Institut für Computergraphik und Algorithmen
Technische Universität Wien
Karlsplatz 13/186/2
A-1040 Wien
AUSTRIA
Tel: +43 (1) 58801-18601
Fax: +43 (1) 58801-18698

Institute of Computer Graphics and Algorithms
Vienna University of Technology
email:
technical-report@cg.tuwien.ac.at
other services:
http://www.cg.tuwien.ac.at/
ftp://ftp.cg.tuwien.ac.at/

TECHNICAL REPORT

Interactive Screen-Space Triangulation for High-Quality Rendering of Point Clouds

Reinhold Preiner Michael Wimmer

TR-186-2-12-01
April 2012

Interactive Screen-Space Triangulation for High-Quality Rendering of Point Clouds

Reinhold Preiner Michael Wimmer

April 4, 2012

Abstract

This technical report documents work that is a precursor to the Auto Splatting technique [PJW12]. We present a rendering method that reconstructs high quality images from unorganized colored point data. While previous real-time image reconstruction approaches for point clouds make use of preprocessed data like point radii or normal estimations, our algorithm only requires position and color data as input and produces a reconstructed color image, normal map and depth map which can instantly be used to apply further deferred lighting passes. Our method performs a world-space neighbor search and a subsequent normal estimation in screen-space, and uses the geometry shader to triangulate the color, normal and depth information of the points. To achieve correct visibility and closed surfaces in the projected image a temporal coherence approach reuses triangulated depth information and provides adaptive neighbor search radii. Our algorithm is especially suitable for in-situ high-quality visualization of big datasets like 3D-scans, making otherwise time-consuming preprocessing steps to reconstruct surface normals or point radii dispensable.

1 Introduction

With the advent of cheap range scanners, large point clouds have become a ubiquitous source of 3D data in many application areas, including archaeology, building history, urban planning, architecture etc. One of the predominant problems when working with such data is that it requires extensive preprocessing before being useful in an application scenario. However, often neither the time nor the resources for preprocessing are available. For example, in an archaeological scanning campaign, archaeologists want to view the acquired data already during acquisition in order to judge whether all important aspects of a monument have been captured.

Figure 1: Left: reconstruction using splats. Right: reconstruction using our new screen-space triangulation.

Several previous approaches have proposed out-of-core rendering systems that are capable of giving a quick preview of even very large models (more than a Billion points). While such previous are already very useful, they suffer from poor display quality. One option is to display points as simple splats with constant size, which leads to holes for sparse regions especially near the viewer, as well as overlaps and general low image quality for more dense regions. Another option is to estimate local point densities from the hierarchies inherent in the underlying out-of-core rendering system and adapt the splat sizes to those densities. However, the quality that can be obtained without knowledge of splat orientation (i.e., normal vector) or more accurate splat sizes is very limited, leading to strong overlap artifacts especially for surfaces viewed edge-on.

Another very large class of algorithms provide very high-quality point cloud rendering with closed surfaces, but rely on accurate normal vector and splat size (and shape) estimation. Unfortunately, such estimations are usually not available for real-world data. In fact, the best way for obtaining such estimations is to create a triangulation of the point cloud and sample all required attributes from the created mesh. However, this is both

very expensive and requires great care and manual intervention to obtain good results.

In this paper, we propose to use the power of current graphics hardware to attempt a high-quality reconstruction of colors and normals directly in screen space, without requiring a lengthy preprocessing phase. Instead of rendering splats as in previous point-cloud previewing systems, we propose to perform a triangulation based directly on screen-space samples, thus obtaining closed surfaces (see Figure 1). This has the advantage that small-scale detail and silhouettes can be preserved, while colors between available samples can be interpolated for a smoother reconstruction. As input for our method, we assume that only point positions and colors are given. Colors are usually available from a camera mounted directly on a range scanner, or from photogrammetric reconstruction. We explicitly do not require information about normal vectors or splat sizes, as this would already require lengthy preprocessing especially for larger datasets.

Contribution

- Our algorithm can render *high-quality* images from colored point clouds *without* the need for common *preprocessing* tasks like normal oder splat radius estimation.
- It produces a *normal map* of the scene in screen space *on the fly*, which can be used for instant deferred shading.
- We introduce a new *screen-space triangulation* method to render triangles that interpolate the color and normal information between the point data in screen space.

