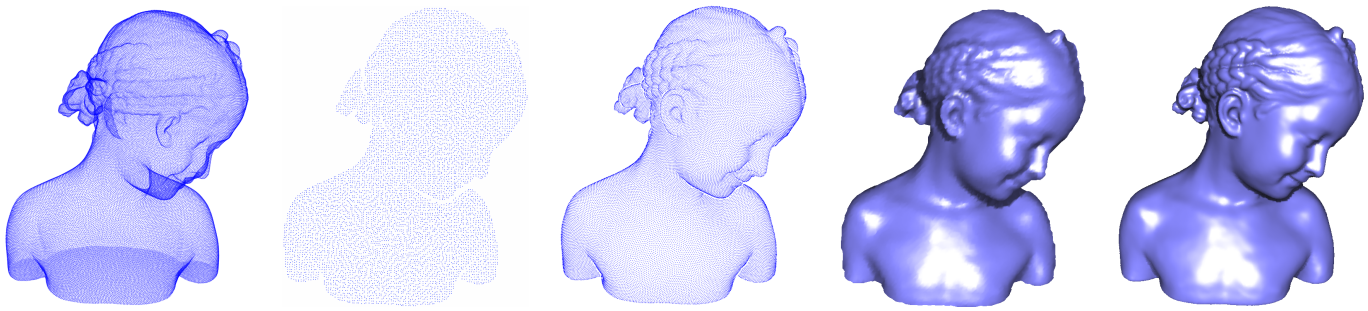
Efficient HPR-based Rendering of Point Clouds

Renan Machado e Silva, Claudio Esperança, Antonio Oliveira
Programa de Engenharia de Sistemas e Computação
COPPE / Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brazil



(a) (b) (c) (d) (e)

Fig. 1. A point cloud (a), small / large subsets of its visible points (b) / (c), and corresponding partial reconstructions (d) / (e).

***Abstract*—Recently, Katz et al. [1] have shown how visibility information for a point cloud may be estimated by the so-called HPR operator. In a nutshell, the operator consists of a simple transformation of the cloud followed by a convex hull computation. Since convex hulls take $O(n \log n)$ time to compute in the worst case, this method has been considered impractical for real-time rendering of medium to large point clouds. In this paper, we describe a GPU implementation of an approximate convex-hull algorithm that runs in $O(n + k)$ time, where k is a parameter of the method. Experiments show that the method is suitable for real-time rendering and partial reconstruction of point clouds with millions of points.**

***Keywords*-point cloud visibility; point based rendering; surface reconstruction**

I. Introduction

Point-based representations have been proposed as an alternative to polygonal meshes, making it possible to describe sampled surfaces without incurring in the cost of storing mesh topology. Thus, objects with very complex and detailed geometry may be represented more tersely with point clouds, as is the case, for instance, of data acquired with 3D scanners. In addition to a 3D position, each sample in a point cloud may contain other attributes such as surface normal or color.

The notion of visibility is well defined for many surface representation schemes such as meshes. The same cannot be said for point clouds, since a point has no area and thus cannot occlude another point. Thus, the related problem of rendering a surface represented by point clouds is usually solved using schemes such as splatting [2], or pyramid-based algorithms [3]. Another option is to estimate a polygonal mesh that interpolates the samples, a problem known as surface reconstruction. It goes without saying that all approaches rely on the point cloud being an ϵ-sampling of a surface, i.e., it must be the case that any disk on the surface with radius bigger than ϵ must contain at least one point. Some approaches require that point samples are accompanied by surface normals, while others try to estimate this information by fitting a plane using nearby samples.

The HPR operator proposed by Katz et al. [1] is a simple algorithm to determine visibility in point clouds without estimating normals or reconstructing the surface. The operator consists of two steps. First, all points of the cloud are transformed by an operation called spherical flipping. Then, the convex hull of the set containing the viewpoint and the transformed points is computed. A point is deemed visible if its transformed version appears as a vertex in the convex hull. Since the algorithm takes place in object space, it is not influenced by screen resolution. Also, it gives good results with dense or sparse clouds, although it does not cope well with noisy data or high-curvature regions [4].

The main disadvantage of the HPR operator is its dependence on a 3D convex hull algorithm, since the problem has been shown to be $\Omega(n \log n)$. Even with the aid of the enhanced computing power of modern GPUs, this is a substantial hindrance to its use for real time rendering of point clouds, given that known algorithms are not well suited to the parallel computing model of GPUs. The main contribution of the present paper is to show how to obtain a "fine enough" sampling of visible points from a point cloud using the HPR operator at interactive rates. The idea is to use an approximate convex hull algorithm suitable for implementation in GPU. In this way, the level of approximation can be calibrated in order to obtain a faithful rendering of the model for a given screen resolution. Moreover, it is shown how a triangle mesh can be computed for such a sampling, making it possible to employ

standard mesh techniques in the rendering of point clouds.

