A Hardware Architecture for Surface Splatting

Tim Weyrich[*†] Simon Heinzle[*] Timo Aila[‡§] Daniel B. Fasnacht[*] Stephan Oetiker[*] Mario Botsch[*]
Cyril Flaig[*] Simon Mall[*] Kaspar Rohrer[*] Norbert Felber[*] Hubert Kaeslin[*] Markus Gross[*]

Abstract

We present a novel architecture for hardware-accelerated rendering of point primitives. Our pipeline implements a refined version of EWA splatting, a high quality method for antialiased rendering of point sampled representations. 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 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 developed a computationally stable evaluation scheme for perspectively corrected splats. We implemented our architecture both on reconfigurable FPGA boards and as an ASIC prototype, and we integrated it into an OpenGL-like software implementation. Our evaluation comprises a detailed performance analysis using scenes of varying complexity.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms;

Keywords: 3D graphics hardware, point-based rendering, surface splatting, rasterization, reordering, data structures

1 Introduction

In recent years, we have witnessed a dramatic increase in the polygonal complexity of graphics models, and triangles project on average to fewer and fewer screen-space pixels. Many applications, such as numerical simulation, surface scanning, or procedural geometry generation, produce millions of geometric primitives. A significant issue arising from increasingly fine geometric detail is that tiny triangles can fall between the sampling points and may thus get completely lost in rasterization. This makes high-quality, antialiased rendering very challenging. Additionally, triangles are not a particularly memory-efficient representation for such detailed objects. It has been suggested [Levoy and Whitted 1985] that point-based representations are well suited for complex, organic shapes with high geometric and appearance detail, whereas for flat surfaces, triangles undoubtedly outperform in memory footprint and rendering performance.

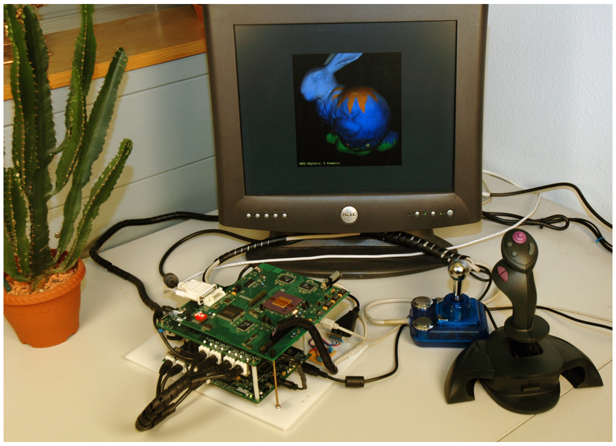


Figure 1: Point rendering system featuring an ASIC implementation of our point rendering architecture. The system runs standalone, independent from a host PC.

In point sample rendering the primitives typically carry all attributes needed for display including geometry and appearance. Early frameworks for point sample rendering [Grossman and Dally 1998; Pfister et al. 2000] suggested simple forward warping of point samples into the frame buffer. In combination with precomputed texture attributes they kept the architecture simple by transferring parts of the rasterization and filtering to the preprocessing stage. Besides increased memory bandwidth requirements, such approaches cannot guarantee closed surfaces, and holes have to be filled in a postprocessing stage. A significant step towards limiting the transfer from off-chip memory to the graphics hardware is the fully procedural graphics architecture proposed by Whitted and Kajiya [2005]. They convert procedural descriptions of geometry and appearance directly into point samples through a programmable sample controller thus keeping external memory bandwidth low. The direct conversion to unconnected points, instead of going through triangles, simplifies the algorithms by eliminating the need for tracking and producing connectivity information.

In this paper, we present a hardware architecture which combines the advantages of triangles and point-based representations. We designed a rendering pipeline based on an advanced variant of so-called Elliptical Weighted Average (EWA) surface splatting, a high-quality point rendering algorithm [Zwicker et al. 2001]. The method renders point samples by superimposing band-limited reconstruction kernels, *surface splats*, in the image domain. It efficiently suppresses aliasing artifacts, and it can guarantee hole-free reconstructions even for moderate sampling densities and under arbitrary transformations. While EWA splatting has been implemented before on conventional GPUs [Ren et al. 2002; Zwicker et al. 2004; Botsch et al. 2005], such architectures do not natively support essential operations, leaving the rendering inefficient and requiring multiple passes. As a recent exception, Zhang and Pajarola [2006] present an implementation that renders all splats in a single pass into multiple frame buffers, before compositing them to a final image. However, their method requires a static presorting of surface splats, which is prohibitive for dynamic scene content.

The design philosophy of the presented hardware architecture is motivated by the desire to seamlessly integrate high quality point

[*] ETH Zurich. Email: [sheinzle,botsch,grossm]@inf.ethz.ch, [soetiker,felber,kaeslin]@iis.ee.ethz.ch, [cflaig,krohrer,dfasnacht,small]@student.ethz.ch
[†] Princeton University. Email: [tweyrich]@cs.princeton.edu
[‡] Helsinki University of Technology
[§] NVIDIA Research. Email: [taila]@nvidia.com

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 have to be accumulated before the final surface can be displayed. Also, splats belonging to different surfaces have to be separated from each other. As we will show, our architecture addresses these issues and renders splats efficiently in single pass. Furthermore, deferred shading of the splat surfaces allows us to apply virtually any fragment shading program to the reconstructed surface.

The central features of our design include the use of a constant-throughput pipelined heap for (re)ordering the memory accesses of the splat primitives for improved 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 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.

2 Related Work

Conventional Graphics Hardware Virtually all commercially available graphics hardware is based on triangle rasterization. In these architectures the transformed and optionally lit vertices are first assembled to triangles [Clark 1982; Akeley 1993; Lindholm et al. 2001]. The triangles are then (possibly) clipped to the view frustum, and rasterization identifies the pixels each triangle overlaps [Fuchs et al. 1985; Pineda 1988]. Finally the fragments are shaded and various tests (e.g., scissor, alpha, stencil, and depth) are performed before frame buffer blending.

