Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Graphics 28 (2004) 869–879
COMPUTERS & GRAPHICS
www.elsevier.com/locate/cag

# Point-based rendering techniques

Miguel Sainz*, Renato Pajarola

*Computer Graphics Lab, Computer Science Department, University of California, Irvine 92697-3425, United States*

**Abstract**

The increasing popularity of points as rendering primitives has led to a variety of different rendering algorithms, and the different implementations compare like apples to oranges. In this paper, we revisit and compare a number of recently developed point-based rendering implementations within a common testbed. Also we briefly summarize a few proposed hierarchical multiresolution point data structures and discuss their advantages. Based on a common view-dependent level-of-detail (LOD) rendering framework, we then examine different hardware accelerated point rendering algorithms. Experimental results are given with respect to performance timing and rendering quality for the different approaches. Additionally, we also compare the point-based rendering techniques to a basic triangle mesh approach.


*Keywords:* Point-based rendering; Level-of-detail; Hardware acceleration

## 1. Introduction

Point-based surface representation and rendering techniques have recently become very popular [10]. In fact, 3D points are the most simple and fundamental geometry-defining entities. Points as display primitives were considered early in [14]; however, they have only recently received increased attention.

Based on their fundamental simplicity, points have motivated a variety of research on topics such as shape modeling [1,23], object capturing [28], simplification [22], processing [15,21], rendering (see Section 2), and hybrid point-polygon methods [5–7,9]. The major challenge of point-based rendering (PBR) algorithms is to achieve a continuous interpolation between discrete point samples that are irregularly distributed on a smooth surface. Furthermore, correct visibility must be supported as well as efficient level-of-detail (LOD) rendering for large data sets.

Different proposed PBR methods have led to a variety of implementations that are as hard to compare as apples and oranges. In this paper, we present an attempt to realistically compare different point-rendering primitives and algorithms by implementing several techniques within the same rendering application framework. This allows a more objective evaluation of available point-rendering alternatives for quality and expected performance with respect to each other. In addition to the point-based rendering methods, we also provide traditional triangle-based rendering performance and quality results for relative comparison. Furthermore, we also compare a few different point-based multiresolution modeling techniques.

*Contributions*: In this paper, we compare a variety of hardware accelerated point-splatting techniques in a common view-dependent LOD rendering framework. This allows an objective evaluation of different exchangeable point-splatting rendering back-ends, also compared to traditional triangle-based polygonal rendering. The compared methods include a wide range of techniques from simple screen-parallel opaque squares to smoothly blended surface-aligned elliptical

*Corresponding author.
*E-mail addresses:* msainz@ics.uci.edu (M. Sainz), pajarola@acm.org (R. Pajarola).

0097-8493/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2004.08.014

disks. The multiresolution models compared in this survey include traditional hierarchical models, in compact as well as sequential format.

In addition to our initial survey [27] we provide more detailed performance and quality comparisons of rendering primitives, including a comparison to conventional polygonal surface rendering. Furthermore, we have also fully integrated the QSplat [26] multiresolution data structure into our testbed, which allows more detailed measurements and comparisons of adaptive LOD-traversal approaches compared to our first study.

## 2. Related work

Since the re-discovery of points as rendering primitives in [11], a stream of new PBR algorithms have been proposed.

Highly space-efficient multiresolution point hierarchies have been proposed in [4,26] along with effective LOD selection and rendering algorithms. Fast high-quality rendering of smoothly blended point samples is achieved by hardware accelerated α-texturing and per-pixel normalization in [2,3,18,25]. Moreover, optimized anisotropic texture sampling can be realized by elliptical splat primitives in object space as proposed in [25,18]. The main hierarchical multiresolution models and hardware accelerated rendering techniques are summarized and evaluated in this paper.

Recent approaches such as [2] or [8] address high-speed rendering of point data by exploiting hardware acceleration and on-board video memory as geometry cache. In contrast to our preliminary study presented in [27], in this paper we do not compare rendering performances using any video memory as geometry cache.

The surface-splatting technique introduced in [24,29] and the differential points proposed in [12,13] offer high-quality surface rendering. The rendering back-end of surface splatting can be accelerated by hardware [25]. Differential points also offer fast rendering but do not offer real-time view-dependent LOD rendering as yet.

In this paper, we focus on comparing the point-splatting primitives and techniques and some of the most generic multiresolution point-rendering systems [4,8,25,26] and [18]. We also compare the point-based rendering techniques to a basic triangle mesh-based rendering.

## 3. Multiresolution representation

### 3.1. Point samples

The data sets considered in our experiments are dense sets of surface point samples organized in a multi-resolution representation. The point samples $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n \in \mathbf{R}^3$ are given in 3D space and may be irregularly distributed on the object's surface. Note that we assume that the discrete point samples satisfy necessary sampling criteria such as the Nyquist condition, and fully define the surface geometry and topology.

The point data consist of attributes for spatial coordinates $\boldsymbol{p}$, normal orientation $\boldsymbol{n}$ and surface color $\boldsymbol{c}$. Furthermore, it is assumed that each point also defines a spatial extent in object space. This *size* information, e.g. a bounding sphere radius $r$, specifies either a circular or an elliptical disk centered at $\boldsymbol{p}$ and perpendicular to $\boldsymbol{n}$. Other attributes optionally include a normal-cone semi-angle $\theta$. For correct visibility, the (elliptical) disks of points must cover the sampled surface nicely without holes and thus overlap each other in object space as illustrated in Fig. 1. Elliptical disks $e$ consist of major and minor axis directions $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ and their lengths.

### 3.2. Multiresolution hierarchies

Besides [11,24], which use specific re-sampling techniques to represent objects as points, most PBR approaches use some sort of a hierarchical space-partitioning data structure as multiresolution representation. The most often proposed structures are octrees of which the region octree with regular subdivision has been favored in [2,4,25]. In [26], a midpoint-split kD tree and in [16,18] an adaptive point octree are used (see also Fig. 2). Fundamentally, the LOD point hierarchy stores aggregate information in each node about all points in that subtree such as centroid position, normal and bounding volume information.

