up to one neighbor per clock cycle. In addition, a more fine-grained multi-threading scheme iterating over 8 sub-threads is used to hide the latency of the accumulation in the loop kernel.

In case the algorithm to be implemented cannot fit into the reconfigurable blocks, our hardware compiler offers the possibility to employ automatic resource sharing. The user can specify a lower rate at which queries can be issued. The lower the rate, the higher the possibility to re-use arithmetic operators for different parts of the algorithm, and the lower the maximum throughput.

6.6.5 Resource requirements and extensions

Our architecture was designed using minimal on-chip resources. As a result, the neighbor search module is very lean and uses a very small number of arithmetic units only, as summarized in Table 6.2. The number of arithmetic units of the processing module depends entirely on the processing operator. Their complexity is therefore limited by the resources of the targeted FPGA device. Table 6.2 shows two such examples.

The prototype has been partitioned into the neighbor search module integrated on a Virtex2 Pro 100 FPGA, and the processing module was integrated on a Virtex2 Pro 70 FPGA. The utilization of the neighbor search FPGA was 23'397 slice flip flops (27% usage) and 33'799 LUTs (38% usage). The utilization of the processing module in the example of a MLS projection procedure based on plane fits [AA04] required 31'389 slice flip flops (47% usage) and 35'016 LUTs (53% usage).

The amount of on-chip RAM required by the current prototype is summarized in Table 6.3, omitting the buffers for PCI transfer and inter-chip communication which are not relevant for the architecture. This table also includes a possible variant using one bigger FPGA only. The architecture then could be equipped with generic shared caches to access the external memory. The tree structure would also be stored off-chip and hence alleviate the current

Arithmetic Unit	***k*d-tree Traversal**	**CNC**	**Covariance Analysis**	**SPSS Projection**
Add/Sub	6	6	38	29
Mul	4	6	49	32
Div	0	0	2	3
Sqrt	0	2	2	2

Table 6.2: *Usage of arithmetic resources for the two units of our neighbor search module, and two processing examples.*

Data	Current Prototype	Off-chip *k*d-tree & shared data cache
Thread data	1.36 kB (87 B/thd)	1.36 kB
Traversal stack	2 kB (8×16 B/thd)	2 kB
*k*d-tree	170 kB (depth: 14)	16 kB (cache)
Priority queue	3 kB (6×32 B/thd)	4 kB
DRAM manager	5.78 kB	5.78 kB
Point data cache	16 kB (p-queue unit)	16 kB (shared cache)
Neighbor caches	8.15 kB (1044B/cache)	1.15 kB (148 B/cache)
Total	206.3 kB	46.3 kB

Table 6.3: *On-chip storage requirements for our current and planned, optimized version of the neighbor search module.*

limitation on the maximum tree depth. Furthermore, such caches would make our current *point data cache* obsolete and reduce the neighbor cache footprint by storing references to the point attributes only. Finally, this would not only optimize on-chip RAM usage, but also reduce the memory bandwidth to access the point data for the *leaf processing* unit, and hence speed up the overall process.

6.6.6 GPU implementation

For comparison, we implemented a *k*d-tree based *k*NN search algorithm using NVIDIA's CUDA. Similar to the FPGA implementation it uses lightweight stacks and priority queues which are stored in local memory. Storing the stacks and priority queues in fast shared memory would limit the number of threads drastically and actually degrade performance compared to using local memory. The neighbor lists are written and read in a large buffer stored in global memory. We implemented the subsequent processing algorithms as a second, distinct kernel. Owing to the GPU's SIMD design, implementing a CNC mechanism is not feasible and would only decrease the performance.

In contrast to the correct implementation, the paper by Zhou et al. [ZHWG08] uses an iterative range search to achieve an approximated *k*-nearest-neighbors search. In a first step, a conservative radius is estimated that contains at least *k* points. Then, the algorithm iteratively performs range searches and adjusts the range towards the true range defined by the *k*-nearest neighbors in each search step. The range adjustment is estimated using a histogram.

Unfortunately, this approach cannot yield the true nearest neighbors unless a high number of iterations is used.

Figure 6.10: *A series of successive smoothing operations. The model size is 134k points and a neighborhood of $k = 16$ has been used for the MLS projections. Only MLS software code had to be replaced by FPGA driver calls.*

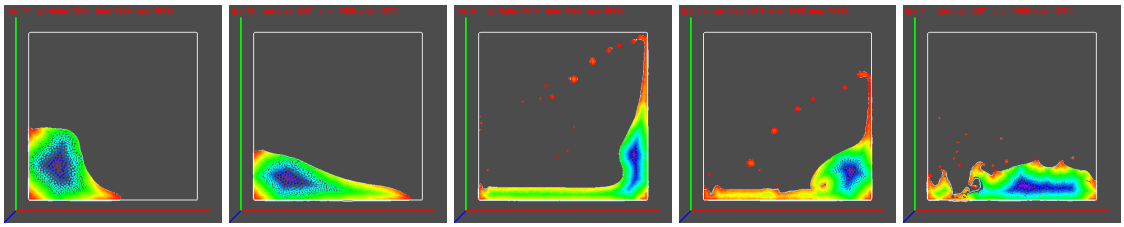


Figure 6.11: *A series of successive simulation steps of the 2D breaking dam scenario. The SPH simulation is using adaptive particle resolutions kNN queries up to $k = 30$. Only kNN software code has been replaced by FPGA driver calls.*

6.7 Results and discussions

To demonstrate the versatility of our architecture, we implemented and analyzed several meshless processing operators. These include a few core operators which are entirely performed on-chip: covariance analysis, iterative MLS projection based on either plane fit [AA04] or spherical fit [GG07], and a meshless adaptation of a nonlinear smoothing operator for surfaces [JDD03]. We integrated these core operators into more complex procedures, such as a MLS based resampling procedure, as well as a normal estimation procedure based on covariance analysis [HDD+92].

We also integrated our prototype into existing publicly available software packages. For instance, in PointShop 3D [ZPKG02] the numerous MLS calls for smoothing, hole filling, and resampling [WPK+04] have been replaced by calls to our drivers. See Figure 6.10 for the smoothing operation. Furthermore, an analysis of a fluid simulation research code [APKG07] based on *smoothed particle hydrodynamics* (SPH) showed that all the computations

involving neighbor search and processing can easily be accelerated, while the host would still be responsible for collision detection and *k*d-tree updates (Figure 6.11).

6.7.1 Performance analysis

Both FPGAs, a Virtex2 Pro 100 and a Virtex2 Pro 70, operate at a clock frequency of 75 MHz. We compare the performance of our architecture to similar CPU and GPU implementations optimized for each platform, on a 2.2 GHz Intel Core Duo 2 equipped with a NVIDIA GeForce 8800 GTS GPU. Our results were obtained for a surface data-set of 100k points in randomized order and with a dummy operator that simply reads the neighbor attributes. Note that our measurements do not include transfer costs from and to the host CPU, since our hardware device lacks an efficient transfer interface between the host and the device; the FPGA included a PCI interface that could only sustain data rates below 100 MB/s for read/write DMA transfers, as opposed to GB/s for the highly optimized PCI express interface of the GPU.

General performance analysis

Figure 6.12 demonstrates the high performance of our design for generic, incoherent *k*NN queries. The achieved on-chip FPGA query performance is about 68% and 30% of the throughput of the CPU and GPU implementations, respectively, although our FPGA clock rate is 30 times lower than that of the CPU, and it consumes considerably fewer resources. Moreover, with the MLS projection additionally enabled, our prototype exhibits the same performance as with the *k*NN queries only, and outperforms the CPU implementation.

Finally, note that when the CNC is disabled our architecture produces fully sorted neighborhoods for free, which is beneficial for a couple of applications. As shown in Figure 6.12 adding such a sort to our CPU and GPU implementations has a non-negligible impact, in particular for the GPU.

