Raw point cloud deferred shading through screen space pyramidal operators

H. Bouchiba[1,2], J-E. Deschaud[1] and F. Goulette[1]

[1] MINES ParisTech, PSL Research University, CAOR - Centre de robotique, Paris, France
[2] Terra3D Research, Paris, France

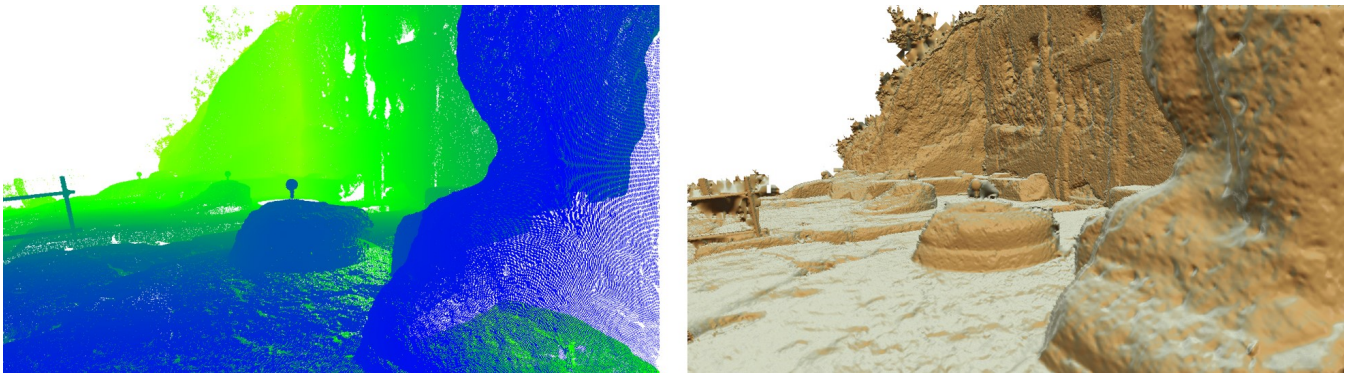

Figure 1: *73.6 million points laser scanned Scalina Etruscan tomb rendered with our algorithm at 120 fps. The surface **(right)** is reconstructed only from a depth buffer **(left)** by efficient pyramidal operators in real-time on the GPU.*

Abstract

We present a novel real-time raw point cloud rendering algorithm based on efficient screen-space pyramidal operators. Our method is based on a pyramidal occlusion-based hidden point removal operator followed by a pyramidal reconstruction by the pull push algorithm. Then a new pyramidal smooth normals estimator enables subsequent deferred shading. We demonstrate on various real-world complex objects and scenes that our method achieves better visual results and is one order of magnitude more efficient comparing to state of the art algorithms.

CCS Concepts

•***Computing methodologies*** → ***Point-based models; Image manipulation;*** *Rasterization;*

1. Introduction

Today's 3D scanners produce massive and detailed 3D point clouds. One way to visualize them is to reconstruct a mesh, which is a computational intensive process. Splatting [BHZK05], i.e. rendering points as ellipses, is another traditional approach to render a filled surface directly from points. This approach still involves heavy precomputing as it needs normals and radii. This is a problem when the dataset is massive or when visualization should be performed on the fly.

We propose a new method to render raw point clouds interactively. The main contribution of the paper is a pyramidal occlusion-based hidden point removal operator, used in conjunction with the pull-push algorithm for rendering point clouds. It is based on a single geometric pass and on screen space operators, it is then independent from the scene complexity. The use of pyramidal operators allows achieving high framerates. The produced high-quality surface and the estimated normals are well suited for subsequent deferred shading.