II. Related work

Determining the visibility of surfaces in a scene is an intensely studied Computer Graphics problem. Several techniques such as z-buffers and ray casting have been developed over the years, most of them requiring some means of sampling the surface in a continuous way. In the case of surfaces represented by point clouds, some ingenuity must be used to effect such a sampling. For instance, one may sample the surface using "thick" rays in the form of cylinders [5] or cones [6], but these techniques are even more computing intensive than the traditional ray-casting of meshes. Ray-casting can also resort to fitting a primitive with positive area – an ellipse, say – on the neighborhood being sampled [7]. A popular alternative is to use *splatting* methods [2], where each point is rendered affecting a small region of the screen, typically using a gaussian blot. The correct visibility is ensured by traversing the point cloud from back to front or using the z-buffer [8]. More recently, pyramid image reconstruction filters were used for "filling out" the spaces between points [3].

Rather than probing the point cloud directly, one may try to obtain a more suitable surface representation such as a polygonal mesh. If the point cloud was obtained from a 3D scanner, then the surface is a height map and thus inherits the regular grid topology used by the device [9], [10]. Some methods do not require an a priori topology, but make use of of the normal vectors which must be known for each point sample [11]. In contrast, other methods such as [12] do not require either topology or normal vectors.

The HPR operator described by Katz et al. [1], unlike other point cloud techniques, tries to establish the visibility of each point directly, i.e., independently of the rendering and without reconstructing the surface. The method does not make use of normal information nor does it require that the cloud be a height map or conform to any known topology. It consists of two steps: inversion and the determining of a convex hull.

The inversion step maps the points to a dual space. Let P be a point sampling of surface S, and C denote the point of view. Then, without loss of generality, P is first translated to a coordinate system with origin in C. The inversion proper is a function which maps a point $p_i \in P$ to some point $\widehat{p_i}$ along the ray from C to p_i in a monotonously decreasing fashion with respect to $||p_i||$. This is equivalent to say that $||\widehat{p_i}||$ decreases as $||p_i||$ increases and vice-versa. While many functions satisfy this requirement, this work employs the *spherical flipping* function, as suggested in [1].

Consider a d-dimensional sphere with radius R centered at the origin C, such that it contains all points in P. Then, *spherical flipping* reflects a point $p_i \in P$ with respect to the sphere according to

$$\widehat{p_i} = f(p_i) = p_i + 2(R - ||p_i||)\frac{p_i}{||p_i||}. \qquad (1)$$

This inversion function maps every point inside the sphere to a corresponding point outside the sphere as shown in Figure 2.

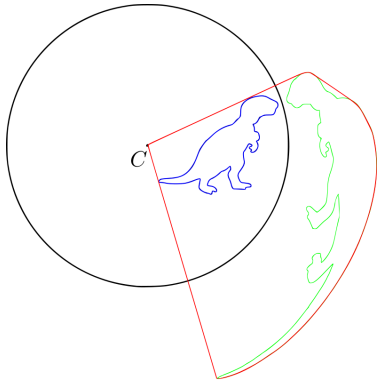


Fig. 2. The green polygon is the spherical flipping of the blue polygon. The red polygon is the convex hull of the set of points in the flipped object plus the center C of the circle.

Let $\widehat{P} = \{\widehat{p_i} = f(p_i) \mid p_i \in P\}$ be the cloud of inverted points. Then, the second step of the method consists of finding the convex hull of set $\widehat{P} \cup \{C\}$ A point p_i is considered to be visible if $\widehat{p_i}$ lies on the convex hull (see Figure 2).

The HPR operator may be applied in point clouds in any number of dimensions, although we are mainly interested in points in $\mathbb{R}^3$. The inversion step is clearly $O(n)$, regardlessly of dimension, where n is the total number of points. The convex hull may be computed $O(n \log n)$, for 2D and 3D point clouds.

The method has been shown to be correct when the point cloud is considered to contain all of the surface points. In this case, every point which the method considers to be visible is indeed visible. Note that some visible points may be considered non-visible, i.e., the method may report false negatives. The number of false negatives is diminished as R grows. In the limit, when R tends to infinity, every visible point will be correctly labeled as such. Larger values of R handle high-curvature regions of the surface. In practice, however, the input is a surface sampling, and thus the output may contain false positives as well as false negatives. The authors deal with this problem by using large R values for dense clouds and smaller R values for sparse clouds.

In a related paper, Mehra et al. [4] show that the HPR operator is very susceptible to noise, and propose a robust variation which is then used to build a global reconstruction of the surface.

III. Approximate Convex Hull

As discussed earlier, the HPR operator, though simple in concept, relies on the computation of a convex hull, which takes $O(n \log n)$ time for a cloud with n points in $\mathbb{R}^3$. In [1], for instance, the authors employ the *QuickHull* [13] algorithm, which computes the convex hull of points in 3D in $O(n \log n)$ time for favorable inputs, but is quadratic in the worst case, making it unsuitable for dealing with large point clouds.

One way of improving the speed of the technique is to use an approximate convex hull algorithm. The fact that the HPR operator is also approximate reinforces this idea, provided

that the errors introduced by one technique and the other are independent. Another contributing thought is that, when used for rendering, the operator could probably work on a sampling that is good enough for the actual screen resolution.