Our FPGA prototype, integrated into the fluid simulation [APKG07], achieved half of the performance of the CPU implementation. The main reason for this is because up to 30 neighbors per query had to be read back over the slow PCI transfer interface. In the case of the smoothing operation of PointShop 3D [WPK+04], our prototype achieved speed ups of a factor of 3, including PCI communication. The reasons for this speed up are two-fold: first, for MLS projections, only the projected positions need to be read back. Second, the *k*d-tree of PointShop 3D is not as highly optimized as our

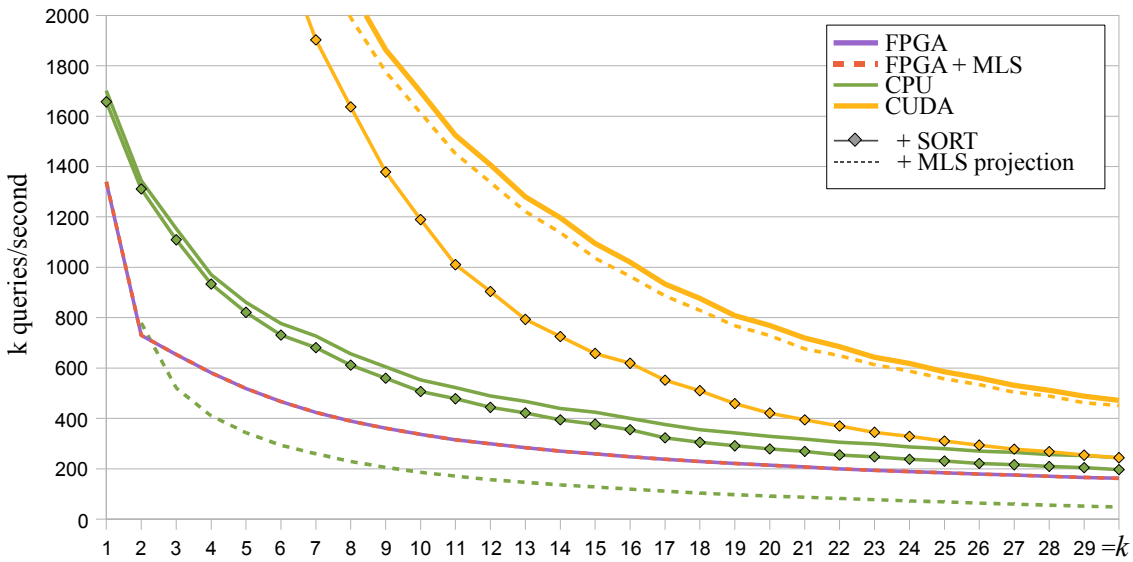


Figure 6.12: *Performance of kNN searches and MLS projections as a function of the number of neighbors k. For this measurements the input points have been used as query points.*

reference CPU implementation used for the other comparisons. Note that using a respective ASIC implementation and a more advanced PCI Express interface, the performance of our system would be considerably higher.

System bottleneck

The bottleneck of our prototype is the *k*d-tree traversal, and it did not outperform the processing module for all tested operators. In particular, the off-chip memory accesses in the *leaf processing unit* represent the limiting bottleneck in the *k*d-tree traversal. Consequently, the nearest neighbor search does not scale well above 4 threads, while supporting up to 16 threads.

More specifically, the theoretical maximum bandwidth to the external DRAM available in our FPGA boards is 1.2 GB/s, which is very small compared to the theoretical 64 GB/s available on the NVIDIA GeForce 8800 GTS. One possible straight-forward solution would therefore be to increase the memory bandwidth, or to employ a least-recently-used caching strategy for the DRAM accesses to increase the overall performance of our system.

Coherent neighbor cache analysis

In order to evaluate the efficiency of our coherent neighbor cache with respect to the level of coherence, we implemented a resampling algorithm that generates $b \times b$ query points for each input point, uniformly spread over its local tangent plane [GGG08]. All results for the CNC were obtained with 8 caches. The best results were obtained for ball queries (Figure 6.13), where even an up-sampling pattern of 2×2 is enough to save up to 75% of the kd-tree traversals, thereby showing the CNC's ability to significantly speed up the overall computation. Figure 6.14 depicts the behavior of the CNC with both exact and $(1+\varepsilon)$-approximate kNN with an upsampling pattern of 8×8. Whereas the cache hit rate remains relatively low for exact kNN especially with such a large neighborhoods, already a small tolerance ($\varepsilon = 0.1$) allows to save more than 50% of the kd-tree traversals. For incoherent queries, the CNC results in a slight overhead due to the search of larger neighborhoods. The GPU implementation does not include a CNC, but owing to its SIMD design and texture caches, its performance significantly drops down as the level of coherence decreases.

While these results clearly demonstrate the general usefulness of our CNC algorithm, they also show the CNC hardware implementation to be slightly less effective than the CPU-based CNC. The reasons for this behavior are twofold. First, from a theoretical point of view, increasing the number of threads while keeping the same number of caches decreases the hit rate for the CNC. This behavior could be compensated by increasing the number of caches. Second, our current prototype consistently checks all caches sequentially while our CPU implementation stops at the first cache hit, at a much lower clock frequency.

An analysis of the $(1+\varepsilon)$-approximate kNN in terms of cache hit rate and relative error can be found in Figures 6.15 and 6.16. These results show that already small values of ε are sufficient to significantly increase the percentage of cache hits, while maintaining a very low error for the MLS projection. In fact, the error is of the same order as the numerical order of the MLS projection in case of exact neighbor queries. Even larger tolerance values like $\varepsilon = 0.5$ lead to visually acceptable results, which is due to the weight function of the MLS projection that results in low influence of the farthest neighbors.

Comparison to Zhou et al. [ZHWG08]

Zhou et al. [ZHWG08] implemented point cloud modeling using a k-nearest-neighbor search on GPUs. Their implementation achieved 9.1 million queries

per second, for a neighborhood of size 10. For the same neighborhood and similar model size, our exact GPU implementation yields about 1.8 million queries per second (Figure 6.12). We believe this performance gap is due to the following three reasons. First, the authors use approximate nearest neighbors search and can therefore tolerate some error at the benefit of performance. Second, the tree built by Zhou et al. is more optimized for neighborhood searches. In fact, the authors claim that mid-point splitting as used for our tree construction results in poor tree structures. Third, the application used by Zhou et al. exhibits high spatial locality, whereas our analysis in Figure 6.12 uses spatially randomized queries. The performance of our GPU implementation however also increases with locality, as can be seen in Figures 6.13 and 6.14.

6.7.2 GPU integration

Our results show that the GPU implementation of kNN search is only slightly faster than our current FPGA prototype and CPU implementation. Moreover, with a MLS projection operator on top of a kNN search, we observe a projection rate between 0.4M and 3.4M projections per second, while the same hardware is able to perform up to 100M of projections per second using precomputed neighborhoods [GGG08]. Actually, the kNN search consumes more than 97% of the computation time. This poor performance is partly due to the divergence in the tree traversal, but even more important, due to the priority queue insertion in $O(\log k)$, which infers many incoherent execution paths. On the other hand, our design optimally parallelizes the different steps of the tree traversal and allows the insertion of one neighbor into the priority queue in a single cycle.