2 Related Work

Our work can be seen in the context of mesh reconstruction from unorganized point clouds, large point cloud rendering, and high-quality surfel rendering. Starting with Hoppe's seminal *reconstruction* work [HDD*92], there has been a plethora of papers treating various aspects of surface reconstruction from point clouds. They all share the property that while providing good results, they require lengthy preprocessing and manual intervention, something we want to avoid in our method. While our screen-space triangulation also attempts a local reconstruction, we do not provide a strict triangulation (i.e., triangles can overlap), but try to achieve the best possible image quality with the available samples.

Point-based rendering has been a very popular research topic in the past decade. However, surprisingly enough, practically all published algorithms (especially [PZvBG00] and many follow-up works) dealt with the problem of rendering models that contain complete information about point orientation and extent, in other words how to render complete surfels and not point samples. Even the approach that is closest to ours, a screen-space pull-push algorithm in order to reconstruct closed surface images from point clouds, relies on precalculated normals and splat radii [MKC07].

Gobetti and Martin [GM04] and Wimmer and Scheiblauer [WS06, SZW09] were the first to abandon the requirement of precalculated normals and allow visualizing huge point clouds in their layered point clouds respectively instant points systems. Both systems focus mainly on the aspect of rendering huge point clouds that do not fit into main memory or even graphics card memory using clever hierarchies. While the former basically relies on a dense enough sampling to avoid holes in the reconstruction, the latter provide a heuristic to render quad-shaped splats to obtain a closed surface. However, quad-shaped splats do not allow for interpolation and are prone to artifacts if not viewed head-on, and are therefore limited in the obtainable image quality.

In this paper, we want to avoid preprocessing while still providing good image quality. In fact, our method only requires an input stream of point positions and colors and is therefore ideally suited to be combined with an out-of-core point rendering system as shown above.

3 Algorithm

3.1 Motivation

First, we try to explain our new algorithm at a higher level. Any surface reconstruction algorithm has to answer the following questions:

- Given a surface sample, what are neighboring surface samples that can be used for a local reconstruction?
- Given the local neighborhood of samples, how should the surface be reconstructed?

In classic reconstruction algorithms, the first step usually entails a k-nearest-neighbor search in a three-dimensional data structure, a luxury we cannot afford when only screen-space samples are available. Instead, we need to rely on two resources: visibility and local

screen-space neighborhoods. In a way, the screen-space grid is a search structure provided by the graphics hardware, however with perspective distortion and only in two dimensions. However, since we are only interested in a reconstruction of the closest surface, this is sufficient. Furthermore, we have to face the problem that we have reconstruct and render at the same time. In particular, rendering requires correct visibility in order to avoid holes. We use is temporal coherence to tackle this problem.

Nearest neighbor search At the heart of our algorithm lies a novel method to quickly find nearest neighbors on the GPU. The idea is to establish communication between potential neighbors. Basically, each point attempts to *register* itself as a neighbor with all points in a given world-space radius. This can be done by rendering a splat that covers the respective screen-space extent. However, a straightforward implementation of this idea would require each point to maintain a list of all points that register with it, which cannot be achieved with current graphics pipelines. Our solution to this problem is to *discretize* the neighborhood of each point into a number of sectors (in screen space) and always maintain the currently nearest point for each sector. In fact, graphics hardware requires us to split this task into two passes: in a first pass, each point determines the *distance* to the nearest neighbor in each sector only, while in a second pass, the actual nearest neighbor in each sector is allowed to register itself.

Normal estimation and triangulation After the neighborhood search, each point knows about its nearest neighbor (if any) in each of its sectors. This information is very powerful and can be used in various ways. For example, it is easy to estimate normal vectors from the neighboring points. Most importantly, since this information is already spatially ordered, it is rather straightforward to create a triangle fan from the point spanning its neighbors. However, this will lead to a triangle being created more than once (up to there times, for each of its vertices). We found that this is still more efficient than trying to avoid multiple rasterizations, and resort to blending instead.