Several approaches for computing approximate convex hulls have been proposed in the past. The central idea initially described by Bentley et al. [14] is to obtain a subsampling of the original set and then compute an exact hull for the smaller set. Most of the various proposed algorithms concentrate on heuristics for obtaining this reduced set. The main concern is to reduce the error by choosing "good" candidates, i.e., points which are likely to lie on the convex hull of the set (see [15] for a survey).

Intuitively, a point $p_i \in P$ is in the convex hull if it is an extreme point for some given direction $\vec{d}$. In other words, if q is an origin point, then $(p_i - q) \cdot \vec{d}$ is maximum over all points in the cloud. Kavan[16] explores this property in an algorithm that computes an approximate convex hull for a set of points. While the method is described for two dimensions, it can be generalized to any number of dimensions. The algorithm is divided into three steps:

1) First, an origin point q inside the hull is selected at cost $O(n)$. This can be easily done by choosing the centroid of the cloud or the center of a bounding box containing the cloud.
2) The plane is divided into k equally spaced sectors centered at q, each covering an angle of $\frac{2\pi}{k}$. All points in the cloud are then assigned to the sector that contains them. For each sector i, establish a direction $\vec{d_i}$ aligned with the bisector of the sector angle, and choose p_i among the points assigned to the sector such that it maximizes $(p_i - q) \cdot \vec{d_i}$. The idea is that the selected p_i is a good estimate of the point which is extreme for direction d_i and, thus, probably a point on the hull. Clearly, this step can be computed in $O(n)$.
3) Finally, refine the estimate for each sector i by comparing each originally selected p_i with the points selected for all the other sectors. If another point $p_j, j \neq i$ is found which is a better estimate for direction d_i, then p_i is updated accordingly. Note that this procedure may remove but not add points to the approximate convex hull. This step takes time $O(k^2)$, since each selected point must be compared with every other selected point.

The extension of this algorithm to three dimensions is straightforward although some care must be taken in order to partition the cloud into sectors of approximately equal size. One can use, for instance, an icosahedron as a reference shape and partition each triangular face into four identical triangles until the desired number of sectors is reached. Another option is to use the algorithm described by Leopardi [17], which partitions a hypersphere into any given number of sectors having the same Lebesgue measure – e.g., perimeter in 2D, or area in 3D. An important observation is that, although the approximate algorithm of Kavan et al. aims at producing a full polytope, in our application there is only need to select points of the cloud which are believed to lie on the hull.

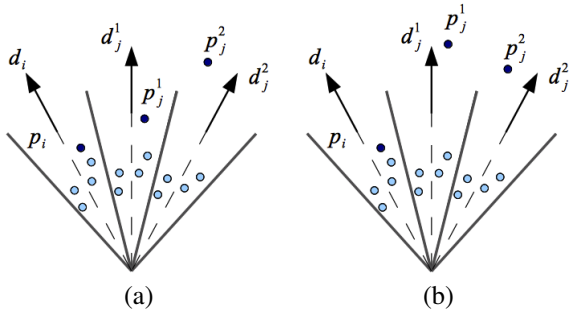


Fig. 3. In (a) candidate p_j^2 is a replacement for both p_j^1 and p_i, while in (b) p_j^2 is a replacement for p_i but not p_j^1; in this case, however, p_i will be replaced by p_j^1.

Unfortunately, the algorithm takes $O(n + k^2)$ time, which makes it advantageous over optimal exact algorithms only if $k \in o(\sqrt{n \log n})$. Note, however, that point clouds obtained with 3D scanners only contain points from the surface of the model. If the model is convex or even if it has relatively few concavities, one may expect that roughly half of the point cloud may be visible. It follows that using the just described algorithm may result in too coarse a sampling, given that a small enough value of k is chosen to make the algorithm suitably fast.

IV. Fast HPR operator

The algorithm described in the previous section is a starting point from which the HPR operator can be computed efficiently. One important observation is that it uses simple data structures which can be traversed simultaneously using a parallel computation model, such as GPU programming. Thus, if the work is split evenly among m processors, steps (1) and (2) can be expected to take $O(n/m)$ time. Another key observation is that step (3) can be computed more efficiently by examining a limited neighborhood of each sector instead of all k sectors. The idea is that the best estimate for each sector can be achieved using a scheme for propagating candidate points. Thus, step (3) can be rewritten as:

3) Refine the estimate for each sector i by comparing each originally selected p_i with the points p_j selected for all *neighbor* sectors $j \in \mathcal{N}(i)$. If any original estimate is changed by this step, repeat it until no better estimate is found for any sector.

The rationale for this modification is that the selected candidate point p_i for a given sector i with bisector d_i is more likely to be replaced by the candidate p_j corresponding to direction d_j if the angle between d_i and d_j is small. Moreover, suppose that candidate p_j^2 for a sector j^2 which is not an immediate neighbor of sector i is found to be a replacement for p_i. Then there is a sector j^1 which *is* a neighbor of i such that (1) p_j^2 is also a replacement for p_j^1, or (2) p_j^2 is not a replacement for p_j^1, but, in this case, p_j^1 is a *better* replacement for p_i (see Figure 3).