These results motivate the integration of our lightweight neighbor search module into such a massively multi-core architecture. Indeed, a dedicated ASIC implementation of our module could be further optimized and run at a much higher frequency and could improve the performance by more than an order of magnitude. Such an integration could be done in a similar manner as the dedicated texturing units, or the programmable tessellation units of current GPUs [Mic10]. In conjunction with the previously proposed rendering architecture for surface splatting in Chapter 5, a full point processing and rendering pipeline could provide extremely high geometric detail at low bandwidth. In such a context, our processing module would then be replaced by more generic computing units. Nevertheless, we emphasize that the processing module still exhibits several advantages. First, it allows to optimally use FPGA devices as co-processors to CPUs or GPUs,

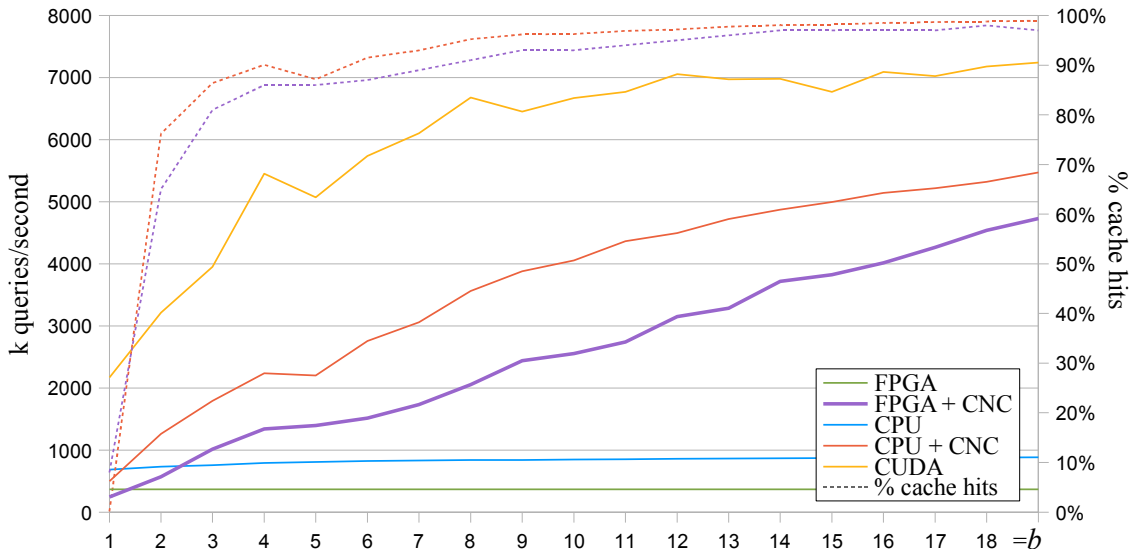


Figure 6.13: *Number of ball queries per second for an increasing level of coherence.*

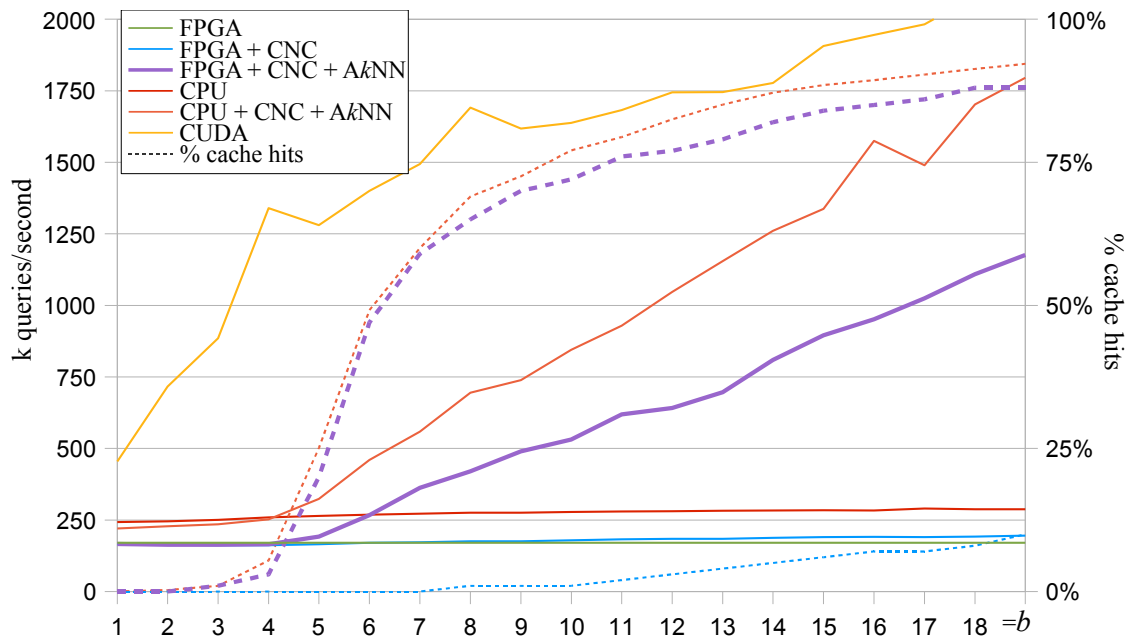


Figure 6.14: *Number of kNN queries per second for an increasing level of coherence. The approximate kNN (AkNN) results with $k = 30$ neighbors were obtained with $\varepsilon = 0.1$.*

which can be expected to become more and more common in the upcoming years [Vah07, Tan06, Int07]. Second, unlike the SIMD design of GPU's microprocessors, our custom design with three sub-kernels allows for optimal throughput, even in the case of varying length neighbor loops.

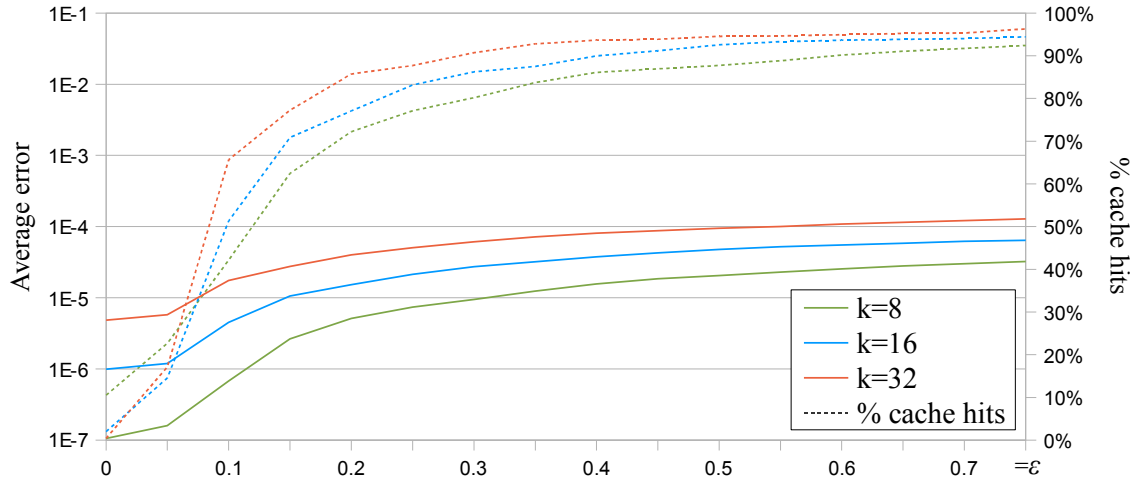


Figure 6.15: *Error versus efficiency of the* $(1+\varepsilon)$*-approximate k-nearest neighbor mechanism as function of the tolerance value* ε*. The error corresponds to the average distance for a model of unit size. The percentages of cache hit were obtained with* 8 *caches and patterns of* 8×8*.*