As an optimization, the frustum clipping operations can be mostly avoided by employing guard bands [Montrym et al. 1997], or altogether removed by using homogeneous rasterization [Olano and Greer 1997]. Modern hardware implementations [Morein 2000] avoid the execution of fragment shader programs for fragments that would be discarded. The current APIs [OpenGL Architecture Review Board and Shreiner 2004; Microsoft 2002] allow this as long as the early culling does not cause side effects.

Experimental Graphics Hardware A few architectures consist of multiple rasterization nodes and create the final image by using composition [Fuchs et al. 1989; Molnar et al. 1992; Torborg and Kajiya 1996]. While the scalability can be particularly good in these architectures, certain newer features such as occlusion queries are difficult to implement efficiently.

SaarCOR [Schmittler et al. 2002] uses triangles but employs ray casting instead of rasterization. A more flexible, programmable ray processing unit [Woop et al. 2005] enables the implementation of advanced effects, such as soft shadows and even some global illumination techniques.

A few experimental hardware architectures have been proposed for the rendering of non-polygonal primitives. WarpEngine [Popescu et al. 2000] uses depth images as its rendering primitive. During the transformation and projection of point samples, additional samples are generated between the original ones in order to avoid holes in the image reconstruction. Herout and Zemcik [2005] describe a prototype architecture that uses circular constant-colored splats as rendering primitives. Image quality, generality, and other issues are not addressed in this architecture.

Whitted and Kajiya [2005] propose making the graphics pipeline fully programmable so that a single processor array would handle geometric as well as shading elements of a procedural description. The processor array outputs point samples, typically without going through triangles. Ideally the sampling density would be controlled adaptively in order to guarantee hole-free reconstruction and avoid superfluous sampling.

Stewart et al. [2004] describe a triangle rasterization-based architecture that maintains a view-independent rendering of the scene. The output images for potentially large number of view points are then reconstructed from the view-independent representation. Meinds and Barenbrug [2002] explain a novel texture mapping architecture that splats texture samples to the screen instead of resorting to the typical filtered texture fetch approach. The authors show high quality texture filtering at modest cost.

Point-Based Rendering The possibility of using points as a rendering primitives was first suggested by Levoy and Whitted [1985]. Since then a large body of follow-up work has been published [Kobbelt and Botsch 2004; Sainz and Pajarola 2004].

The reconstruction of continuous (i.e. hole-free) images from a discrete set of surface samples can be done by image-space reconstruction techniques [Grossman and Dally 1998; Pfister et al. 2000] or by object-space resampling. The techniques from the latter category dynamically adjust the sampling rate so that the density of projected points meets the pixel resolution. Since this depends on the current viewing parameters, the resampling has to be done dynamically for each frame. Examples are dynamic sampling of procedural geometries [Stamminger and Drettakis 2001], the randomized z-buffer [Wand et al. 2001], and the rendering of moving least squares (MLS) surfaces [Alexa et al. 2001; Fleishman et al. 2003; Adams et al. 2005].

3 Overview

3.1 EWA Surface Splatting

EWA splatting [Zwicker et al. 2001] was designed for high-quality direct visualization of point-sampled surfaces. Rendering pure point-samples of surface attributes, such as position $\boldsymbol{c}$, normal vector $\boldsymbol{n}$, and reflectance properties, by simple forward-projection inevitably causes holes in the resulting image in case of insufficient sampling density. In contrast to this, the EWA splatting framework assumes *splats* as input data, which additionally have two tangent axes $(\boldsymbol{u}, \boldsymbol{v})$ and corresponding radii, such that they represent disks or ellipses in object space, and by this augment points with a spatial extent. As a consequence, the mutual overlap of surface *splats* in object space guarantees a watertight rendering without holes or gaps in between samples. Starting with [Zwicker et al. 2004; Botsch et al. 2004], perspective accurate variants preserve this property even under extreme projections.

To achieve high visual quality, EWA splatting assigns an elliptical Gaussian reconstruction kernel to each splat, which results in an elliptical footprint when projected into image space [Westover 1990]. Using these footprint kernels, the individual color contributions of overlapping splats are weighted and accumulated, such that a final per-fragment normalization, i.e., a division of each fragment's accumulated color by its sum of weights, results in a smooth surface reconstruction in image space (see Figure 2).

Similarly to Heckbert's texture filtering approach [1989], the projected reconstruction filters are additionally convolved with a band-limiting image-space pre-filter in order to avoid aliasing artifacts under minification. If the object-space reconstruction filter as well as the image-space pre-filter are Gaussians, and if the splat projection is approximated by an affine mapping, the convolution of both filters again gives a Gaussian, which is then referred to as the image-space EWA filter. Since its inception, various improvements

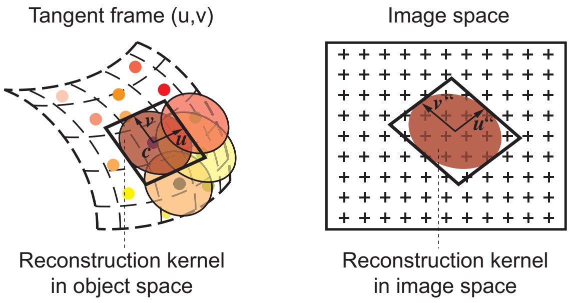

Tangent frame (u,v)
Image space
v
c
u
v'
u'
Reconstruction kernel in object space
Reconstruction kernel in image space

Figure 2: Elliptical surface splats, defined by center $\boldsymbol{c}$ and tangential vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, sample surface attributes in object space. Their elliptical projection is sampled and accumulated in image space to reconstruct an image of the surface.

of the EWA splatting algorithm have been proposed [Räsänen 2002; Botsch et al. 2005; Guennebaud et al. 2006].

While EWA splatting can be implemented on modern programmable GPUs, the result is a relatively inefficient multi-pass algorithm. In particular, we have identified the following performance issues:

- 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.

