Rendering of Complex Heterogenous Scenes using Progressive Blue Surfels

SASCHA BRANDT, Paderborn University
CLAUDIUS JÄHN, DeepL.com
MATTHIAS FISCHER, Paderborn University
FRIEDHELM MEYER AUF DER HEIDE, Paderborn University

We present a technique for rendering highly complex 3D scenes in real-time by generating uniformly distributed points on the scene’s visible surfaces. The technique is applicable to a wide range of scene types, like scenes directly based on complex and detailed CAD data consisting of billions of polygons (in contrast to scenes handcrafted solely for visualization). This allows to visualize such scenes smoothly even in VR on a HMD with good image quality, while maintaining the necessary frame-rates. In contrast to other point based rendering methods, we place points in an approximated blue noise distribution only on visible surfaces and store them in a highly GPU efficient data structure, allowing to progressively refine the number of rendered points to maximize the image quality for a given target frame rate. Our evaluation shows that scenes consisting of a high amount of polygons can be rendered with interactive frame rates with good visual quality on standard hardware.

CCS Concepts: • Computing methodologies → Rasterization; Visibility; Image-based rendering; Virtual reality; Point-based models;

Additional Key Words and Phrases: surfels, splats, point rendering, continuous level of detail, blue noise, virtual reality, CAD

1 INTRODUCTION

Real time rendering of highly complex scenes used to be a major topic in computer graphics for a considerable time. In recent years, the interest dropped as one might consider the problem solved considering the vast computational power of modern graphics hardware and the rich set on different rendering techniques like culling and approximation techniques.

A new challenge arises from the newest generation of head mounted displays (HMD) like the Oculus Rift or its competitors – high framerates are no longer a nice-to-have feature but become a hard constraint on what content can actually be shown to the user. To avoid nausea it is necessary to keep a high frame-rate of 90 fps while rendering in stereo at a high resolution. Although, the lower frame-rate limit of 90 fps can be reduced to 45 fps by using techniques such as time warping (van Waveren 2016), it is still challenging to display highly complex 3d scenes in stereo at the required resolutions. Furthermore, due to the distortion by the HMD lenses, it is usually required to render to an oversized frame buffer that exceeds the HMDs resolution to achieve the required image quality at the focal area. But, this also means, that pixels at the border get skewed and we would render unnecessary high details. Modern game engines adapted to these requirements and allow the rendering of virtual scenes with an impressive visual quality on head mounted displays; but only if the scenes are specially designed for visualization purposes or have a low complexity.

Another challenge is posed by the rendering of complex CAD data. It can be a requirement to visualize multiple machines in a virtual machine hall for interactive design reviews for, e.g., planning
presents. These virtual prototypes are often visualized in a big
cave-like system with multiple projection screens to provide a multi-user
virtual reality view on the 3d scene to analyse and discuss potential
problems and improvements of a technical system. However, the
real-time requirements for such interactive virtual design reviews are
usually much lower than for, e.g., games (20 – 30 fps) and for
the image quality, visualizing functionality is usually more impor-
tant than realism. For virtual design reviews of actual machines,
factories, or buildings, the underlying data is not created for the
visualization itself but is based on potentially highly complex 3d
CAD-data. Converting such CAD data into a suitable virtual scene
typically requires expert knowledge, manual work and a substantial
amount of time.

We present a new approach for rendering very large and complex
scenes supporting a wide range of input scenes, including CAD
data, fast enough for displaying on HMDs. Our approach is fast,
r obust, easy to implement, requires only minimal user interaction
for preparing an input scene, and offers good visual quality, which
automatically adapts to the required framerate and available GPU
computation power. Because our method does not have any require-
ments for the type of scene (e.g. landscape, architecture, machines), it
can render any scene equally well. No time-consuming manual work
for converting CAD data is required. This is achieved by combining
and extending ideas from image-based and point-based rendering,
visibility algorithms and approximation algorithms.

1.1 Outline

The basic idea of our technique is to approximate complex parts
of the scene having a small projected size by a set of points with
much lower complexity. In contrast to other point based approx-
imation algorithms, the points are not placed on all surfaces, but
only on surfaces that are visible if the approximated part is seen
from the outside (external visibility). To minimize the number of
points needed to cover a surface without visible holes, the place-
ment algorithm aims at maximizing the minimal distances between
neighboring points (and thereby aims at creating a blue noise dis-
tribution). Unlike other techniques distributing points evenly on
a three dimensional surface, our algorithm creates a particular or-
dering of the distributed points: Each prefix of the complete set
of points of an approximated part maximizes the closest distances
between neighboring points. Choosing a larger prefix results in a
smaller distance between points and in a denser coverage of the
surface. This allows to dynamically choose as few points for a part
of the scene as are necessary to cover each of its projected pixels
with high probability. The sorted surface points are created in a
preprocessing step and are stored in ordinary vertex buffers. During
runtime a traversal algorithm determines which parts of the scene
are rendered using the original geometry and which ones are ap-
proximated using a subset of points. The number of rendered points
is determined by the available projected size of the rendered part
and the overall rendering budget for that frame. A huge benefit of
this arrangement is, that rendering one array of points requires only
a single draw call to the GPU with a variable length parameter. To
our knowledge, other current progressive simplification methods
still need to perform complex operations for the dynamic simplifica-
tion or refinement on the CPU or GPU, or require a certain structure
of the simplified object.

Our technique consists of two steps: In a preprocessing step, the
surfels are generated based on the scene’s original geometry. We
describe the generation of the initial surfel set, the sampling process
and the surfel generation for hierarchical scenes in Section 3. The
second step is to render the precalculated surfel approximations
during real-time rendering. We present the rendering in Section 4. In
Section 5 we present experimental results evaluating the overhead
of the preprocessing and rendering time and visual quality of the
rendering for a highly complex virtual scene. In Section 6 we discuss
limitations and possible extensions to the technique.