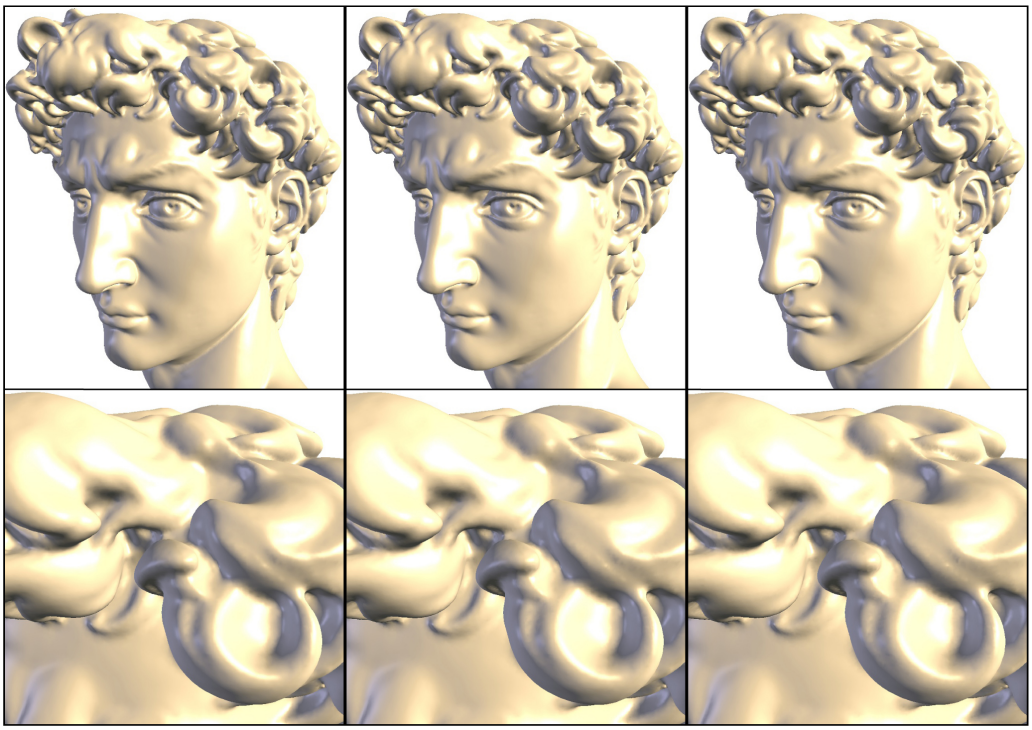


Figure 6.16: *MLS reconstructions with, from left to right, exact kNN (34 s), and AkNN with* $\varepsilon = 0.2$ *(12 s) and* $\varepsilon = 0.5$ *(10 s). Without our CNC the reconstruction takes 54 s.*

6.8 Conclusion

We presented a novel hardware architecture for efficient nearest-neighbor searches and generic meshless processing operators. In particular, our *k*d-tree based neighbor search module features a novel and dedicated caching mechanism exploiting the spatial coherence of the queries. Our results show that neighbor searches can be accelerated efficiently by identifying independent parts in terms of memory access. Our architecture is implemented in a fully pipelined and multi-threaded manner and suggests that its lightweight design could be easily integrated into existing computing or graphics architectures, and hence be used to speed up applications depending heavily on data structure operations. When scaled to realistic clock rates, our implementation achieves speedups of an order of magnitude compared to reference implementations. Our experimental results prove the high benefit of a dedicated neighbor search hardware.

This architecture should be combined with the rendering architecture presented in Chapter 5 for a general point chip: while the processing architecture would be responsible for the processing and supersampling of a coarse input point cloud in the sense of modern tessellation units, the rendering architecture then would rasterize the generated samples in screen space.

A current limitation of the design is its off-chip tree construction. An extended architecture could construct or update the tree on-chip to avoid expensive host communication. Zhou et al. [ZHWG08] suggest that the tree construction should be optimized for *k*-nearest-neighbor searches, and their method could be used to construct the tree. Furthermore, a more configurable spatial data structure processors in order to support a wider range of data structures and applications could be very useful for a broader range of application.

CHAPTER 7

Conclusion

This thesis presented an analysis of point-based graphics in the context of hardware acceleration. First we extended the EWA surface splatting framework to the time domain, and then analyzed and identified the main bottlenecks and shortcomings of existing hardware architectures with respect to point rendering. We then introduced novel architectural elements and concepts to enhance hardware support for point based rendering. Finally, we investigated a generalized hardware architecture for the processing of point sets to support novel algorithms, such as tessellation or surface smoothing, which showed great potential to accelerate point-based graphics and to alleviate shortcomings of previous architectures.

7.1 Review of principal contributions

We extended the EWA surface splatting framework by a time domain to support motion-blur for point-based rendering. Conceptually we replaced the 2D Gaussian kernels encoding the surface domain by 3D Gaussian kernels encoding the surface in the time domain. The 3D kernels unify the spatial and temporal component, and represent a linearized sampling of a moving surface in time. The derived result naturally fits into the EWA splatting algorithm in such that the final image can be computed as a weighted sum of warped

and bandlimited kernels. We presented a rendering algorithm with strong parallels to the original EWA rendering: the 3D kernels are integrated along the viewing ray and accumulated with kernels belonging to the same surface patch, which are then combined with temporally overlapping surfaces using an A-Buffer like approach. Additionally to the correct rendering approach, we provided an approximative implementation of the entire point rendering pipeline using vertex, geometry and fragment program capabilities of current GPUs. The correct algorithm was able to produce high-quality motion blur for point based objects, the GPU implementation was limited in terms of visual quality and performance due to incompatibilites of modern GPUs with point-based rendering.

Based on the results of the previous analysis, we then introduced a rendering architecture for surface splatting to overcome the limitations of the available graphic processors. The architecture used a refined version of EWA surface splatting optimized for hardware implementations. Central to our design decisions was the seamless integration of the architecture into conventional, OpenGL-like graphics pipelines, and the architecture was designed to complement triangle rendering instead of replacing it. We strived to re-use as much existing hardware concepts such as the vertex and fragment processor, however the fundamental differences of EWA surface splatting also required some novel design concepts. The rasterization part has been completely redesigned to support a ternary depth test and the reconstruction buffer, which can only be released as soon as a surface has been completely reconstructed. Furthermore, we introduced an on-chip pipelined heap to reorder screen space points and to make accesses to the reconstruction buffer memory more coherent. We implemented different versions of the pipeline both on reconfigurable FPGA boards and ASIC prototypes to demonstrate the performance of the proposed pipeline, and we integrated the pipeline into an OpenGL-like software implementation to prove the usability of the combined triangle-splat rendering approach.

Finally, we generalized the rendering architecture by developing a hardware architecture for the processing of unstructured point sets. Our new design supported the two computationally most expensive operations on point sets: the neighborhood search and the stream processing of the found neighborhood. A central design decision was the fixed function *k*d-tree-based neighborhood search in conjunction with a novel caching mechanism to exploit the spatial coherency of many point processing operators. The stream processing was then achieved using the reconfigurability of FPGA devices: we developed a hardware compiler able to transform assembler-like code to FPGA blocks which in turn can directly be plugged into the processing module. The prototype implementation showed versatility and very good

results even at low clock frequencies and at low memory bandwidth. The neighbor search module proved to be very lean and lightweight, and could be integrated as valuable co-processor block in other hardware architectures at low cost. Although the processing module was very similar to available stream processing units, the concept of FPGA reconfigurability could be used for hybrid systems containing FPGA logic.

7.2 Discussion and future work

This thesis investigated the hardware acceleration of point-based graphics by combining general purpose processing with small fixed function units targeted for point-processing, for example the k-nearest-neighbor search or the ternary depth test for the rasterization of splats. Within the time-frame of this thesis, the state-of-the-art GPU architectures became more and more programmable and converged very close towards multi-core CPUs. While many parts of the original pipeline such as geometry processing or tessellation became programmable, there are still open issues that limit point-graphics algorithms to take full advantage of such architectures. We see the following areas for future work.

Programmable tessellation units. With the advent of programmable tessellation new possibilities for upsampling of point sampled data have become possible. However, point based graphics relies strongly on unstructured point sets, whereas tessellation requires explicit knowledge of neighbors. One of the questions that arises is how to incorporate a nearest neighborhood search tightly with tessellation units in order to provide maximum performance for upsampling of point based models to the pixel level. Moreover, it is not yet clear how this tessellation could be effectively combined with polygonal meshes, or if there is potential to even sample polygonal models with points for more efficient rendering.