The time complexity for the modified step (3) in sequential computers depends on the number of iterations necessary for the propagation to cease, say, k'. If each sector has a constant number of neighbors, then the total cost will be $O(kk')$. In the worst case, $k' \approx k$, yielding the same time complexity of the original algorithm. Notice, however, that a large k' means that candidate points are assigned to large angular intervals, causing the hull to have correspondingly large faces. It is reasonable to assume that this will be a rare occurrence when dealing with dense point clouds such as those obtained with 3D scanners, leading us to expect that k' is significantly smaller than k. Moreover, the propagation scheme just discussed can be easily implemented in parallel architectures, thus enhancing the performance of the process as a whole.

V. Fast HPR-based rendering of point clouds

This section discusses in detail an implementation of the algorithm outlined in the previous section, as well as the extensions required to obtain a partial reconstruction of the surface which can then be used for rendering. Our prototype uses the CUDA toolkit [18] for executing most of the work in parallel in a GPU.

A. Defining the sectors

The first step consists of establishing an appropriate coordinate system for defining the sectors where the points of the cloud will be distributed. For this purpose, an enclosing sphere for the cloud is computed having center at C_e and radius r. In our implementation, C_e is the centroid of the cloud and r is the distance from C_e to the furthest point in the cloud. Then, a coordinate system is built where the origin is at C, the position of the observer, with the x axis passing through C_e (see Figure 4).

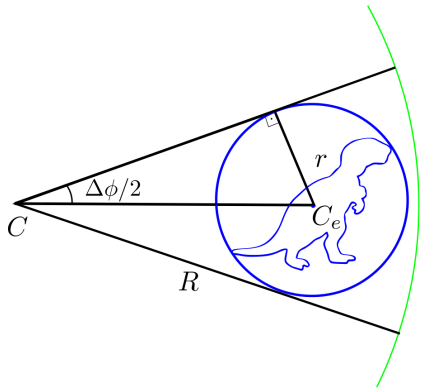


Fig. 4. Coordinate system for defining the sectors. View point C lies at the origin, with the x axis passing through C_e, the center of the enclosing sphere of the point cloud.

The sectors are defined by dividing the horizontal and vertical angles of the viewing frustum regularly in a grid-like manner. The region of the frustum containing the enclosing sphere will be symmetrical, covering an angle $\Delta\phi$ given by

$$\Delta\phi = 2\sin^{-1}\frac{r}{|C - C_e|}.$$

Using spherical coordinates, the frustum will then correspond to ranges in φ and θ given by

$$\begin{aligned} \varphi &\in [-\tfrac{\Delta\phi}{2}, +\tfrac{\Delta\phi}{2}], \\ \theta &\in [\tfrac{\pi}{2} - \tfrac{\Delta\phi}{2}, \tfrac{\pi}{2} + \tfrac{\Delta\phi}{2}]. \end{aligned}$$

In order to produce k sectors, these angular ranges are regularly sampled $\sqrt{k}$ times in each direction. The sectors thus formed will have a pyramid shape and the directions $\vec{d}_i$ to be minimized will be aligned with the φ and θ bisectors. Notice that sectors will not be identical due to the fact that any given angle interval in the φ coordinate will correspond to smaller sections closer to the poles, i.e., for θ near 0 or π. Thus, this particular way of defining sectors is only adequate when the view point is far from the cloud so as to yield a relatively narrow frustum. Whereas there are methods which do not impose this restriction and yet yield more uniform sectors – see Section III for some suggestions –, this scheme has the advantage of making it easy to visit the up to 8 neighbors of a given sector, which is necessary for the candidate propagation step (see Section V-C).

This step of the algorithm is implemented by two CUDA kernels which process all points in the cloud. The first kernel computes the centroid C_e, while the second computes r. These two are implemented as *parallel prefix scans* [19] which take $O(n/m + \log n)$ time each, using m processors.

B. Computing sector candidate points

Once C_e and r are known, and k is established by some means, the angular interval $\Delta\phi$ can be computed, thus defining the geometry of all sectors. At this point, another kernel performs a simple parallel scan of all points in the cloud with the following goals:

1) applying an affine transformation to the cloud so as to move the view point to the origin of the coordinate system and C_e to some point on the x axis,
2) computing the spherical flip of each point, storing it in an array P of size n, and
3) assigning a sector number for each point, storing it in an array *SECTOR* of size n.

The sector number of a point is an integer number between 0 and $k-1$ which can be determined by computing its spherical coordinates and finding the proper angular interval in φ and θ where it lies. For instance, if a point lies in the i'th interval in the φ direction and the j'th interval in the θ direction, then its sector number is $i\sqrt{k} + j$. Notice that all computation in this kernel is done independently for each point and thus the kernel runs in $O(n/m)$ time.

