DOI: 10.1111/cgf.12265

COMPUTER GRAPHICS *forum*
Volume 33 (2014), number 1 pp. 178–189

Image Space Rendering of Point Clouds Using the HPR Operator

R. Machado e Silva, C. Esperança, R. Marroquim and A. A. F. Oliveira

Computer and Systems Engineering Program, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil
renangms@gmail.com; {esperanc, marroquim, oliveira}@cos.ufrj.br

Abstract

The hidden point removal (HPR) operator introduced by Katz et al. [KTB07] provides an elegant solution for the problem of estimating the visibility of points in point samplings of surfaces. Since the method requires computing the three-dimensional convex hull of a set with the same cardinality as the original cloud, the method has been largely viewed as impractical for real-time rendering of medium to large clouds. In this paper we examine how the HPR operator can be used more efficiently by combining several image space techniques, including an approximate convex hull algorithm, cloud sampling, and GPU programming. Experiments show that this combination permits faster renderings without overly compromising the accuracy.

Keywords: surface reconstruction rendering, visibility determination

ACM CCS: I.3.5 [Computer Graphics]: Computational Geometry and Object Modelling—Geometric algorithms; languages and systems.

1. Introduction

The term 'point cloud' refers to a collection of points in 3D space which typically are taken from the external surface of an object. This sampling is most often produced with the aid of 3D scanners, but may also be the result of some modelling process. In addition to position, each sample in a point cloud may contain other information about the surface such as normal or colour.

Point clouds have also been proposed as an alternative to other geometric representations such as polygonal meshes, for instance [GP07]. In fact, dense polygonal meshes can in many cases dispense with the topology information and faithfully represent the object solely by means of its vertices. In this case, surface reconstruction, that is, the process of estimating mesh topology using only vertex positions is an important and actively researched problem.

Another related problem is that of point cloud visibility. In short, assuming that the cloud is a sampling of a surface, a cloud point is deemed visible to a given observer if the corresponding surface point is also visible. Although surface reconstruction can be used to determine this information, direct methods have also been proposed. Most of these, however, can be categorized *rendering* methods, such as Surface Splatting [ZPvBG01]. In other words, rather than merely determining whether a given point is visible, such methods perform a rendering of the implied object surface. Moreover, such methods usually work in *image space*, that is, they solve the problem for a particular rectangular grid of pixels representing the image.

On the other hand, the hidden point removal (HPR) operator proposed by Katz *et al*. [KTB07] is a simple algorithm to determine visibility in point clouds without rendering them or performing surface reconstruction. The operator consists of two steps. First, all points of the cloud are transformed by an operation called *inversion*. 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. The method produces good results with dense or sparse clouds by calibrating the inversion transformation with respect to the cloud density. This requires the point sampling to be fairly uniform. Formally speaking, it requires that the cloud be an ϵ-sampling of a surface, that is, it must be the case that any disk on the surface with radius bigger than ϵ must contain at least one point of the cloud.

Rather than using a rendering method to determine visibility, it is possible to use the visibility obtained with the HPR operator to perform a rendering of the cloud [KTB07]. This application, however, is hindered by the fact that it depends on the computation

of a 3D convex hull algorithm, a costly operation that has been show to run in $\Omega(n \log n)$ time. Inasmuch as this can be alleviated, say, by using the enhanced computing power of modern GPUs, the fact that the method runs in object space means that it is not subject to adaptation to the conditions of the rendering. In other words, the method will perform essentially the same work regardlessly of the kind of projection being employed or the resolution of the screen.

This paper, an extended version of [SEO12], describes an efficient way of implementing the HPR operator so that it can be used for rendering point clouds at interactive rates. To accomplish this, we use several image-space techniques. The main idea is to compute an approximate convex hull using an angular grid, which makes it straightforward to use GPU programming models. The rest of the paper is divided as follows: Section 2 reviews earlier work proposed for point cloud rendering, with special emphasis on the HPR operator. Section 3 describes how convex hull approximations can be produced using spatial decomposition of the point cloud, discussing in detail our own tackle on the problem by refining a solution proposed by Kavan *et al*. [KKZ06]. Section 4 explains the general form of our rendering algorithm. Section 5 describes a specific implementation of the algorithm using shaders; Section 6 discusses how to estimate values for the algorithm parameters so as to obtain the desired trade-off between rendering quality and speed. Section 7 presents the results of several experiments comparing with the original approach of Katz *et al*. [KTB07]. Finally, in Section 8 we present a few concluding remarks and directions for future research.

2. Related Work

In *Computer Graphics*, the problem of visibility consists of determining which parts of the objects in a scene should actually appear in a rendering of that scene. This problem assumes different characteristics depending on the kind of representation used for the objects. A common instance of the visibility problem is known as the hidden surface problem, and corresponds to objects represented by their boundary surfaces. Well-known techniques such as z-buffers and ray casting require some means of sampling the surface in a continuous way. In the case of surfaces represented by point clouds, the fabric of the surface must be inferred in an indirect way. For instance, one may sample the surface using 'thick' rays in the form of cylinders [SJ00] or cones [WS03], but these techniques are even more computing intensive than the traditional ray-casting of surfaces represented by meshes. Ray-casting can also resort to fitting a primitive with positive area—an ellipse, say—on the neighbourhood being sampled [WS05]. A popular alternative is to use *splatting* methods [SP04], 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 [TC10]. Another interesting idea is to use multi-resolution reconstruction filters for 'filling out' the spaces between points [MKC08].