Micro-point rendering instead of micro-polygon rendering. Current tessellation units are able to generate highly detailed triangle meshes containing millions of tiny triangles. Although micro-polygon rendering methods have been devised for such cases, points are a much simpler representation. However, in its current EWA formulation, such micro-points are not yet being rendered as efficient as possible. By developing a simpler splatting scheme, micro-points could probably outperform micro-polygons by an order of magnitude, while requiring less complicated hardware architectures with smaller memory footprint.

A complete point-graphics hardware pipeline. Ideally, the full hardware graphics pipeline would include both the generation of point rendering primitives from a sparse input point point cloud as well as the rendering of those primitives using EWA surface splatting. Furthermore, a tessellation unit that converts triangle representations to point representations on the fly would provide a general purpose rendering architecture that supports even the traditional rendering and production pipelines. The generation of point primitives would be performed using an architecture similar to our general point processor used as general purpose tessellator, and the output would then be forwarded to the EWA surface splatting architecture. Such a combined architecture looks very promising as the input bandwidth could be decreased by orders of magnitude, and the architecture could adaptively upsample micro-points to match the resolution required for the display.

A programmable data structure processor. The fixed-function architecture for neighborhood queries showed that parts of a recursive search on tree-like structures can be parallelized quite efficiently. To make such an architecture more general, those separate fixed function parts could be replaced by small lightweight processors performing data structure shaders, and could be used as co-processor to current architectures. Furthermore, the general caches could be replaced by programmable caching controllers to adapt to special access patterns of different data structure searches. A programmable data structure processor then could be used to implement recursive searches on hierarchical data-structures more efficiently and with higher hardware utilization as the data-parallel multi-core GPUs. Therefore, the GPU itself would concentrate on the stream processing of data being supplied by the data structure processor.

APPENDIX A

Notation

This appendix covers the used notation, split into the associated chapters.

A.1 General mathematical notation

$\mathbb{N}$ ..... Set natural numbers.

$\mathbb{R}$ ..... Set of real numbers.

$\mathbb{R}^n$ ..... Set of real n-vectors.

$\mathbb{R}^{m \times n}$ ..... Set of real $m \times n$ matrices with m rows and n columns.

$\mathbf{a} \in \mathbb{R}^n$ ..... Bold lowercase letters are denoting column vectors of any dimension.

$\mathbf{a}_{xy}$ ..... Two component sub-vector of $\mathbf{a}$, containing the x and y component only.

$\mathbf{A} \in \mathbb{R}^{m \times n}$ ..... Bold uppercase letters are denoting matrices of any dimension.

$\mathbf{A}_{2 \times 2}$ ..... 2×2 sub-matrix of $\mathbf{A}$, containing the upper $2x2$ block of $\mathbf{A}$

$a(\cdot), A(\cdot) \in \mathbb{R}$ ..... Scalar functions.

$\mathbf{a}(\cdot) \in \mathbb{R}^n$ ..... Vector functions.

$\mathcal{A} = \{\ldots\}$ ..... Set of properties or values.
$*$ ..... Convolution operator.
$\mathbf{a}^T, \mathbf{A}^T$ ..... Transposed vector / matrix.
$\mathbf{A}^{-1}$ ..... Inverse of matrix $\mathbf{A}$.
$\|\mathbf{a}\|$ ..... L2-norm of a vector $\mathbf{a}$.
$|\mathbf{A}|$ ..... Determinant of matrix $\mathbf{A}$.
$e^{\cdot}$ ..... Exponential function.
$\mathrm{diag}(a_1,\ldots,a_n)$ ..... $n \times n$ diagonal matrix, with $a_1,\ldots,a_n$ as diagonal elements

A.2 Processing of point sets

$\mathcal{S}$ ..... Defining set of a surface.
$\hat{\mathcal{S}}$ ..... Approximating set of a surface.
$\theta(x)$ ..... Weight function.
$\mathbf{p}_i$ ..... Points of a surface.
$\mathbf{n}_i$ ..... Associated normals of the surface.
$\mathbf{x}$ ..... A point in space.
$\mathbf{q}$ ..... Query point.
r ..... Search radius.
e_i ..... Tolerance distance.
a_i ..... Memory location.

A.3 EWA surface splatting

$\mathcal{P}_k$ ..... Point k
$r_k(\cdot)$ ..... Elliptical 2D reconstruction kernel of $\mathcal{P}_k$.
$R_k(\cdot)$ ..... Ellipsoidal 3D reconstruction kernel of $\mathcal{P}_k$.
$\mathbf{a}_k^{[1]}, \mathbf{a}_k^{[2]}$ ..... Axis vectors spanning coordinate frame of $\mathbf{r}_k(\cdot)$.
$\mathbf{a}_k^{[1]}, \mathbf{a}_k^{[2]}, \mathbf{a}_k^{[3]}$ ..... Axis vectors spanning coordinate frame of $\mathbb{R}_k(\cdot)$.
$\mathbf{c}_k$ ..... Center of coordinate frame of $\mathcal{P}_k$.
$\mathbf{s}_k$ ..... Attribute sample vector of $\mathcal{P}_k$.
$\mathbf{u}$ ..... Point in tangent frame.

$\mathbf{x}'$ ..... Point in screen frame.

$\mathbf{x}$ ..... Point in world frame.

$\mathbf{M}$ ..... Modelview matrix.

$\mathbf{P}$ ..... Projection matrix.

$\mathbf{T}_k$ ..... Axis transformation from coordinate frame of $\mathcal{P}_k$ to object space.

$\mathbf{V}_k$ ..... Variance matrix of $\mathcal{P}_k$.

$G^2_{\mathbf{V}}(\cdot)$ ..... 2-dimensional Gaussian with 3x3 variance matrix $\mathbf{V}$.

$G^3_{\mathbf{V}}(\cdot)$ ..... 3-dimensional Gaussian with 4x4 variance matrix $\mathbf{V}$.

$f(\cdot)$ ..... Source space attribute function of surface.

$\mathbf{m}(\cdot)$ ..... Projective mapping from source space to screen space.

$g(\cdot)$ ..... Screen space attribute function of surface.

$h(\cdot)$ ..... Screen space anti-alias pre-filter.

$\rho_k(\cdot)$ ..... Filtered screen space resampling kernel.

$v(\cdot)$ ..... Visibility function.

$i \prec_I j$ ..... Cyclic ordering criterion, defined in Section 5.5.2.

APPENDIX B

Glossary

A-Buffer ..... Anti-aliased, area-averaged, accumulation buffer, defined by Carpenter [Car84]

APSS ..... Algebraic point set surfaces, defined by Guennebaud et al. [GG07].

AkNN ..... Approximated k spatially nearest neighbors.

API ..... Application programming interface.

ASIC ..... Application specific integrated circuit.

ATI ..... ATI Technologies, a manufacturer of GPUs.

C++ ..... A programming language.

CNC ..... Coherent neighbor cache.

CPU ..... Central processing unit. Main processing chip inside a computer system.

CUDA ..... Compute unified device architecture, a software development kit for GPUs and marketing term by NVIDIA.

DDR ..... Double data rate.

DDR2, DDR3 ..... Double data rate protocols for DRAM.

DMA ..... Direct memory access.

DRAM ..... Dynamic random access memory.

DSP ..... Digital signal processor.

DVI ..... Digital visual interface.

EWA ..... Elliptical weighted average.

EWA splatting ..... Elliptical weighted average surface splatting, a method to render high-quality anti-aliased images from 3D point models.

ϵN ..... Range search for all points contained in a given euclidian radius.

FIFO ..... First in first out.

FPGA ..... Field programmable gate array, which can be considered reconfigurable logic.

GPU ..... Graphics processing unit, a chip used for acceleration of 3D graphics.

IC ..... Integrated circuit.

IEEE ..... Institution of Electrical and Electronics Engineers, a non-profit professional organization.

k-d tree ..... Spatial data structure commonly used for point graphics and ray tracing.

*k*NN ..... *k* spatially nearest neighbors.

LRU ..... Least recently used, a processing order often employed in caches.

LUT ..... Look-up table, an element commonly found in FPGAs.

