

# High-Quality Point-Based Rendering Using Fast Single-Pass Interpolation

Markus Schütz  
Institute of Computer Graphics  
Vienna University of Technology  
Vienna / Austria  
mschuetz@potree.org

Michael Wimmer  
Institute of Computer Graphics  
Vienna University of Technology  
Vienna / Austria  
wimmer@cg.tuwien.ac.at

**Abstract**—We present a method to improve the visual quality of point cloud renderings through a nearest-neighbor-like interpolation of points. This allows applications to render points at larger sizes in order to reduce holes, without reducing the readability of fine details due to occluding points. The implementation requires only few modifications to existing shaders, making it eligible to be integrated in software applications without major design changes.

**Index Terms**—Computer graphics, point clouds, WebGL

## I. INTRODUCTION

3D scanning methods such as laser scanning or photogrammetry produce enormous amounts of point cloud data. Unlike polygon meshes, point clouds do not contain connectivity between points, and surface normals are not always available. Due to the missing connectivity and normals, points are often rendered using screen-aligned squares or circles. If the size of these primitives is too small, holes appear, and if the size is too large, points will occlude each other and reduce the visibility of high-frequency features such as text.

This paper presents a method that allows using larger point sizes in order to avoid holes and at the same time, solve undesirable occlusions by performing a nearest-neighbor-like interpolation of points. The interpolation is achieved by rendering points as 3d shapes through manipulation of fragment depths. Additional passes are not required.

Our method can be seen as a trade-off between the performance of the commonly used screen-aligned square and circle primitives, and the high quality of multi-pass splatting algorithms.

## II. RELATED WORK

Related works include high-quality point-based rendering as well as fast Voronoi diagram generation methods.

### A. High-Quality Splatting

Previous high-quality splatting methods for the GPU [1] require three rendering passes. First, a visibility pass builds a depth map with a small offset. The blending or attribute pass then builds a weighted sum of all attributes that pass the depth test. The last pass normalizes attribute values by dividing the weighted sum of attributes by the sum of weights. These methods also render points as oriented disks or ellipses.

The results have a very high quality. The need for three rendering passes, however, significantly reduces performance, and rendering oriented disks requires normals, which are not always available.

Deferred Blending [2] is a GPU-accelerated method that is able to render opaque point clouds in a single geometry pass and an additional compositing pass. This method is also able to render simple transparency effects in a two-pass approach and higher-quality transparencies in 3 passes.

These methods have in common that they require multiple rendering passes. They achieve high-quality results with smoothly blended points at a high performance cost.

### B. Voronoi Diagram Generation

The results of our method closely resemble Voronoi diagrams. In fact, the idea of rendering points as 3D shapes has already been used in previous works for fast generation of Voronoi diagrams. Kenneth et al. [3] create two-dimensional Voronoi diagrams by rendering points as cones and lines as tents with a cone at each corner.

Jump flooding [4] uses a flooding algorithm to generate Voronoi diagrams by repeatedly spreading pixels in a texture until the whole texture is filled. Instead of distributing pixels to their closest empty neighbors in each step, they are propagated over larger distances, thus reducing the number of necessary repetitions.

## III. INTERPOLATION SHADER

This section covers the theory as well as implementation details of our interpolation shader.

The idea behind our method is similar to creating Voronoi diagrams by rendering points as cone meshes. [3]. However, instead of using meshes, points are rendered as view-aligned quads, as commonly used in point cloud renderers. The three-dimensional shape is achieved by adding an additional offset to the fragment depth values. This offset depends on the distance to the center of the quad and the type of weight function.

Figure 1 shows shapes produced by different weight functions. Weights are calculated for each fragment and subsequently used as an offset to the depth value. For spheres, the weight function is not defined for fragments outside the sphere's boundaries. These fragments are therefore discarded,

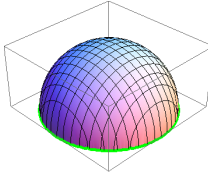
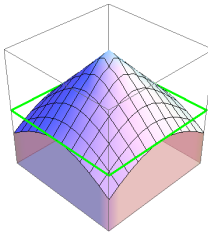
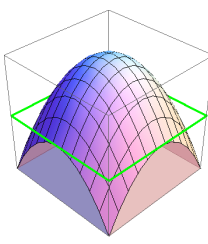
	Weight function	Shape
sphere	$\sqrt{1 - (u^2 + v^2)}$	
cone	$1 - \sqrt{(u^2 + v^2)}$	
paraboloid	$1 - (u^2 + v^2)$	

Fig. 1. Shapes produced by different weight functions.  $u, v \in [-1, 1]$

resulting in circular shapes on screen. The cone and paraboloid weight functions are well defined for all fragments and can therefore be used for squares as well.

