Evaluating the visibility algorithm of point-based graphics for real-time applications

Rodrigo Braga
Alexandre Valdetaro
Gustavo Nunes
Alberto Raposo
Bruno Feijó
Pontifical Catholic University of Rio de Janeiro (PUC-Rio) - Department of Informatics

Abstract

In this paper we present a detailed study and some improvements of the technique "Direct Visibility of Point Sets" [Katz et al. 2007], which proposes an operator that can distinguish visible points from occluded points in a point cloud. The operator is not originally intended for real-time usage due to its computational complexity, so we present an improvement that maintains the visual quality and allows it to be used in real-time. We also expose a problem that is not evident in the original work, the operator requires the computation of a convex hull with a set of points. This convex hull is calculated with QuickHull algorithm [Barber et al. 1996], but the point set generated into the operator causes a worst case entry for QuickHull, making it slower than expected. We also provide some options for speeding the convex hull algorithm that trade off visual quality for performance.

Keywords:: Point-based graphics, Visibility, Point Cloud, Point Rendering, Real-time, QuickHull

Author's Contact:

{rodrigo,alexandre,gustavo}@xtunt.com
{bfeijo,abraposo}@inf.puc-rio.br

1 Introduction

Although point-based rendering has been receiveing increasing attention since the last decade, treating points as primitives is still a challenge. Despite the recent research, point based graphics is still non mainstream, and is current applied only to specific niches. Nevertheless, it may be a very promising area. Two factors can be considered as a motivatvation for the current usage of point primitives, procedural shape definition and automated shape acquisition. The inherent scale and complexity of these shapes can make great use of a simple representation such as points. However, proper rendering of points requires at least a proper acquisition technique and an understanding of a correct rendering process.

Also, if one looks more carefully into the possiblities for the future of mainstream rendering pipeline, a comparison between the current polygon based graphics and point based graphics may show some clear advantages to the latter. The polygonal complexity of triangle based models increases with each passing year. Moreover, the processing of these models is getting more and more complex, and the granularity of the triangles is so great that it may even become unnecessary to fill a polygon as it is already in a sub-pixel size. In such scenario a point based rendering can be used as a similar representation, however with a set of clear advantages: Unlimited level of detail, smoother animation, no self-collision and discontinuities among others.

The point based graphics area can be coarsely subdivided in two main areas, surface reconstruction and direct point rendering. Most of the techniques in current point-based graphics area are dedicated to surface reconstruction, where the actual point cloud is used only in a pre-processing stage to build a polygon-based mesh. Then the generated mesh can be rendered via regular polygon pipeline.

The direct point rendering area is where the point cloud is rendered as point primitives. It is still a rather unexplored area, due to inherent problems of using points as primitives. One of these problems is self occlusion. As a point can never be an occluder, and there is no counter-clockwise/clockwise culling for it, the occluded parts of a point model are always visible. The technique "Direct Visibility of Point Sets" [Katz et al. 2007] proposes an operator, that can be applied to a point set, based on a view, to determine which points should be visible and which should be occluded.

In this paper we present a careful analysis of Katz's operator usage. We propose a simple method of using the operator in real-time without compromising visual quality. Also, we expose some inherent problems of the operator that makes it fall into worst case scenarios of convex hull algorithms, which are necessary for the selection of visible points.

2 Related Work

Determination of the geometry visibility and occlusion is often an overlooked problem in general computer graphics, as both z-buffer for polygonal models and ray tracing rendering pipelines provides simple ways to cull the occluded geometry. However, when dealing with point primitives, there is no such straightforward approach for detection of visible points based on a point of view.

During ray tracing, since points are discrete primitives and rays have no volume, there is no simple collision. [Schaufler and Jensen 2000] give a "volume" to rays, [Rusinkiewicz and Levoy 2000], [Dutré et al. 2000], [Zwicker et al. 2001], [Wu and Kobbelt 2004], [Guennebaud et al. 2004] treats the points as an "area" with the normals pointing to the observer. So there may be collision and visibility can be determined. However, ray tracing approaches generally require some acceleration structures such as patial hierarchies [Gross and Pfister 2007].

If dealing with a unstructured raw point set, z-buffer algorithms can be a better option than ray tracing. Splatting is a forward-projection approach that determines visibility with the z-buffer, [Sainz and Pajarola 2004] and [Dachsbacher et al. 2003]. Although efficient and simpler, z-buffer splatting can suffer in the visual quality if compared to ray tracing solutions.