LVDS ..... Low voltage differential signaling.

MHz ..... Clock frequency of ICs: the number of clock ticks per second in millions.

MLS ..... Moving least squares.

mW ..... Milli-Watt.

NVIDIA ..... A manufacturer of GPUs.

OpenGL ..... Open Graphics Library, a rendering pipeline.

PCB ..... Printed circuit board.

PCI ..... Peripheral Component Interface.

PSS ..... Point set surfaces.

SIMD ..... Single instruction, multiple data.

SRAM ..... Static random access memory.

SPH ..... Smoothed particle hydrodynamics.

SPSS ..... Simple point set surfaces as defined by Adamson and Alexa [AA04].

T&L ..... Transform and lighting.

USB ..... Universal serial bus protocol.

Virtex2 Pro ..... A FPGA chip manufactured by Xilinx.

Xilinx ..... A manufacturer of FPGA chips.

Z-Buffer ..... Depth buffer.

Bibliography

[AA03] Anders Adamson and Marc Alexa. Ray tracing point set surfaces. In *Conference on Shape Modeling and Applications*, 2003.

[AA04] M. Alexa and A. Adamson. On normals and projection operators for surfaces defined by point sets. In *Proceedings of Symposium on Point Based Graphics*, pages 149–155, 2004.

[AA06] Anders Adamson and Marc Alexa. Point-sampled cell complexes. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 671–680, New York, NY, USA, 2006. ACM Press.

[AA09] Marc Alexa and Anders Adamson. Interpolatory point set surfaces—convexity and hermite data. *ACM Transactions on Graphics*, 28(2):1–10, 2009.

[ABCO+01] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. Silva. Point set surfaces. In *Proc. IEEE Visualization*, pages 21–28, San Diego, CA, 2001.

[ABCO+03] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Computing and rendering point set surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):3–15, 2003.

[AH04] Julien Reptin Adam Herout, Pavel Zemcik. Hardware implementation of ewa for point-based three-dimensional graphics rendering. *International Conference on Computer Vision and Graphics*, 32:593–598, 2004.

[AK04a] Nina Amenta and Yong Joo Kil. Defining point-set surfaces. *ACM Transactions on Graphics (Proc. SIGGRAPH 2004)*, 23(3):264–270, 2004.

[AK04b] Nina Amenta and Yong Joo Kil. The domain of a point set surface. pages 139–147, 2004.

[Ake93] Kurt Akeley. RealityEngine graphics. In *Computer Graphics (Proc. ACM SIGGRAPH '93)*, pages 109–116, 1993.

[AKP+05] Bart Adams, Richard Keiser, Mark Pauly, Leonidas Guibas, Markus Gross, and Philip Dutré. Efficient raytracing of deforming point-sampled surfaces. *Computer Graphics Forum*, 24(3), 2005.

[AMS03] Tomas Akenine-Möller and Jacob Ström. Graphics for the masses: a hardware rasterization architecture for mobile phones. *ACM Transactions on Graphics (Proc. SIGGRAPH '03)*, 22(3):801–808, 2003.

[APKG07] Bart Adams, Mark Pauly, Richard Keiser, and Leonidas J. Guibas. Adaptively sampled particle fluids. In *ACM Transactions on Graphics (Proc. ACM SIGGRAPH 2007)*, volume 26, page 48, 2007.

[Ben85] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, pages 509–517, 1985.

[BFMZ94] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen J. Scher Zagier. Frameless rendering: double buffering considered harmful. In *SIGGRAPH*, pages 175–176. ACM, 1994.

[BHZK05] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's GPUs. In *Proc. Eurographics Symposium on Point-Based Graphics 2005*, pages 17–24, Stony Brook, Long Island, USA, 2005.

[BSK04] M. Botsch, M. Spernat, and L. Kobbelt. Phong splatting. In *Proc. Eurographics Symposium on Point-Based Graphics 2004*, pages 25–32, Zurich, Switzerland, 2004.

[Car84] L. Carpenter. The a-buffer, an antialiased hidden surface method. In *Computer Graphics*, volume 18 of *Computer Graphics (Proc. ACM SIGGRAPH '84)*, pages 103–108, 1984.

[Cat84] Edwin Catmull. An analytic visible surface algorithm for independent pixel processing. In *SIGGRAPH*, pages 109–115. ACM, 1984.

[Cla82] James Clark. The geometry engine: A VLSI geometry system for graphics. In *Computer Graphics (Proc. ACM SIGGRAPH '82)*, volume 16, pages 127–133. ACM, 1982.

[Coo86] R. L. Cook. Stochastic sampling in computer graphics. In *SIGGRAPH*, pages 51–72. ACM, 1986.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *SIGGRAPH*, pages 137–145. ACM, 1984.

[CW93] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *SIGGRAPH*, pages 279–288. ACM, 1993.

[Dog05] Michael Doggett. Xenos: XBOX360 GPU. Presentation at Eurographics, 2005.

[DWS+88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a VLSI system for high performance graphics. In *Computer Graphics (ACM SIGGRAPH '88)*, volume 22, pages 21–30. ACM, 1988.

[ETH+09] Kevin Egan, Yu-Ting Tseng, Nicolas Holzschuch, Frédo Durand, and Ravi Ramamoorthi. Frequency analysis and sheared reconstruction for rendering motion blur. *ACM Transactions on Graphics (SIGGRAPH)*, 28(3):1–13, 2009.

[Eve01] Cass Everitt. Interactive order-independent transparency. Technical report, Nvidia, 2001.

[FB74] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[FBF77] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, pages 209–226, 1977.

[FGH+85] Henry Fuchs, Jack Goldfeather, Jeff Hultquist, Susan Spach, John Austin, Frederick Brooks, John Eyles, and John Poulton. Fast spheres, shadows, textures, transparencies, and imgage enhancements in pixel-planes. In *Computer Graphics (Proc. ACM SIGGRAPH '85)*, volume 19, pages 111–120. ACM, 1985.

[Fil79] Y. V. Silva Filho. Average case analysis of region search in balanced k-d trees. *Information Processing Letters*, 8(5), 1979.

[FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *Computer Graphics (Proc. ACM SIGGRAPH 1980)*, pages 124–133, 1980.

[FLB+09] Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 59–68, 2009.

[FPE+89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Computer Graphics (Proc. ACM SIGGRAPH '89)*, volume 23, pages 79–88. ACM, 1989.

[FS05] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 15–22, 2005.

[GBP06] G. Guennebaud, L. Barthe, and M. Paulin. Splat/mesh blending, perspective rasterization and transparency for point-based rendering. In *Proc. Eurographics Symposium on Point-Based Graphics 2006*, Boston, MA, 2006.

[GD98] J. P. Grossman and W. Dally. Point sample rendering. In *Rendering Techniques '98*, pages 181–192. Springer, 1998.

[GG07] Gaël Guennebaud and Markus Gross. Algebraic point set surfaces. *ACM Transactions on Graphics (Proc. ACM SIGGRAPH 2007)*, 26(3):23, 2007.

[GGG08] Gaël Guennebaud, Marcell Germann, and Markus Gross. Dynamic sampling and rendering of algebraic point set surfaces. *Computer Graphics Forum*, 27(2):653–662, 2008.

[GKBP05] Pascal Gautron, Jaroslav Krivánek, Kadi Bouatouch, and Sumanta Pattanaik. Radiance cache splatting: A GPU-friendly global illumination algorithm. In *Proc. Eurographics Symposium on Rendering*, pages 55–64, 2005.

[GM04] X. Guan and K. Mueller. Point-based surface rendering with motion blur. In *Point-Based Graphics*. Eurographics, 2004.

[GP07] Markus Gross and Hanspeter Pfister. *Point-Based Graphics*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann Publishers, 2007.

[GPSS07] Johannes Günther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Symposium on Interactive Ray Tracing*, pages 113–118, 2007.