2. Related work

One approach to render points directly without reconstructing an explicit mesh is to perform surface splatting [BHZK05]. However, surface splatting involves costly computation of normals and radii in preprocessing, which quality is also highly sensitive to real-

DOI: 10.2312/egs.20181036

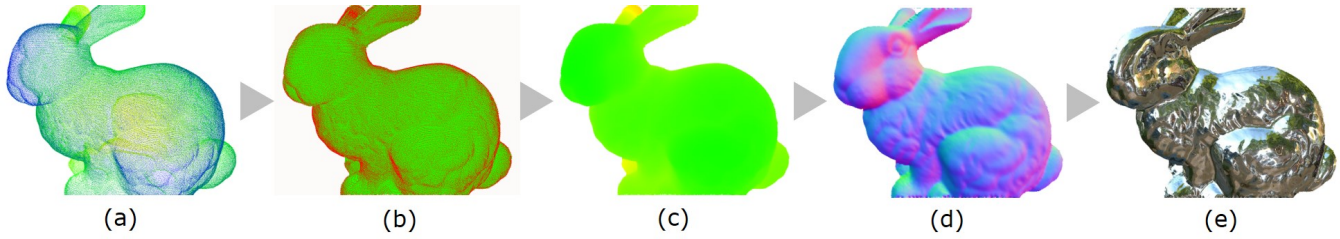

Figure 2: *Overview of our method pipeline.* ***(a)*** *Geometry rendering with one pixel per point.* ***(b)*** *Hidden Point Removal (HPR), in red: foreground visible points, in green: hidden points.* ***(c)*** *Depth and attributes reconstruction [Kra09].* ***(d)*** *Normals estimation.* ***(e)*** *Shading.*

world point cloud artifacts: noise, holes, misalignment and outliers. In addition, surface splatting fails to reconstruct complex structures and produces artifacts on regions with high curvature.

Another way to render point clouds efficiently without preprocessing is to increase the size of the points. [SW15] shows that this simple method can achieve good quality results by drawing, for each point, camera aligned paraboloids instead of flat squares. This method however fails to generate a smooth filled surface, it is then only suited for image-based rendering of colored point clouds.

Finally, the most efficient approach for point cloud rendering is to use screen space operators. [MKC08] use the pull-push algorithm to reconstruct a filled color and depth buffer. However, they rely on precomputed normals to perform both reconstruction and shading. [PJW12] use screen space nearest neighbor queries to compute normals and radii to feed [BHZK05] splatting algorithm in real-time. Their algorithm is however very computational intensive. [PGA11] use an interesting screen-space visibility operator to remove the non-visible parts of the scene. The space between points is then filled by an iterated median filter. Such iterative filling algorithms, also as [RL08], are limited and cannot achieve high quality surfaces. In addition, the above cited methods do not handle scenes with high depth differences as they are based on fixed neighborhood sizes in image space.

3. Proposed method

Our method uses as input a framebuffer with depth and an optional color after a single geometric pass that renders one pixel per point. The method is then based on three screen space pyramidal operators. The first one is a Hidden Point Removal (HPR) operator that suppresses the points seen through the model due to the discrete sampling of the points. It also labels the background and the foreground of the final image. Then we use the pull push algorithm [Kra09] to reconstruct a depth and color filled framebuffer. The depth texture pyramid built in the push phase is then reused to estimate filtered normals in the final framebuffer. This later can then be used to perform deferred shading on the reconstructed surface. This pipeline is depicted in Figure 2.

The only parameter of our method is an approximate metric scale s_0 provided by the user that should be on the order of magnitude of the point sampling.

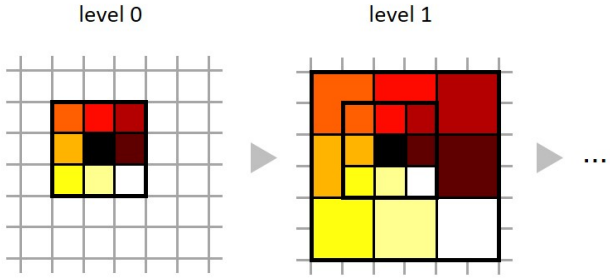

Figure 3: *Pyramidal neighborhood pattern around a central pixel (in black) used to compute the visibility of each point.*

3.1. Pyramidal and adaptive hidden point removal

A new HPR operator is used to remove the supposed hidden parts of the point cloud seen by transparency over it. This operator is applied to the whole image (even on the background pixels). As [PGA11] we use an occlusion-based operator. But unlike [PGA11], our operator is adaptive with depth ; thus, it handles better scenes with high depth differences. It is also one order of magnitude more efficient as it is computed thanks to an image pyramid.