Also, the normal of a point can determine if it belongs to a self-occluded part of a model. However, a simple normal estimation can be complicated, [Hoppe et al. 1992] introduced an approach of finding the k-nearest neighbours of a point and taking the normal of the total least squares best-fitting plane of the neighbourhood as the normal at the point. [Mitra et al. 2004] approach was to take only points in a certain adaptative radius around a point.

These cited approaches all determine the visibility during rendering. [Katz et al. 2007] proposes an operator HPR (Hidden Point Removal) to solve visibility regardless of rendering. Also, there is no surface continuity and sampling regularity requirements, and no requisition for points to be associated with normals. The HPR, however, requires a computation of a convex hull for the whole point set at every frame, limiting its real-time applicability.

This paper proposes a method to enable HPR operator to be used to calculate visibility in real-time. There is also a section that exposes a problem with the Quickhull algorithm when used to calculate the convex hull of the point-set output by the HPR.

3 The HPR Operator

The HPR operator is composed by a viewpoint (camera position) *C* and a set of points *P*. Each point p_i is considered a sampling of a surface *S*. The operator is responsible to determine which points are visible from *C*. The identification of these visible points is made through two steps, discussed below:

3.1 Inversion

There are various ways to perform the inversion. In the original paper, the HPR author focus on spherical flipping. To realize the inversion by spherical flipping, it's necessary to define a D-dimensional sphere that contains a radius *R* and a center *C*. The value of *C* is easy to define because it's the camera position. However, to automatically define the radius *R* is not immediate, because it's necessary to evaluate the density and ϵ-visible, described in [Katz et al. 2007]. To simplify this definition, the alternative solution is define the radius, manually. In other words, set a value for radius that includes all points.

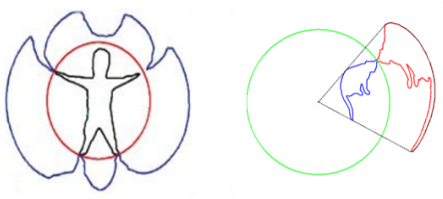


Figure 1: *Spherical flipping*

Intuitively, spherical flipping reflects every point p_i internal to the sphere along the ray from *C* to p_i to its image outside the sphere by applying the following equation:

$$\hat{p_i} = f(p_i) = p_i + 2(R - \|p_i\|)\frac{p_i}{\|p_i\|}$$

3.2 Convex Hull

The convex hull of a set Q of points is the smallest convex polygon P for which each point Q is either on the boundary of P or in its interiors. The Figure 2 illustrates the concept.

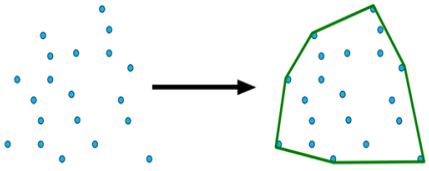


Figure 2: *Illustration of Convex Hull in 2D*

To construct the convex hull, it is important to consider all reflected points of *P*: $\hat{P} = \{\hat{p_i} = f(p_i) | p_i \in P\}$ and the camera position *C*. The inclusion of *C* is important since points on the back side of the object may otherwise lie on the convex hull, when *C* is external to *P*.

After the execution of convex hull algorithm, a new set of elements is created. This set is composed by all reflected points of $\hat{P}$ that reside on the convex hull and by the camera position. Each point, excepted for camera position, are considered visible from *C*.

In Figure 3, the black line represents the convex hull and the red line represents the visible points. For this example, only the dorse and tail of cat contains visible points from *C*. For the complete proof please refer to [Katz et al. 2007].

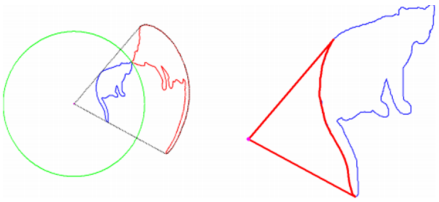


Figure 3: *Illustration of visible points*

4 QuickHull

HPR author is not concerned with real-time application of the HPR, so there is no detailed performance analysis. In the original work, the QuickHull [Barber et al. 1996] algorithm is used to calculate the convex hull of the output points from HPR. The algorithm is an efficient divide and conquer strategy that recursively subdivides the space in tetrahedrons eliminating points that do not belong to the convex hull. For any general case convex hull the Quickhull can be an appropriate choice, whereas the expected complexity for an average case is O(n.logn). However, if the recursive splits are not well balanced, the time complexity can easily get to a prohibitive $O(n^2)[Mucke2009]$.