[Gra85] Charles W. Grant. Integrated analytic spatial and temporal anti-aliasing for polyhedra in 4-space. In *SIGGRAPH*, pages 79–84. ACM, 1985.

[Gum03] S. Gumhold. Splatting illuminated ellipsoids with depth correction. In *Proc. 8th International Fall Workshop on Vision, Modelling and Visualization 2003*, pages 245–252, 2003.

[HA90] Paul Haeberli and Kurt Akeley. The accumulation buffer: hardware support for high-quality rendering. In *Computer Graphics (Proc. ACM SIGGRAPH '90)*, volume 24, pages 309–318. ACM, 1990.

[HAM07] Jon Hasselgren and Thomas Akenine-Möller. PCU: the programmable culling unit. In *SIGGRAPH*, page 92, 2007.

[HDD+92] H. Hoppe, T. DeRose, T. Duchampt, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. In *Computer Graphics*, SIGGRAPH 92 Proceedings, pages 71–78, Chicago, IL, 1992.

[Hec89] P. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Science, 1989.

[HKL+99] K. E. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Computer Graphics (Proc. ACM SIGGRAPH 99)*, pages 277–286, 1999.

[HZ05] Adam Herout and Pavel Zemcik. Hardware pipeline for rendering clouds of circular points. In *Proc. WSCG 2005*, pages 17–22, 2005.

[IK01] A. Ioannou and M. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high speed networks. In *Proc. IEEE Int. Conf. on Communications*, 2001.

[IMRV97] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multidimensional spaces. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 618–625, 1997.

[Int07] Intel. Intel QuickAssist technology accelerator abstraction layer white paper. In *Platform-level Services for Accelerators Intel Whitepaper*, 2007.

[JDD03] Thouis R. Jones, Frédo Durand, and Mathieu Desbrun. Non-iterative, feature-preserving mesh smoothing. *ACM Transactions on Graphics*, 22(3):943–949, 2003.

[JLBM05] Gregory S. Johnson, Juhyun Lee, Christopher A. Burns, and William R. Mark. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Transactions on Graphics*, 24(4):1462–1482, 2005.

[KB83] Jonathan Korein and Norman Badler. Temporal anti-aliasing in computer generated animation. In *SIGGRAPH*, pages 377–388. ACM, 1983.

[Lev01] D. Levin. Mesh-independent surface interpolation. In *Advances in Computational Mathematics*, 2001.

[Lev03] D. Levin. Mesh-independent surface interpolation. In *Geometric Modeling for Scientific Visualization*, pages 181–187, 2003.

[LH96] M. Levoy and P. Hanrahan. Light field rendering. In *Computer Graphics*, SIGGRAPH 96 Proceedings, pages 31–42, New Orleans, LS, 1996.

[LKM01] Erik Lindholm, Mark J. Kligard, and Henry Moreton. A user-programmable vertex engine. In *Computer Graphics (Proc. ACM SIGGRAPH '01)*, pages 149–158, 2001.

[LW85] M. Levoy and T. Whitted. The use of points as display primitives. Technical Report TR 85-022, The University of North Carolina at Chapel Hill, Department of Computer Science, 1985.

[Max90] Nelson Max. Polygon-based post-process motion blur. *The Visual Computer*, 6:308–314, 1990.

[MB02] Koen Meinds and Bart Barenbrug. Resample hardware for 3D graphics. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 17–26, 2002.

[MBDM97] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: a real-time graphics system. In *Computer Graphics (Proc. ACM SIGGRAPH '97)*, pages 293–302. ACM Press, 1997.

[MEP92] Steven Molnar, John Eyles, and John Poulton. PixelFlow: high-speed rendering using image composition. In *Computer Graphics (Proc. ACM SIGGRAPH '92)*, volume 26, pages 231–240. ACM, 1992.

[Mes] Mesa. The Mesa 3D graphics library. http://www.mesa3d.org/.

[Mic10] Microsoft. The DirectX software development kit. http://msdn.microsoft.com/directx, 2010.

[ML85] Nelson L. Max and Douglas M. Lerner. A two-and-a-half-D motion-blur algorithm. *SIGGRAPH*, pages 85–93, 1985.

[MM02] Vincent C. H. Ma and Michael D. McCool. Low latency photon mapping using block hashing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 89–99, 2002.

[MMG+98] Joel McCormack, Robert McNamara, Christopher Gianos, Larry Seiler, Norman P. Jouppi, and Ken Correll. Neon: a single-chip 3d workstation graphics accelerator. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 123–132, New York, NY, USA, 1998. ACM.

[MMS+98] Klaus Mueller, Torsten Mller, J. Edward Swan, Roger Crawfis, Naeem Shareef, and Roni Yagel. Splatting errors and antialiasing. *IEEE TVCG*, 4(2):178–191, 1998.

[Mor00] Steve Morein. ATI Radeon – HyperZ Technology. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, Hot3D session, 2000.

[MWA+08] Mateusz Majer, Stefan Wildermann, Josef Angermeier, Stefan Hanke, and Jürgen Teich. Co-design architecture and implementation for point-based rendering on fpgas. In *Proceedings of the IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 142–148, 2008.

[NHS84] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[NVI] NVIDIA. The NV_half_float OpenGL extension. Specification Document.

[NVI07] NVIDIA. CUDA: Compute unified device architecture. http://www.nvidia.com/cuda, 2007.

[OG97] Marc Olano and Trey Greer. Triangle scan conversion using 2d homogeneous coordinates. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 89–95, 1997.

[OGG09] Cengiz Oztireli, Gael Guennebaud, and Markus Gross. Feature preserving point set surfaces based on non-linear kernel regression. *Computer Graphics Forum*, 28(2), 2009.

[OvL80] M.H. Overmars and J. van Leeuwen. Dynamic multi-dimensional data structures based on quad- and k-d trees. Technical Report RUU-CS-80-02, Institute of Information and Computing Sciences, Utrecht University, 1980.

[PC83] Michael Potmesil and Indranil Chakravarty. Modeling motion blur in computer-generated images. In *SIGGRAPH*, pages 389–399. ACM, 1983.

[PDC+03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 41–50, 2003.

[PEL+00] Voicu Popescu, John Eyles, Anselmo Lastra, Josh Steinhurst, Nick England, and Lars Nyland. The WarpEngine: An architecture for the post-polygonal age. In *Computer Graphics (Proc. ACM SIGGRAPH '00)*, pages 433–442, 2000.

[PGSS07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum (Proc. of EUROGRAPHICS 2007)*, 26(3), September 2007. Proceedings of Eurographics.

[Pin88] Juan Pineda. A parallel algorithm for polygon rasterization. In *Computer Graphics (Proc. ACM SIGGRAPH '88)*, volume 22, pages 17–20. ACM, 1988.

[PKKG03] M. Pauly, R. Keiser, L. Kobbelt, and M. Gross. Shape modeling with point-sampled geometry. *ACM Transactions on Graphics (Proc. SIGGRAPH '03)*, 22(3):641–650, 2003.

[PZvBG00] H. Pfister, M. Zwicker, J. van Baar, and M Gross. Surfels: Surface elements as rendering primitives. In *Computer Graphics (Proc. ACM SIGGRAPH '00)*, pages 335–342, 2000.

[RÖ2] Jussi Räsänen. Surface splatting: Theory, extensions and implementation. Master's thesis, Helsinki University of Technology, 2002.

[Ree83] W. T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. In *Computer Graphics*, volume 17 of *SIGGRAPH 83 Proceedings*, pages 359–376, 1983.

[RK88] V. N. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Trans. Comput.*, 37(12):1657–1665, 1988.

[RL00] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, pages 343–352, Los Angeles, CA, 2000.

[Rob81] John T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD '81: Proceedings of the 1981*

ACM SIGMOD international conference on Management of data, pages 10–18, 1981.

[RPZ02] L. Ren, H. Pfister, and M. Zwicker. Object-space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Computer Graphics Forum*, 21(3):461–470, 2002.