An extremely memory-efficient point LOD hierarchy is given in [26]. Aggressive quantization techniques and look-up tables are used to reduce the cost to represent a point $\boldsymbol{p}$ and bounding sphere radius $r$ by only 13 bits, as well as the normal $\boldsymbol{n}$ to 14 bits. The color $\boldsymbol{c}$ is quantized to a 5–6–5 bit and the normal-cone semi-angle $\theta$ to 2 bits. The tree structure uses 3 bits in each node to encode the number of children. The LOD hierarchy is laid out in breadth-first order in an array, with each

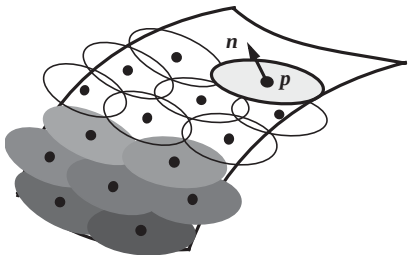


Fig. 1. Elliptical disks covering a smooth and curved 3D surface.

Fig. 2. Example hierarchical spatial subdivision of an adaptive point-octree.

group of siblings sharing one pointer (index) to their list of consecutive child nodes.

In [4], an octree is proposed that implicitly encodes the point coordinates $\boldsymbol{p}$ as the center of a cell in the recursive octree subdivision. A byte code of the subdivision provides the tree branching information at each node. The normal $\boldsymbol{n}$ and color $\boldsymbol{c}$ are quantized to less than 2 bytes and 1 byte, respectively. No bounding sphere size is used as it is implicit in the hierarchy and a normal-cone semi-angle is optionally maintained in non-leaf nodes only. During hierarchy traversal, due to the lack of explicit parent–child links, back-tracking at a node is only supported by actively skipping its entire subtree without performing any operations.

Such compact encodings of the LOD hierarchy and point attributes lead to storage costs of only a few bytes per point, which in turn allows the representation of several 100 million points within the 4 GB virtual memory addressing limit of 32 bit systems. This is a significant benefit over methods with more complex node formats.

The LOD hierarchy in [16,18] does not apply any quantization to point attributes $\boldsymbol{p}$, $\boldsymbol{n}$, $r$, $\theta$ or $\boldsymbol{c}$, and keeps additional optional information to define oriented elliptical disks instead of circular disks. The hierarchy can easily be linearized in breadth-first order with each node storing the index to the first child and branching factor. This allows a simple implementation for general use and supports efficient back-tracking in recursive traversal algorithms. The simple data formats and flexibility come at the expense of storage cost compared to the methods outlined above.

In our experiments in Section 6, we test the compact LOD hierarchy of [26] alongside the simple LOD framework of [16,18].

### 3.3. Sequential multiresolution models

In [8], a nested bounding sphere hierarchy, i.e. that of [26], is sequentialized in such a way that no more explicit nor implicit parent–child relation is maintained. The LOD decision if a node $h$ or its parent $g$ in the multiresolution hierarchy is drawn is entirely delegated to the individual nodes, avoiding any recursive traversal of a hierarchical LOD data structure. The so de-referenced nodes are then linearly sorted with respect to decreasing LOD importance. In Section 4.3, we explain in more detail how this is used in the rendering stages. This sequential LOD data structure is also compared to the hierarchical LOD structures outlined above in Section 6.

## 4. LOD selection

### 4.1. View-dependent LOD metric

One of the basic error metrics of many view-dependent LOD methods is the perspective projection of an object-space geometry deviation to screen space. In PBR, the screen-space projection of the spatial extent of a point sample is usually considered. Given a point's spatial extent area $A$ in object space and its distance $d$ to the viewpoint, a screen-space error metric can be defined as $\varepsilon = f \times A/d^2$, which is the perspective projected point sample's spatial size, corrected by a factor $f$ for the surface normal orientation if applicable.

For bounding-sphere hierarchies [4,8,26] the extent is basically $A = \pi r^2$ for a given bounding-sphere radius $r$. Note that in [8], this area measure is further cleverly refined by approximating the actual occupancy of points in the subtree. For object-space elliptical disks as in [16,18], the extent is computed as $A = \pi r \kappa r$ with $r$ being the major ellipse axis length and $\kappa$ the axis' aspect ratio.

For a viewpoint $\boldsymbol{v}$ and a point $\boldsymbol{p}$ with normal $\boldsymbol{n}$, the factor $f$ can be computed as $f = \boldsymbol{n}(\boldsymbol{v} - \boldsymbol{p})/|\boldsymbol{v} - \boldsymbol{p}|$. For non-leaf nodes in the LOD hierarchy, this should be corrected for the bounding normal-cone semi-angle $\theta$ (see [16,18] for details on that).

As outlined in Section 5, we compare three different point-rendering primitives: (1) OpenGL point, (2) depth-sprite or (3) triangle-based splatting primitives.

Correspondingly, we adjust the view-dependent error metric given above as follows for the different primitives:

*Points*: Renders a screen-aligned disk for each point. Thus, the extent is computed as the disk size $A = \pi r^2$ using the bounding sphere radius $r$. The factor $f$ is set to 1.0 as no orientation with respect to the surface normal is considered.

*Sprites*: Renders a surface-normal oriented disk for each point. Hence the extent is computed as $A = \pi r^2$ with the bounding sphere defining the disk radius $r$. It also uses the conservative orientation factor $f = \cos(\gamma - \theta)$, the cosine of the angle $\gamma$ between the normal $\boldsymbol{n}$ and the view direction $|\boldsymbol{v}-\boldsymbol{p}|$ minus the normal-cone semi-angle $\theta$ (and clamped to zero appropriately).

*Triangles*: Renders a tangential elliptical disk for each point. Thus, it computes the extent as $A = \pi r\kappa r$ and sets the orientation factor to $f = \cos(\gamma - \theta)$ as for sprites.

Therefore, given a screen-space error tolerance $\tau$ and viewpoint $\boldsymbol{v}$, an LOD point $(\boldsymbol{p}, \boldsymbol{n}, r, \theta, \kappa)$ can be rendered if $fA|\boldsymbol{v}-\boldsymbol{p}|^{-2} \leqslant \tau$, and must be split into smaller LOD point samples otherwise, if not a leaf node.