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 [TL94], [CL96]. Some methods do not require an a priori topology, but make use of the normal vectors which must be known for each point sample [KBH06]. In contrast, other methods such as [HDD*92] do not require either topology or normal vectors.

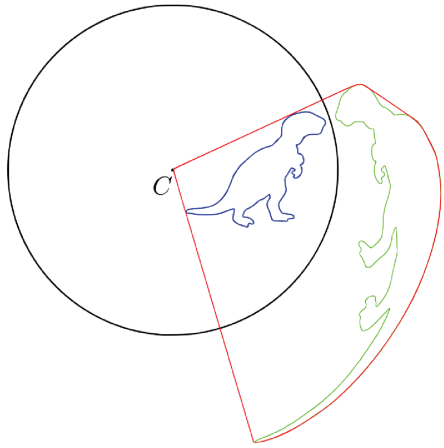

Figure 1: *The green polygon is the spherical flipping of the blue polygon. The red polygon is the convex hull of the set of points in the flipped object plus the centre C of the circle.*

2.1. The HPR operator

The HPR operator described by Katz *et al*. [KTB07], unlike other point cloud techniques, tries to establish the visibility of each point directly, that is, independently of the rendering and without reconstructing the surface. Thus, unlike the previously discussed approaches, it is an object space technique, although the present paper, in a sense, is all about how to mix in some image space techniques.

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, Katz *et al*. employ the *spherical flipping* function.

Consider a d-dimensional sphere with radius R centred 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

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

This inversion function maps every point inside the sphere to a corresponding point outside the sphere as shown in Figure 1.

Table 1: *Performance comparison.*

			Exact HPR		Approx. HPR						
						FPS					
						CPU		CUDA		Shader	
Model	R	k	#vis pts	fps	#vis pts	pts	Faces	pts	Faces	pts	Splatting
Bimba	3100	254 016	27 722	2.9	23 627	20	18	100	77	126	51
Armadillo	5400	857 476	61 393	1.1	50 337	7.5	7	55	36	69	45
Happy Buddha	6904	2 569 609	16 2887	0.4	126 306	2.6	2.3	7.6	6	8.5	6.2
Buddha	9500	2 350 089	218 632	0.3	170 972	2.5	2.2	11	7	18.5	11.5
Asian Dragon	25 637	5 919 489	503 988	0.1	498 656	1	0.8	7.8	4.3	5.2	3

Let $\widehat{P} = \{\widehat{p_i} = f(p_i) | p_i \in P\}$ 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 1).

The HPR operator may be applied for point clouds in any number of dimensions, although we are mainly interested in points in $\Re^3$. 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.

In the original paper [KTB07] the authors show that the method is 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, that is, the method may report false negatives. The number of false negatives diminishes as R grows. In the limit, when R tends to infinity, every visible point will be correctly labelled 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. Katz *et al*. 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*. [MTSM10] 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.

3. Approximate Convex Hull

As mentioned earlier, the HPR operator requires the computation of the convex hull of the inverted cloud, which takes $O(n \log n)$ time for a cloud with n points in $\Re^3$. In [KTB07], for instance, the authors employ the *QuickHull* [BDH96] algorithm, which computes the convex hull of points in 3D in $O(n \log n)$ time for favourable inputs, but is quadratic in the worst case. Even the highly optimized implementation used in the original paper has sluggish performance (see, for instance, the results for the Happy Buddha model shown in Table 1) making it unsuitable for dealing with large point clouds. In fact, as with many convex hull algorithms, QuickHull performs best when the number of points in the hull is small when compared with the size of the cloud, which is not to be expected when the HPR operator is applied to clouds which are samplings of a surface model.

Clearly, some way of improving the efficiency of convex hull calculations will greatly benefit the usefulness of the HPR operator. Taking advantage of the substantial raw power of modern GPU's immediately comes to mind. Surprisingly, although implementing 2D algorithms in GPU is relatively straightforward, only more recently have we seen works describing GPU-assisted convex hull algorithms for 3D clouds [TO12], [SGES12], [GCTH11]. While significant performance increases have in general been reported, we have not yet evaluated these approaches in the scope of the problem at hand.

Another 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. When used for rendering, the idea is to apply the HPR operator on a subsampling of the cloud which has a high probability of containing convex hull points but is just dense enough for the actual screen resolution. An additional advantage of this idea is that this kind of screen-based sampling fits naturally with GPU computing models.

Several approaches for computing approximate convex hulls have been proposed in the past. The central idea initially described by Bentley *et al*. [BPF82] 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, that is, points which are likely to lie on the convex hull of the set (see [KS95] 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 *et al*. [KKZ06] explore this property in an algorithm that computes an approximate convex hull for a set of points. 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.

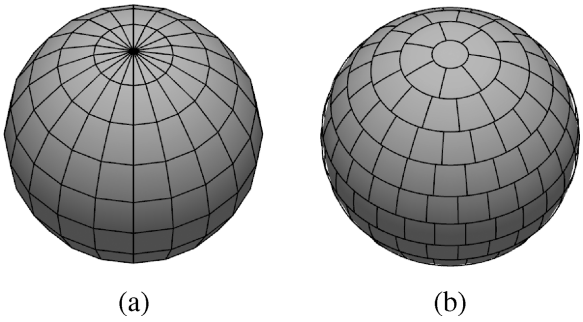

Figure 2: *Two ways of dividing a sphere into k partitions: (a) angular grid, and (b) method of Leopardi.*

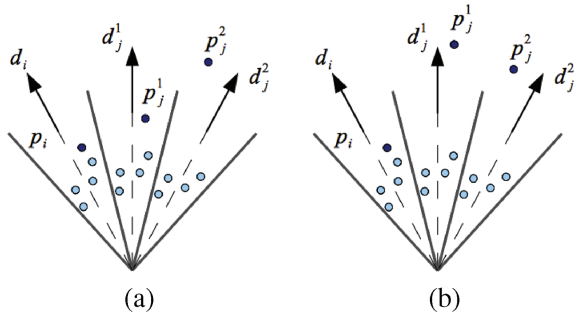

Figure 3: *In (a) candidate p_j^2 is a replacement for both p_j^1 and p_i, while in (b) p_j^2 is a replacement for p_i but not p_j^1; in this case, however, p_i will be replaced by p_j^1.*

(2) The plane is divided into k equally spaced sectors centred 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.