2 RELATED WORK

A lot of research has gone into the area of rendering highly complex
3d scenes in real time and vast amount of techniques has been de-
veloped over the years. The usual approach is to reduce the amount
of data that has to be processed by the graphics hardware by culling
invisible parts of a 3d scene and reducing the complexity of some
objects by replacing it with a simplification (level-of-detail) where
feasible.

For level-of-detail (LOD) based algorithms the 3d scene is usually
partitioned into a hierarchical data structure where each level pro-
vides an approximation of the underlying subtree (HLOD (Erikson
et al. 2001)). The approximations can consist of a discrete set of
geometric models with varying complexity (Luebke et al. 2003),
image-based simplifications (Aliaga et al. 1999; Oliveira et al. 2000;
Stüß et al. 2010), point-based simplifications (Alexa et al. 2004; Gross
2009; Kobbelt and Botsch 2004), or progressive level-of-detail ap-
proximations (Derzapf and Guthe 2012; Hoppe 1996; Yoon et al.
2004).

Progressive LODs have the advantage, that the degree of ab-
straction can be chosen dynamically at run time dependent on the
observers position and viewport, and therefore there is a continuous
transition between different detail levels without popping artifacts
that occur when switching between discrete models. Progressive
Meshes were introduced by Hoppe (Hoppe 1996). A Mesh is pro-
gressively refined or simplified by performing a sequence of split
or collapse operations on the vertices and edges on a mesh. This
idea was later combined with the idea of HLOD to allow for a hier-
archy of progressive simplifications (Yoon et al. 2004). A problem
of progressive meshes is, that they require a certain structure of the
mesh (i.e. 2-manifold geometry), and do not translate well to the
GPU, because of the dependencies between operations and vertices.

Although there are a few attempts for progressive meshes on the
GPU (Derzapf and Guthe 2012; Hu et al. 2009), using progressive
point-based approximations are often better suited, since they usu-
ally don’t require neighborhood dependencies between points.
Dachsbacher et al. (Dachsbacher et al. 2003) proposed a progressive
point-based LOD technique that allows adaptive rendering of point
clouds completely on the GPU. They transfer points effectively to
the GPU by transforming it into a sequential point tree which can
be traversed directly on the GPU by sequential vertex processing.
A similar approach to progressive point rendering as progressive
meshes by Hoppe (Hoppe 1996) was proposed by Wu et al. (Wu et al.
we order the points in a vertex buffer such that each prefix presents a good approximation and the detail level can be chosen by simply specifying the number of points to be rendered. This can be achieved by ordering the points by an approximate greedy permutation or farthest-first traversal (Eppstein et al. 2015), a well known technique from image sampling (Eldar et al. 1997). Using a farthest point strategy, Eldar et al. (Eldar et al. 1997) showed that such a sampling scheme possesses good blue noise characteristics. Having a blue noise quality for a set of point samples (either 2d or 3d) is a desirable property. It guarantees high spatial uniformity and low regularity of the distribution which avoids aliasing artifacts when displaying the points (Yan et al. 2015). Moenning and Dodgson (Moenning and Dodgson 2003) used a farthest point strategy (FastFPS) for the simplification of point clouds and they also hinted at the usefulness of this strategy for progressive point rendering. However, their algorithm requires a dense point cloud or an implicit surface as input.

For the rendering of point clouds, the most common practice is the usage of surfels or splats as introduced by Pfister et al. (Pfister et al. 2000) and Rusinkiewicz et al. (Rusinkiewicz and Levoy 2000). A surfel is a n-tuple which encodes a 3d position together with an orientation and shading attributes that locally approximate an object’s surface. The point cloud can then be rendered using hardware accelerated point sprites (Botsch and Kobbelt 2003; Coconu and Hege 2002) or screen space splatting (Guennebaud et al. 2004; Preiner et al. 2012; Zwicker et al. 2001). Multiple techniques have been developed over the years to improve the visual quality of the rendered surfels, and this is still an active research area.

One major difference of available point-based rendering techniques to our method is how we acquire the initial point set which then gets refined. We only sample the externally visible surfaces of an object by rendering it from multiple directions and using the rendered pixels as basis for the surfel computations. A similar approach for sampling visibility was proposed by Eikel et al. (Eikel et al. 2013). They use the rasterization hardware to compute the direction-dependent visibility of objects in a bounding sphere hierarchy. This allows for efficient rendering of nested object hierarchies without the need of time consuming occlusion culling or high memory consumption for precomputed visibility.

3 Generation of Surfels Approximations

In the following, we describe the generation of surfel approximations (LODs) for complex scenes. A surfel approximation is stored in a single contiguous vertex buffer object where each vertex entry represents a surfel, where a surfel consists of a 3d position, a normal vector and material properties (e.g., color). The order of the surfels in a vertex buffer gives an approximation of the underlying object that can be progressively refined by simply adjusting the number of rendered surfels. This allows for an efficient, cache-friendly rendering of a surfel approximation independent of the desired detail level, by simply rendering only a prefix of the a single vertex buffer object.

We assume the scenes to be represented by a hierarchically organized scene graph, preferably representing the spatial structure of the scene. Scenes originating from CAD data often already provide a suitable structure (object hierarchies, assembly groups). If no such structure is available, commonly used spatial data structures can be applied, e.g. a loose octree (Ulrich 2000). We assume that the scene’s geometry is stored in the leaf nodes of the scene graph and that the geometry is represented by polygonal data, although any renderable opaque surface representation can be used.