The HPR output points unfortunately do not compose a well balanced space for an input to the Quickhull algorithm. The points are concentrated on a thin outer layer of a semi-sphere, and the semi-sphere comprises a very small portion of the whole sphere. In such arrangement, every split executed by the Quickhull can only eliminate few points. As shown at Figure 4, the points after the HPR operator are almost at the same plane, through the steps of the Quickhull algorithm a thin layer is selected and only few points are discarded as not being at the convex hull of the flipped point cloud. In that way, many steps of Quickhull algorithm has to be made in order to resolve the final convex hull.

By producing a small perturbation in the points positions, a random addition of +-0.01 of sphere radius to each coordinate, we were able to enhance the calculating speed of the convex hull by an order of magnitude. See the table 1 for actual values.

Stanford Models	Without Perturbation	With Perturbation
Bunny	4.5	0.9
Dragon	150	70

Table 1: *Execution time of QuickHull (in seconds)*

The PC setup that held the tests was an Intel Core i7 920 2.66Ghz with 6GB RAM and a NVIDIA Geforce 480GTX video card.

5 Multithreading

Even though the visible points must be rendered at every frame, they may not need to be calculated at every frame. In our implementation, we decided to maintain two separate buffers for the points: the points backbuffer and the points frontbuffer. The work of these buffers is analogous to the Graphics APIs buffers. The frontbuffer is the buffer that contains the last calculated with the HPR visible points and these are the points passed to the GPU for rendering. The backbuffer is constantly used by an auxiliary thread that keeps calculating the HPR and Convex Hull for all the points. Whenever the auxiliary thread is done calculating, it raises a flag. This flag is watched by the main thread and means that the backbuffer needs to be copied to the frontbuffer, so the to-be-rendered points are updated. We used this multi-threaded approach with a model of a size that the auxiliary thread can calculate HPR and convex hull every 30 frames in a 60 fps ratio. There was no visible difference even with a camera speed of half the model bounding sphere's radius per second. Although it can't still be considered an efficient approach for real-time direct point visibility calculation, with our approach it can already be used for models that are not massive.

6 The Algorithm

Before using the HPR operator and its optimizations, it's necessary to load a point cloud that represents a specific 3D model. After loading this model, ie., after obtaining at least, the information about of each point position, it's possible to use the HPR operator. The implementation of this operator is very simple, composed of the next three steps:

1. Perform the inversion of model's points, ie, apply the inversion described in Section 3. To do this, it is necessary to define a value to radius *R* as it is required for spherical flipping implementation. For this paper, the value of *R* will be set manually based on these facts: high values of *R* are utilized for a dense point cloud and low values of *R* are utilized for sparse point cloud. It's important to highlight that for high values of *R*, some points that should be occluded pass in the threshold of convexhull and eventually become visible.

2. Applies the Quickhull algorithm on the reflected points to define which are the visible points for the camera. The camera position must be considered on the computation of the Quickhull. The visible points are all the points that lies on the Quickhull.

3. At this moment, all points considered visible by the camera are known. However, these points are still transformed by spherical flipping. So, they cannot be rendered because they are not points of the original model. To solve this problem, is necessary to create a mapping between the original points of the model with the transformed spherical flippled points. Once this mapping is done, it's easy to know what are the transformed and visible points in the original model. Now, with the information about the original and visible points, just send these points to be rendered

The algorithm is executed using the multithreading concept, described in Section 5.

Algorithm 1 HPR Algorithm

$points[], flippedPoints[], quickhullPoints[];$

$points[] := loadPointsFromModel();$

$CreateThread();$
$InitializeThread(DoWork);$

proc DoWork() $\equiv$
  $flippedPoints[] := sphericalFlipping(points, radius);$
  $quickhullPoints[] := quickhull(flippedPoints);$
  $mappingPoints();$
.

$DrawPoints();$

7 Conclusion and Future Works

The optimizations made to the HPR operator algorithm improved the performance and interactivity in point-based graphics applications. The use of threads for computing the visibility of points allowed a real-time rendering for not complex models.

Another important factor was the gain obtained by performing a perturbation on the position of each point in the original model. This modification allowed a reduction of 80% on the time execution of Quickhull's algorithm for the Stanford Bunny's model (69451 points) and a reduction of 53% on time execution for the Stanford Dragon's model (871414 points) - see Figures 5 and 6.