Note that the bounding sphere and normal-cone attributes allow for effective visibility culling in a LOD-selection algorithm. Given the normals $N_{1..4}$ of a four-sided view frustum and the viewpoint $\boldsymbol{v}$, a point sample is outside the view frustum if $(\boldsymbol{v}-\boldsymbol{p}) \bullet N_j > r$ for any of the normals $N_{1\,4}$. A point is considered back-face culled if the angle between $(\boldsymbol{v}-\boldsymbol{p})$ and normal $\boldsymbol{n}$, minus $\theta$, is larger than $90^\circ$.

### 4.2. Hierarchical LOD

The basic view-dependent LOD-selection algorithm consists of a depth- or breadth-first traversal of the multiresolution hierarchy as illustrated in Fig. 3. The basic traversal not only incorporates the view-dependent LOD metric as outlined in Section 4.1 to decide whether a node is rendered or further refined, but also includes back-tracking based on view frustum and back-face culling.

In [26], a top–down traversal based on an error threshold $\tau$ is performed using the explicit tree organization of its bounding sphere hierarchy. A major implementation detail is that from the compressed node representation (see Section 3.2) the rendering attributes are decompressed on-the-fly for rendering.

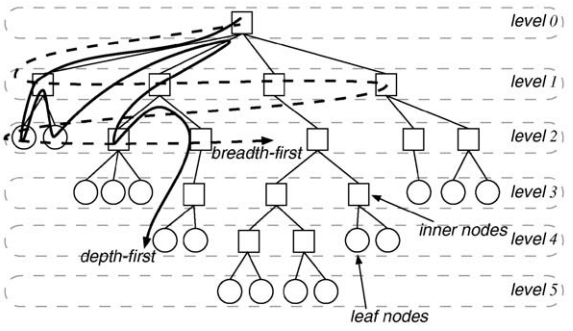


Fig. 3. Example of LOD-hierarchy organization and traversal orders.

Alternatively, a depth- or breadth-first traversal of the LOD hierarchy is used in [4]. Note that the traversal in this approach incorporates a fast incremental computation of the geometric transformation and projection of points (on the main CPU). This is particularly useful in a software-rendering system as in [4]. Due to the lack of an explicit octree structure, back-tracking due to culling is only supported by actively skipping all nodes in a subtree when traversing the linearized octree. This may limit culling efficiency as the data have to be scanned at least up to the level and index of the highest-resolution point selected for rendering.

The explicit octree in [16,18] allows depth- and breadth-first traversal algorithms and effective back-tracking, e.g. to support culling. A level-by-level linearization, and embedding of the hierarchy into an array, improves memory access coherency for breadth-first traversal order due to its sequential access to nodes, see also Fig. 3.

### 4.3. Sequential LOD

From the projective error metric $\varepsilon = A/d^2$, with $d$ being the distance between point $\boldsymbol{p}$ and viewpoint $\boldsymbol{v}$, and $A$ being the size of the point sample in object space, one can define the minimal distance $rmin_h$ at which node $h$ will be split for a given error tolerance $\varepsilon$ by $rmin_h = \sqrt{A_h/\varepsilon}$. Consequently, a maximal distance $rmax_h$ is defined at which the node will be merged based on the $rmin_g$ of its parent $g$. Compensated for the distance to the parent node, we get a merge distance of $rmax_h = rmin_g + |\boldsymbol{p}_h - \boldsymbol{p}_g|$. This concept has been introduced in [8] to de-couple individual nodes in the LOD hierarchy such that each one can individually be selected and rendered based on its merge and split distances $rmax_h$ and $rmin_h$, which may occasionally result in rendering nodes for which it was determined already to display the parent node (see also [8] for more details). Note that in the preprocess, $\varepsilon$ can be set to 1.0 to construct the sequential point trees [8].

To determine all points to be rendered they are ordered according to *rmax*. For a given viewpoint $\boldsymbol{v}$ and error threshold $\tau$, the LOD selection proceeds as follows: Based on the bounding sphere of point 0, with radius $r_0$ and center $\boldsymbol{p}_0$, the limiting merge and split distances $dmin = \sqrt{\tau} \times (|\boldsymbol{p}_0 - \boldsymbol{v}| + r_0)$ and $dmax = \sqrt{\tau} \cdot (|\boldsymbol{p}_0 - \boldsymbol{v}| - r_0)$ are computed. Within the *rmax*-ordered points, only the conservative range $[lo,hi]$ of points must be considered for rendering as shown in Fig. 4, with $lo$ and $hi$ being the smallest, respectively largest, index for which $rmin_{lo} \leqslant dmin$ and $rmax_{lo} \geqslant dmax$. The entire range of points from $\boldsymbol{p}_{lo}$ to $\boldsymbol{p}_{hi}$ is then processed by the graphics

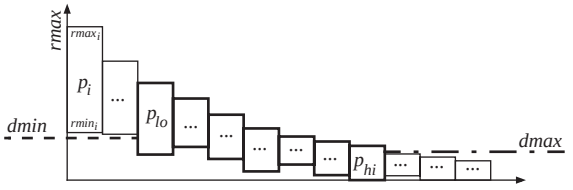


Fig. 4. Ordering of points with respect to their *rmax* values and selection of LOD-range.

hardware. Using a vertex program the individual per-point LOD test $rmin_i \leqslant \sqrt{\tau} \cdot |\boldsymbol{p}_i - \boldsymbol{v}| \wedge rmax_i \geqslant \sqrt{\tau} \cdot |\boldsymbol{p}_i - \boldsymbol{v}|$ is performed and any points $\boldsymbol{p}_i$ failing this test are ignored and discarded in the vertex program.

Note that this sequential LOD-range selection does not support any visibility culling on the CPU before submitting the very conservative range of points to the GPU. Furthermore, in [8] the entire LOD point hierarchy is stored in on-board video memory. For models exceeding the available video memory capacity, severe limitations may incur as the highly conservative range of points $\boldsymbol{p}_{lo}$ to $\boldsymbol{p}_{hi}$ must then be submitted over the AGP bus prior to being processed by the GPU.