Visibility and temporal coherence One major difference to offline reconstruction algorithms is that we need to reconstruct *and* render the point cloud at the same time. In particular, we only want to reconstruct the surfaces visible to the viewer, thus requiring visibility. However, determining visibility already requires a reconstructed surface to avoid background surfaces to shine through the spaces between samples. In our system we solve this by resorting to temporal coherence: in each frame, the currently reconstructed surface is written to the depth buffer, while the reconstruction algorithm itself accesses depth information from the previous frame (through reprojection as also used in [SJW07]). This will of course lead to short-term holes for parts of the scene that appear in a new frame (due to rotation or disocclusions), however these holes are quickly filled in subsequent frames.

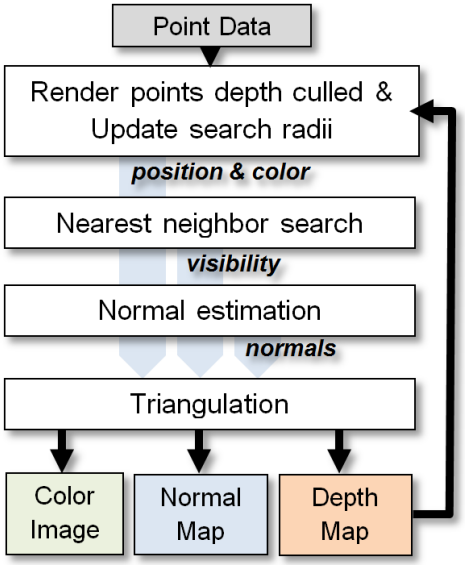


Figure 2: Overview of the steps and data-flow of our algorithm.

3.2 Overview

Our algorithm is fed with unorganized, colored point data, and outputs a reconstructed image, a screen-space normal map and a depth map of the current frame. Figure 2 shows an overview over the steps of our algorithm.

In the first step, the point cloud data (position and colors) are rendered into the frame buffer with pixel size 1. In this render pass, the points are depth-culled against the reprojected depth map that results from the previous frame – in this way we can reuse already obtained reconstructed surface depth information to get progressively better visibility over time. Further, in the first pass also the neighbor search radii of the visible points are updated. (Neighbor search radii and their adjustment are described in detail in Section 3.6).

The second step performs a screen-space neighbor search for each visible point in the frame buffer. This

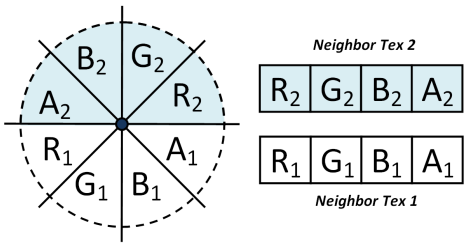


Figure 3: Storage layout for the nearest neighbors in the 8 surrounding screen-space sections of a point.

step consists of two passes, after which we retrieve the screen coordinates of at most 8 nearest neighbors per point. With the resulting neighborhood-information, we subsequently perform a normal estimation pass on the points in screen-space.

Finally, having the view-space depth, the estimated world-space normals and the color per point in the frame buffer, we perform a triangulation pass on the points, using the geometry shader. This results in three output buffers (color map, depth map and normal map).

In the following, the individual steps of our algorithm are described in detail. Besides the first pass, which renders the point data into the frame buffer (object-pass), all remaining passes are screen-space passes that operate on those framebuffer pixels that contain point information. For convenience, we will refer to such pixels as "points".

3.3 Nearest neighbor search

In order to be able to perform a convenient triangulation of the points in the framebuffer, each point must store its world-space nearest neighbor points. Our algorithm stores up to eight nearest neighbors per point. We subdivide the screen-space region around each point in 8 sections and use two RGBA textures in which we render the pixel ID of the nearest neighbor point for each of this sections. Figure 3 illustrates this concept. The unique pixel IDs we store in the texture channels are simply calculated from the screen positions of the points.

Neighbor retrieval is performed by a two-pass method that each time renders a point-sprite splat for each point, with size equal to the projection of the world-space search radius of the point. We use a vertex buffer object (VBO) containing the uv-coordinates for each pixel in the viewport, and pass it through a vertex shader that renders the splats. The fragment shader then is able to write information to each of the neighbor points within the splat radius.

Let P be any point in the framebuffer, and Q be a point covered by the screen-space splat of P. In both passes, the fragment shader first calculates the world-space position of Q. If it lies outside the world-space search radius of P, Q is discarded. Otherwise, we write to the output texture channel that corresponds to the sector of Q in which P is positioned. To be able to write selectively to one specific channel, while leaving the others untouched, we use different blending operations in each pass.