The direction $\vec{d}_i$ pointing to the center of each sector must also be computed by means of a kernel which builds an array of size k called DIR in $O(k/m)$ time. Once this is done, another kernel is fired to compute the projection of each spherically flipped point on its sector central direction. In short, an array called *DIST* of size n is computed by a parallel scan of all points such that

$$\mathit{DIST}[i] = P[i] \cdot DIR[\mathit{SECTOR}[i]].$$

Finally, a candidate extreme point for each sector must be computed by examining only the points assigned to the sector. This requires reordering the array P containing the inverted cloud so that points assigned to the same sector are contiguous in memory. This is accomplished by means of a parallel sort operation which uses the values in *SECTOR* as keys. Our prototype uses the GPU-optimized radix sort algorithm described by Merrill and Grimshaw [20] as implemented in the Thrust [21] library. Although no explicit complexity bounds are mentioned by the authors, optimal parallel sort algorithms are believed to run in $O((n \log n)/m)$ time. Once P is sorted, sector candidate points are computed with a *segmented scan* kernel [22] taking $O(n/m + \log n)$ time. The result of this computation is stored in an array called MAX of size k which contains the indices of the candidate points.

C. Candidate point propagation

In this step, the candidate initially considered as extreme point for a given sector may be replaced by a candidate assigned to one of the up to eight sectors sharing an edge or a vertex in the angular grid. This is a critical phase of the algorithm since it must be repeated a number of times until no sector candidates are replaced.

Unfortunately, counting the number of candidate replacements is complicated by the occurrence of empty sectors, i.e., sectors for which no candidates have been estimated in the previous iterations. Empty sectors can be attributed to two causes, namely, (1) the sector corresponds to a region outside the object projection, or (2) the sector is inside the object projection, but no point of the cloud lies inside it (see Figure 5). Clearly, the propagation process must ignore sectors in the first case, but not those in the second case. Thus, the propagation algorithm makes use of an auxiliary array named *EMPTY*, of size k such that *EMPTY*$[i]$ is *true* if sector i is empty, and likely to be of type (1). This array is populated along with the initial candidate points in the previous step. In order to be reasonably sure that it does not contain empty sectors of type (2), we observe that empty sectors of this type become more likely as k increases. In consequence, if k/n is above a given threshold – we use 25% in our experiments – *EMPTY* is computed for a coarser angular grid. Thus, for instance, if the *EMPTY* array has $k/4$ elements, each of its elements will be *false* only if no point of the cloud falls on a 2×2 sector neighborhood of the finer grid with k elements.

Each propagation step is computed by a simple parallel scan in $O(k/m)$ time. A global variable *CHANGED* is set to *true* if any candidate replacement is done on a sector not marked as empty. Notice that there is no need for using atomic operations to ensure non-simultaneous write access to that variable, since any access to *CHANGED* is enough to guarantee that another propagation step must be conducted.

Another important consideration is whether a *gather* or a *scatter* strategy is more adequate for this step. In a *gather* strategy, the thread examining sector s visits its neighbors looking for a replacement for its current candidate, whereas in a *scatter* strategy the candidate at s is considered as a replacement for each of its neighbors. In the former approach, each thread may alter a single sector, while in the latter, concurrent modifications may take place. In our GPU implementation, for simplicity, only the gather strategy is used. However, a CPU implementation developed as a means for comparison, uses a scatter strategy, so that each successive iteration visits only sectors which had their candidates changed in the previous iteration. This reduces considerably the number of sectors visited in each step, especially when many propagation steps are necessary due to a very high k.

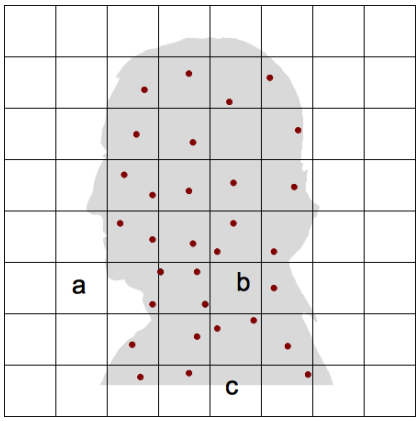
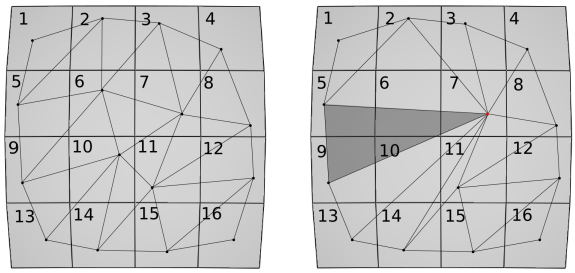


Fig. 5. Sectors lying outside the object projection such as (a) will have no candidates and need not be visited in the propagation process. Sectors such as (b) are enclosed in the object projection but have no samples inside the cloud must be visited in the propagation. Sectors such as (c) correspond to a borderline case.