3.2 Design Overview

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

The new algorithms used by our splat rasterization unit are described in Sections 4.1 and 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 4.3) and extended blending functionality (Sections 4.4 and 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 [Deering et al. 1988; Botsch et al. 2005] for splats, and as a result any fragment shading program can be used. Figure 3 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, and direct feeding of normalized dirty fragments to the shader. Note that if our pipeline was embedded into a GPU that already supported this functionality for off-screen surfaces, those resources could be reused.

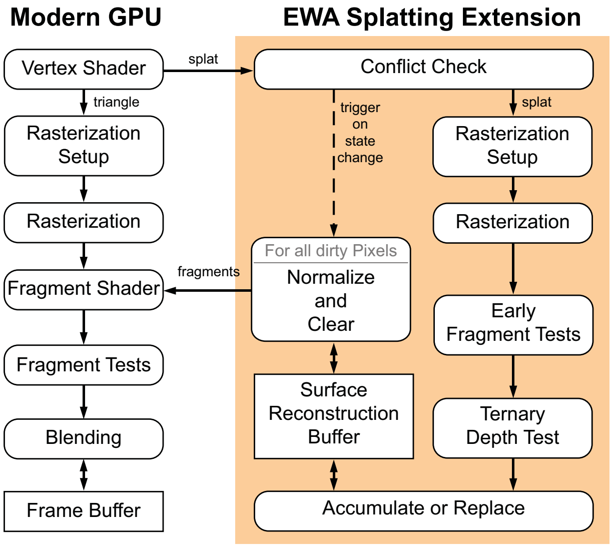


Figure 3: The integration of EWA surface splatting into a conventional graphics pipeline. The key element is the surface reconstruction buffer. On state changes, readily reconstructed fragments are fed back into the traditional graphics pipeline.

In order to significantly improve the caching efficiency over typical frame buffer caches, we propose a novel reordering stage in Section 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.

The rest of this paper is organized as follows. Our algorithms and hardware architecture are described in Sections 4 and 5, respectively. The characteristics of our FPGA, ASIC, and Mesa implementations are outlined (Section 6), and the seamless integration of the new functionality into existing APIs is covered in Section 6.3. Finally, Section 7 provides test results, and Section 8 discusses the limitations and weaknesses of the current design and lists potential future work.

4 Rendering Pipeline

We propose an EWA surface splatting pipeline that has been optimized for hardware implementations.

A splat is defined by its center $\boldsymbol{c}$, and two tangential vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, as illustrated in Figure 2. The tangential vectors span the splat's local coordinate system that carries a Gaussian reconstruction kernel. $\boldsymbol{u}$ and $\boldsymbol{v}$ may be skewed to allow for the direct deformation of splat geometry [Pauly et al. 2003]. In addition, each splat has a variable-length attribute vector $\boldsymbol{a}$ that contains surface attributes to be used by vertex and fragment shading units.

The splat transform and lighting computations are similar to vertex processing, and thus the existing vertex shader units can be reused. Table 1 lists the computations required in various stages of our EWA surface splatting pipeline. The corresponding triangle pipeline operations are shown for comparison purposes only, and may not exactly match any particular GPU. 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.

Symbols	
$\boldsymbol{c},\boldsymbol{n},\boldsymbol{u},\boldsymbol{v}$	Center, normal, and two tangent vectors
$\boldsymbol{c}',\boldsymbol{n}',\boldsymbol{u}',\boldsymbol{v}'$	Transformed to camera space
$\boldsymbol{c}''$	Center projected to screen space
$\boldsymbol{a}_{s,v,f}$	Splat, vertex, and fragment attribute vectors
M,P	Model-view matrix, projection matrix
T	Transformation from clip space to canonical splat
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

Transform and lighting (using vertex shader)	
EWA Surface Splat	*Triangle (for each vertex)*
$\boldsymbol{c}'' = P M \boldsymbol{c}$	$\boldsymbol{c}'' = P M \boldsymbol{c}$
$\boldsymbol{c}' = M \boldsymbol{c}$	$\boldsymbol{c}' = M \boldsymbol{c}$
$\boldsymbol{u}' = M \boldsymbol{u}$	
$\boldsymbol{v}' = M \boldsymbol{v}$	
$\tilde{\boldsymbol{n}}' = \boldsymbol{u}' \times \boldsymbol{v}'$	$\tilde{\boldsymbol{n}}' = M^{-\top} \boldsymbol{n}$
$\langle \boldsymbol{c}', \tilde{\boldsymbol{n}}' \rangle > 0 \, ? \Rightarrow$ backface, kill	
$\boldsymbol{n}' = \tilde{\boldsymbol{n}}' / \lVert \tilde{\boldsymbol{n}}' \rVert$	$\boldsymbol{n}' = \tilde{\boldsymbol{n}}' / \lVert \tilde{\boldsymbol{n}}' \rVert$
$\boldsymbol{a}_s = \text{lighting}(\boldsymbol{a}, \boldsymbol{c}', \boldsymbol{n}')$	$\boldsymbol{a}_v = \text{lighting}(\boldsymbol{a}, \boldsymbol{c}', \boldsymbol{n}')$