While the method is described for two dimensions, it can be generalized to any number of dimensions. The extension of this algorithm to three dimensions is straightforward although some care must be taken in order to partition the cloud into k angular sectors. In principle, any kind of spherical tiling scheme can be used for this purpose. One way for doing it is to project a cube or any regular polyhedron onto the sphere, subdividing each face a suitable amount of times. This has the disadvantage of producing not quite identical partitions. Another option is to use the algorithm described by Leopardi [Leo06], which partitions a hypersphere into any given number of sectors having the same Lebesgue measure—for example, perimeter in 2D, or area in 3D (see Figure 2b). In our implementation, however we use a simple angular grid based on spherical coordinates (Figure 2a), since, unlike the schemes mentioned earlier, a grid provides a simple way of visiting neighbouring partitions. Although this arrangement also produces partitions of different sizes, we use a small region on the sphere close to the equator where quadrants have very similar sizes.

The algorithm proposed by Kavan *et al.* takes $O(n + k^2)$ time, which makes it advantageous over optimal exact algorithms only if $k \ll \sqrt{n \log n}$. 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 Kavan *et al.*'s algorithm may result in too coarse a sampling, given that a small enough value of k must be chosen to make the algorithm suitably fast. One observes, however, that step (3) can be computed more efficiently by examining a limited neighbourhood 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 *neighbour* sectors $j \in N(i)$. 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 neighbour of sector i is found to be a replacement for p_i. Then there is a sector j^1 which *is* a neighbour 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) depends on the number of propagation steps actually performed. The first step clearly takes $O(k)$ time, but the next steps will only work on sectors which were changed on the previous steps. In the worst case, all sectors will have changed at every step, and the process will finish in k steps, yielding the same $O(k^2)$ performance of the original algorithm. Notice, however, that a large number of propagation steps 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. More importantly, we will show that in some cases it is beneficial to limit the number of propagation steps so that large faces are not formed.

4. HPR-based Rendering of Point Clouds Using CUDA

The algorithm for computing approximate convex hulls described in the previous is a starting point from which the HPR operator can be computed efficiently. Since it uses simple data structures which can

be traversed simultaneously, they are amenable to implementation in parallel architectures such as GPU's. Roughly speaking, if the work is split evenly among m processors, steps (1) and (2) can be expected to take $O(n/m)$ time, while the propagation phase should take $O(\ell k/m)$ time, where ℓ is the maximum number of propagation steps allowed.

Two main GPU programming paradigms are prevalent nowadays. In the first one, *shader programming*, the programmer stays within the general graphics pipeline framework, that is, he/she prepares code pieces called *shaders* which replace the conventional stages in the graphics pipeline, still using graphics data structures such as pixel fragments and vertices. Shader programming must employ a shading language such as *OpenGL Shading Language* GLSL, which extends the OpenGL graphics programming dialect.

On the other hand, *general purpose computing on graphics processing units* (GPGPU) takes a broader view on the process of building programs which can be run on GPUs, in the sense that they mostly dispense with graphics pipeline-related structures for code and data. Currently, OpenCL is the dominating open GPGPU programming language, whereas NVidia's graphics cards may also be programmed using the CUDA [cud] toolkit. It should also be mentioned that the distinctions between these two paradigms are being somewhat blurred by initiatives such as *compute shaders*, supported by OpenGL versions 4.3 onward.

In this section, we discuss a CUDA implementation of the algorithm outlined in the previous section, whereas a shader version is discussed in the following section.

4.1. 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 centre 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).

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

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

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

$$\begin{aligned} \varphi &\in [-\frac{\Delta\phi}{2}, +\frac{\Delta\phi}{2}], \\ \theta &\in [\frac{\pi}{2} - \frac{\Delta\phi}{2}, \frac{\pi}{2} + \frac{\Delta\phi}{2}]. \end{aligned} \tag{3}$$

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, that is, 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 3 for some suggestions, this scheme has the advantage of making it easy to visit the up to eight neighbours of a given sector, which is necessary for the candidate propagation step (see Section 4.3).

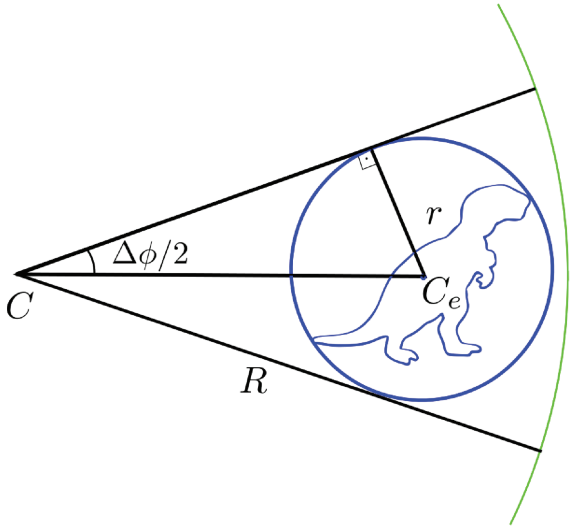

Figure 4: *Coordinate system for defining the sectors. View point C lies at the origin, with the x axis passing through C_e, the centre of the enclosing sphere of the point cloud.*

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* [SHZO07] which take $O(n/m + \log n)$ time each, using m processors.