## 5. Rendering

In this section, we present the most common and efficient point-rendering primitives that exploit hardware acceleration. We briefly outline the different primitives and techniques below and discuss the most significant components to be considered for implementation. The qualitative and quantitative performance differences of using various point primitives are more accurately given in the experimental results in Section 6.

### 5.1. Point primitives

The first and probably most important decision, with a direct impact on the display quality and performance, corresponds to the choice of primitive to render a point sample.

*Points*: Several approaches (i.e. [4,8,26]) have proposed to use simple OpenGL point primitives, which have the advantage of a low cost per primitive (3D position, color and normal if lighting is required). The primitive is drawn on screen as a fixed sized screen-aligned square, or rounded point with GL_POINT_SMOOTH enabled. Moreover, with the use of vertex and fragment programs and recent extensions, the size of points can be calculated on a per-primitive basis to reflect the actual projected screen size of a point sample, improving the visual quality by avoiding conservatively large points and holes between rendered points.

*Sprites*: Another choice for point primitives consists of using NV_SPRITES as promoted in [2] which can be considered as textured points. This primitive combines the simplicity of points for geometry submission to the graphics card with the flexibility of texture mapping with blending kernels to support smooth interpolation of discrete points and hence improved visual quality. As presented in [2], with some effort these sprites can be modified to represent surface-normal oriented disks, rendered with proper per-pixel depth values using graphics card programmability. Moreover, smooth blending can be achieved by computing a per-pixel $\alpha$-value in the fragment program.

*Triangles*: The third hardware supported class of primitives that can be used to render disks or ellipses are $\alpha$-texture mapped polygons. In [18], triangles are used with an $\alpha$-texture which segments an elliptical disk out of the planar triangle as desired (using $\alpha$-tests). In fact, the $\alpha$-texture can describe any desired shape, and moreover it can define a blending kernel mapped onto the point splat primitive. The system presented in [26] also allows the use of oriented solid polygonal shapes which tend to run significantly slower as they are made up of many vertices and not using a triangle with $\alpha$-texture. The use of more complex primitives than simple points has the advantage of added flexibility with respect to $\alpha$-texture mapping and blending kernels. In our experiments, we render elliptical disks as $\alpha$-textured triangles, thus rendering one triangle for each point primitive to be displayed.

In summary, each point can be represented and rendered in different ways: a screen-aligned square or round region of pixels which is referred to as *point* and rendered as an OpenGL point primitive; a circular disk, oriented perpendicular to the surface normal, referred to as *disk* and rendered as an NV_SPRITES point primitive; an elliptical disk, oriented perpendicular to the surface normal, referred to as *ellipse* or *elliptical disk* and rendered as $\alpha$-texture mapped triangle.

Note that considering float values for the position and normal, bytes for the color channels and extra parameters such as the point size (float), texture coordinates for the blending kernels, etc., we are dealing with point structures from 28 bytes to 40 bytes for the different approaches. Of course, quantization can be applied such as in [26] at the expense of on-the-fly decoding.

### 5.2. Rendering passes

Depending on the type of point primitives chosen for display, different rendering strategies are necessary. The key factor for this is if blending kernels are used on the points. If blending is performed, then it is necessary to ensure that only overlapping and front-facing points closest to the viewpoint are combined. If there exist

front-facing points farther away, occluded from the viewpoint, they must be ignored for blending.

This can be achieved by carefully selecting just the closest overlapping points. Commonly a two-pass $\varepsilon$–$z$ buffer rendering approach [2,11,17,18,25] works efficiently: the first pass initializes the $z$-buffer to generate a depth mask without rendering to the color buffer, and the second pass only performs $z$-buffer tests for each pixel fragment against some $\varepsilon$-offset of the $z$ value from the first pass.

Hence, when rendering opaque point primitives with no blending, only a single rendering pass is performed, but for blended primitives a two-pass approach is required. Although the first pass is less expensive than the second one, it still requires the geometry to be processed twice by the graphics hardware.

### 5.3. Normalization

The normalization problem appears only for blended primitives where it is difficult to guarantee that overlapping blending weights partition unity accurately on a per-pixel basis. Conservatively reduced blending values avoid overflow of blending weights in the $\alpha$-channel and achieve correctly (proportionally) blended, but underweighted colors in an intermediate image.

A simple solution to compensate for this intensity defect is to download the final frame buffer to the CPU and normalize on a per-pixel basis the weighted colors by the accumulated blending weights stored in the $\alpha$-channel which performs reasonably well. An improved solution using graphics hardware was presented in [19,20] using NVIDIA register combiners which have since been replaced by more efficient fragment programming techniques in [2,8,17,18].

## 6. Experiments

### 6.1. Test environment

To perform comparative and objective experiments, the different point-rendering primitives outlined in Section 5 have all been integrated into a single common view-dependent LOD rendering framework. In particular, our PBR system *Confetti* [18] has been extended to incorporate the following features for this survey:

- Multiple point-rendering primitives: resizeable GL points, orientable NV_SPRITES and triangles with smooth $\alpha$-texture blending.
- One-pass rendering algorithm for opaque resizeable points or disks, and two-pass rendering for smoothly blended point primitives.
- A fragment shader based per-pixel normalization algorithm for blended primitives.
- Addition of QSplat and Sequential Point Tree multiresolution data structures for view-dependent LOD rendering.

All experiments reported in this section were performed on a Dell Pentium4 PC with 2.4 GHz CPU, 1 GB of main memory and an NVIDIA GeForce 5900 GPU. The driver version used was 52.16 from NVIDIA. We have exhaustively tested 4 different models using our modified *Confetti* PBR system with different rendering pipelines. The original models used are Balljoint (137,062 points), Female (302,948 points), David head (2,000,646 points) and David 2 mm (4,129,534 points).

We observed that in many previous experiments it was not fully clear what the actual timing measures included in the reported tests. For example, it makes a huge difference to compute a point-rendering rate based on timed function calls and full object size, or based on *observed frame rate* and actual number of *visible* and *drawn* points. Also, it is not advisable to measure the pure rendering performance by simply timing the functions that issue any OpenGL calls each frame as these functions may return before the actual rendering has completed on the graphics card. Moreover, no information is usually given on the type and duration of the animation to perform the tests. In the experiments reported in this paper, we tried to be as specific as possible about actually observed real-world numbers which could be expected in other applications implementing similar techniques.