Rasterization setup (fixed function)	
EWA Surface Splat, $i \in [1 \dots 3]$	*Triangle*, $i \in [1 \dots 3]$
$T = \begin{pmatrix} \boldsymbol{u}'\boldsymbol{v}'\boldsymbol{c}' \\ 0\,0\,1 \end{pmatrix}^\top P^\top \in \mathbb{R}^{3\times 4}$ $d = \langle (1,1,-1), (T_4 \star T_4) \rangle$ $\boldsymbol{f} = \frac{1}{d}(1,1,-1)$ Center of projected splat: $p_i = \langle \boldsymbol{f}, (T_i \star T_4) \rangle$ Half extents of bounds: $h_i = \sqrt{p_i^2 - \langle \boldsymbol{f}, (T_i \star T_i) \rangle}$	$\boldsymbol{p}_i = (c''_{i_x}, c''_{i_y})$ $A = [(\boldsymbol{p}_3 - \boldsymbol{p}_1) \times (\boldsymbol{p}_2 - \boldsymbol{p}_1)]_z$ $A < 0 \, ? \Rightarrow$ backface, kill Edge functions: $\boldsymbol{N}_1 = (\boldsymbol{p}_{1_y} - \boldsymbol{p}_{2_y}, \boldsymbol{p}_{2_x} - \boldsymbol{p}_{1_x})$ $\boldsymbol{N}_2 = (\boldsymbol{p}_{2_y} - \boldsymbol{p}_{3_y}, \boldsymbol{p}_{3_x} - \boldsymbol{p}_{2_x})$ $\boldsymbol{N}_3 = (\boldsymbol{p}_{3_y} - \boldsymbol{p}_{1_y}, \boldsymbol{p}_{1_x} - \boldsymbol{p}_{3_x})$ $\boldsymbol{e}_i = (\boldsymbol{N}_i, -\langle \boldsymbol{N}_i, \boldsymbol{p}_1 \rangle) \in \mathbb{R}^{1\times 3}$ Bounding rectangle: $\boldsymbol{br}_{\min} = \min(\boldsymbol{p}_i)$ $\boldsymbol{br}_{\max} = \max(\boldsymbol{p}_i)$

Rasterization (fixed function)	
EWA Surface Splat $\forall (x,y) \in [c''_x \pm h_x, c''_y \pm h_y]$	*Triangle* $\forall (x,y) \in \boldsymbol{br}, i \in [1 \dots 3]$
$\boldsymbol{k} = -T_1 + x T_4$ $\boldsymbol{l} = -T_2 + y T_4$ $\boldsymbol{s} = l_4 (k_1, k_2) - k_4 (l_1, l_2)$ Distance to splat's origin: $\rho^2 = \langle \boldsymbol{s}, \boldsymbol{s} \rangle / (k_1 l_2 - k_2 l_1)^2$ $\rho^2 > \text{cutoff}^2 \, ? \Rightarrow$ outside, kill $w_f = k(\rho^2)$	Barycentric coordinates: $b_i = \langle \boldsymbol{e}_i, (x,y,1) \rangle / A$ $b_i < 0 \, ? \Rightarrow$ outside, kill Perspective correction: $c_i = b_i w_1 w_2 w_3 / w_i$ $r = \sum c_i$ $b'_i = c_i / r$ $\boldsymbol{a}_f = \sum b'_i \boldsymbol{a}_{v_i}$

Table 1: *Left:* Our splat rendering pipeline. *Right:* The corresponding triangle pipeline operations are shown for comparison purposes. The computation of per-fragment attributes $\boldsymbol{a}_f$ for EWA splats is described in Sections 4.3–4.5.

4.1 Rasterization Setup

The rasterization setup computes per-splat variables for the subsequent rasterization unit, which then 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 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.

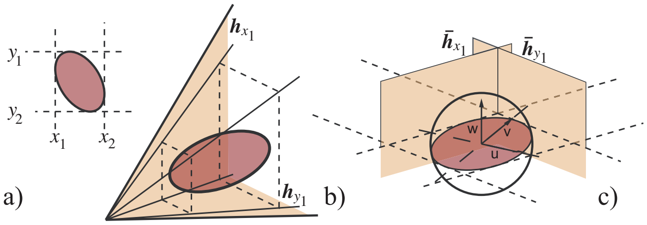


Figure 4: Splat bounding box computation. The screen-space bounding rectangle (a) corresponds to the splat's bounding frustum in camera space (b). Transformation into local splat coordinates maps the frustum planes to tangential planes of the unit sphere (c).

Previous bounding rectangle computations for perspective accurate splats have used either a centralized conic representation of the splat [Zwicker et al. 2004] or relied on fairly conservative approximations. According to our simulations, the method of Botsch et al. [2004] overestimates the bounding rectangle by an average factor of four, as it is limited to square bounding boxes leading to inefficient rasterization. Gumhold's [2003] computation does not provide axis-aligned bounding boxes, which would result in setup 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. When the rendering primitives are tiny, it is beneficial to reduce the setup cost at the expense of increased per-fragment cost [Montrym et al. 1997].

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 $\boldsymbol{c}'$, $\boldsymbol{u}'$, and $\boldsymbol{v}'$ in camera space and computes the axis-aligned bounding rectangle under 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 $\boldsymbol{u}'$, $\boldsymbol{v}'$, and $\boldsymbol{w}'$, where $\boldsymbol{w}'$ is reduced to zero length. The splat in clip space is built by this unit sphere under the projective mapping PS with

$$S = \begin{pmatrix} \boldsymbol{u}' & \boldsymbol{v}' & \boldsymbol{0} & \boldsymbol{c}' \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (1)$$

Finding the bounding rectangle coordinates x_1, x_2, y_1, and y_2 (see Figure 4a), corresponds to finding planes $\boldsymbol{h}_x = (-1,0,0,x)^\top$ and $\boldsymbol{h}_y = (0,-1,0,y)^\top, x \in \{x_1,x_2\}$ and $y \in \{y_1,y_2\}$, that are adjacent to the splat in clip space (Figure 4b). Mapping these planes to the local splat coordinate system by $(PS)^{-1}$ yields $\bar{\boldsymbol{h}}_c = (PS)^\top \boldsymbol{h}_c$, $c \in \{x,y\}$, which are tangential planes to the unit sphere in $(\boldsymbol{u}',\boldsymbol{v}',\boldsymbol{w}')$ (see Figure 4c). Hence, the splat's bounding rectangle can be determined by solving for x and y, so that $\bar{\boldsymbol{h}}_x$ and $\bar{\boldsymbol{h}}_y$ have unit-distances to the origin. Analogously, the depth extent of the splat can be computed starting from $\boldsymbol{h}_z = (0,0,-1,z)^\top$. Note how sending homogeneous planes through the inverse transpose of $(PS)^{-1}$ avoids explicit inversion of S (and P). This is the reason for stability in degenerate cases. This computation can be considered a special case of the quadric bounding box computation by Sigg et al. [2006].