4.2. 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.
(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 ith interval in the φ direction and the jth 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 centre 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

$$DIST[i] = P[i] \cdot DIR[SECTOR[i]]. \tag{4}$$

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 [mer11] as implemented in the Thrust [HB10] 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 [SHGO11] 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.

4.3. 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, that is, 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, that is, using the average occupancy rate calculated for the actual grid, we linearly estimate the size of the grid for the $EMPTY$ map. Thus, for instance, if the $EMPTY$ array has $k/4$ elements, each of its

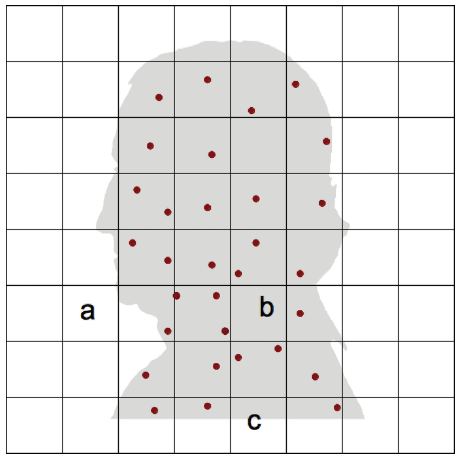

Figure 5: *Sectors lying outside the object projection such as (a) will have no candidates and need not be visited in the propagation process. Sectors such as (b) must be visited in the propagation process, since they are enclosed in the object projection but have no samples inside the cloud. Sectors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

elements will be *false* only if no point of the cloud falls on a 2×2 sector neighbourhood 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 neighbours 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 neighbours. 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.

4.4. Partial view-dependent reconstruction

In [KTB07], 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

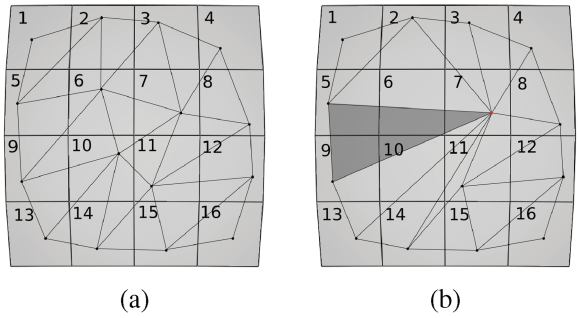

Figure 6: *In (a) the extreme point of each sector lies within the sector, while in (b) the candidates of sectors 6 and 10 lie within sector 7. The shaded triangle is generated while visiting sector 5.*

still be inferred by visiting the angular grid and generating up to two triangles for each 2×2 sector neighbourhood. 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 neighbouring sectors. Invalid triangles are trivially eliminated by requiring all three points of each triangle to be distinct. In Figure 6(b), for instance, no triangles are generated while visiting sector 6, while only one triangle is generated for sector 2.

As pointed out by [KTB07], 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. We have adopted a similar procedure in our approach.

5. Shader Implementation

The shader implementation heavily uses the concepts of render-to-texture and multipasses. We have entirely avoided reading back any information from the GPU during the many different passes. We have also avoided extensive use of texture fetches by recomputing some values in every shader pass, that is instead of storing the sector directions they are recomputed as needed.

To shorten the description of the following passes, we briefly describe a common GPGPU technique to use a buffer as input primitive, hereafter called *buffer processing*. By mapping a quad primitive to fit exactly the viewport's dimensions, the fragment shader is called once for each buffer pixel. In this way, we simulate processing each pixel of a 2D buffer in parallel using shaders. When it is necessary to read and write from the same buffer, we employ a ping-pong scheme: read from a first buffer and write to a second one, and invert roles in each subsequent pass.

5.1. Point projection

We project all points to an offscreen buffer of size $(\sqrt{k}, \sqrt{k})$ corresponding to the k sectors. Each buffer pixel represents one sector as illustrated in Figure 7.

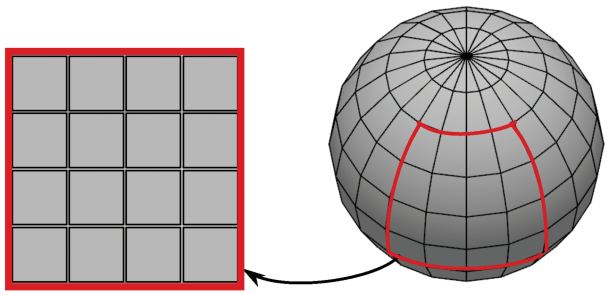

Figure 7: *Mapping of the sectors into an offscreen-buffer. The buffer size is* 4×4 *just for illustrative purposes.*

In the *vertex shader* we compute the first two steps described in Section 4.2, and then force the output position to project the point to the corresponding sector/fragment. In the *fragment shader* we compute the sector's direction and the distance as in Equation (4). To keep only the best candidate, depth test is enabled and the fragment depth value set to $(1.0 - DIST)$. The result is written to two different buffers, one containing the direction DIR, and the other the original vertex world coordinates and the vertex ID to facilitate some future operations.