D. Partial view-dependent reconstruction

In [1], a "quick and dirty" view-dependent reconstruction of the visible surface is displayed by rendering not only the vertices but also the faces (triangles) of the convex hull. In their case, this can be done at no extra cost since the topology of the hull (triangulation) is always computed by the quickhull algorithm. In our method, however, the hull is never computed per se, but a triangulation may still be inferred by visiting the angular grid and generating up to 2 triangles for each 2×2 sector neighborhood. Thus, while visiting sector i, four points may be used to form two triangles, namely, the points whose indices are $MAX[i]$, $MAX[i+1]$, $MAX[i+\sqrt{k}]$, and $MAX[i+\sqrt{k}+1]$ (see Figure 6a). Notice, however, that the propagation process may have assigned the same candidate to several neighboring sectors. Invalid triangles are trivially eliminated by requiring all three points of each triangle to be distinct. In Figure 6b, for instance, no triangles are generated while visiting sector 6, while only one triangle is generated for sector 2.

As pointed out by [1], some triangles must be filtered out since their vertices are not likely to be contiguous in a "real" surface reconstruction. They suggest removing triangles having edges longer than a certain threshold. A similar procedure is adopted in our prototype.



(a) (b)

Fig. 6. In (a) the extreme point of each sector lies within the sector, while in (b) the candidates of sectors 6 and 10 lie within sector 7. The shaded triangle is generated while visiting sector 5.

VI. Results

In order to assess the correctness and usefulness of the techniques just described, a series of experiments were conducted. The first batch of experiments aim at showing that the proposed approximate convex hull algorithm yields equivalent visual results when compared with the HPR operator using an exact convex hull. The second batch of experiments demonstrate the performance gains obtained by using a GPU-based and a conventional (i.e., CPU only) implementation of our algorithm with respect to an implementation employing a fast implementation of the well-known QuickHull algorithm [13].

All experiments were conducted on a workstation equipped an Intel i7 CPU running at 2.4 GHz and 8Gb memory. The graphics board uses an NVidia GTX 470 GPU with 1Gb memory. All software prototypes were written in C++ and OpenGL. Exact convex hulls are computed using the Qhull [23] library. GPU algorithms were written using *C for CUDA*, under CUDA 4.0 and the Thrust [21] library.

As a reference, Table I shows relevant information about the various models used in the experiments. Notice that although the original models are meshes, only the vertices are used as input point clouds for the HPR algorithm.

TABLE I
Models used in the experiments

Model name	Vertices	Faces	Example
Bimba	74,764	149,524	Figure 1
Armadillo	172,974	345,944	Figure 7
Happy Buddha	543,652	1,087,716	Figure 7
Buddha	719,560	1,500,000	Figure 7
Asian Dragon	3,609,455	7,218,906	Figure 8

A. Visual Experiments

Figure 7 shows sample renderings obtained with the method described in Section V-D. As with the original HPR method, best results were obtained by choosing an optimal value for the spherical flipping parameter R. Rather than performing the costly procedure suggested in [1], a "good" value for R was determined by mere visual inspection. On the other hand, the values for parameter k were calibrated so as to yield a number of visible points within 95% of that obtained by the exact HPR algorithm (see Section VI-B).

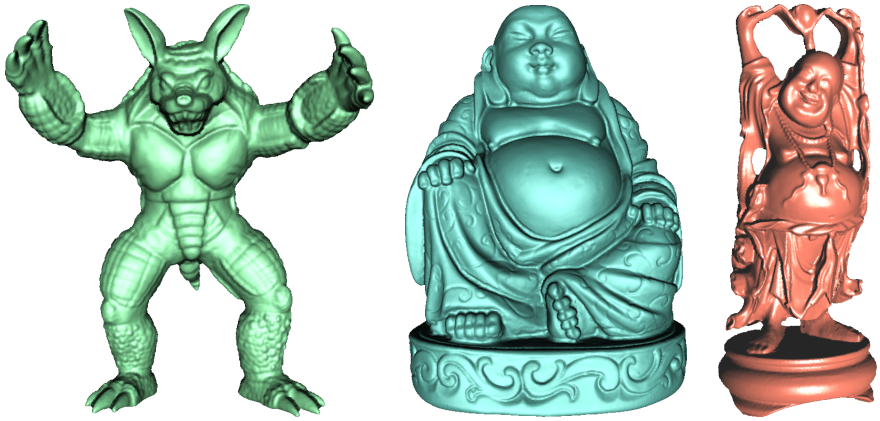
Since the proposed algorithm depends on the number of sectors, it is useful to conduct a visual inspection of the results obtained for different values of k. Figure 1a shows the point rendering of the original Bimba model containing $74{,}764$ points. In Figures 1b and 1d the visible point set and partial reconstruction are rendered with $k = 10{,}000$. Similar renderings are shown in Figures 1c and 1e for $k = 850{,}000$. These last results are indistinguishable from what was obtained with the exact HPR algorithm. Indeed, for the model and pose shown in Figure 1, the renderings for $k = 850{,}000$ classify $26{,}598$ points as visible, while the original HPR yields $27{,}351$ visible points for the same value of R.