As future work we intend to make feasible to use the HPR operator for massive models. An alternative is the implementation of the Quickhull 3D algorithm using a full parallel approach with GPGPU. With the completion of this improvement, there is the possibility that the performance gain of this approach allows a greater degree of interactivity for more complex models.

Acknowledgment

The authors would like to thank CNPq, CAPES, FAPERJ, Tecgraf, ICAD/VisionLab and Petrobras for the financing support.

References

BARBER, C. B., DOBKIN, D. P., AND HUHDANPAA, H. 1996. The quickhull algorithm for convex hulls. *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE 22*, 4, 469–483.

DACHSBACHER, C., VOGELGSANG, C., AND STAMMINGER, M. 2003. Sequential point trees. In *ACM SIGGRAPH 2003 Papers*, ACM, New York, NY, USA, SIGGRAPH '03, 657–662.

DUTRÉ, P., TOLE, P., AND GREENBERG, D. P. 2000. Approximate visibility for illumination computations using point clouds. Tech. rep.

GROSS, M., AND PFISTER, H. 2007. *Point-Based Graphics (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

GUENNEBAUD, BARTHE, L., AND PAULIN, M. 2004. Deferred splatting. *Computer Graphics Forum 23*, 653–660.

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

KATZ, S., TAL, A., AND BASRI, R. 2007. Direct visibility of point sets. *ACM Trans. Graph. 26* (July).

MITRA, N. J., NGUYEN, A., AND GUIBAS, L. 2004. Estimating surface normals in noisy point cloud data. In *special issue of International Journal of Computational Geometry and Applications*, vol. 14, 261–276.

MUCKE, E. P. 2009. Quickhull: Computing convex hulls quickly. *Computing in Science and Engineering 11*, 5, 54–57. Generated by Odysci - http://www.odysci.com/article/1010112990070929.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, 343–352.

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

SCHAUFLER, G., AND JENSEN, H. W. 2000. Ray tracing point sampled geometry. In *In Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, 319–328.

WU, J., AND KOBBELT, L. 2004. Optimized sub-sampling of point sets for surface splatting. *Comput. Graph. Forum*, 643–652.

ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROSS, M. H. 2001. Surface splatting. In *SIGGRAPH'01*, 371–378.