Point clouds are usually rendered with view-aligned rather than camera-facing quads. Applying weights as depth offsets leads to rendering distorted shapes because of the perspective projection of view-aligned quads. Figure 2 shows how using a paraboloid weight function results in rendering distorted paraboloids. In practice, this has shown to work fine and to significantly increase quality at a low cost despite the distortion. Figure 3 shows the same points rendered as simple view-aligned squares and paraboloids. In the latter case, the point closest to the camera is less likely to occlude all points behind it.

All of the listed weight functions reduce occlusion problems. There is a subtle difference, though, and we decided to chose the paraboloid function for some of its properties. First of all, it is the simplest one to calculate. It is also defined for all fragments, unlike the spherical function, and can therefore be used with square-shaped point primitives as well. But most importantly, the intersections between paraboloids at different distances remain straight, whereas the intersections of cones and spheres appear rounded.

All of the weight functions assume that the radius of the point is 1 and the generated weights range from -1 to 1. The final depth offset is obtained by multiplying the weight by the world-space point radius. If the world-space radius is not known, it can be approximated by taking the pixel size and inverting the projection, as described in Section III-A.

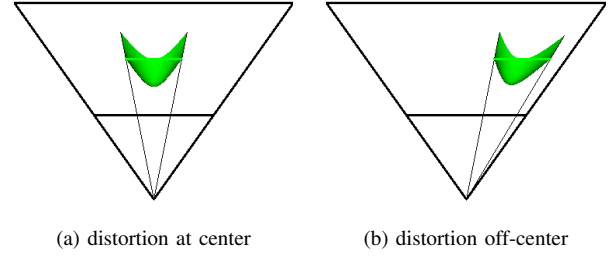


Fig. 2. The resulting shapes are distorted due to perspective projection.

### A. Implementation

The implementation requires field of view in radians and screen height in pixels as additional uniform inputs to the vertex shader and the projection matrix as additional input to the fragment shader. Field of view and screen height are used to approximate the world-space point radius from the pixel size of the point primitive. The projection matrix is used to recalculate the projected depth after modifying the view-space depth value.

Even if the world-space point radius is known, it may still be necessary to approximate it from the pixel size instead. For example, since this method is sensible to overdraw, our implementation limits the point size to a maximum of 50 pixels. The pixel size is therefore no longer guaranteed to be the screen projection of the radius.

For the approximation, the projection factor from a world-space radius to screen-space pixel size is calculated. The pixel size is then divided by the projection factor to obtain an approximation of the world-space radius. This also allows integrating the algorithm into systems that use fixed or camera-dependent point sizes with no assumptions of point radii.

```
float projFactor = 1.0 / tan(fov / 2.0);
projFactor = projFactor / -vViewPos.z;
projFactor = projFactor * screenHeight / 2.0;
...
vRadius = gl_PointSize / projFactor;
```

The fragment shader provides coordinates that indicate the fragment position inside the point primitive. To calculate the weight, these coordinates have to be transformed from an interval of [0,1] to an interval of [-1,1]. The following code sample uses the paraboloid weight function.

```
float u = 2.0 * gl_PointCoord.x - 1.0;
float v = 2.0 * gl_PointCoord.y - 1.0;
float w = 1.0 - ( u*u + v*v );
```

The weight is multiplied by the radius of the point and added to the screen-space depth value. The resulting position is then projected and its projected depth value is used as the new fragment depth.

```
vec4 pos = vec4(vViewPos, 1.0);
pos.z += w * vRadius;
pos = projectionMatrix * pos;
pos = pos / pos.w;
gl_FragDepthEXT = (pos.z + 1.0) / 2.0;
```

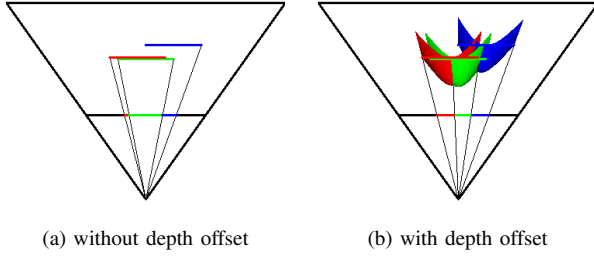


Fig. 3. Top view showing (a) points occluding other points behind them and (b) using a fragment depth offset to reduce undesirable occlusions.