5.2. Propagation

After the first pass, we have a buffer with k cells with the best candidate for each sector. To propagate there are two equivalent strategies: propagate the best candidate by using a diffusion gather approach; or use a fixed size kernel to gather neighbouring fragments. The former works better when a large propagation is needed, as it does not overload a thread processing the fragment shader.

The buffer is then processed and each sector/fragment collects information about its neighbours' directions, recomputes the sector direction and keeps the best candidates information. We call the set of buffers with the propagated candidates the *visible buffer*, as it contains the best direction for each sector, the corresponding vertex ID and its original world coordinates.

5.3. Normal computation

To compute a normal direction for each visible point, we process the visible buffer and sample all neighbours of a fragment. We circulate the neighbours of the central candidate c_0 in a clockwise manner, as shown in Figure 8(a). When a candidate $c_1 \neq c_0$ is found, it is marked. We continue circulating until we find a third candidate $c_2 \neq c_1 \neq c_0$ to form a triangle with the central candidate. At this point we compute the normal $n = (c_2 - c_0) \times (c_1 - c_0)$ and set $c_1 = c_2$.We repeat this process until we have completed the circulation checking all possible triangles . To avoid degenerated triangles a maximum distance of 3 pixels is permitted to form a triangle during circulation, as illustrated in Figures 8(b) and (c).

In a second pass, a diffusion process similar to the direction propagation step is conducted, accumulating all normals belonging to the same candidate.

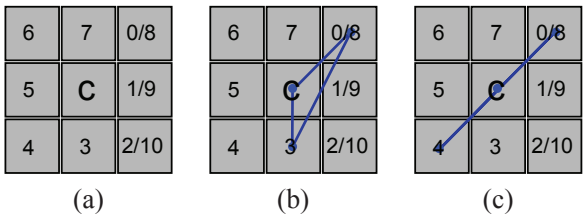

Figure 8: *(a) Circulation order to compute the normal in the fragment shader. (b) A valid triangle configuration* $(0, 3, c)$ *and (c) an invalid configuration* $(0, 4, c)$.

5.4. Point rendering

To render the points we avoid the naive approach of reading back the visible buffer to collect all visible points after the propagation phase. Alternatively, we could sample the visible buffer at each fragment, and check if there is a visible point in its original position. However, for large screen sizes or k values, it is cheaper to simply reproject all points and check if the vertex IDs matches the corresponding sector in the visible buffer. If it is the best candidate, then it is rendered, otherwise the shader discards the point.

5.5. Screen space surface reconstruction

At this point we have buffers containing the visible points and their normals. To reconstruct the surface using a splatting technique, it is necessary to know the size of each point, which is usually defined by the local density. We can profit from the normal propagation pass and simultaneously compute a splat size for each visible point. To achieve this, we add to the visible buffer information about the original sector coordinates of the projected point. Thus, during normal propagation, we also propagate the largest distance to the original sector's position. We have tested our implementation using the approach by Marroquim *et al*. [MKC08], since it can make direct use of these buffers without further processing.

6. Algorithm Calibration

The original HPR algorithm depends on just one parameter, namely, R, the radius of the sphere used for the inversion transformation. The proposed algorithm also depends on k, the size of the angular grid, and on ℓ, the number of iterations allowed for propagation.

In [KTB07], an optimal value for R is estimated by a process whereby the cloud is repeatedly rendered from opposite sides with varying values of R trying to maximize the total number disjoint visible points. The rationale is that a high number of common visible points indicates too many false positives (R is too high), whereas a low number of disjoint visible points indicates too many false negative (R is too low). It is possible, however, to estimate a reasonable value for R if the point density of the surface for the given viewpoint is known. Let us consider two front-facing points lying 1 unit distant from the observer and separated by an angle of α (see Figure 9). Then, a convex hull face spanning the two inverted points will exactly occlude a point distant $d + 1$ units from the observer if

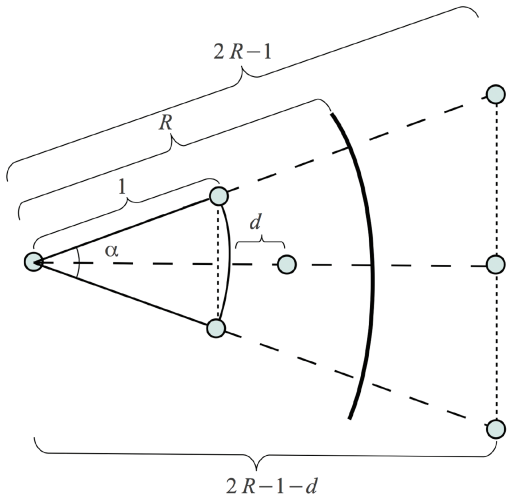

Figure 9: *Spherical inversion of two points angularly separated by* α *and their ability to occlude a point* d *units farther away.*

$$\frac{2R - 1 - d}{2R - 1} = \cos\frac{\alpha}{2}.$$

Thus, stipulating a small value for d, a lower bound for R may be computed, given that the angular separation α between two visible points is known. The same coarse sampling used to obtain the *EMPTY* map can be used estimate this information in the way described below.