Table 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.

4.2 Rasterization

Our rasterization unit traverses all sample positions within the bounding box, for each sample computing the intersection point $\boldsymbol{s}$ 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 $\boldsymbol{s}$ 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 the reconstruction kernel function.

A conic-based rasterization [Zwicker et al. 2004] was not used due to its high setup cost and numerical instabilities. Botsch et al. [2004] 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.

Table 1 shows how the ray-splat intersection is efficiently computed. 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. $\boldsymbol{s}$ follows by intersecting the projected planes $\boldsymbol{k}$ and $\boldsymbol{l}$ with the splat plane. Due to the degeneracy of the splat's coordinate system, the computation of $\boldsymbol{s}$ and ρ^2 requires very few operations.

We use the EWA screen-space filter approximation by Botsch et al. [2005]. 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.

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

$$\begin{cases} \textit{z-fail,} & z_d < z_s - \varepsilon \\ \textit{z-pass,} & z_d > z_s + \varepsilon \\ \textit{z-blend,} & \text{otherwise} \end{cases} . \qquad (2)$$

The additional case *z-blend* triggers the accumulation of overlapping splats. An ε-band around incoming splat fragments defines whether a fragment is blended to the surface reconstruction buffer. The size of the ε is computed from the splat's depth range h_z as $\varepsilon := h_z \cdot \varepsilon_{\Delta z} + \varepsilon_{\text{bias}}$, 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. [2000] and Räsänen [2002] use deep depth buffers to store additional information for the blending criterion. However, in our tests the benefits did not justify the additional storage costs and complexity of a deep depth buffer.

4.4 Attribute Accumulation

Our pipeline supports two kinds of splat attributes: typical *continuous* attributes can be blended, whereas *cardinal* ones (e.g., material identifiers) cannot. Unless the ternary depth test fails, all continuous attributes in $\boldsymbol{a}_s$ are weighted by the local kernel value w_f. If the test resulted in *z-pass*, weight w_d and attributes $\boldsymbol{a}_d$ currently stored in the reconstruction buffer are replaced by

$$(w_d,\boldsymbol{a}_d) \quad := \quad \begin{cases} (w_f, w_f \cdot \boldsymbol{a}_s), & \text{for continuous attributes} \\ (w_f, \boldsymbol{a}_s), & \text{for cardinal attributes} \end{cases} \qquad (3)$$

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

$$(w_d,\boldsymbol{a}_d) \quad := \quad (w_d + w_f, \boldsymbol{a}_d + w_f \cdot \boldsymbol{a}_s) . \qquad (4)$$

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

$$(w_d,\boldsymbol{a}_d) \quad := \quad \begin{cases} (w_f, \boldsymbol{a}_s), & w_d < w_f \\ (w_d, \boldsymbol{a}_d), & w_d \geq w_f \end{cases} . \qquad (5)$$

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. [1999].)

4.5 Normalization

Once the accumulation is complete, all continuous attributes must be normalized ($\boldsymbol{a}_f = \boldsymbol{a}_d / w_d$) before sending the fragments to the fragment shader units.

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.

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), some of the fragment tests can be executed already at the time of splat rasterization.

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 provides an overview over the proposed hardware extension. The remainder discusses the extensions in more detail.

5.1 Rasterization Setup and Splat Splitting

Rasterization setup was implemented as a fully pipelined fixed-function design, which computes the bounding rectangle of a splat along with the depth extents. 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 [Morein 2000], 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. 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 amount of additional data is generally not significant.

5.2 Splat Reordering

Motivation The small size of EWA splats along with the relatively high overdraw during accumulation put an exceptionally high

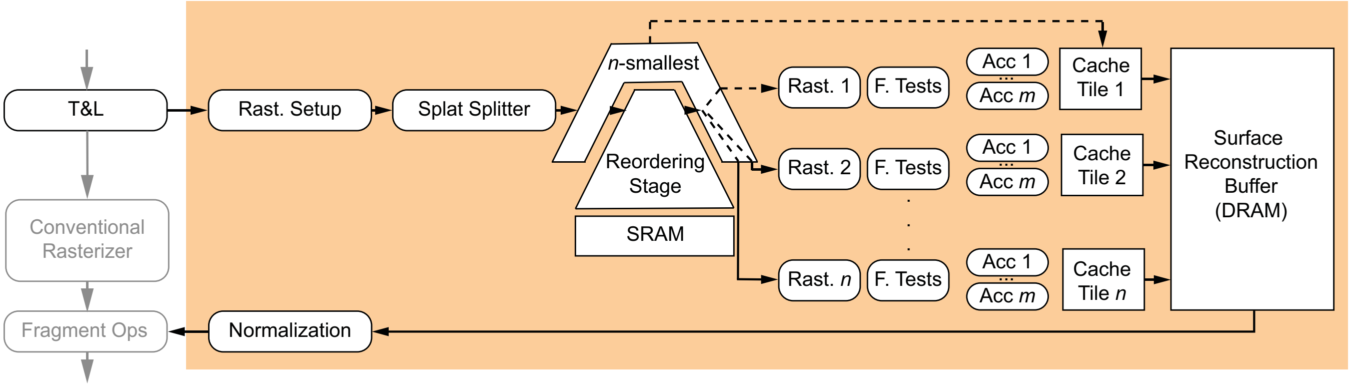


Figure 5: Hardware architecture of the EWA surface splatting extension.

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 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. 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 7).

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. 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 $i \prec j$ as

$$\left\{ \begin{array}{ll} \mathit{false}, & i < I_c \le j \\ \mathit{true}, & j < I_c \le i \\ i < j, & \text{otherwise} \end{array} \right. \tag{6}$$

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 $I_c \le i < I'_c$ (interpreted cyclically) in the data structure.