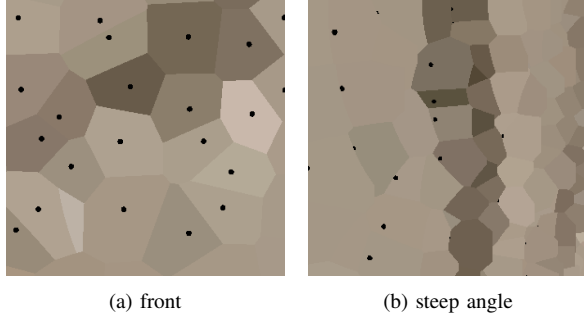


Fig. 4. Point centers are indicated by black dots. (a) Front view showing similarities of the results to a Voronoi diagram. (b) Similarities to Voronoi diagrams decrease at steep angles.

#### IV. RESULTS AND LIMITATIONS

In this section, we show images of our results and comparisons to screen-aligned squares and circles. We also compare results to a three-pass high-quality splatting method using screen-aligned circles because our datasets do not contain the normals necessary for rendering oriented splats.

The results of this method, as seen in Figure 4, show strong similarities to a Voronoi diagram.

Figure 5 and 7 show images generated by the different rendering methods. Squares and circles both suffer from occlusions. Camera rotations also cause flickering as points change their order and occluding points suddenly become occluded points. The interpolation and high-quality splat-rendering modes do not suffer from this problem.

Due to its nearest-neighbor-like behavior, our method is as susceptible to noise as squares and circles. The high-quality splatting methods, on the other hand, blend multiple points together and therefore reduce the impact of noise, as shown in Figure 6.

#### V. PERFORMANCE

All performance tests were done on a notebook with an Intel Core i7-4712MQ and a NVIDIA GTX 860M. We used WebGL and the Chrome web browser to render into a 1920x955 pixel canvas element.

Figure 8 shows frames per second (FPS) for different modes and point sizes. The size parameter is a multiplier. A value of 0 results in a size of 1 pixel. With a value of 1, pixel

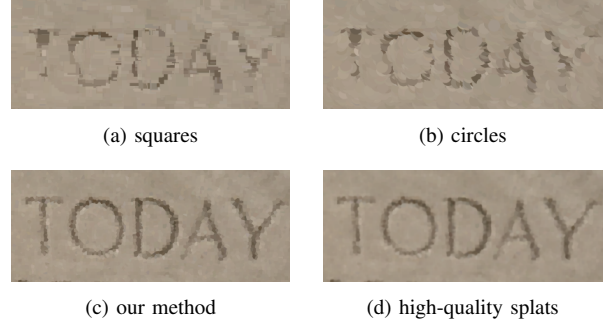


Fig. 5. Squares (a) and circles (b) suffer from occlusions. Our method (c) and high-quality splats (d) improve readability of high-frequency details such as text.

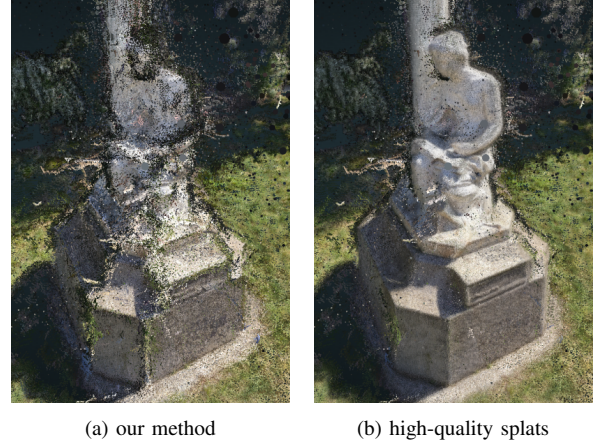


Fig. 6. A limitation of our approach: It does not improve noisy datasets. High-quality splats are better suited in such cases.