3.1.1. Point cloud pyramid construction

The first step consists in building an image space point cloud hierarchy from the input depth buffer. The depth buffer is first unprojected from the screen projective space to the camera Cartesian frame and stored to a (x,y,z) float texture. This texture is the level 0 of the pyramid and then the other levels are built recursively. For the level l each pixel (i^l, j^l) is obtained by keeping the point with the minimum screen depth in the $\{2i^l, 2i^l+1\} \times \{2j^l, 2j^l+1\}$ neighborhood in the $l-1$ level texture. This pyramid holds then a fine to coarse subsampled representation of the input depth buffer. It can be interpreted as an efficient way to gather neighboring pixel information.

3.1.2. Adaptive occlusion operator

The second step is an approximate occlusion computation for each point of the input framebuffer. The operator is applied to the whole image. Background pixels depth is set to the camera far clip plane. Given an input point x, its neighborhood is computed thanks to the

Figure 4: *Closeup on HPR results. In red points removed by the HPR:* ***(left)*** *fixed neighborhood, using [PGA11]* ***(right)*** *adaptive neighborhood, using our method.*

pyramid, with the pattern described in Figure 3. Each point y in its neighborhood is associated to one of the 8 sectors, represented by a different color in Figure 3. For each sector we keep the neighbor with the minimum occlusion value : $\left(1 - \frac{y-x}{\|y-x\|} \cdot \frac{-y}{\|y\|}\right)$. The mean of the 8 occlusion values is then compared to a threshold, typically 0.1 (same value for all datasets). The pixels below this value are labeled as holes and the others as visible (see Figure 2b).

Compared to exhaustive neighborhood [PGA11], using pyramidal neighborhood improves the complexity of the algorithm by decreasing the total number of visited neighbors. Indeed, given a pixel radius r_0 the extensive neighborhood visits $\sim r_0^2$ pixels whereas the pyramidal one visits $\sim \log r_0$.

To handle scenes with high depth differences, we use an adaptive neighborhood size. To do so, we use one level of the pyramid as coarse depth map. Experiments shows that the 4th level of the pyramid gives good results. This depth map is then used to estimate the level of the neighborhood thanks to the following formula:

$$l(z_i) = \log\left(\frac{s_{hpr} h}{2\tan(\theta)} \frac{1}{z_i}\right) / \log 2 \qquad (1)$$

Where h is the viewport height θ the half of the vertical field of view, z_i the coarse depth value for a given pixel, and s_{hpr} a metric size of the HPR projected neighborhood. Experiments shows that $s_{hpr} = 10 s_0$ is a good value for a wide range of datasets. A comparison between adaptive and fixed size neighborhood is presented in Figure 4. The adaptive operator does not overestimate the hidden points on the scene, particularly under the bridge in Figure 4.

3.2. Pyramidal reconstruction by pull push algorithm

Given the visibility mask obtained after the previous phase, we filter out hidden points from the input framebuffer. They are then considered as background points and their initial weight is set to 0. We then reconstruct a filled framebuffer thanks to the weighted pull push algorithm in 2D, introduced by [GGSC96] and improved by [Kra09]. We then recover the background in the filled framebuffer by taking visible background pixels obtained by the HPR operator.

3.3. Pyramidal normals estimation

The pull push algorithm applied to depth reconstruction yields a noisy surface and thus noisy normals if estimated from the finest

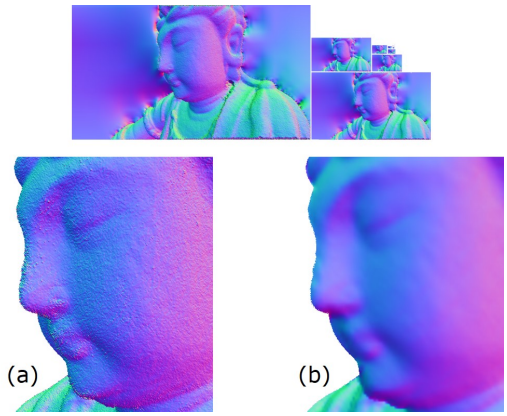

Figure 5: *Pyramidal normals estimation. Closeup on: (a) noisy normals estimated from the level 0, (b) smoothed normals.*