In the first pass, the fragment shader writes the world-space distance from P to Q, with per-channel minimum blending enabled. (OpenGL provides the `glBlendEquation(GL_MIN)` command to leave the lowest value per channel). After this step, each point knows the distance to its nearest neighbor per section. In the second pass, all splats are rendered the same way, but this time with additive blending enabled. In the fragment shader, P tests the world-space distance between P and Q against the minimum value of the according section calculated in the previous pass, and writes its ID if he finds to be the nearest neighbor in this section. This mechanism could be refered to as a radial two-pass visiblity test. Figure 4 gives an example for this approach.

Note that this technique doesn't produce a real triangulation of the point clouds in terms of a mesh. Due to the subdivision of the screen-space region around a point, triangle overlappings may occur. We address this issue by blending the rendered triangles later in the triangulation step. Notice further that searching for neighbors within a screen-space splat is sufficient to cover all points that lie within the world-space search radius of a point P, i.e. if in 3D a point Q lies within the search sphere of P, then its projection Q' lies within the projection of the world-space sphere in 2D.

In the subsequent passes, each point in the framebuffer is able to lookup its neighbor's IDs which can be unwrapped to their screen coordinate. With this coordinate it can

- lookup its neighbors' view-space depths,
- calculate their world-space positions by unprojection, and
- lookup any neighbor information like their estimated surface normals.

3.4 Normal estimation

After we have determined the pixel IDs of the nearest neighbor points, we perform a fast screen-space normal estimation pass. Contrary to the two previous neighbor

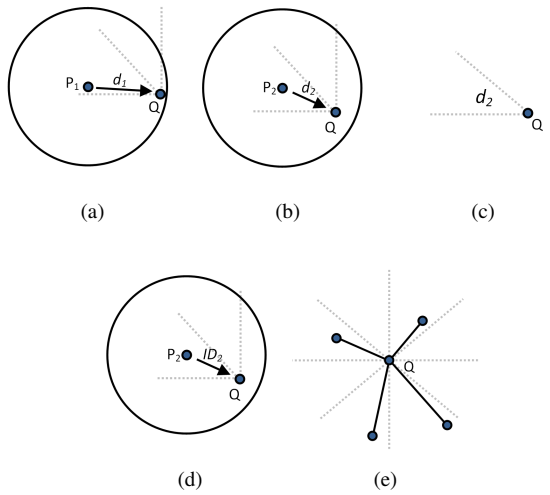


Figure 4: Example for the two pass neighbor search procedure. Upper row: in the first pass, both $P1$ and $P2$ lie in the same section of Q, thus writing their distances to the same texture channel (4a and 4b). Since $P2$ has a lower distance than $P1$, $d2$ remains in the section (4c). Lower row: In the second pass, each point looks up the minimum distance again. $P2$ finds that the stored distance equals its own distance and stores its ID in Q (4d). Finally, Q contains the nearest neighbor IDs for each section (4e).

search passes, this is done by a simple per-pixel operation on the framebuffer. Since the neighbor IDs of any point P are stored in circular order in the neighbor textures, we are able to reproduce the circular sequence of the neighbors in terms of a triangle fan, in which P is the origin. For each triangle in this fan we determine the world-space position of the two adjacent neighbors and retrieve its face normal. The world-space normal of P can then be simply estimated by the average of the surrounding triangle's face normals. Note that with this method we do not reproduce any inside/outside information, thus the normal are always oriented towards the viewer.

3.5 Triangulation

In the last step, we invoke the geometry shader to triangulate the triangle fan that is span up be each point with its surrounding neighbors. Again we make use of the circular order of the neighbors in its screen-space sections. A triangle is only rendered if the opening angle between its neighbor points is lower than π to avoid

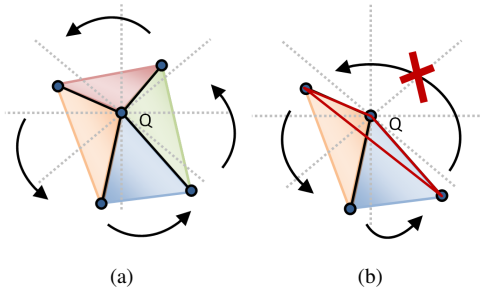


Figure 5: 5a) Example triangulation of a point Q and its neighbors. The geometry shader generates a triangle fan is in circular order of the sections. 5b) If in circular order, two neighbors span a angle $\geq \pi$, no triangle is rendered in order to avoid overlappings.

triangles overlapping within a fan. Figure 5 shows this triangulation concept.