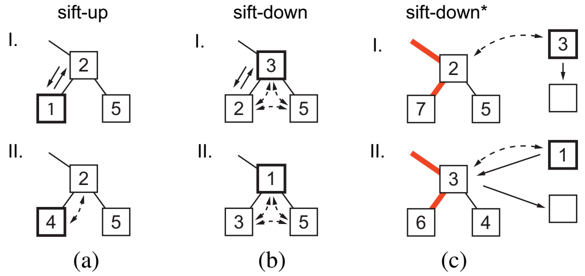


Figure 6: Heap operations. Roman numbers refer to two independent examples of each operation. Dashed arrows denote required comparisons, solid arrows move operations, and bold boxes the new element. (a) In SIFT-UP ON INSERTION the new element is inserted to the last leaf node, ascending in the hierarchy 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 reestablished. (c) In modified SIFT-DOWN* ON INSERTION, auxiliary registers between heap levels allow to propagate inserted elements downward. The path for this operation is known in advance (in red).

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 it has no storage overhead and the child and parent nodes of a given node index can be found using simple shift and increment operations on the indices.

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 6 (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 [Rao and Kumar 1988].

This can be achieved by adding auxiliary buffers between the heap levels that can hold one element each, see Figure 6 (c). For SIFT-DOWN*, the path that the operation takes through the heap tree is known in advance: the propagation is going to traverse all heap levels before finally an element is stored at the first free leaf, that is, 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 7 shows the resulting comparison operations along the propagation path of an inserted element.

SIFT-DOWN ON DELETION usually requires the last element of the heap to be moved to the root node before it sifts down the hierarchy. This procedure has to be adapted if non-empty auxiliary buffers exist. In this case, the element with the highest target ad-

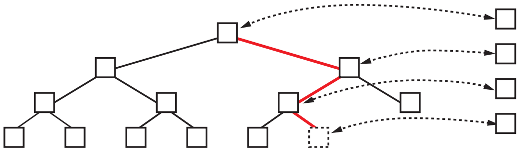


Figure 7: The path an insertion process takes on SIFT-DOWN* is known in advance. Elements are propagated using a chain of auxiliary buffers. Dashed arrows denote element comparisons.

dress 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 [2001] proposed for network switches. Each heap level is accompanied by a 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 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. The improvement offered by the reordering stage over conventional caches will be investigated in Section 7.

5.3 Rasterization and Early Tests

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 5.1, it makes sense to map each rasterizer exclusively to a single cache tile. Let n rasterizers process tile indices $I_1 \ldots I_n$. 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 $I_{n+1} \succ \{I_1 \ldots I_n\}$ 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, further reducing the memory bottleneck. Retrieving n keys in parallel, however, requires intercepting keys $I_1 \ldots I_n$ before the reordering stage and by-passing them directly to the respective FIFOs.

After the rasterization, side-effect free, early fragment tests are performed.

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

The surface reconstruction buffer needs to provide space for all attributes at a precision that exceeds 8 bits per channel. Hence, it usually requires more space than the frame buffer and 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.

6 Implementations

We evaluated the proposed rendering architecture using three prototype implementations.

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 [Zwicker et al. 2004].

The ASIC has been manufactured in a 0.25 μm process using 25 mm^2 die area, and runs at 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 8 shows the final system.

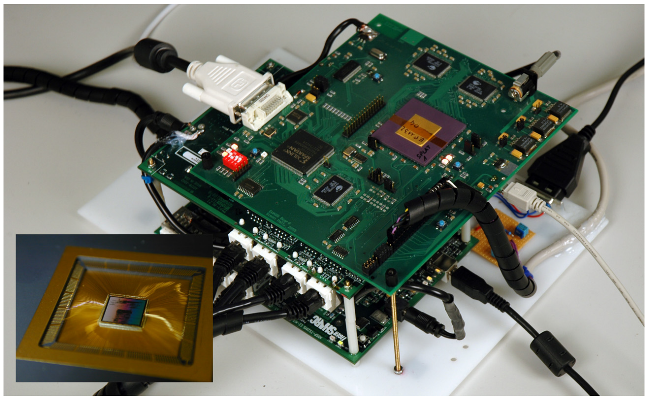


Figure 8: Our prototype board with an ASIC implementation of the EWA splatting rasterizer.

6.2 FPGA Implementation

In order to further investigate our architecture, we aimed at a complete 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 and the rasterization pipeline. The two boards are inter-connected using a 2 GB/s LVDS communication. See Figure 9 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 re-

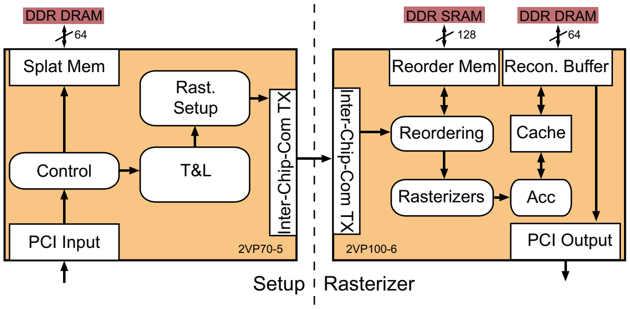


Figure 9: Two FPGAs, coupled to implement a fixed-function T&L stage followed by the proposed rasterization architecture.

ordering 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.

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 blending two fragments every cycle, leading to a theoretical peak performance of 140 M blended fragments/s. Additionally, FLIPQUAD sampling [Akenine-Möller and Ström 2003] is supported for improved silhouette antialiasing. Larger attribute vectors lead to a decreased pixel output. Our design currently supports up to 14 attributes per splat.

In order to simulate the embedding of our extension into an existing GPU, we pass the normalized output from the second FPGA to an NVidia GeForce 6800 GT 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.