Let e be the number of non-empty cells in the *EMPTY* map and n be the size of the cloud ($n \gg e$). Then, the problem consists of estimating what angular grid resolution would lead to a certain ratio a between non-empty sectors and the total number of sectors s. This problem is similar to several probability problems known collectively as 'balls in bins' [RSM98]. The probability that at least one of n balls falls in a given bin chosen from s possible bins is given by

$$a = 1 - \frac{(s-1)^n}{s^n},$$

which, when solved for s gives

$$s = -\frac{1}{(1-a)^{\frac{1}{n}} - 1}. \tag{5}$$

Thus, if the coarse grid *EMPTY* has e of its m sectors occupied, then we may infer that a finer grid with k sectors will offer $s = ke/m$ sectors for occupation. Clearly, we would like to estimate a value for k which would yield one point per sector. Unfortunately, the balls and bins analogy does not fit our problem for high values of a. For instance, $a = 1$ yields $s = 1$. Nevertheless, using $a = 0.5$ we may estimate the angle separation α as twice the angular size of the resulting sector.

The method outlined above can also be used to estimate ℓ, the maximum number of propagation steps needed to cover the sectors between any two visible points. For doing this, however, we must

Figure 10: *Example renderings of partial view-dependent reconstructions: (a) Bimba (74 764 pts), (b) Armadillo (172 974 pts), (c) Buddha (719 560 pts), (d) Happy Buddha (543 652 pts) and (e) Asian Dragon (3 609 455 pts).*

guess the total number of visible points (as opposed to the total number of points in the cloud). It is possible to modify the creation of the $EMPTY$ map to perform this estimation. The idea is to compute for each sector the closest point to the observer and to consider visible the other points that fall within the sector and are roughly at the same distance. Alternatively, we may consider, say, that half of the cloud points are visible and thus overestimate ℓ somewhat.

7. 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 aims to assess how well our algorithm fares with respect to the original HPR algorithm. The second batch of experiments demonstrate the performance gains obtained by using GPU-based and conventional (i.e. CPU only) implementations of our algorithm with respect to an implementation employing a fast implementation of the well-known QuickHull algorithm [BDH96].

All experiments were conducted on a workstation equipped with 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 [qhu] library. The GPGPU implementation uses CUDA 4.0 and the Thrust [HB10] library.

As a reference, Figure 10 show renderings obtained with our techniques for all 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.

7.1. Accuracy experiments

A fair comparison between the proposed algorithm and the original HPR method hinges on the choice of the value of R, k and ℓ. In [KTB07] it is suggested that the best visibility results obtained by the HPR operator correspond to a value of R that yields the smallest total number of false positives and false negatives. For instance, Figure 11(a) shows a pose of the Armadillo model obtained with the exact HPR method for $R = 7400$, which yields the lowest error with respect to the real visibility (obtained with the mesh model), namely 6227 false negatives, 2808 false positives and 58037 correct points, that is, 15%. Our approximate method with $k = 4 \times 10^6$ for the same R yields an error of 16% (see Figure 11b), with 7434 false negatives, 2073 false positives and 56 830 correct points. On the other hand, using the value $R = 5400$ as estimated using the technique described in Section 6 yields the results shown in Figures 11(c) and (d) with errors 16% and 18% for the exact and approximate HPR algorithms, respectively. Arguably, the results for lower R look nicer than those for higher R. This is due to the fact that some false positives appear as distracting 'holes' in the surface.

A better understanding of the strengths and weaknesses of both algorithms can be obtained by examining the differences between the results shown in Figures 11(c) and (d). In Figure 12(a), red dots and blue dots are points considered visible by the exact algorithm but not by the approximate algorithm and vice versa. Thus, we observe that our approach does a better job in the areas around the snout and the hand fingers. This is due to the fact that by using a small number of propagation steps avoids eliminating visible points which are farther away from the observer. On the other hand, by using an angular grid with a fixed resolution, many visible points near the silhouette are missed. This is confirmed by Figure 12(b) which shows the false positives (blue) and false negatives (red) for Figure 11(d).

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 for the desired screen resolution and viewing angle.

As an example, Figure 13 shows three renderings of the Buddha model composed of 1.5 million points obtained with the proposed algorithm using low, medium and high-density angular grids. The last picture is a rendering of the same pose with the exact algorithm. Visual inspection shows little, if any, difference between the result of

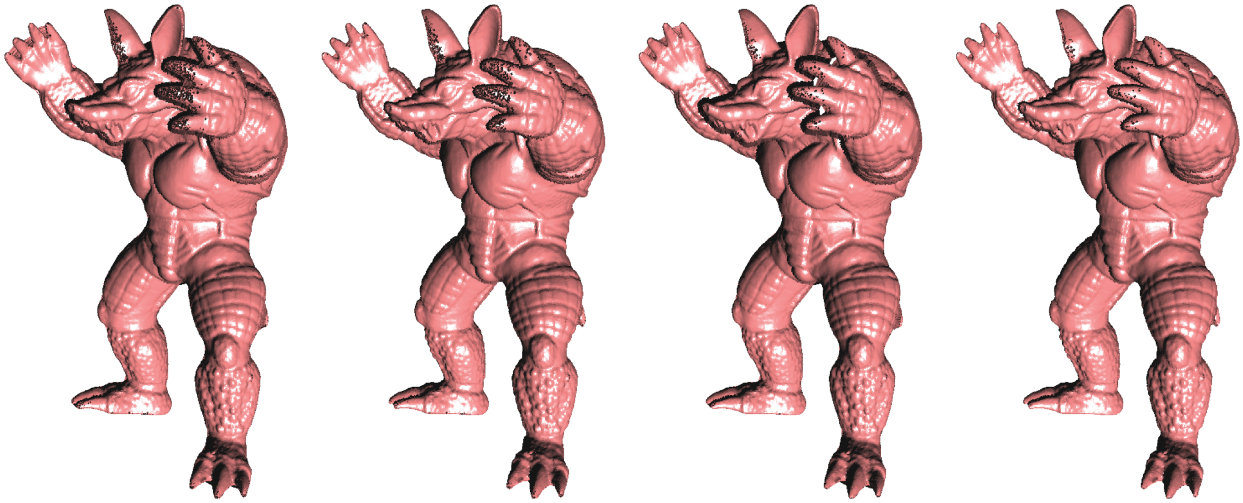

Figure 11: *Panels (a) and (b) show a pose of the Armadillo model rendered for optimal R with the exact and the approximate algorithm, while (c) and (d) show corresponding results for an automatically estimated R. Note: these renderings use per-face (flat) normals so as to highlight individual faces.*