An important property of the HPR operator is that it tends to produce better results for denser point clouds. Thus, while the operator is able to produce detailed renderings of point clouds with millions of points, computing an exact convex hull with millions of vertices is costly both in time and memory. Our method based on approximate convex hulls, however, scales well for dense clouds. The reason for this is that it uses an angular grid for obtaining a subsampling of the cloud which can be tuned to the desired screen resolution and viewing angle. As an example, in Figure 8 it is shown a rendering of the Asian Dragon model composed of $3{,}609{,}455$ vertices using the proposed method with $k = 820{,}000$. This rendering is obtained by our GPU prototype at 11 FPS. Although the rendering uses only $98{,}097$ visible points, very little visual detail is lost when compared with the rendering of the full mesh.

B. Performance Experiments

Clearly, the accuracy of the proposed method hinges on the size of the angular grid as given by parameter k. As k increases, more visible points are detected, at a cost of increased processing time. A crucial question then is how dense a grid should be used in order to produce roughly the same number of visible points as the exact algorithm. The chart shown in Figure 9 plots the number of visible points as a function of k for the Happy Buddha model, which contains 543,652 points. For the particular pose used in the experiment, the maximum number of visible points is roughly $96{,}000$, reached for k near $4{,}000{,}000$ which means that increasing k above that value is ineffectual. In practice, one might either establish a value for k as small multiple of the total number of points in the cloud, or probe for a "good enough" value by increasing k until the number of visible points levels off.

Finally, in order to compare the speedup obtained with the proposed algorithm with respect to the original HPR formulation, tests were conducted for clouds of different sizes. The results are shown in Table II. In order to provide a fair comparison, the values used for k in these experiments were established so as to yield roughly the same number of visible points as the exact implementation – thus, the number of visible points shown in the table are only approximate. Clearly, when dealing with denser clouds, the proposed method becomes less competitive, since it has to deal with a large number of empty cells. In particular, the CPU implementation performs



(a) (b) (c)

Fig. 7. Example renderings of partial view-dependent reconstructions: (a) Armadillo, (b) Buddha, (c) Happy Buddha.

Fig. 8. Renderings of the Asian Dragon model: Original mesh (top) and HPR with $k = 820{,}000$ (bottom).

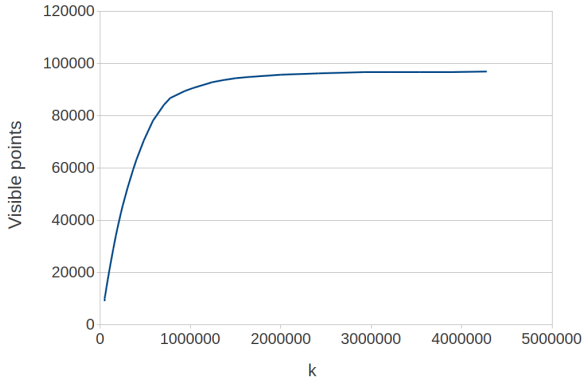


Fig. 9. Number of visible points as a function of k for the Happy Buddha.

TABLE II
PERFORMANCE COMPARISON

Model name	Total Points	Visible Points	FPS Exact	FPS Approx.	
				CPU	GPU
Bimba	74,764	$\sim$ 27,000	2	5	50
Armadillo	172,974	$\sim$ 63,000	1	2	25
Buddha	719,560	$\sim$ 250,000	0.3	0.18	1.7

worse than the exact method for the Buddha model since, in that case, less than 5% of the sectors are occupied with distinct candidate points.

It must be emphasized that the results of Table II were obtained with k values which are unnecessarily large for most applications. For instance, in the case of the Buddha model, it is possible to obtain renderings which are almost identical to Figure 7b at 20 frames per second using $k = 450{,}000$ and only 115,000 visible points.

C. Limitations

One of the main strengths of the proposed technique is the possibility of calibrating the amount of visual detail simply by choosing an appropriate value for k which, in turn, controls how well the convex hull is approximated. This dependency of the method on the fineness of the angular grid is also the source of its main limitation. As k increases, sectors become less and less occupied, i.e., more empty sectors of both type (1) and (2) are generated (see Section V-C). Since empty sectors are also represented in the data structures, this leads to a waste of memory, which is especially scarce in the case of our GPU implementation. A chart demonstrating this behavior is shown in Figure 10.

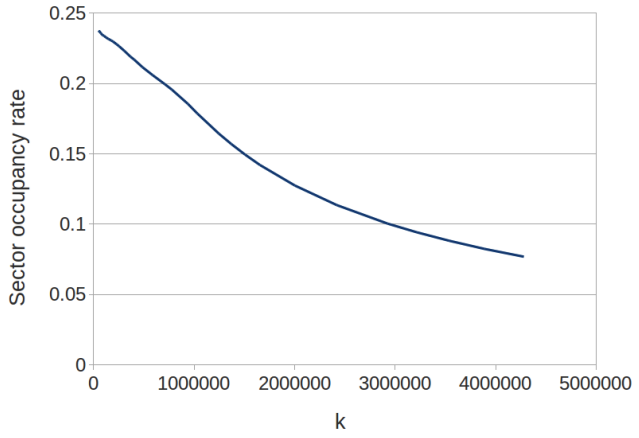


Fig. 10. Sector occupancy rate as a function of k for the Happy Buddha model.