Dataset	geometry pass	[PGA11] HPR	our HPR
Scalina	1.4 ms	58 ms (25)	**2.7 ms**
Water-moon	2.0 ms	21 ms (15)	**2.1 ms**
Ajaccio	11.4 ms	21.6 ms (15)	**2.3 ms**

Table 1: *Per frame processing time comparison between, [PGA11] (in parentheses the equivalent radius size in pixels), and our method.*

reconstructed depth. In order to estimate smooth normals, we introduced a new adaptive and pyramidal operator. First we compute normals on each depth texture obtained in the push phase of the previous algorithm in order to obtain a multiresolution normal map. Normals are computed by simple cross product between neighboring pixels. Then given a pixel in the target normal map, we first compute a radius from its depth. We perform a linear interpolation between the normals at the two nearest levels in the pyramid. Normals inside each level are obtained by bilinear interpolation. The results are illustrated in Figure 5.

4. Results

We implemented our method in C++ with OpenGL. All the images have been rendered at 1248 x 768 on a 4.2 Ghz Intel Core i7 with an Nvidia GTX 1080. We show in Figure 1 and Figure 6 that our method achieves good visual results on a wide range of point clouds types from photogrammetry (b) to noisy mobile mapping (c).

Table 1 shows that our new HPR operator, which is usually the bottleneck of the pipeline, is ten times more efficient than state of the art. The processing time is constant over various datasets and is similar to the geometric pass.

5. Conclusion

We introduced a new pipeline for efficient raw point cloud rendering. This pipeline achieves visual results that surpasses state of the art, thanks to the combination of depth reconstruction by pull push and to a new pyramidal normal estimator. Our method is independent from the scene complexity by the use of a single geometry