[Sam06] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*, chapter Multidimensional Point Data, pages 1–190. Morgan Kaufmann, 2006.

[SBM04] Jason Stewart, Eric Bennett, and Leonard McMillan. Pixelview: A view-independent graphics rendering architecture. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 75–84, 2004.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[She68] Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the ACM national conference*, pages 517–524, 1968.

[Shi93] Mikio Shinya. Spatial anti-aliasing for animation sequences with spatio-temporal filtering. In *SIGGRAPH*, pages 289–296. ACM, 1993.

[SPW02] K. Sung, A. Pearce, and C. Wang. Spatial-temporal antialiasing. *IEEE TVCG*, 8(2):144–153, 2002.

[SWBG06a] C. Sigg, T. Weyrich, M. Botsch, and M. Gross. GPU-based ray-casting of quadratic surfaces. In *Proc. Eurographics Symposium on Point-Based Graphics 2006*, pages 59–65, Boston, MA, 2006.

[SWBG06b] Christian Sigg, Tim Weyrich, Mario Botsch, and Markus Gross. GPU-based ray-casting of quadratic surfaces. In *Point-Based Graphics*, pages 59–65. Eurographics, 2006.

[SWS02] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR: A hardware achitecture for ray tracing. In *Proc. Workshop on Graphics Hardware 2002*, pages 27–36, 2002.

[Tan06] Yankin Tanurhan. Processors and FPGAs quo vadis. *IEEE Computer Magazine*, 39(11):106–108, 2006.

[THM+03] M. Teschner, B. Heidelberger, M. Mller, D. Pomeranerts, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proc. Vision, Modeling, Visualization VMV*, pages 47–54, 2003.

[TK96] Jay Torborg and James T. Kajiya. Talisman: commodity realtime 3D graphics for the PC. In *Computer Graphics (Proc. ACM SIGGRAPH '96)*, pages 353–363, 1996.

[TMF+07] Hiroyuki Takeda, Student Member, Sina Farsiu, Peyman Milanfar, and Senior Member. Kernel regression for image processing and reconstruction. *IEEE Transactions on Image Processing*, 16:349–366, 2007.

[Vah07] Frank Vahid. It's time to stop calling circuits hardware. *IEEE Computer Magazine*, 40(9):106–108, 2007.

[WBS06] Sven Woop, Erik Brunvand, and Philipp Slusallek. Estimating performance of a ray-tracing asic design. In *Proc. IEEE Symposium on Interactive Ray Tracing*, 2006.

[WBWS01] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153–164, 2001.

[WHA+07] Tim Weyrich, Simon Heinzle, Timo Aila, Daniel Fasnacht, Stephan Oetiker, Mario Botsch, Cyril Flaig, Simon Mall, Kaspar Rohrer, Norbert Felber, Hubert Kaeslin, and Markus Gross. A hardware architecture for surface splatting. *ACM Transactions on Graphics (Proc. ACM SIGGRAPH 2007)*, 26(3):90–11, 2007.

[Whi84] Mary Whitton. Memory design for raster graphics displays. *IEEE Computer Graphics and Applications*, 4(3):48–65, 1984.

[WK05] Turner Whitted and James Kajiya. Fully procedural graphics. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 81–90, 2005.

[WMS] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*.

[WMS06] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-kd trees for hardware accelerated ray tracing of dynamic scenes. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 67–77, 2006.

[WPK+04] T. Weyrich, M. Pauly, R. Keiser, S. Heinzle, S. Scandella, and M. Gross. Post-processing of scanned 3D surface data. In *Proc. Eurographics Symposium on Point-Based Graphics 2004*, pages 85–94, Zurich, Switzerland, 2004.

[WSS05] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (SIGGRAPH 2005)*, 24(3):434–444, 2005.

[WZ96] Matthias M. Wloka and Robert C. Zeleznik. Interactive real-time motion blur. *The Visual Computer*, 12(6):283–295, 1996.

[YHGT10] Jason C. Yang, Austin Hensley, Holger Gruen, and Nicolas Thibieroz. Real-time concurrent linked list construction on the GPU. In *Eurographics Symposium on Rendering*, pages 277–286, 2010.

[ZHWG08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions Graphics (SIGGRAPH Asia)*, 27(5):1–11, 2008.

[ZP06] Y. Zhang and R. Pajarola. Single-pass point rendering and transparent shading. In *Point-Based Graphics*, pages 37–48. Eurographics, 2006.

[ZP09] Herout Adam Zemcik Pavel, Marsik Lukas. Point cloud rendering in fpga. In *Proc. WSCG*, pages 517–524, 2009.

[ZPBG01] M. Zwicker, H. Pfister., J. Van Baar, and M. Gross. Surface splatting. In *Computer Graphics (Proc. ACM SIGGRAPH '01)*, pages 371–378, Los Angeles, CA, 2001.

[ZPBG02] M. Zwicker, H. Pfister, J. Van Baar, and M. Gross. EWA splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):223–238, 2002.

[ZPKG02] M. Zwicker, M. Pauly, O. Knoll, and M. Gross. Pointshop 3D: An interactive system for point-based surface editing. In *Computer Graphics*, SIGGRAPH 2002 Proceedings, pages 322–329, San Antonio, TX, 2002.

[ZRB+04] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate splatting. In *Proc. Graphics Interface*, pages 247–254, 2004.

Curriculum Vitae

Simon Heinzle

Personal Data

Oct. 12, 1981	Born in Feldkirch, Austria
Nationality	Austria

Education

Sep. 4, 2010	Ph.D. defense.
Apr. 2006 – Sep. 2010	Research assistant and Ph. D. student at the Computer Graphics Laboratory of the Swiss Federal Institute of Technology (ETH) Zurich, Prof. Markus Gross.
Mar. 2006	Diploma degree in Computer Science.
Oct. 2000 – Jan. 2006	Diploma Studies of Computer Science, ETH Zurich, Switzerland. Specialization: Computer Graphics; Complementary studies: VLSI design and testing.

Awards

June 2008	Best Paper Award "A Hardware Processing Unit for Point Sets" at Graphics Hardware 2008.

Scientific Publications

S. HEINZLE, J. WOLF, Y. KANAMORI, T. WEYRICH, T. NISHITA, and M. GROSS. Motion Blur for EWA Surface Splatting. In *Computer Graphics Forum (Proceedings of Eurographics 2010)*, Norrköping, Sweden, May 2010.

S. HEINZLE, G. GUENNEBAUD, M. BOTSCH, and M. GROSS. A Hardware Processing Unit for Point Sets. In *Proceedings of the 23rd SIGGRAPH/Eurographics Conference on Graphics Hardware*, Sarajevo, Bosnia and Herzegovina, June 2008. Was awarded with the Best Paper Award.

S. HEINZLE, O. SAURER, S. AXMANN, D. BROWARNIK, A. SCHMIDT, F. CARBOGNANI, P. LUETHI, N. FELBER, and M. GROSS. A Transform, Lighting and Setup ASIC for Surface Splatting. In *Proceedings of International Symposium on Circuits and Systems (ISCAS)*, Seattle, USA, May 2008.

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 In *Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, San Diego, August 2007.

T. WEYRICH, M. PAULY, R. KEISER, S. HEINZLE, S. SCANDELLA, and M. GROSS. Post-processing of Scanned 3D Surface Data In *Proceedings of Eurographics Symposium on Point-Based Graphics*, Zurich, Switzerland, June 2004.

Employment

From Oct. 2010	Post-doctoral researcher at The Walt Disney Company Schweiz AG, Disney Research Zurich.
Apr. 2006 – Sep. 2010	Research assistant at ETH Zurich, Zurich, Switzerland.
Oct. 2003 – Mar. 2005	Teaching assistant at ETH Zurich, Zurich, Switzerland
Mar. 2005 – Jun. 2005	Internship at AdNovum Informatik AG, Zurich, Switzerland.