We begin by describing the generation of a surfel approximation for a single object, i.e., a single node in the scene graph. First, we generate an initial set of surfel candidates, which is described in Subsection 3.1. Then, we progressively sample the initial set of candidates to achieve the desired surfel approximation of the object (Subsection 3.2). Finally, we describe the hierarchical generation of surfel approximations for an entire scene graph (Subsection 3.4).

3.1 Creating the initial set of surfels

The first step for creating the surfel approximation for a single node in a scene graph is to determine an initial set of possible surfels from which the resulting surfels for the node’s approximation will be drawn. We generate these initial samples using the rasterization hardware by rendering the node’s subtree from multiple directions into a set of G-buffers. This allows us to capture the visible surface of a subtree as seen from outside of it’s bounding volume. In practice, it has shown that rendering from the eight corners of the node’s bounding box, using orthographic projection directed to the center of the node, gives a sufficiently good approximation of the visible surface for most applications. We use multi-target rendering to render the node into multiple output buffers in a single rendering pass, each output buffer encoding a different surface property. One buffer contains the pixels’ 3d position relative to the node’s local coordinate system, another buffer encodes the surface normal in the same coordinate system. At least one buffer is used to encode the pixels material properties. In the simplest version, the ambient, diffuse and specular color values are combined into a single color value. To allow for more complex lighting of the approximation during rendering, further properties can be captured in additional buffers,
like PBR material parameters. As this step uses the standard rendering pipeline, even complex shaders for surface modification and generation can be used if they provide the desired output and only produce opaque surfaces. To assure a sufficiently dense coverage of the surface with surfels, the resolution for rendering the buffers for one direction should at least match the intended resolution used at runtime. A much higher resolution unnecessarily increases the preprocessing time and requires a larger sample size during the following adaptive sampling phase. An example for the created buffers for a single object is shown in Figure 2. After the buffers have been filled, a surfel entry (a record storing a 3D position, normal, color etc.) is added to the initial set of surfels for each covered pixel.

3.2 Progressive Sampling

After creating the initial surfel set, we select and sort a subset of the surfels. The goal is to achieve a sequence of surfels in which for each prefix, the minimal closest distance between neighboring surfels is maximized (greedy permutation), i.e., the first surfels are evenly spread over different parts of the object while further surfels start covering the complete surface until small details are refined at the end of the sequence. In order to approximate such a sequence, we apply a random sampling technique: We start by selecting and removing a random starting surfel from the input set (the initial set of surfels) and add it to the output sequence. Now, we select a uniformly chosen random subset of the remaining input set of fixed size (in practice, \( \sim 200 \) samples yield a reasonably good quality result). We determine the candidate with the largest distance to all surfels chosen before (e.g., using an octree data structure over the chosen surfels), append it to the result sequence, and remove it from the input set. Samples that are at close vicinity to the chosen samples can also be removed from the input set. The other samples remain in the input set for the next round. The sampling step is repeated until a desired number of chosen surfels is reached or the input set becomes empty. The number of created surfels influences the point size that can be chosen during rendering of the node and therefore the quality of the approximation.

In order to speed up the sampling process, we apply a heuristic: After each \( k \) iterations, the number of candidates chosen from the random sample set is increased by one (e.g., \( k = 500 \)). This increases the preprocessing speed while reducing the quality of the distribution only slightly as shown in Subsection 3.3.

3.3 Quality of Sampling Distribution

Since the goal is to cover the entire surface of an object with as few surfels as possible to get the best possible image quality when rendering a surfel approximation, a uniform distribution of the points is very important. A greedy permutation (a sequence of points where every point is as far away as possible from all previous points) has the property, that the points of every prefix are uniformly distributed with blue noise characteristics which reduces aliasing artifacts when displaying the points and therefore gives a good approximation of the underlying surface. We use a simple randomized sampling algorithm, which runs independently from the size of the input data, to quickly get an approximate greedy permutation. Although we might loose some of the desired properties, our experiments show, that our method still gives a reasonably good approximation of a greedy permutation while performing much faster than an exact greedy permutation (computing an exact greedy permutation for 10k points of the bunny model shown in Figure 3 took us \( \sim 15s \) while our method only took \( \sim 180ms \)).

Figure 3 shows surfel prefixes of different sizes (1k, 5k, 10k) for the Stanford bunny model for different sample strategies. Point sizes are reduced to show distributions. Below: Violin plots for the minimum relative point distances for each prefix and strategy.

---

1http://graphics.stanford.edu/data/3Dscanrep/
the minimum distance of the greedy permutation approach while significantly better than the random approach. This is an indicator that our method yields a good surface coverage for a fixed prefix length and therefore allows for fast rendering with a good image quality in comparison to the other two solutions. The image quality is further examined in Section 5.

3.4 Hierarchical generation

For the rendering of complex scenes, we hierarchically generate surfel approximations for multiple subtrees of the existing scene graph structure of a scene. This can be done by traversing the scene graph in a top-down or bottom-up order and generate surfel approximations for each node that exceeds a certain complexity (e.g., generate a surfel approximation of 10k − 100k surfels for each subtree that consists of more than 10k triangles). When generating the surfel approximations bottom-up, one can use the already computed surfels of the child nodes instead of the original geometry to speed up the rendering step for the initial surfel sampling (see Subsection 3.1). Otherwise, any existing approximation or culling technique can be used to generate the images for the initial sampling process. This also allows for easy out-of-core generation of the surfel sets.

For animated or moving objects, one should generate the surfel approximations separately from the static scene parts, since already computed approximations cannot easily be modified without breaking the desired distribution qualities. Unfortunately, complex deforming animations cannot easily be handled by our method, but it shouldn’t be too difficult to incorporate bone weights for skeletal animation in the vertices of the surfel buffer.