In order to have averaged results, each test with different parameters has been performed as a rotation around the object with a rotation step of $2^\circ$, and averaged over all frames. The viewer is located at varying distances from the object for the different tests. If not specified otherwise, the projected bounding box of the objects fills the entire viewport area of the $640 \times 480$ window.

### 6.2. Data and program complexity

In order to obtain maximum performance in all the rendering algorithms, we rely on several OpenGL performance tricks and extensions. At initialization time, the points of the complete hierarchy are stored in main memory in a single array. Then the indices of the selected data from the LOD stage are assembled into an integer array and a single rendering call is issued to the GPU using OpenGL vertex arrays, which minimizes the overhead due to excessive function calls for drawing individual primitives, as well as improves the AGP transfer. For the smaller models that use video memory to store the geometry, we use the extension ARB_vertex_buffer which performs as good or sometimes better than the NV_array_range extension.

The basic data type for each individual point primitive (GL_POINTS or NV_SPRITES) that is sent to the GPU consists of 3 floats for the point position, 3 floats for the point normal, 1 float for the splat radius and 3 bytes if color information is available, making a total of 40 bytes including padding. In the case of elliptical disks, we use $\alpha$-texture mapped triangles as primitives and, due to the nature of the OpenGL vertex arrays, each vertex requires to submit 3 floats for the position, 3 floats for the normal, 2 floats for the texture coordinates, 1 float for the point radius and 3 bytes for the color, making a total of 40 bytes per vertex and hence 120 bytes per elliptical primitive.

With respect to vertex and fragment shader used, we have mainly relied on the Cg language from NVIDIA and their Cg compiler version 1.2.1001 using the ARB profile for the shaders. It is well known that the compiled code can be further optimized manually by very skilled programmers, but that is not what the majority of users will do. Moreover, we have run the tests with per-vertex illumination and Gouraud shading enabled using two positional light sources that contribute considerably to the complexity of the shaders. Table 1 provides the amount of instructions and registers used by each approach in the different rendering passes.

### 6.3. Results

In the first experiment reported in Fig. 5, we compare the rendering performance of different rendering primitives. In this experiment we only measured the pure rendering time of the system, and thus how much time is needed to draw a number of points but without incorporating the time spent to assemble that data set before rendering. The measured rendering time $t_{\text{render}}$ includes the glutSwapBuffers() calls as only measuring glDrawElements() would not sufficiently account for the actual time spent by the graphics hardware to complete the rendering. The numbers in Fig. 5 are achieved by

Table 1
Instruction (and temporal register) counts of the vertex (VP) and fragment (FP) shaders for the different rendering

| Rendering primitives | First pass ($\varepsilon$–z buffering) | | Second pass (rendering) | | | Normalization |
|---|---|---|---|---|---|---|
| | VP | FP | VP (lights) | VP (no lights) | FP | FP |
| Points | — | — | 55 (7) | 14 (1) | — | — |
| Sprites | 20 (1) | 21 (3) | 60 (7) | 21 (1) | 24 (3) | 6 (4) |
| Triangles | 13 (2) | — | 51 (7) | 6 (0) | — | 6 (4) |
| Sequential points | — | — | 70 (7) | 32 (3) | — | — |
| Sequential sprites | 36 (3) | 21 (3) | 75 (7) | 37 (3) | 24 (3) | 6 (4) |

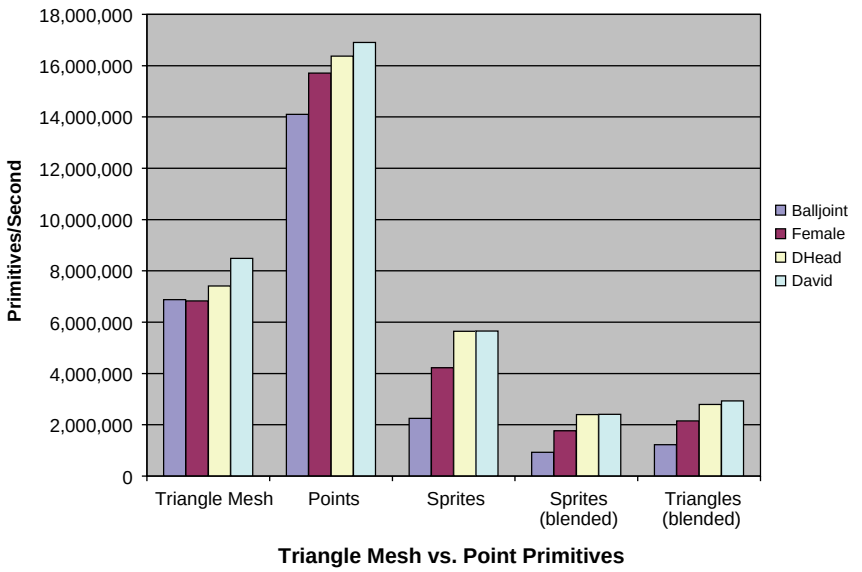


Fig. 5. Performance of triangle mesh rendering compared to point primitives.