Each triangle contains its color, its normal, and the viewspace depth as vertex attribute. Using triangulation by the geometry shader, each of these attributes is automatically interpolated over the render target pixels.

At the rasterization stage, we average the color and the normal information of the rendered triangles by additive blending and a subsequent normalization pass of the values. For the depth value, we need to use minimum blending to get a correct depth map of the scene. However, the triangulation of each of the three values can be performend in one pass, using a OpenGL feature that allows to set different blending operations for both the RGB and the alpha channels (`glBlendEquationSeparate`).

After this final pass, we obtain three distinct output buffers: a color image of the scene, a normal map, and a linear depth map. For extended rendering requirements, these buffers can instantly be used to performed deferred shading on the scene. Figure 6 shows an example phong shaded point cloud model.

3.6 Search radii adjustment

Inherently to our method, the choice of a convenient world-space neighbor search radius for a point is critically for retrieving its neighbor IDs. The size of this radius is constrained to two sides: Too small search radii lead to points not finding each other, while too large search radii result in rendering too large point splats an thus in a lot of unnecessary neighbor checks, wasting computation time. To achieve good results in our point cloud scenes we use an adaptive search radii approach

that adjusts the radii by a feedback mechanism between the points that makes use of temporal coherence.

This mechanism works in two steps: In each frame, after the neighbor search we execute a feedback pass where each point looks up its surrounding neighbors and writes its distance to them as feedback information back to their position. (This is done by the geometry shader that emits a single vertex at each neighbor's position). This is performed using maximum blending, i.e. we retrieve for each point P the distance of the farest point Q that sees P as its nearest neighbor. When projecting the point data into the framebuffer in the next frame, each point looks up both its previous search radius and the feedback values given by the neighbors in the last frame. With this information we can adjust the search radii for the current frame by the following simple ruleset, which is parameterized by some user-defined maximum connection distance for the points:

- If the previous search radius of P is zero, then this point was not visible in the previous frame (occluded or outside the viewport). In this case, we initialize the new point with some average search radius between zero and the maximum connection distance.
- Otherwise, look at the maximum distance of a point claiming P to be his neighbor. If this distance has increased in comparison to the previous frame, increase further.
- If the radius has not increased, increase further until reaching some threshold size (which is chosen relative to the last maximum distance), where we stop increasing and set the search radius back to the last maximum neighbor distance.

While these simple rules perform well in our test scenes, other heuristic rule sets are imaginable and could be easily implementable within this feedback framework.

4 Results

4.1 Image Quality

In order to measure the quality of the rendered images, we observe the quality of both the reconstruction of texture and the reconstruction of shape. Figure 9 compares different image renderings of a advertising pillar point cloud model. We compare point splats with both size and empirically precalculated size [SZW09] and manually adjusted fixed size with our screen-space triangulation method. While the empirical point size ensures splat sizes that produce dense surfaces, it is less suited for high frequency textures due to geometric noise. Adjusting the splat size manually to a minimal surface covering size produces far better results, but still shows some box artifacts. The result of our method is comparable to the manually adjusted image, but even improves the image quality by omitting box artifacts, resulting in a smoother image.

Another example is given by Figure 8, which compares a view from domitilla catacomb rendered once with preprocessed splat sizes and once with screen-space triangulation. Notice that our approach reconstructs the wall surfaces preserving their fine detailed writings, while in the splatted image they are partially not visible any more.

Comparing the quality of the reconstructed scene geometries, we observe major differences to other approaches. Rendering box splats inherently produces staircase artifacts, and too large splat sizes destroy silhouettes of detailed geometry. In contrary, our screen-space triangulation technique is able to reconstruct detailed silhouettes, but is more sensible to noise in the data, as often present in scanned point clouds. Figure 7 shows a scanned chandelier from the inside of a cathedral.