4 RENDERING PROGRESSIVE BLUE SURFELS

In this section, we describe the rendering of Progressive Blue Surfels during an interactive walkthrough of a complex scene. The goal is to replace entire subtrees of a scene graph with their corresponding surfel approximation (LOD) as long as the visible surface of the original geometry can be covered by the oriented discs defined by the surfels and as long as the image quality (and run time) suffices for the intended application. Given a fixed surfel size, we can easily compute the required prefix of the surfel approximation dependent on the distance of the approximated object to the observer to cover all pixels of the object in screen space (see Subsection 4.1). An example of different prefix length (100, 1k, 10k, 100k) of a blue surfel approximation of the UNC power plant model (The Walkthru Group 2001) can be seen in Figure 4 compared to the model without any approximation (4e). It also shows a zoomed in view (upper left boxes of Figure 4a-4d) for each of the surfel prefixes as they would actually be rendered with the corresponding prefix length.

We render the surfels as oriented discs by using OpenGL point primitives together with a fragment shader as described in Subsection 4.2. When a surfel approximation for a scene node cannot sufficiently cover the visible geometry of the subtree anymore (i.e., when getting too close to the object), we blend between the node’s approximation and its children’s approximations or original geometry by gradually decreasing the number of rendered surfels for the node while increasing the number of rendered surfels of the child nodes or rendering the original geometry. Finally, in Subsection 4.3, we describe a simple extension to our rendering algorithm that adaptively tries to keep a desired frame-rate while maximizing the possible image quality, and in Subsection 4.4, we describe how our method can be used for simple fixed foveated rendering for head-mounted displays.

4.1 Rendering a surfel prefix

A surfel approximation for a single object is stored in a contiguous vertex buffer on the GPU. We can choose the quality of the approximation by simply adjusting the number of rendered point primitives from this buffer. Ideally, we choose the rendered prefix of the buffer such that the entire surface of the object is covered by surfels without holes, i.e., we choose the prefix length in such a way, that every pixel of the rendered object is covered by at least one surfel. That means, for a given surfel radius r, we want to find a minimal prefix length p s.t. every other surfel in the entire surfel set (which approximates the surface) is covered by a surfel of radius r in the prefix. To find this value p, we use the close relation of greedy permutations to r-nets. An r-net of a point set X is a subset S ⊂ X s.t. no point in S is within a distance of r of each other and every point in X is within a distance of r to a point in S. Now, each prefix
of a greedy permutation is an \( r \)-net for \( r \) equal to the minimum distance between points in this prefix (Eppstein et al. 2015). Using this, and the fact, that for Euclidean metrics in \( \mathbb{R}^d \), every ball of radius \( r \) can be covered by \( \Theta(2^d) \) balls of radius \( r/2 \) (Gupta et al. 2003), we can easily estimate the prefix length \( p \) for a given radius \( r \):

\[
P(r) = p_m \cdot \left( \frac{r_m}{r} \right)^2
\]

Here, \( r_m \) is the precomputed median minimum distance for a fixed prefix length \( p_m \) (e.g., \( p_m = 1000 \)), which can easily be computed during the preprocessing phase (see Subsection 3.2). We use the median of the minimum distances between points to compensate for our used heuristic. We furthermore use the simplified assumption, that the generated surfels lie on a 2-manifold surface (which does not have to be the case), i.e. every surfel disk covers \( \sim 2^2 = 4 \) surfels with half the radius (in \( \mathbb{R}^2 \) this value should be \( \sim 7 \), but 4 gives a good enough estimation).

To get a covering of all pixels in screen-space, we choose \( r \) proportional to the projected distance between two pixels relative to the approximated object’s local coordinate system. For this, we take two neighboring pixels in screen-space, project them onto the view plane going through a point at the object (e.g., the closest point to the bounding box of the object from the camera position) and compute the distance \( d_p \) between these two points (in the object’s local coordinate system). Now, we compute \( r \) by \( r = \frac{s}{d_p} \), where \( s \) is the desired surfel size in pixels (which corresponds with \texttt{glPointSize} in \texttt{OpenGL}).

### 4.2 Drawing oriented discs

We render each surfel of a surfel prefix as a point primitive with a fixed size, i.e., a square in screen space with fixed pixel dimensions. To draw the points as oriented discs (using the stored normals of a surfel), we use a fragment shader. For each rendered fragment of a point primitive, we project the fragment in screen space back onto the plane defined by the surfels position and normal. The fragment is discarded if the distance to the center of the surfel is larger than the size of the surfel in object space. This results in opaque elliptic discs as can be seen in Figure 4.

In this step, it is also possible to use extended filtering methods for surfels as, e.g., EWA filtering as described by Botsch et al. (Botsch et al. 2005), to blend between the colors of neighboring surfels. However, when rendering massive scenes using our method, such filtering methods can become too slow very quickly and for our purposes, it was enough to only render elliptical splats without further filtering. But, as shown in Section 5, we still achieve a reasonable good image quality.

### 4.3 Adaptive rendering

Due to the progressive nature of our method, it is easy to extend our algorithm for rendering complex scenes with an adaptive level-of-detail mechanism that tries to keep a desired frame-rate while maximizing the possible image quality. The image quality of a surfel approximation depends mainly on the size of the rendered surfels (smaller is better) while the frame-rate depends on the number of rendered surfels and polygon count of the original geometry.