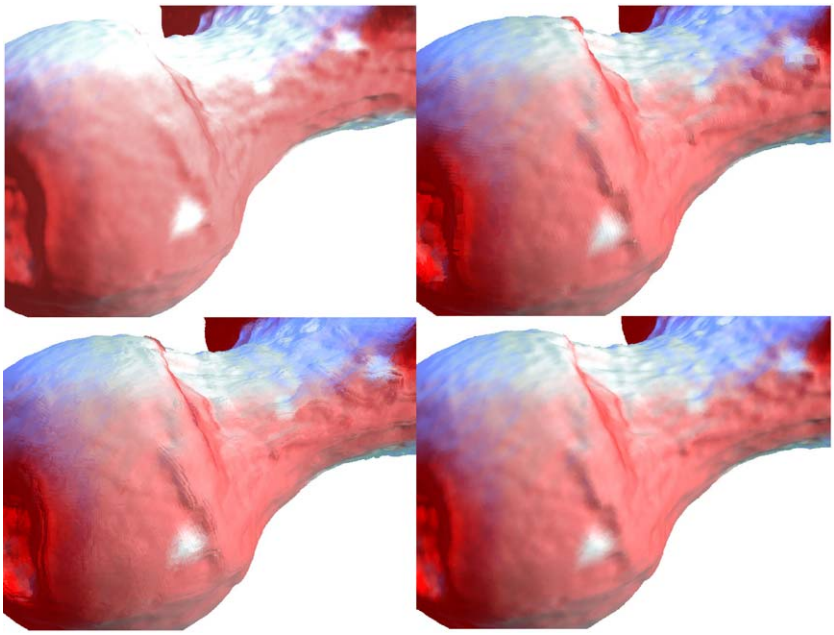

Fig. 6. Display results of rendering primitives (upper-left to lower-right): triangle mesh, resizeable points, surface-normal oriented disks, and blended surface-normal oriented elliptical disks.

accumulating the number of displayed points over all frames and dividing it by the accumulated $t_{\text{render}}$ time. No culling was performed on the CPU or the GPU for these tests and the geometry data were submitted as vertex arrays and list of indices to the graphics card.

Fig. 5 clearly indicates that simple point primitives such as resizeable GL_POINTS can provide significant performance advantages of polygonal rendering. This performance, however, comes at the expense of continuous interpolation between vertices which is automatically given for triangle meshes. Higher quality point-based rendering using surface-normal oriented disks or even smoothly blended primitives still performs lower than triangle meshes at this point. In Fig. 6, the differences in display behavior are shown between the different primitives. We observe that the simple resizeable points provide a good display quality, and this at a very high performance. The blended primitives show best-possible display quality fully comparable to polygonal rendering; however, some continuing improvements in performance are necessary to catch up with conventional polygonal meshes. While competitive in performance, the opaque surface-normal oriented disks do not provide a very good rendering quality for this model and are, in fact, visually less pleasing than the much faster simple points. Further point-rendering quality can be achieved by the use of phong splatting as introduced in Ref. [3].

Fig. 7 exhibits the antialiasing properties of the different point-rendering primitives. Clearly, the object-aligned (elliptical) disk primitives offer a better sampling in object space, and also in projected image space, of the surface than the screen-aligned resizeable points, which also accounts for fewer artifacts in visibility occlusion. Furthermore, disks (sprites) and ellipses (triangles) can be rendered with $\alpha$-texture which allows blending together neighboring and overlapping point primitives. Therefore, these primitives generally provide a superior image quality. Differences between circular and elliptical disks stem from the different surface-covering flexibility of the two primitives. Ellipses, in general, can adapt better to local sampling anisotropy and thus can result in better surface covering and rendering eventually. Note that while the blended primitives offer some object space and LOD-based antialiasing, they do not support a low-pass filtering in image space as the EWA splatting provides [25].

In Fig. 8, the performance measured in frames-per-second (FPS) is reported for the different LOD data structures and for three different viewing configurations. Resizeable points were used as rendering primitives for these tests and the frame rate was measured as the inverse of the cumulative times $t_{\text{frame}} = t_{\text{render}} + t_{\text{LOD}}$ of performing LOD selection and point rendering. Corresponding screen shots are given in Fig. 9. The zoomed-in view exhibits significant view-frustum culling. For the low-density point models Balljoint and Female, a large screen-space error tolerance was set in the zoomed-out view to allow for significant LOD simplification in that configuration. The octree data structure of *Confetti* [18],

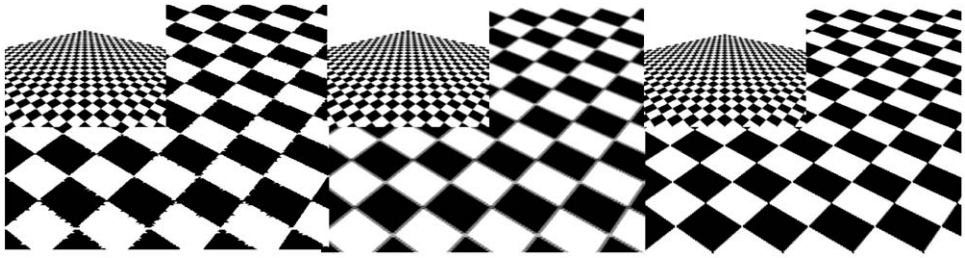

Fig. 7. Demonstration of antialiasing properties using resizeable points, blended surface-normal oriented disks, and blended surface-normal oriented elliptical disks (left to right).