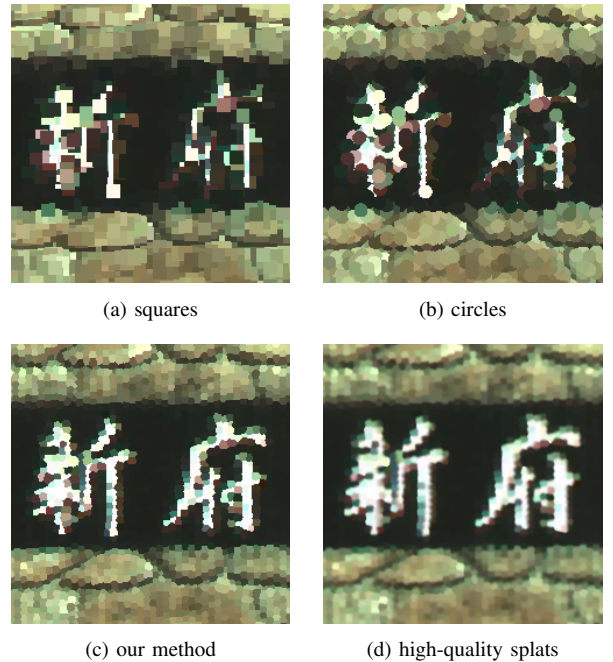


Fig. 7. Improved readability of text with our method, comparable to high-quality splats.

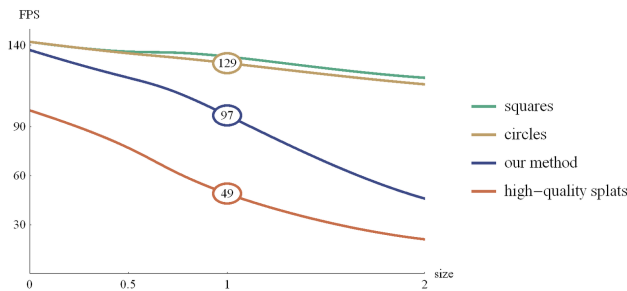


Fig. 8. Performance of squares, circles, interpolation and high-quality splats (in top-to-bottom order) in frames per second (FPS). A size factor of 1 covers holes while minimizing overdraw. Lower values cause holes while larger values increase overdraw.

size is chosen in a way to close holes but minimize overdraw. Lower values lead to holes and larger values cause increasingly higher overdraw. Too much overdraw is problematic since interpolation and high-quality splatting depend on features that do not allow for early depth testing.

## VI. CONCLUSION AND FUTURE WORK

We have presented a single-pass method that significantly increases quality at a lower impact on performance than previous high-quality methods that require two or even three rendering passes. Implementation is simple and requires adding a few lines of code, as described in Section III-A, to existing shaders.

It is especially useful for close-up views of datasets with sharp features such as text or edges.

This method can be seen as a trade-off between the performance of screen-aligned squares and circles, and the high quality of multi-pass splatting algorithms.

This method was developed for the WebGL point cloud renderer Potree [5] with the help of the three.js library. [6]. A reference implementation of this method is available in the Potree github repository.

Manipulating the fragment depth can disable some GPU optimizations such as early depth testing. Other possible approaches to render points as paraboloids are geometry shaders and instancing. We did not explore these options since WebGL does not support geometry shaders at this time and instancing is not supported by three.js.

## VII. ACKNOWLEDGEMENTS

This research was supported by the EU FP7 project HARVEST4D (no. 323567) [7]. The statue is part of the Arene de Lutece dataset, courtesy of HARVEST4D. The point cloud depicting the Japanese sign is courtesy of Anan Survey [8].

## REFERENCES

- [1] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt, “High-quality surface splatting on today’s gpu,” in *Proceedings of the Second Eurographics / IEEE VGTC Conference on Point-Based Graphics*, ser. SPBG’05. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2005, pp. 17–24. [Online]. Available: <http://dx.doi.org/10.2312/SPBG/SPBG05/017-024>
- [2] Y. Zhang and R. Pajarola, “Deferred blending: Image composition for single-pass point rendering,” *Comput. Graph.*, vol. 31, no. 2, pp. 175–189, Apr. 2007. [Online]. Available: <http://dx.doi.org/10.1016/j.cag.2006.11.012>
- [3] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver, “Fast computation of generalized voronoi diagrams using graphics hardware,” in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286. [Online]. Available: <http://dx.doi.org/10.1145/311535.311567>
- [4] G. Rong and T.-S. Tan, “Jump flooding in gpu with applications to voronoi diagram and distance transform,” in *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, ser. I3D ’06. New York, NY, USA: ACM, 2006, pp. 109–116. [Online]. Available: <http://doi.acm.org/10.1145/1111411.1111431>
- [5] M. Schütz, “Potree,” <http://potree.org>, accessed: 2015-07-02.
- [6] R. Cabello, “three.js,” <http://threejs.org/>, accessed: 2015-07-02.
- [7] “Harvest4d,” <https://harvest4d.org/>, accessed: 2015-04-16.
- [8] “Anan survey,” <http://anan.skr.jp/>, accessed: 2015-02-14.