Now, for our method, the number of rendered surfels can be directly derived from the desired surfel size (or vice versa) to cover the visible surface of the original geometry (see Subsection 4.1). Therefore, we can easily reduce the frame-time by increasing the size of the rendered surfels and therefore reducing the image quality. This allows for a simple reactive algorithm that increases or decreases the surfel size based on the frame-time of the last rendered frame. We do this by calculating the moving average of the surfel size \( s_{old} \) of the last 3 frames and the surfel size weighted by the deviation factor of the last frame-time \( t_{frame} \) to the target frame-time \( t_{target} \):

\[
s_{new} = \frac{s_{old} \cdot 3 + s_{old} \cdot t_{frame}/t_{target}}{4}
\]

To avoid flickering, we only modify this value when the deviation factor reaches a certain threshold, e.g. when it falls outside of the interval \([0.9, 1.1]\). We also clamp the value by a minimum size of 1px and a small maximum size (e.g., 8px) to avoid surfels that are too big.

### 4.4 Foveated rendering

To allow efficient rendering for head-mounted displays (HMDs), foveated rendering is a great method to significantly improve performance on current HMDs, even without eye-tracking capabilities.

The basic idea is, to decrease rendering complexity and quality in the periphery of the viewpoint while maintaining high fidelity in the focal area (fovea). This is possible due to the distortion by the HMD lenses which skews pixel at the border of the frame buffers for each eye.

With our method it is easily possible to implement a simple foveated rendering technique. For this, we defined different fovea zones in screen space for each eye with different quality settings for the surfel sizes. We then simply interpolate the size of the surfels between zones on a per-object basis to achieve a gradual increase in surfel size to reduce the complexity towards the periphery. This allows for a smooth change in quality which is very important for rendering on HMDs since popping artifacts are especially noticable in the peripheral view.

### 5 RESULTS

In this section we describe the experimental results of our proposed rendering method. In Subsection 5.1 we describe the hardware configuration and the test scene used for evaluating the performance and visual quality of our method. In Subsection 5.2 we examine the preprocessing time of our method. Subsequent, the running time and visual quality during rendering is discussed (Subsection 5.3).

#### 5.1 Benchmark

We implemented Progressive Blue Surfels in our experimental rendering framework. All measurements of the subsequent evaluations were performed using a workstation PC (Intel Core i7-6700 with 4 \( \times \) 3.4 GHz, 32 GB RAM, NVIDIA GeForce GTX 1060). For experiments involving Head-Mounted Displays we used the Oculus Rift CV1 and stereo rendering using an oversized framebuffer of \( 2 \times 1344 \times 1600 \). An example image rendered using our method on the Oculus Rift can be seen in Figure 6.

For our tests, we create a large heterogeneous scene which consists of various parts (see Figure 5). The basic scene is a \( 5km \times 3km \) terrain
Rendering of Complex Heterogenous Scenes using Progressive Blue Surfels

Fig. 5. Overview of the scene used for evaluation with highlighted scene parts. 1 - Pompeii; 2,3 - Car factory; 4,5 - Bakery; 6 - Power Plants

Table 1. Number of Objects, LODs (surfel approximations), Triangles, Surfels, and memory consumption of each scene part. The numbers show the total number including instanced geometry (objects that share the same memory), while the numbers inside parentheses show the number of unique geometry (without instancing).

| Scene part | Objects (unique) | LODs (unique) | Triangles (unique) | Surfels (unique) | Triangle Memory | Surfel Memory | Total Memory |
|------------|------------------|---------------|--------------------|------------------|----------------|--------------|--------------|
| Terrain    | 30518 (524)      | 30259 (265)   | 1.74G (1.73M)      | 1.76G (24.74M)   | 59.11 MB       | 494.79 MB    | 553.90 MB    |
| Pompeii    | 33167 (26878)    | 5547 (2145)   | 62.93M (40.35M)    | 40.28M (29.01M)  | 1.79 GB        | 580.26 MB    | 2.37 GB      |
| Car Factory| 1768 (179)       | 893 (62)      | 75.85M (5.97M)     | 17.36M (2.13M)   | 130.71 MB      | 42.60 MB     | 173.32 MB    |
| Bakery     | 7257 (511)       | 6432 (363)    | 267.80M (27.76M)   | 111.26M (10.23M) | 857.13 MB      | 204.76 MB    | 1.06 GB      |
| Power Plants| 4560 (304)      | 2145 (143)    | 186.00M (12.40M)   | 9.67M (6.64M)    | 364.15 MB      | 132.90 MB    | 497.05 MB    |
| Total      | 77495 (28397)    | 45419 (3240)  | 2.35G (86.23M)     | 2.04G (72.76M)   | 3.20 GB        | 1.43 GB      | 4.66 GB      |

chunk with roads and a high number of trees. On this terrain we placed a set of smaller scenes that each highlight different strengths of our proposed rendering algorithm:

**Terrain** The terrain consists of 260 tiles, each having a size of 255m×255m and containing 4914 Triangles (hexagonal grid). On the terrain we placed 30k trees (randomly selected from 6 unique trees with ~40k~70k triangles each). The other scene parts are connected by roads consisting of 258 simple road segments with ~100~5000 triangles each.

**Pompeii** Highly detailed model of pompeii generated using CityEngine2 (Müller et al. 2006) (Figure 5 view 1). It consists of a high amount of small objects with various materials.

**Car Factory** A large car factory created during a student project consisting of multiple factory halls with moderately complex machinery and car parts (Figure 5 view 2&3).

**Bakery** A smaller factory hall with 5 high detailed triangulated CAD models of donut production lines provided by WP Kemper GmbH3 (Figure 5 view 4&5). Each production line contains ~50M triangles.

---

2http://www.esri.com/software/cityengine/industries/procedural-pompeii
3https://www.wp-kemper.de
Power Plants 16 copies of the UNC Power Plant model (The Walkthru Group 2001) with ~12M triangles each (Figure 5 view 6).