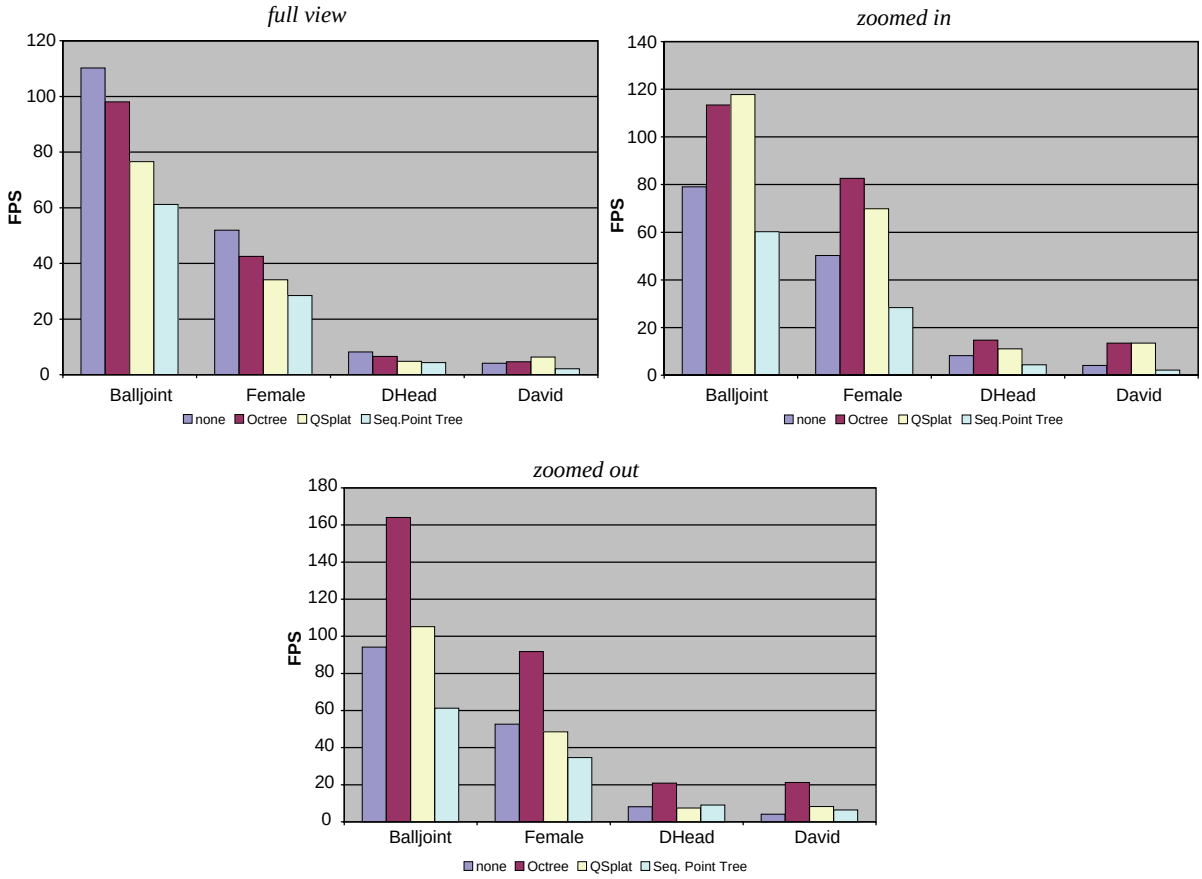


Fig. 8. Performance of different LOD approaches: no LOD, Confetti octree [18], QSplat octree [26], and Sequential Point Tree data structures [8].

the native QSplat data structure [26] as well as a re-implementation of Sequential Point Trees (SPTs) [8] were compared to a plain full-resolution rendering approach (labelled *none*).

From the performed experiments, one can clearly see the benefit of doing LOD simplification over rendering brute-force the full-resolution point set. Only in the full-view configuration for the smaller two models, the brute-force rendering showed notable benefits over doing a LOD traversal and rendering. It can also be observed that the *Confetti* octree generally performs better than the QSplat data structure, which can be attributed to its efficient LOD selection [16] and to the more costly on-the-fly decoding of attributes in QSplat [26].

When measuring observable frame rates, the SPTs were not competitive in the performed tests, which may stem from a number of different reasons. First off, SPTs


Fig. 9. Viewing configurations for LOD data structure performance tests.

Table 2
Timing break-up for the different LOD approaches, with total average frame time being the sum of LOD and render time. Number of points denotes amount of primitives actually sent to the graphics hardware

| LOD method | Zoomed out | | | Full view | | | Zoomed in | | |
|---|---|---|---|---|---|---|---|---|---|
| | LOD time (ms) | Render time (ms) | # Points | LOD time (ms) | Render time (ms) | # Points | LOD time (ms) | Render time (ms) | # Points |
| None | 0 | 122 | 2,000,606 | 0 | 122 | 2,000,606 | 0 | 122 | 2,000,606 |
| Octree | 35 | 13 | 148,158 | 86 | 65 | 1,042,306 | 40 | 28 | 443,791 |
| Qsplat | 91 | 44 | 687,051 | 121 | 86 | 1,097,956 | 58 | 32 | 493,014 |
| SPTs | 0 | 110 | 1,342,261 | 0 | 230 | 2,802,620 | 0 | 230 | 2,802,620 |

perform a very conservative (but fast) LOD selection on the main CPU. Additionally, SPTs do not allow simple and effective culling on the main CPU which, in conjunction with the conservative LOD selection, results in an extensively large number of points submitted to the graphics hardware for rendering. Finally, as the fine-grain LOD selection and culling is done on a per-vertex level in the graphics card via vertex programs, SPTs have significantly higher vertex processing costs. The LOD selection cost for SPTs is virtually zero for all configurations. However, the rendering time reached 110/230/230 ms per frame on average for the David head model for the different views as 1.3/2.8/2.8 M points have to be processed for final LOD selection, culling, visibility and rendering in the complex vertex program of the SPT. In contrast, the *Confetti* octree spends 35/86/40 ms to perform hierarchical visibility culling (cutting down 50% back-facing points and any parts that are outside the view frustum) and view-dependent LOD selection; and 13/65/28 ms for rendering. Thus the *Confetti* octree totals in a 48/151/68 ms per frame cost and hence an overall performance advantage of up to a factor of 3 over SPTs. Table 2 demonstrates the limiting conservative LOD selection and culling factors that are in effect for the SPT-based LOD selection approach, which from our experiences are the main reasons for the observed performance. Additional numerical results can be found in [27].

## 7. Conclusion

The contributions of this paper lie in a careful comparison of the performance of different point-rendering primitives as well as different point-based multiresolution hierarchies and view-dependent LOD selection algorithms. The experiments include realistic side-by-side evaluation of the compared approaches and techniques that can be anticipated by prospective users of such methods. Within a common framework, we show what the different implementations can offer in terms of rendering quality and performance. This allows a potential customer of point-based rendering technology to make informed decisions on what rendering primitives and LOD algorithms to consider for the particular point-rendering task at hand.

A main conclusion we would like to draw here is that the observable frame rate (FPS) is the ultimate performance measure, and not any isolated points-per-second rendering rate. In [27], we have demonstrated that simple point primitives can achieve mind-boggling 160 M point-rendering rates for some models. However, normalized for the observable frame rate, this generally reduces to a throughput of maybe 10 M points.