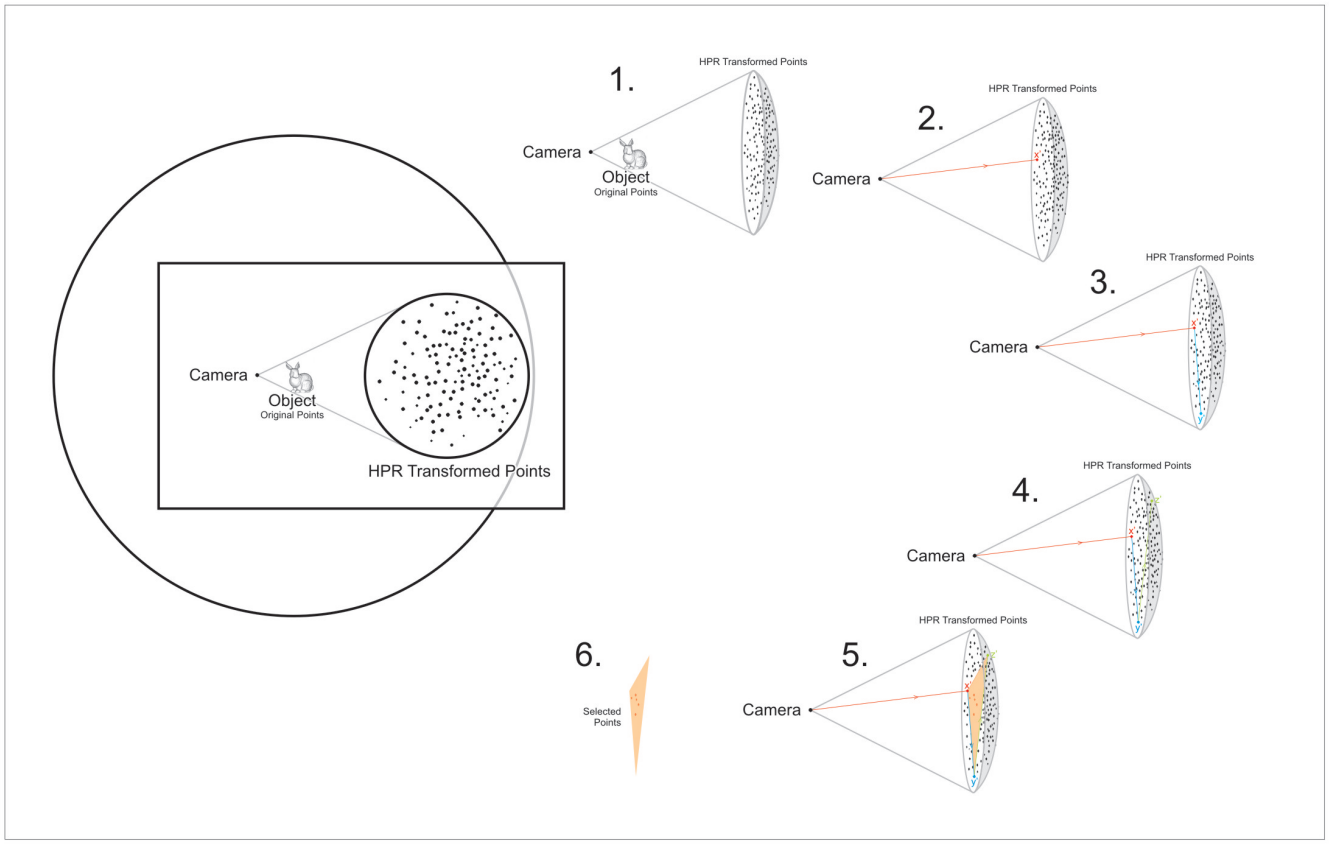
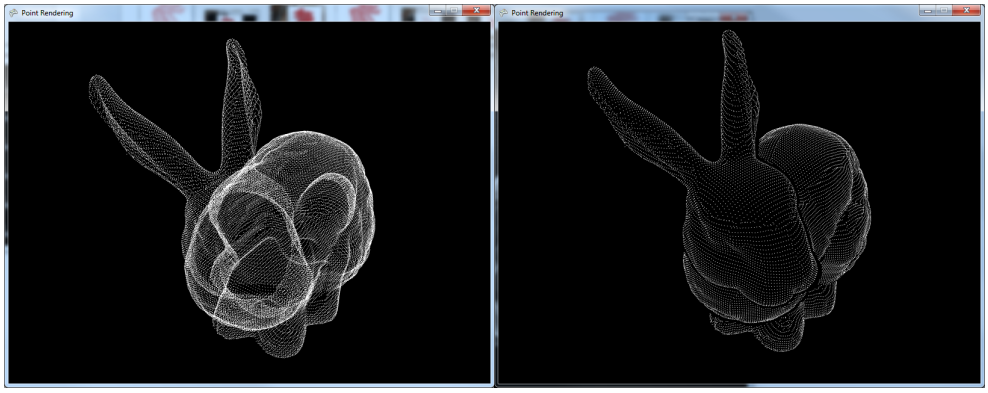
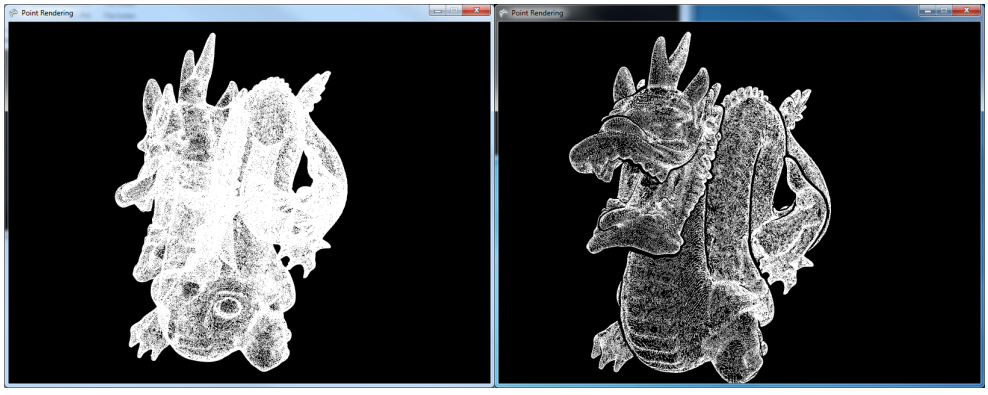


Figure 4: *Quickhull optimization schema*



(a) *Bunny with no occlussion* **(b)** *Bunny with occlussion*

Figure 5: *Bunny mesh*



(a) *Dragon with no occlussion* **(b)** *Dragon with occlussion*

Figure 6: *Dragon mesh*