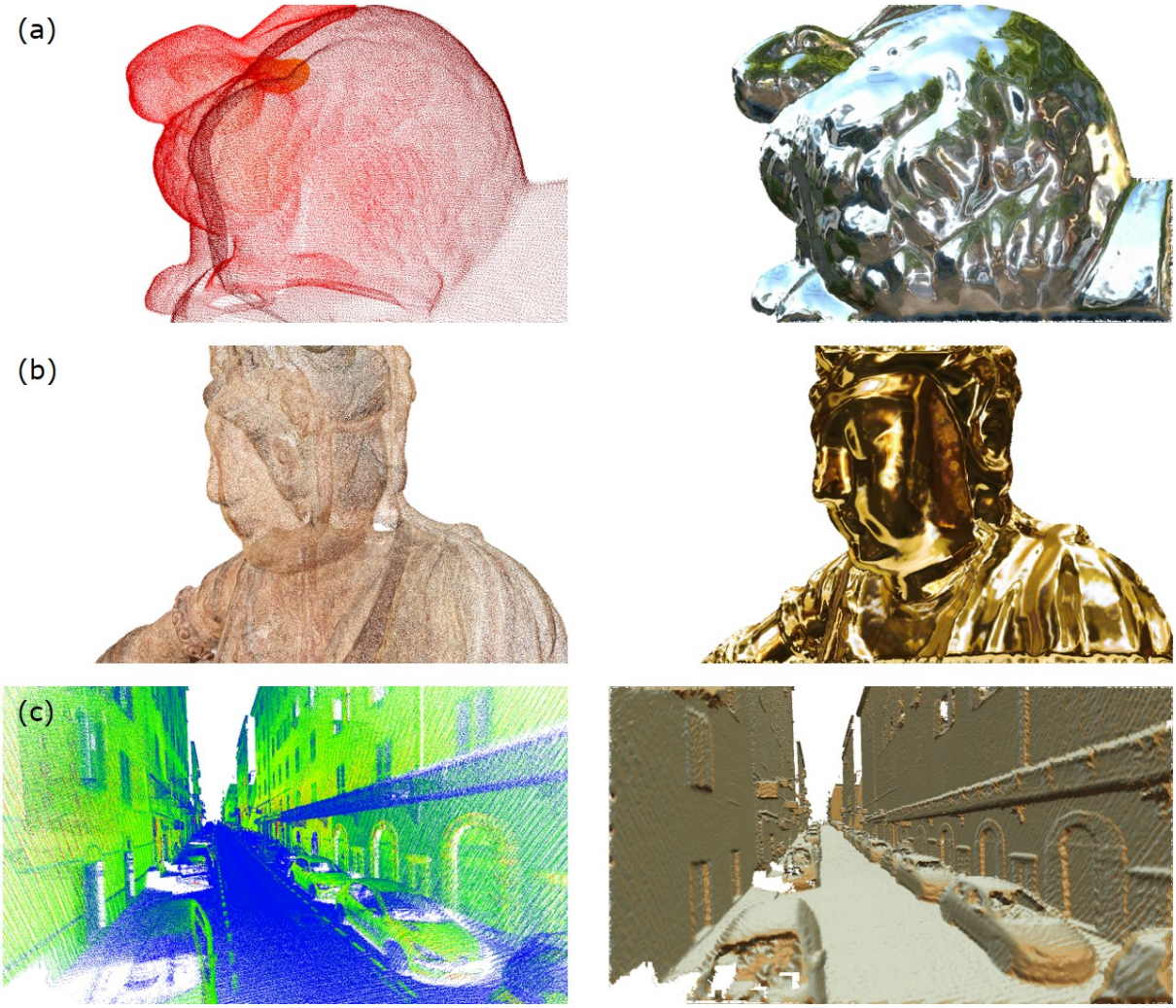

Figure 6: ***(left)*** *raw point cloud views.* ***(right)*** *shaded reconstructed model with our algorithm.* ***(a)*** *Stanford Bunny dataset 350k points.* ***(b)*** *Water-moon Asian statue 6 million points.* ***(c)*** *Ajaccio city 2.7 billion points, a noisy mobile mapping scene. This dataset is rendered with an algorithm similar to [WS06] points that feeds the GPU with approximately 30 million points per frame.*

pass and screen space operators. It finally achieves low computation times enabling high framerate demanding applications, such as virtual reality.

References

[BHZK05] BOTSCH M., HORNUNG A., ZWICKER M., KOBBELT L.: High-quality surface splatting on today's GPUs. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.* (June 2005), pp. 17–141. 1, 2

[GGSC96] GORTLER S. J., GRZESZCZUK R., SZELISKI R., COHEN M. F.: The Lumigraph. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 43–54. 3

[Kra09] KRAUS M.: The pull-push algorithm revisited. *Proceedings GRAPP 2009* (2009). 2, 3

[MKC08] MARROQUIM R., KRAUS M., CAVALCANTI P. R.: Efficient image reconstruction for point-based and line-based rendering. *Computers & Graphics 32*, 2 (Apr. 2008), 189–203. 2

[PGA11] PINTUS R., GOBBETTI E., AGUS M.: Real-time Rendering of Massive Unstructured Raw Point Clouds Using Screen-space Operators. VAST'11, Eurographics Association, pp. 105–112. 2, 3

[PJW12] PREINER R., JESCHKE S., WIMMER M.: Auto Splats: Dynamic Point Cloud Visualization on the GPU. In *EGPGV* (2012), pp. 139–148. 2

[RL08] ROSENTHAL P., LINSEN L.: Image-space point cloud rendering. In *Proceedings of Computer Graphics International* (2008), pp. 136–143. 2

[SW15] SCHÃIJTZ M., WIMMER M.: High-quality point-based rendering using fast single-pass interpolation. In *2015 Digital Heritage* (Sept. 2015), vol. 1, pp. 369–372. 2

[WS06] WIMMER M., SCHEIBLAUER C.: Instant Points: Fast Rendering of Unprocessed Point Clouds. In *SPBG* (2006), Citeseer, pp. 129–136. 4