While not included numerically in this paper, initial comparisons to view-dependent LOD triangle mesh approaches have shown that their view-dependent LOD mesh generation is generally more costly than

simple point-based LOD selection, thus making PBR systems more amenable to dynamically changing LODs. We have also noticed that highly optimized LOD-mesh approaches often only work for well-behaving manifold triangle mesh data sets, which is often not the case for large complicated models. Hence, for generic surface data that may not have been preprocessed and cleaned, points provide a better alternative than meshes.

## Acknowledgements

We would like to thank the Stanford *3D Scanning Repository* and *Digital Michelangelo* projects as well as *Cyberware* for freely providing geometric models to the research community. We would also like to thank Mario Botsch and Leif Kobbelt for sharing some of their code, and we would like to thank the guest editors as well as many of the participants of the inaugural Symposium on Point-Based Graphics (in Zürich 2004) for their suggestions and discussions on this topic.

## References

[1] Adams B, Dutre P. Boolean operations on surfel-bounded solids using programmable graphics hardware. In: Proceedings of the symposium on point-based graphics. Eurographics; 2004. p. 19–24.

[2] Botsch M, Kobbelt L. High-quality point-based rendering on modern GPUs. In: Proceedings of the Pacific graphics 2003. IEEE, Computer Society Press; 2003. p. 335–43.

[3] Botsch M, Spernat M, Kobbelt L. Phong splatting. In: Proceedings of the symposium on point-based graphics. Eurographics; 2004. p. 25–32.

[4] Botsch M, Wiratanaya A, Kobbelt L. Efficient high quality rendering of point sampled geometry. In: Proceedings Eurographics workshop on rendering. 2002. p. 53–64.

[5] Chen B, Nguyen MX. POP: a hybrid point and polygon rendering system for large data. In: Proceedings IEEE visualization 2001, 2001. p. 45–52.

[6] Coconu L, Hege H-C. Hardware-oriented point-based rendering of complex scenes. In: Proceedings Eurographics workshop on rendering. 2002. p. 43–52.

[7] Cohen JD, Aliaga DG, Zhang W. Hybrid simplification: combining multi-resolution polygon and point rendering. In: Proceedings IEEE visualization 2001, 2001. p. 37–44.

[8] Dachsbacher C, Vogelgsang C, Stamminger M. Sequential point trees. In: Proceedings ACM SIGGRAPH 03. ACM Press; 2003. p. 657–62.

[9] Dey TK, Hudson J. PMR: point to mesh rendering, a feature-based approach. In: Proceedings IEEE visualization 2002. Computer Society Press; 2002. p. 155–62.

[10] Gross MH. Are points the better graphics primitives? Computer Graphics Forum 2001;20(3) Plenary Talk Eurographics 2001.

[11] Grossman JP, Dally WJ. Point sample rendering. In: Proceedings Eurographics rendering workshop 98. Eurographics; 1998. p. 181–92.

[12] Kalaiah A, Varshney A. Differential point rendering. In: Proceedings Eurographics Workshop on rendering techniques. Springer-Verlag; 2001. p. 139–50.

[13] Kalaiah A, Varshney A. Modeling and rendering points with local geometry. IEEE Transactions on Visualization and Computer Graphics 2003;9(1):30–42.

[14] Levoy M, Whitted T. The use of points as display primitives. Technical Report TR 85-022, Department of Computer Science, University of North Carolina at Chapel Hill, 1985.

[15] Mitra NJ, Nguyen A. Estimating surface normals in noisy point cloud data. In: Proceedings of the symposium on computational geometry. ACM; 2003. p. 322–8.

[16] Pajarola R. Efficient level-of-details for point based rendering. In: Proceedings IASTED international conference on computer graphics and imaging (CGIM 2003). 2003.

[17] Pajarola R, Sainz M, Guidotti P. Object-space point blending and splatting. ACM SIGGRAPH Sketches & Applications Catalogue; 2003.

[18] Pajarola R, Sainz M, Guidotti P. Confetti: object-space point blending and splatting. IEEE Transactions on Visualization and Computer Graphics 2004.

[19] Pajarola R, Sainz M, Meng Y. Depth-mesh objects: fast depth-image meshing and warping. Technical Report UCI-ICS-03-02, The School of Information and Computer Science, University of California Irvine, 2003.

[20] Pajarola R, Sainz M, Meng Y. DMesh: fast depth-image meshing and warping. International Journal of Image and Graphics (IJIG) 2004;4(4):1–29.

[21] Pauly M, Gross M. Spectral processing of point-sampled geometry. In: Proceedings ACM SIGGRAPH 2001. ACM Press; 2001, p. 379–86.

[22] Pauly M, Gross M, Kobbelt LP. Efficient simplification of point-sampled surfaces. In: Proceedings IEEE visualization 2002. Computer Society Press; 2002. p. 163–70.

[23] Pauly M, Keiser R, Kobbelt L, Gross M. Shape modeling with point-sampled geometry. In: Proceedings ACM SIGGRAPH 2003. ACM Press; 2003. p. 641–50.

[24] Pfister H, Zwicker M, van Baar J, Gross M. Surfels: surface elements as rendering primitives. In: Proceedings SIGGRAPH 2000. ACM SIGGRAPH; 2000. p. 335–42.

[25] Ren L, Pfister H, Zwicker M. Object space EWA surface splatting: a hardware accelerated approach to high quality point rendering. In: Proceedings EUROGRAPHICS 2002. pp. 461–70. (Also in Computer Graphics Forum 21(3)).

[26] Rusinkiewicz S, Levoy M. Qsplat: a multiresolution point rendering system for large meshes. In: Proceedings SIGGRAPH 2000. ACM SIGGRAPH; 2000. p. 343–52.

[27] Sainz M, Pajarola R, Lario R. Points reloaded: point-based rendering revisited. In: Proceedings symposium on point-based graphics. Eurographics; 2004. p. 121–8.

[28] Sainz M, Pajarola R, Susin A, Mercade A. A simple approach for point-based object capturing and rendering. IEEE Computer Graphics & Applications 2004;24(4):24–33.

[29] Zwicker M, Pfister H, van Baar J, Gross M. Surface splatting. In: Proceedings SIGGRAPH 2001. ACM SIGGRAPH 2001. p. 371–8.