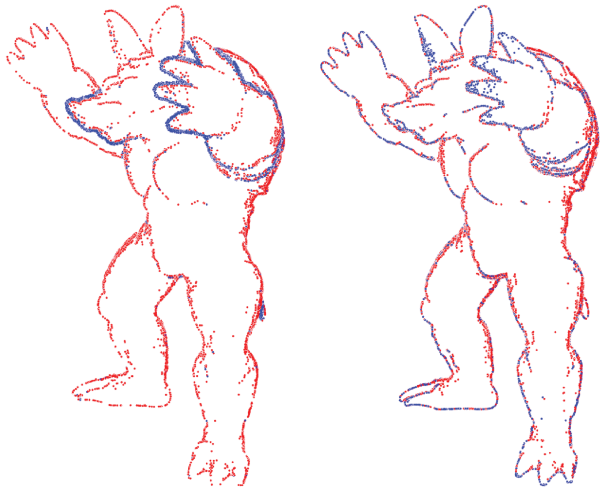

Figure 12: *Error comparisons for renderings shown in Figures 11(c) and (d). (a) shows how the algorithms fail to detect points which are truly visible: red dots are points which considered visible only by the exact algorithm, while the blue dots are points considered visible only by the approximate algorithm. (b) shows visibility errors solely for the approximate algorithm: red points are false negatives while blue points are false positives.*

our algorithm with high resolution and the exact algorithm, despite the fact that the former detects roughly 200 000 visible points versus 240 000 for the latter. As with the Armadillo experiment discussed above, most points considered visible by the exact but not by the approximate algorithm are concentrated on the silhouette, where they have little visual importance.

7.2. Performance experiments

A natural question for the proposed method is how well it compares performance-wise with the exact algorithm. As demonstrated with the accuracy experiments shown in Section 7.1, however, it is not possible to obtain exactly the same results with both algorithms since this would require an angular grid of almost infinite size. For this reason, in the following experiments we are only interested in comparing the performance of the various implementations of the approximate algorithm. Thus, in Table 1, the experiments with the exact algorithm show a number of visible points which are up to 20% higher than the corresponding experiments with the approximate algorithm, in spite of producing no significant visual differences. The experiments used the same point clouds used in the accuracy experiments, with k and R estimated using the calibration method discussed in Section 6.

Note that in Table 1 for the approximate HPR we distinguish the performances of the visibility algorithm implementations from those of the partial surface reconstruction (points vs. faces), since our approach makes it possible to classify a point of the cloud as visible or invisible without ever computing the convex hull topology, unlike the exact HPR algorithm. In general, we observe that both GPU implementations have roughly the same performance, which is from three to eight times better than the CPU implementation, which, nevertheless, is quite practical for small to medium-sized clouds, especially if a lower value for k is used.

Clearly, the speed of the method is inversely proportional to 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 14 plots the number of visible points as a function of k for the Happy Buddha model,

Figure 13: *(a)–(c) show a pose of the Buddha model rendered with the approximate HPR algorithm for k equal to* 40*K*, 100*K and* 10*M*, *respectively, while (d) shows the result of the exact HPR. Note: these renderings use per-face (flat) normals so as to highlight individual faces.*

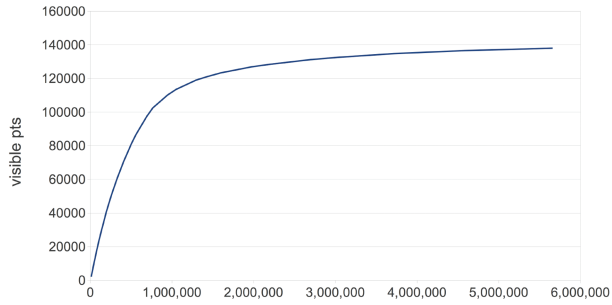

Figure 14: *Number of visible points as a function of k for the Happy Buddha model.*

which contains 543 652 points. For the particular pose used in the experiment, the maximum number of visible points is roughly 135 000, 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.

8. Conclusion

The algorithm for computing approximate convex hulls as described in this paper can be viewed as a practical alternative for several applications where the hull is used for estimates of volume, area or other integral properties. In particular, we have shown that it can be used for computing the visibility of point clouds as per the HPR operator introduced by Katz *et al*.

The image-based techniques described for implementing the HPR algorithm have as a main advantage the fact that it can trade-off accuracy for speed. Additionally, they are very amenable to parallelization as our GPU-based implementations demonstrate. It is important to notice that the algorithm needs to project all points whenever the view—or the model—changes. Thus, model inspection can easily be implemented in such a way that a coarse rendering is shown when the user is changing viewing parameters rapidly, but a fine rendering is shown when the interaction ceases.

When used in high-quality mode, the results obtained are arguably indistinguishable from those obtained with the exact HPR algorithm. In fact, in some cases, as with the example shown in Figure 11, the visual quality of the renderings look even nicer.