In total the scene consist of ~2.35G triangles in 77495 individual objects (28397 unique triangle meshes). A more detailed breakdown of the scene geometry and memory consumption of each part, as well as the generated surfel approximations, can be gathered from Table 1.

5.2 Preprocessing

![Image](image_url)

Table 2. Preprocessing times of the power plant model for varying surfel counts (10k, 50k, 100k).

| Surfel count | 10k  | 50k  | 100k |
|--------------|------|------|------|
| Render to texture (8x) | 12 ms | 12 ms | 12 ms |
| Creating initial surfel set (1.75M entries) | 82 ms | 82 ms | 82 ms |
| Sampling (sample size 200) | 105 ms | 311 ms | 592 ms |
| **Total** | 199 ms | 405 ms | 686 ms |

In this section we examine the preprocessing time of our method. Table 2 shows the preprocessing times for generating surfel approximations of various sizes for a single object (the UNC power plant (The Walkthru Group 2001)). The initial samples were generated using a resolution of 1024x1024 using 8 directions. For the progressive sampling, a sample size of 200 samples per round were chosen. The only part that depends on the approximated scene geometry is the first rendering step from multiple directions from which the initial sample set is created. However, this is usually only a small part for the generation and can be sped up by using preexisting culling or approximation techniques, or by using previously computed surfel approximations (when approximating larger subtrees of a scene graph). The time taken for creating the initial surfel set is primarily due to transferring the data from the GPU to main memory and depends on the resolution used for creating the samples and (to some degree) the shape of the approximated object. The major portion of the computation is the sampling part, which (due to our randomized sampling technique) only depends on the intended target size of the surfel approximation. This step can easily be multithreaded when generating LODs for an entire scene.

Our benchmark scene contains a total number 3248 unique surfel approximations of varying sizes (1k~200k surfels). We bounded the surfel count for an object by the minimum of 200k surfels and half the complexity of the approximated object (number of triangles). Objects with a complexity of below 1000 triangles were only approximated in groups in a higher hierarchy level. The total preprocessing time for the scene took only ~23.62 minutes on a single thread.

5.3 Rendering Performance & Image Quality

In this section we examine how our proposed method performs in terms of real-time rendering performance and image quality. The image quality was measured by comparing the approximated image with the image rendered without LOD using the hierarchical Structural SIMilarity index method proposed by Wang et al. (Wang et al. 2004). An example of this method is shown in Figure 7. The upper image shows a camera view rendered with our method, while the middle image shows a view without any approximations. The bottom image shows the SSIM image which highlights the differences of both images from which the SSIM index (0.81) is computed. Especially noticeable is the difference at the trees and the crane of the power plant. The trees and crane rendered with our method seem more volumetric since our method cannot handle thin or finely detailed objects very well. However, this fact can also be utilized to achieve some degree of anti-aliasing.

For the evaluation of our method, we placed cameras at various representative positions in our benchmark scene to show different aspects of our algorithm. For each camera position we measured the average frame time to render a single image at different resolutions (1k, 2.5k, 4k) as well as stereo rendering for head-mounted displays (2x1344x1600). We computed the SSIM index for these positions at each resolution, except for HMD rendering since the value might get a wrong impression due to the distortion of the HMD lenses.
Table 3 shows the measured camera positions with the resulting statistics for number of draw calls, rendering time, and image quality of this specific view for each of the different resolutions. The last column additionally shows a combined box & violin plot of the distribution of frame times measured at uniformly distributed positions at the specific region of the view for each of the cardinal directions. For the overview, we measured ~10000 positions at a height of 200m above ground with a slight downward tilt for each camera view. For the other views, we measured ~1000 positions (~100 for the bakery) at ground level (2m).

In general, we achieved high frame rates of at least 30 fps for each camera position and resolution while achieving a relatively good SSIM index of at least 0.68 for the overview (which is mainly due to the large amount of trees) and at least 0.9 for camera positions closer to the ground level. For resolution of 1920x1080 we even achieve frame rates of at least 60 fps throughout the entire scene. The lowest frame rates can be observed at Pompeii and the Bakery, which are the most complex parts of our benchmark scene. Here, we have high number of complex scene objects close to the observer, which cannot be effectively approximated anymore by our algorithm alone (see, e.g., Figure 8). Still, the median frame times in these cases are still at least around 15 ms (~66 fps) for a 4k resolution and at least 13 ms (~76 fps) for HMD stereo rendering as can be seen in the box plot in the rightmost column of Table 3.

![Image of a complex scene](image-url)
Although we did not achieve the desired 90 fps for stereo rendering on an HMD everywhere, we were still able to keep the frame rates (mostly) in the range of 45–90 fps which is the range used for effective time-warping to reduce nausea (van Waveren 2016). These frame rates can certainly be improved using further specialized techniques for HMD rendering, like fixed foveated rendering or multiresolution framebuffers. 

6 CONCLUSION AND FUTURE WORK

We have presented an efficient point-based algorithm for generating and rendering continuous approximations of highly complex scenes in real time, even for VR applications using head-mounted displays on standard consumer hardware. Our method can handle a large variety of scene types, including complex CAD data. It can robustly create approximations of almost any surface requiring only little user interaction and parameterization. The image quality is reasonably good (with room for improvement) and, due to the continuous nature, there are almost no visible popping artifacts when navigating a scene. Using our method combined with other culling and level-of-detail techniques, we are certain that we can achieve an even better performance, and with better point based filtering to improve the visual quality it might also be applicable in the context of games.