The quality of the normal buffer obtained by our algorithm strongly depends on the quality of the data. Figure 10 shows our resulting normal map on the armadillo model, showing good results when comparing to the preprocessed normal map. However, since we perform a simple neighbor face-normal averaging in screen-space, noisy geometry data results in bad normals.

4.2 Performance

All our performance tests were executed on a platform with a Intel Xeon X5550 2.67GHz CPU with 72 GB RAM, and a GeForce GTX480 GPU with 1,5 GB dedicated video RAM.

Table 1 lists the time consumption in milliseconds of the shader passes of our algorithm in the three different fly-through scenes that are seen in our submission video, for two different viewport sizes. The largest point workload is given in the Domitilla catacomb scene, thus noticeably affecting performance. As one would have guessed, the triangulation pass that invokes the geometry shader to triangulate all our neighbor triangles is identified to be the biggest bottleneck in our whole shader-pipeline. The second most time consuming operation is given by the two neighbor search passes. Certainly, the number of points that are projected to the framebuffer is also a performance criteria, but in our

tests do not top the triangulation and neighbor search passes. In each of our test scenes, we observe that the viewport size is another critical factor for our performance, which is a common issue for all screen-space algorithms.

5 Limitations

The first major drawback of the screen-space nearest neighbor search is that we subdivide the region around a point only in screen-space. Thus we have no depth resolution for storing the nearest world-space neighbor. For viewing angles orthogonal to planar or moderately curved surfaces, this method is of advantage, while points on surfaces orthogonal to the viewing plane suffer a higher probability of producing false triangles. In most cases, this error is smoothed by the blending of the rasterized triangles.

A second major problem of the algorithm is finding a good estimation for the neighbor search radii. Despite using an iterative adaptation mechanism for the search radii, the initial search position of a new point is always a compromise between image quality (use larger initial search radii to reduce visible temporal coherence artifacts) and performance (use smaller initial search radii to save unnecessary search computation time).

6 Conclusion and future work

We have introduced a new method that is able to render interactive high-quality images from unprocessed point clouds given only their position and color information. In addition to the color image output, our method provides a normal map and a depth map which together can be used for subsequent deferred shading passes. In order to display closed surfaces, we use a screen-space triangulation technique that interpolates the color and the estimated normal information between neighboring points. Our method is able to reconstruct high-frequency surface textures, where simple splats would produce much more blurry images or even unidentifiable textures.

Currently, our method works best on scenes with a point density that does not already produce nearly dense images on its own, since such cases would result in mostly pixel-sized triangles. For such situations, a hybrid approach that closes pixel holes on highly dense screen regions by quick filtering while triangulating sparser regions could be an interesting field for future research. Other points for future improvements are a more sophisticated determination of convenient search radii and the investigation of different search radius adaptation mechanisms.

References

[GM04] GOBBETTI E., MARTON F.: Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Comput. Graph. 28*, 6 (2004), 815–826.

[HDD*92] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., STUETZLE W.: Surface reconstruction from unorganized points. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1992), ACM, pp. 71–78.

[MKC07] MARROQUIM R., KRAUS M., CAVALCANTI P. R.: Efficient point-based rendering using image reconstruction. In *Proceedings Symposium on Point-Based Graphics* (2007), pp. 101–108.

[PJW12] PREINER R., JESCHKE S., WIMMER M.: Auto splats: Dynamic point cloud visualization on the gpu, May 2012.

[PZvBG00] PFISTER H., ZWICKER M., VAN BAAR J., GROSS M.: Surfels: surface elements as rendering primitives. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), ACM Press/Addison-Wesley Publishing Co., pp. 335–342.

[SJW07] SCHERZER D., JESCHKE S., WIMMER M.: Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering)* (June 2007), Kautz J., Pattanaik S., (Eds.), Eurographics, Eurographics Association, pp. 45–50.

[SZW09] SCHEIBLAUER C., ZIMMERMANN N., WIMMER M.: Interactive domitilla catacomb exploration, Sept. 2009.

[WS06] WIMMER M., SCHEIBLAUER C.: Instant points, July 2006.