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.

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 10 shows two screenshots.

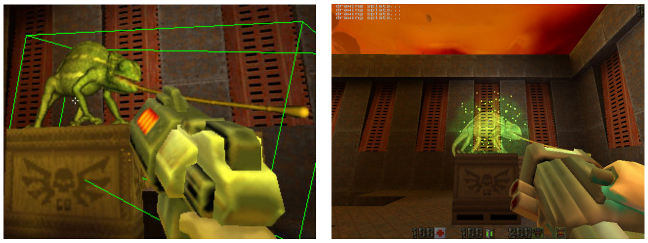


Figure 10: The integration of splat objects into a complex OpenGL application did not require any changes of the surrounding GL code.

7 Results

Table 2 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 antialiasing quality. Scene 3 combines a wide range of splat sizes in a single view, see Figure 11. The displayed statistics were obtained in 512 × 512 resolution with FLIPQUAD supersampling enabled. The screen-shots partly show deferred shading as described in Section 6.2.

7.1 Performance Measurements

The theoretical peak performance of our 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

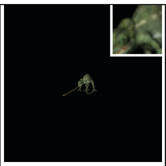
	 Scene 1	 **Scene 2**	 **Scene 3**
Splat count	101 685	101 685	465 878
Coverage	31.2%	1.3%	43.5%
Overdraw	6.8	27.7	17.6
Samples / splat	19.36	3.5	12.6
Shading	deferred	T&L	deferred
Attributes	14	3	6

Table 2: Test scenes used for evaluation, rendered using the FPGA.

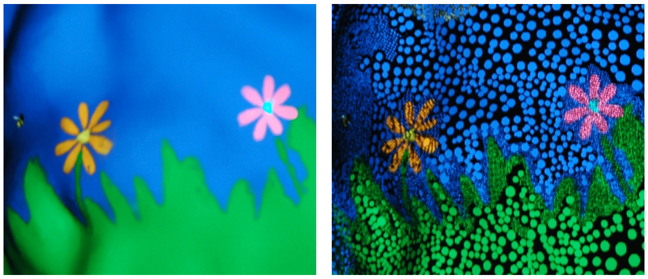
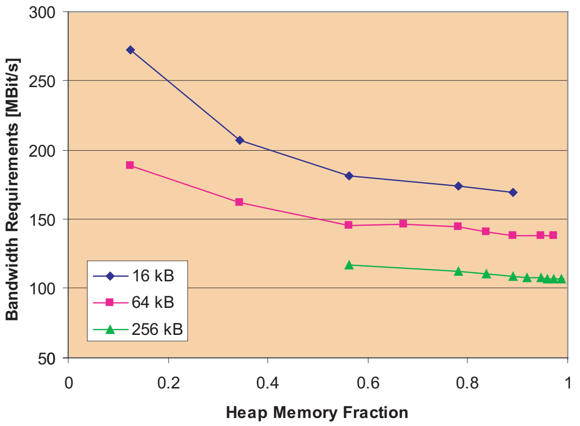


Figure 11: ASIC rendering of an irregularly sampled surface. On the right, the splats have been artificially shrunk to illustrate the sample distribution. Images show photographs taken off a screen.



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.

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 backface culling, driving the ASIC close to its maximum fill rate. The system is presented in the accompanying video.

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. 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 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 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. Although the 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, we are planing an upgrade to a next-generation FPGA for further investigation.

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.

7.2 Scalability

Our FPGA's theoretical peak performance of up to 17.5 M splats per second is comparable to a splat rasterizer running on the latest GPUs. The Nvidia 7800GTX runs at 550 MHz and has external memory bandwidth of 55.4 Gb/s, while ATI Radeon X1800 XT offers 625 Mhz and 48 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 0.11\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 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 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. In terms of gate count our design is very small compared to modern GPUs, and thus duplication of the necessary units seems realistic for getting impressive, say half a billion splats per second, performance.

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 [Gautron et al. 2005] 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, but unfortunately we were unable to obtain samples in time, being forced to use older and smaller chips. In the future, we plan to exploit the parallelism further, carry out more detailed performance and scalability analysis, and also use a wider range of test scenes and applications. Additionally we are investigating the use of geometry shaders to dynamically sample higher-order surfaces into surface splats.

As the proposed EWA splatting pipeline does not utilize deep buffers, it can reconstruct only one layer (e.g. the closest one) of a splat surface for each pixel. 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 [Everitt 2001]. Shadow map lookups, on the other hand, can be implemented in the fragment shader and require no additional units.

Acknowledgments

Many thanks go to Matthias Brändli for the back-end design of the ASIC and to Hanspeter Mathys for his support with the ASIC board production. Tomas Akenine-Möller for helpful suggestions with the text. This research has partly been supported by a grant from the Department of Computer Science, ETH Zurich. FPU IP cores were donated by Arithmatica Inc.

References

Adams, B., Keiser, R., Pauly, M., Guibas, L., Gross, M., and Dutré, P. 2005. Efficient raytracing of deforming point-sampled surfaces. *Computer Graphics Forum 24*, 3.

Akeley, K. 1993. RealityEngine graphics. In *Computer Graphics (Proc. ACM SIGGRAPH '93)*, 109–116.

Akenine-Möller, T., and Ström, J. 2003. Graphics for the masses: a hardware rasterization architecture for mobile phones. *ACM Transactions on Graphics (Proc. SIGGRAPH '03) 22*, 3, 801–808.

Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., and Silva, C. 2001. Point set surfaces. In *Proc. IEEE Visualization*, 21–28.

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

Botsch, M., Hornung, A., Zwicker, M., and Kobbelt, L. 2005. High-quality surface splatting on today's GPUs. In *Proc. Eurographics Symposium on Point-Based Graphics 2005*, 17–24.

Clark, J. 1982. The geometry engine: A VLSI geometry system for graphics. In *Computer Graphics (Proc. ACM SIGGRAPH '82)*, ACM, vol. 16, 127–133.