However, there are some limitations. Since our method is mostly intended for objects with a small projected size on the screen, very complex geometry, which covers a large portion of the screen, cannot be effectively approximated. Also, since we uniformly distribute points on the visible surface of an object, we might draw unnecessarily many points for long objects where one part is close to the observer while other parts are further away. Currently, the only way to circumvent these problems is to cut these objects into smaller parts and approximate each part seperately, which results in more draw calls and possibly higher memory consumption for the surfel approximations.

Another type of objects that cannot be handled well, are objects with thin surfaces or walls. Since our sampling method currently does not incorporate the normal of a sampling point, it might happen that we incorrectly distribute the surfels on the surface of such objects, which results in holes during rendering. In future work, we would like to include (approximate) geodetic distances in our generation of greedy permutations to better capture the surface of an object.

Furthermore, we want to include better filtering of color values of the surfels, since our current sampling process does not take the colors into account. It can happen that samples are taken during the preprocessing step from pixels with colors of minor importance, as can be seen in Figure 4 (blue disc in the first few pictures). This could be circumvented, e.g., by averaging the color values of early prefixes in an additional post-processing step. Another idea would be, to generate the samples from different mip-levels of the rendered images.

Due to the progressive nature of our method, it is an ideal basis for streaming applications in out-of-core systems and mobile rendering. In future work, we also want to use our method for on-the-fly generation of approximations at run time, e.g. for very large, procedurally generated worlds.

REFERENCES

Marc Alexa, Markus Gross, Mark Pauly, Hanspeter Pfister, Marc Stammbinger, and Matthias Zwicker. 2004. Point-based Computer Graphics. In ACM SIGGRAPH 2004 Course Notes (SIGGRAPH ’04). ACM, New York, NY, USA. https://doi.org/10.1145/1103900.1103907

Daniel Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erickson, Kenny Hoff, Tom Hudson, Wolfgang Strasserling, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manocha. 1999. MMR: an interactive massive model rendering system using geometric and image-based acceleration. In Proceedings of the 1999 symposium on Interactive 3D graphics (ID ’99). ACM, New York, NY, USA, 199–206. https://doi.org/10.1145/300023.300054

Mario Botsch, Alexander Hormung, Matthias Zwicker, and Leif Kobbelt. 2005. High-Quality Surface Splatting on Today’s GPUs. In Symposium on Point Based Graphics, Stony Brook, NY, USA, 2005. Proceedings, Marc Alexa, Szymon Rusinkiewicz, Mark Pauly, and Matthias Zwicker (Eds.). Eurographics Association, 17–24. https://doi.org/10.2312/SPBG/SPBG05/017-024

M. Botsch and L. Kobbelt. 2003. High-quality point-based rendering on modern GPUs. In 11th Pacific Conference on Computer Graphics and Applications, 2003. Proceedings. 335–343. https://doi.org/10.1109/PCCA.2003.1283275

Liviu Coconu and Hans-Christian Hege. 2002. Hardware-Accelerated Point-Based Rendering of Complex Scenes. In Proceedings of the 13th Eurographics workshop on Rendering (EGWR ’02). P. Debevec and S. Gibson (Eds.). Eurographics Association, 43–52. http://www.cg.technion.ac.il/graphics/course/542/556/papers/5

Carsten Dachsbacher, Christian Vogelsang, and Marc Stammbinger. 2003. Sequential Point Trees. ACM Trans. Graph. 22, 3 (July 2003), 657–662. https://doi.org/10.1145/882622.882321

Evgenij Derzapf and Michael Guthe. 2012. Dependency-Free Parallel Progressive Meshes. Comput. Graph. Forum 31, 8 (2012), 2288–2302. https://doi.org/10.1111/j.1467-8659.2012.03154.x

Benjamin Ekel, Claudius Jahn, Matthias Fischer, and Friedhelm Meyer auf der Heide. 2013. Spherical Visibility Sampling. Comput. Graph. Forum 32, 4 (2013), 49–58. https://doi.org/10.1111/cgf.12150

Y. Eldar, M. Lindenbaum, M. Porat, and Y. Zeevi. 1997. The farthest point strategy for progressive image sampling. IEEE Transactions on Image Processing. 6, 9 (Sep 1997), 1305–1315. https://doi.org/10.1109/83.623193

David Epstein, Sariel Har-Peled, and Anastasios Sidiropoulos. 2015. Approximate Greedy Clustering and Distance Selection for Graph Metrics. CoRR abs/1507.01555 (2015). http://arxiv.org/abs/1507.01555

Carl Erickson, Dinesh Manocha, and William V. Baxter III. 2001. HLODs for faster display of large static and dynamic environments. In Proceedings of the 2001 Symposium on Interactive 3D Graphics, S3ID 2001, Chapel Hill, NC, USA, March 26–29, 2001, John F. Hughes and Carlo H. Séquin (Eds.). ACM, 111–120. https://portal.acm.org/citation.cfm?id=364338.364576

Markus Gross. 2009. Point based graphics. In ACM SIGGRAPH 2009 Courses on -SIGGRAPH ’09. ACM Press. https://doi.org/10.1145/1667239.1667257

Gaël Guennebaud, Loïc Barthe, and Mathias Paulin. 2004. Dynamic surfel set refinement for high-quality rendering. Computers & Graphics 28, 6 (2004), 827 – 838. https://doi.org/10.1016/j.cag.2004.08.011

Anupam Gupta, Robert Krauthgamer, and James R. Lee. 2003. Bounded Geometries, Fractals, and Low-Distortion Embeddings. In Proceedings of the 44th Symposium on Foundations of Computer Science (FOCS 2003). IEEE Computer Society, Cambridge, MA, USA, 534–543. https://doi.org/10.1109/SFCS.2003.1238226