Scene fly-through Viewport size	Mayan Pyramid 1280x600	Mayan Pyramid 800x600	Domitilla Catacomb 1280x600	Domitilla Catacomb 800x600	Armadillo 1280x600	Armadillo 800x600
Reproject Depth Map	0.9	0.7	1.2	0.8	0.7	0.4
Project Points	1.0	1.0	11.8	11.3	0.4	0.5
Neighbor Search Pass 1	9.4	5.6	34.1	27.0	3.1	2.8
Neighbor Search Pass 2	8.3	5.0	23.0	19.8	2.9	2.4
Normal Estimation	1.2	0.8	2.0	1.3	0.6	0.6
Search Radii Feedback	4.3	3.4	6.1	3.6	4.8	3.0
Triangulation	15.9	11.1	27.9	18.9	12.3	7.9

Table 1: Listing of the time consumption of the individual shader passes of our algorithm in ms for the three scenes at different viewport sizes.

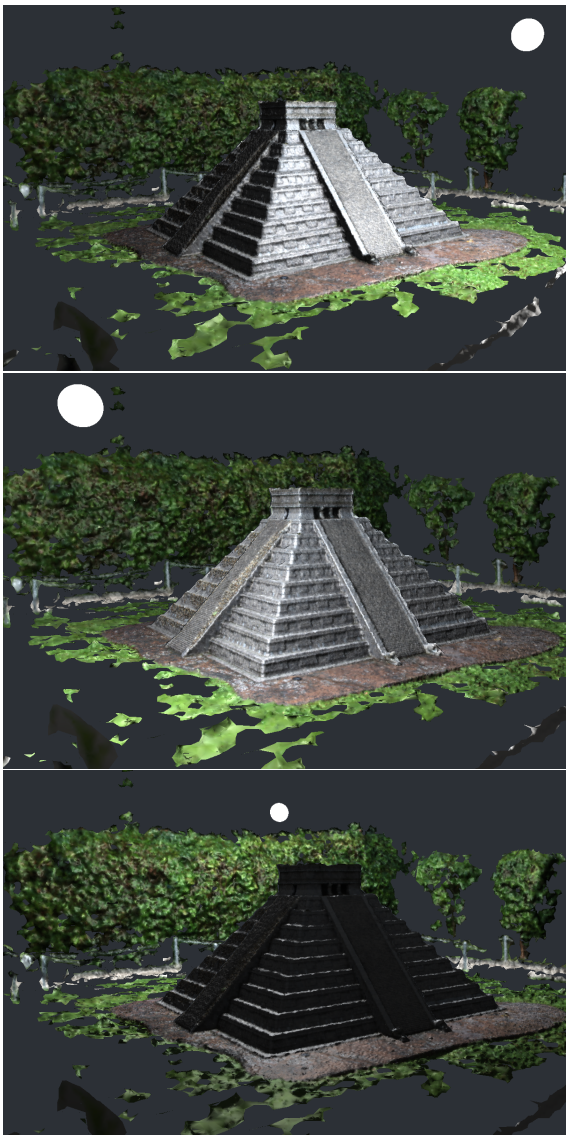


Figure 6: Deferred shaded phong illumination of the pyramid point cloud model under different light positions.

(a) precalculated splat size

(b) screen-space triangulation

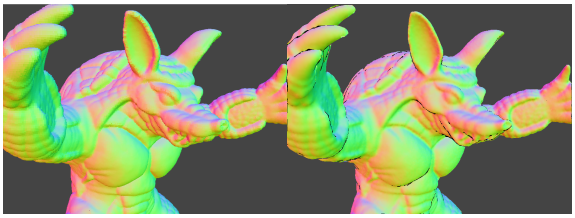
Figure 7: A complex-geometry scan from a chandelier, rendered with with precalculated splat sizes and with screen-space triangulation. Note that the triangulation approach is much better suited to reconstruct even difficult silhouettes in point clouds.

(a) precalculated splat size

(b) screen-space triangulation

Figure 8: Comparison between our approach and a precalculated splat size rendering in the domitilla catacomb. Notice the writing on the wall in the lower right section on the image.

Figure 9: A scanned pillar point cloud rendered with an empirical splat size by [SZW09] (left), a manually chosen optimal fixed plat size (center), and our image reconstruction approach (right).



(a) preprocessed normals rendered as splats

(b) screen-space triangulated normal estimations

Figure 10: Comparison between preprocessed normals in a point splat rendering (left), and the normal map retrieved with by our screen-space triangulation technique(right).