Deering, M., Winner, S., Schediwy, B., Duffy, C., and Hunt, N. 1988. The triangle processor and normal vector shader: a VLSI system for high performance graphics. In *Computer Graphics (ACM SIGGRAPH '88)*, ACM, vol. 22, 21–30.

Everitt, C. 2001. Interactive order-independent transparency. Tech. rep., Nvidia.

Fleishman, S., Cohen-Or, D., Alexa, M., and Silva, C. T. 2003. Progressive point set surfaces. *ACM Transactions on Graphics 22*, 4.

Fuchs, H., Goldfeather, J., Hultquist, J., Spach, S., Austin, J., Brooks, F., Eyles, J., and Poulton, J. 1985. Fast spheres, shadows, textures, transparencies, and imgage enhancements in pixel-planes. In *Computer Graphics (Proc. ACM SIGGRAPH '85)*, ACM, vol. 19, 111–120.

Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., and Israel, L. 1989. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Computer Graphics (Proc. ACM SIGGRAPH '89)*, ACM, vol. 23, 79–88.

Gautron, P., Křivánek, J., Bouatouch, K., and Pattanaik, S. 2005. Radiance cache splatting: A GPU-friendly global illumination algorithm. In *Proc. Eurographics Symposium on Rendering*, 55–64.

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

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

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

Heckbert, P. 1989. *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Science.

Herout, A., and Zemcik, P. 2005. Hardware pipeline for rendering clouds of circular points. In *Proc. WSCG 2005*, 17–22.

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

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

Kobbelt, L., and Botsch, M. 2004. A survey of point-based techniques in computer graphics. *Computers & Graphics 28*, 801–814.

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

Lindholm, E., Kligard, M. J., and Moreton, H. 2001. A user-programmable vertex engine. In *Computer Graphics (Proc. ACM SIGGRAPH '01)*, 149–158.

Meinds, K., and Barenbrug, B. 2002. Resample hardware for 3D graphics. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 17–26.

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

Microsoft, 2002. Direct3D 9.0. http://microsoft.com/directx/.

Molnar, S., Eyles, J., and Poulton, J. 1992. PixelFlow: high-speed rendering using image composition. In *Computer Graphics (Proc. ACM SIGGRAPH '92)*, ACM, vol. 26, 231–240.

Montrym, J. S., Baum, D. R., Dignam, D. L., and Migdal, C. J. 1997. InfiniteReality: a real-time graphics system. In *Computer Graphics (Proc. ACM SIGGRAPH '97)*, ACM Press, 293–302.

Morein, S., 2000. ATI Radeon – HyperZ Technology. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, Hot3D session.

Olano, M., and Greer, T. 1997. Triangle scan conversion using 2d homogeneous coordinates. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 89–95.

OpenGL Architecture Review Board, and Shreiner, D. 2004. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.4*. Addison Wesley.

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

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

Pineda, J. 1988. A parallel algorithm for polygon rasterization. In *Computer Graphics (Proc. ACM SIGGRAPH '88)*, ACM, vol. 22, 17–20.

Popescu, V., Eyles, J., Lastra, A., Steinhurst, J., England, N., and Nyland, L. 2000. The WarpEngine: An architecture for the post-polygonal age. In *Computer Graphics (Proc. ACM SIGGRAPH '00)*, 433–442.

RAO, V. N., AND KUMAR, V. 1988. Concurrent access of priority queues. *IEEE Trans. Comput. 37*, 12, 1657–1665.

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

REN, L., PFISTER, H., AND ZWICKER, M. 2002. Object-space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Computer Graphics Forum 21*, 3, 461–470.

SAINZ, M., AND PAJAROLA, R. 2004. Point-based rendering techniques. *Computers & Graphics 28*, 869–879.

SCHMITTLER, J., WALD, I., AND SLUSALLEK, P. 2002. SaarCOR: A hardware achitecture for ray tracing. In *Proc. Workshop on Graphics Hardware 2002*, 27–36.

SIGG, C., WEYRICH, T., BOTSCH, M., AND GROSS, M. 2006. Gpu-based ray-casting of quadratic surfaces. In *Proc. Eurographics Symposium on Point-Based Graphics 2006*, 59–65.

STAMMINGER, M., AND DRETTAKIS, G. 2001. Interactive sampling and rendering for complex and procedural geometry. In *Proc. 12th Eurographics Workshop on Rendering*, 151–162.

STEWART, J., BENNETT, E., AND MCMILLAN, L. 2004. Pixelview: A view-independent graphics rendering architecture. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 75–84.

TORBORG, J., AND KAJIYA, J. T. 1996. Talisman: commodity realtime 3D graphics for the PC. In *Computer Graphics (Proc. ACM SIGGRAPH '96)*, 353–363.

WAND, M., FISCHER, M., PETER, I., MEYER AUF DER HEIDE, F., AND STRASSER, W. 2001. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. In *Computer Graphics (ACM SIGGRAPH '01)*, ACM Press, 361–370.

WESTOVER, L. 1990. Footprint evaluation for volume rendering. In *Computer Graphics (Proc. ACM SIGGRAPH '90)*, ACM, 367–376.

WHITTED, T., AND KAJIYA, J. 2005. Fully procedural graphics. In *Proc. ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 81–90.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (SIGGRAPH 2005) 24*, 3, 434–444.

ZHANG, Y., AND PAJAROLA, R. 2006. Single-pass point rendering and transparent shading. In *Proc. Eurographics Symposium on Point-Based Graphics 2006*, 37–48.

ZWICKER, M., PFISTER., H., BAAR, J. V., AND GROSS, M. 2001. Surface splatting. In *Computer Graphics (Proc. ACM SIGGRAPH '01)*, 371–378.

ZWICKER, M., RÄSÄNEN, J., BOTSCH, M., DACHSBACHER, C., AND PAULY, M. 2004. Perspective accurate splatting. In *Proc. Graphics Interface*, 247–254.