Hugues Hoppe. 1996. Progressive meshes. In Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (SIGGRAPH ’96). ACM, New York, NY, USA, 99–108. https://doi.org/10.1145/237170.237216

Liang Hu, Pedro V. Sander, and Hugues Hoppe. 2008. Parallel view-dependent refinement of progressive meshes. In Proceedings of the 2009 Symposium on Interactive 3D Graphics, S3ID 2009, February 27 - March 1, 2009, Boston, Massachusetts, USA, Eric Haines, Morgan McGuire, Daniel G. Aliaga, Manuel M. Oliveira, and Stephen N. Spencer (Eds.). ACM, 169–176. https://doi.org/10.1145/1507149.1507177

Leif Kobbelt and Mario Botsch. 2004. A survey of point-based techniques in computer graphics. Computers & Graphics 28, 6 (2004), 801 – 814. https://doi.org/10.1016/j.cag.2004.08.009

David Luebke, Martin Reddy, Jonathan d. Cohen, Aminthab Varshney, Benjamin Watson, and Robert Huebner. 2003. Level of Detail for 3D Graphics. Morgan Kaufman Publishers, San Francisco, USA.

Carsten Moenning and Neil A. Dodgson. 2003. Fast Marching farthest point sampling. In Graphics 2003 - Posters. Eurographics Association. https://doi.org/10.2312/egp.20031024

Pascal Müller, Tijl Verbeeck, Peter Wonka, Iken Paap, and Luc J. Van Gool. 2006. Procedural 3D Reconstruction of Pupe Buildings in Xkipché. In VAST 2006: The 7th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage, Nicosia, Cyprus, 2006. Proceedings, Marinos Ioannides, David B. Arnold, Franco Nicolucci, and Katerina Mania (Eds.). Eurographics Association, 139–146. https://doi.org/10.2312/VAST/VAST06/139-146
Rendering of Complex Heterogenous Scenes using Progressive Blue Surfels

Manuel M. Oliveira, Gary Bishop, and David McAllister. 2000. Relief texture mapping. In Proceedings of the 27th annual conference on Computer graphics and interactive techniques. ACM Press/Addison-Wesley Publishing Co., 359–368. https://doi.org/10.1145/344779.344947

Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus H. Gross. 2000. Surfels: surface elements as rendering primitives. In Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2000, New Orleans, LA, USA, July 23-28, 2000, Judith R. Brown and Kurt Akeley (Eds.). ACM, 335–342. https://doi.org/10.1145/344779.344936

Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. 2012. Auto Splats: Dynamic Point Cloud Visualization on the GPU. In Eurographics Symposium on Parallel Graphics and Visualization, EGPGV 2012, Cagliari, Italy, May 13-14, 2012. Proceedings, Hank Childs, Torsten Kuhlen, and Fabio Marton (Eds.). Eurographics Association, 139–148. https://doi.org/10.2312/EGPGV/EGPGV12/139-148

Szymon Rusinkiewicz and Marc Levoy. 2000. QSplat: a multiresolution point rendering system for large meshes. In Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2000, New Orleans, LA, USA, July 23-28, 2000, Judith R. Brown and Kurt Akeley (Eds.). ACM, 343–352. https://doi.org/10.1145/344779.344940

Tim Süß, Claudia Jahn, and Matthias Fischer. 2010. Asynchronous Parallel Reliefboard Computation for Scene Object Approximation. In Proceedings of the 10th Eurographics Symposium on Parallel Graphics and Visualization (EGPGV ’10). Eurographics Association, Eurographics Association, Norrköping, Sweden, 43–51.

The Walkthru Group. 2001. Power Plant Model. Internet page. (March 2001). http://gamma.cs.unc.edu/POWERPLANT/ University of North Carolina at Chapel Hill.

Thatcher Ulrich. 2000. Loose Octrees. In Game Programming Gems, Mark DeLoura (Ed.). Charles River Media, Boston, MA, USA, Chapter 4.11, 444–453.

J. M. P. van Waveren. 2016. The asynchronous time warp for virtual reality on consumer hardware. In Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology, VRST 2016, Munich, Germany, 2-4 November, 2016, Dieter Kranzlmüller and Gudrun Klinker (Eds.). ACM, 37–46. https://doi.org/10.1145/2993369.2993375

Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, and Eero P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. Image Processing, IEEE Transactions on 13, 4 (April 2004), 600–612. https://doi.org/10.1109/TIP.2003.819861

Jianhua Wu, Zhuo Zhang, and Leif Kobbelt. 2005. Progressive Splatting. In Symposium on Point Based Graphics, Stony Brook, NY, USA, 2005. Proceedings, Marc Alexa, Szymon Rusinkiewicz, Mark Pauly, and Matthias Zwicker (Eds.). Eurographics Association, Eurographics Association, Norrköping, Sweden, 43–51.

Dong-Ming Yan, Jianwei Guo, Bin Wang, Xiaopeng Zhang, and Peter Wonka. 2015. A Survey of Blue-Noise Sampling and Its Applications. J. Comput. Sci. Technol. 30, 3 (2015), 439–452. https://doi.org/10.1007/s11390-015-5535-0

Sung-Eui Yoon, Brian Salomon, Russell Gayle, and Dinesh Manocha. 2004. QuickVDR: interactive view-dependent rendering of massive models. In 31. International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2004, Los Angeles, California, USA, August 8-12, 2004, Sketches, Ronen Barzel (Ed.). ACM, 22. https://doi.org/10.1145/1186223.1186251

Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. 2001. Surface Splatting. In Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ‘01). ACM, New York, NY, USA, 371–378. https://doi.org/10.1145/383259.383300