The practical value of the proposed technique is enhanced by being able to estimate the key parameters of the algorithm using a simple pre-sampling of the point cloud. It should be mentioned, however, that these estimates work better with more uniformly sampled surfaces.

One important limitation of the proposed technique is the fact that it may use up a lot of memory for high values of k. A possible solution for this shortcoming is to use the *EMPTY* array to implement a more efficient memory management where only non-empty sectors need to be stored and processed.

Acknowledgements

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

[BDH96] Barber C. B., Dobkin D. P., Huhdanpaa H.: The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software 22*, 4 (1996), 469–483.

[BPF82] Bentley J. L., Preparata F. P., Faust M. G.: Approximation algorithms for convex hulls. *Communications of the ACM 25*, 1 (1982), 64–68.

[CL96] Curless B., Levoy M.: 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* (1996), pp. 303–312.

[cud] CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[GCTH11] Gao M., Cao T.-T., Tan T.-S., Huang Z.: ghull: a three-dimensional convex hull algorithm for graphics hardware. In *Symposium on Interactive 3D Graphics and Games* (San Francisco, CA, USA, 2011), I3D ' 11, pp. 204–204.

[GP07] Gross M., Pfister H. : *Point Based Graphics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2007.

[HB10] Hoberock J., Bell N.: Thrust: A parallel template library (2010). http://code.google.com/p/thrust. Accessed December 2012.

[HDD*92] Hoppe H., DeRose T., Duchamp T., McDonald J., Stuetzle W.: Surface reconstruction from unorganized points. In *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques* (1992), pp. 71–78.

[KBH06] Kazhdan M., Bolitho M., Hoppe H.: Poisson surface reconstruction. In *SGP '06: Proceedings of the Fourth Eurographics Symposium on Geometry Processing* (Aire-la-Ville, Switzerland, 2006), Eurographics Association, pp. 61–70.

[KKZ06] Kavan L., Kolingerova I., Zara J.: Fast approximation of convex hull. In *Proceedings of the 2nd IASTED International Conference on Advances in Computer Science and Technology* (2006), pp. 101–104.

[KS95] Kim C. E., Stojmenović I.: Sequential and parallel approximate convex hull algorithms. *Computers and Artificial Intelligence 14*, 6 (1995), 597–610.

[KTB07] Katz S., Tal A., Basri R.: Direct visibility of point sets. In *SIGGRAPH '07: ACM SIGGRAPH 2007 Papers* (2007), p. 24.

[Leo06] Leopardi P., : A partition of the unit sphere into regions of equal area and small diameter. *Electronic Transactions on Numerical Analysis 25* (2006), 309–327.

[mer11] Merrill D., Grimshaw A.: High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters 21*, 02 (2011), pp. 245–272.

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

[MTSM10] Mehra R., Tripathi P., Sheffer A., Mitra N. J.: Visibility of noisy point cloud data. *Computers & Graphics 34*, 3 (2010), 219–230.

[qhu] Qhull code for convex hull, Delaunay triangulation, Voronoi diagram, and halfspace intersection about a point. http://www.qhull.org. Accessed December 2012.

[RSM98] Raab M., Steger A., München T. U.: Balls into bins—a simple and tight analysis. In *RANDOM'98, LNCS 1518*, M. Luby, J. Rolim, and M. Serna (Eds.) (Springer-Verlag, Berlin, Heidelberg, 1998), pp. 159–170.

[SEO12] Silva R. M., Esperanca C., Oliveira A.: Efficient hpr-based rendering of point clouds. In *Proceedings of the 2012 25th SIBGRAPI Conference on Graphics, Patterns and Images* (2012), pp. 126–133.

[SGES12] Stein A., Geva E., El-Sana J.: Cudahull: Fast parallel 3d convex hull on the GPU. *Computers & Graphics 36*, 4 (2012), 265–271.

[SHGO11] Sengupta S., Harris M., Garland M., Owens J. D.: Efficient parallel scan algorithms for many-core GPUs. In *Scientific Computing with Multicore and Accelerators*. J. Kurzak, D. A. Bader, J. Dongarra (Eds.), (Taylor & Francis, Boca Raton, FL, January 2011), ch. 19, pp. 413–442.

[SHZO07] Sengupta S., Harris M., Zhang Y., Owens J. D.: Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2007), pp. 97–106.

[SJ00] Schaufler G., Jensen H. W.: Ray tracing point sampled geometry. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000* (2000), pp. 319–328.

[SP04] Sainz M., Pajarola R.: Point-based rendering techniques. *Computers & Graphics 28*, 6 (2004), 869–879.

[TC10] Tavares D., Comba J.: Efficient approximate visibility of point sets on the GPU. In *23rd SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, (September 2010), pp. 239–246.

[TL94] Turk G., Levoy M.: Zippered polygon meshes from range images. In *SIGGRAPH '94: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques* (1994), pp. 311–318.

[TO12] Tzeng S., Owens J. D.: Finding convex hulls using quickhull on the gpu. *CoRR abs/1201.2936* (2012). http://arxiv.org/abs/1201.2936. Accessed December 2012.

[WS03] Wand M., Strasser W.: Multi-resolution point-sample raytracing. In *Graphics Interface 2003 Conference Proceedings* (2003).

[WS05] Wald I., Seidel H.-P.: Interactive Ray Tracing of Point Based Models. In *Proceedings of 2005 Symposium on Point Based Graphics* (2005).

[ZPvBG01] Zwicker M., Pfister H., van Baar J., Gross M.: Surface splatting. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (2001), pp. 371–378.