In our experiments, this limitation was not important for models with up to a million points, such as those used in the performance comparison (Table II), since the number of visible points obtained by our prototype are comparable to those obtained by the exact HPR. For a larger model such as the Asian Dragon shown in Figure 8, however, the exact HPR returns over a million visible points, while the best rendering we were able to obtain with our prototype detected only about half as many. It should be noted, nevertheless, that the exact HPR takes over 16s to render a single frame, while our prototype does half as good a job at 2 frames per second.

VII. Conclusion

The use of an approximate convex hull algorithm makes it possible to turn the HPR operator into a more practical tool for interactively rendering point clouds. By setting an appropriate value for the parameter k, it is possible to favor either the speed of the rendering or the quality of the result. Also, due to its simplicity, the convex hull algorithm described in Section IV, can be easily implemented in GPU for an additional performance boost.

Two extensions to the method are planned to be tackled in the near future. First, the $EMPTY$ array should make it possible to implement a more efficient memory management where only non-empty sectors need to be stored and processed. Second, more intelligence should be incorporated into our prototype so as to yield as good as possible renderings without having to fiddle with parameters k and R.

Acknowledgements

To Diego Nehab, for educating us in GPU programming. The models Bimba, Happy Buddha, Armadillo and Asian Dragon are courtesy of the Stanford 3D Scanning Repository. The model Buddha is courtesy of the AIM@SHAPE Shape Repository.

References

[1] S. Katz, A. Tal, and R. Basri, "Direct visibility of point sets," in *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*. New York, NY, USA: ACM, 2007, p. 24.

[2] M. Sainz and R. Pajarola, "Point-based rendering techniques," vol. 28, 2004, pp. 869–879.

[3] R. Marroquim, M. Kraus, and P. R. Cavalcanti, "Efficient point-based rendering using image reconstruction," in *Symposium on Point-Based Graphics 2007, Prague-Czech Republic*, September 2007.

[4] R. Mehra, P. Tripathi, A. Sheffer, and N. J. Mitra, "Visibility of noisy point cloud data," *Computers and Graphics*, vol. 34, no. 3, pp. 219–230, 2010.

[5] G. Schaufler and H. W. Jensen, "Ray tracing point sampled geometry," in *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*. London, UK: Springer-Verlag, 2000, pp. 319–328. [Online]. Available: http://portal.acm.org/citation.cfm?id=647652.732143

[6] M. Wand and W. Straßer, "Multi-resolution point-sample raytracing," in *Graphics Interface 2003 Conference Proceedings*, 2003.

[7] I. Wald and H.-P. Seidel, "Interactive Ray Tracing of Point Based Models," in *Proceedings of 2005 Symposium on Point Based Graphics*, 2005.

[8] D. Tavares and J. Comba, "Efficient approximate visibility of point sets on the GPU," in *Graphics, Patterns and Images (SIBGRAPI), 2010 23rd SIBGRAPI Conference on*, 30 2010-sept. 3 2010, pp. 239 –246.

[9] G. Turk and M. Levoy, "Zippered polygon meshes from range images," in *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1994, pp. 311–318.

[10] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1996, pp. 303–312.

[11] M. Kazhdan, M. Bolitho, and H. Hoppe, "Poisson surface reconstruction," in *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006, pp. 61–70.

[12] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Surface reconstruction from unorganized points," in *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1992, pp. 71–78.

[13] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Trans. Math. Softw.*, vol. 22, no. 4, pp. 469–483, 1996.

[14] J. L. Bentley, F. P. Preparata, and M. G. Faust, "Approximation algorithms for convex hulls," *Commun. ACM*, vol. 25, no. 1, pp. 64–68, 1982.

[15] C. E. Kim and I. Stojmenović, "Sequential and parallel approximate convex hull algorithms," *Computers and Artificial Intelligence*, vol. 14, no. 6, pp. 597–610, 1995.

[16] L. Kavan, I. Kolingerova, and J. Zara, "Fast approximation of convex hull," in *Proceedings of the 2nd IASTED international conference on Advances in computer science and technology*. Anaheim, CA, USA: ACTA Press, 2006, pp. 101–104.

[17] P. Leopardi, "A partition of the unit sphere into regions of equal area and small diameter," in *Electronic Transactions on Numerical Analysis*, vol. 25, 2006, pp. 309–327.

[18] "CUDA," version 4.0. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[19] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, ser. GH '07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 97–106.

[20] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011.

[21] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010, version 1.4.0. [Online]. Available: http://www.meganewtons.com/

[22] S. Sengupta, M. Harris, M. Garland, and J. D. Owens, "Efficient parallel scan algorithms for many-core GPUs," in *Scientific Computing with Multicore and Accelerators*. Taylor & Francis, 2011, ch. 19, pp. 413–442.

[23] "Qhull code for convex hull, delaunay triangulation, voronoi diagram, and halfspace intersection about a point." [Online]. Available: http://www.qhull.org/