Eurographics Symposium on Point-Based Graphics (2004)
M. Alexa, M. Gross, H. Pfister, S. Rusinkiewicz (Editors)

Points Reloaded: Point-Based Rendering Revisited

Miguel Sainz[†] Renato Pajarola[*] Roberto Lario[‡]

[†*]Computer Graphics Lab
Computer Science Department
University of California Irvine

[‡]Dpto. Arquitectura de
Computadores y Automática
Universidad Complutense Madrid

Abstract

The increasing popularity of points as rendering primitives has led to a variety of different rendering algorithms, and in particular the different implementations compare like apples to oranges. In this paper we revisit a number of recently developed point-based rendering implementations. We briefly summarize a few proposed hierarchical multiresolution point data structures and their advantages. Based on a common multiresolution framework we then describe and examine different hardware accelerated point rendering algorithms. Experimental results are given with respect to performance timing and rendering quality for the different approaches.

Categories and Subject Descriptiors (according to ACM CCS): I.3 Computer Graphics, I.3.5 Computational Geometry and Object Modeling, I.3.6 Methodology and Techniques

1. Introduction

Point-based surface representation and rendering techniques have recently become very popular [Gro01]. In fact, 3D points are the most simple and fundamental geometry-defining entities. Points as display primitives were considered early in [LW85], however, have only recently received increased attention.

Based on their fundamental simplicity, points have motivated a variety of research on topics such as shape modeling [PKKG03], object capturing [SPSM04], simplification [PGK02], processing [PG01, MN03], rendering (see Section 2), and hybrid point-polygon methods [CN01, CH02, CAZ01, DH02]. 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.

The different PBR methods that have been proposed have led to a variety of implementations that are as hard to compare as apples and oranges. In this paper we present an initial attempt to realistically compare different point-rendering algorithms by implementing several techniques within the same multiresolution modeling and rendering application framework. This allows a more objective evaluation of available point-rendering alternatives for quality and expected relative speed-up with respect to each other. Note that our experiments focus on the back-end of the point-rendering pipeline that, given a set of points to be displayed, performs visibility splatting and optional blending.

However, besides the back-end rendering also the multiresolution model and LOD selection algorithm have a significant influence on the overall display performance. While the rendering back-end is easily interchangeable between applications, the multiresolution model is often very specific and not as exchangeable. Therefore, we limit ourselves to explain qualitative advantages of a few general multiresolution representations, and concentrate quantitative experiments mainly on the rendering-back end.

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. The compared methods include a wide range of techniques from simple screen-parallel opaque squares to smoothly blended surface-aligned elliptical disks.

Figure 1: *David head model rendered in Confetti as smoothly blended depth-sprites at a rate of 2M points/sec and 2 frames/sec.*

2. Related Work

Since the re-discovery of points as rendering primitives in [GD98] a stream of new PBR algorithms have been proposed.

Highly space-efficient multiresolution point-hierarchies have been proposed in [RL00, BWK02] 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 [RPZ02, BK03, PSG04]. Optimized anisotropic texture sampling can be realized by elliptical splat primitives in object-space as proposed in [RPZ02, PSG04]. The main hierarchical models and hardware accelerated rendering techniques are summarized and evaluated in this paper.

Recent approaches such as [BK03] or [DVS03] address high-speed rendering of point data by exploiting hardware acceleration and on-board video memory as geometry cache. We compare the various methods for different model sizes, also with and without using video memory to cache point data on the graphics card.

The surface splatting technique introduced in [PZvBG00, ZPvBG01] and the differential points proposed in [KV01, KV03] offer high-quality surface rendering. The rendering back-end of surface splatting can be accelerated by hardware [RPZ02]. Differential points also offer fast rendering but do not offer interactive view-dependent LOD-rendering as of yet.

In this paper we focus on comparing the point splatting techniques of the most generic multiresolution point rendering systems [RL00], [BWK02], [DVS03] and [PSG04]. These can directly be applied to any large surface objects and work with the original input vertices of the surface.

3. Multiresolution Representation

3.1. Point Samples

The data sets considered in this paper are dense sets of surface point-samples organized in a multiresolution representation. The point samples $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n \in \mathbf{R}^3$ are 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 consists of attributes for spatial coordinates $\boldsymbol{p}$, normal orientation $\boldsymbol{n}$ and surface color $\boldsymbol{c}$. Furthermore, it is assumed that each point also contains information about its spatial extent in object-space. This *size* information, e.g. a bounding sphere radius r, specifies a circular or elliptical disk centered at $\boldsymbol{p}$ and perpendicular to $\boldsymbol{n}$. Other attributes optionally include a normal-cone semi-angle θ. For correct visibility the (elliptical) disks of points must cover the sampled object nicely without holes and thus overlap each other in object-space as shown in Figure 2. Elliptical disks e consists of major and minor axis directions $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ and their lengths.

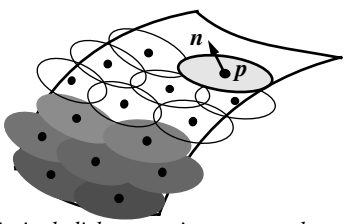

Figure 2: *Elliptical disks covering a smooth and curved 3D surface.*

3.2. Hierarchy

Besides [GD98, PZvBG00], which use specific re-sampling techniques to represent objects as points, most 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 [RPZ02, BWK02, BK03]. In [RL00] a midpoint-split kD-tree and in [Paj03, PSG04] an adaptive point-octree are used. The latter two offer data-adaptive hierarchies with fewer nodes than regular subdivision approaches (see also Figure 3).

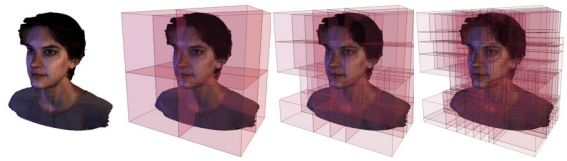

Figure 3: *Example hierarchical spatial subdivision of an adaptive point-octree.*

Basically, a 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 [RL00]. 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 5-6-5 bit and the normal-cone semi-angle θ 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 group of siblings sharing one pointer (index) to their list of consecutive child nodes.

In [BWK02] 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 2bytes and 1byte 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 4GB virtual memory addressing limit of 32bit systems. This is a significant benefit over methods with more complex node formats.

The LOD-hierarchy in [Paj03, PSG04] does not apply any quantization to point attributes $\boldsymbol{p}$, $\mathbf{n}$, r, θ or $\boldsymbol{c}$. In fact, it additionally keeps optional information do 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. However, this representation cannot compete in terms of memory cost with the methods outlined above. In this paper, this simple octree structure serves as the com-

mon LOD framework to compare the performance of the different point-rendering algorithms described in Section 5.

3.3. Sequential Point Trees

In [DVS03] a nested bounding sphere hierarchy (i.e. [RL00]) 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 is drawn is entirely delegated to the individual nodes. The so de-referenced nodes are then linearly sorted with respect to decreasing LOD importance. How this is used in the rendering stages is described in more detail in Section 4.3.

4. LOD Selection

4.1. View-dependent LOD Metric

The basic error metric of most view-dependent LOD methods is the perspective projection of an object-space 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 given as $\varepsilon = f \cdot A/d^2$, which is the perspective projected point sample extent, corrected by a factor f for the surface normal orientation if applicable.

For bounding-sphere hierarchies [RL00, BWK02, DVS03] the extent is basically $A = \pi \cdot r^2$ for a given bounding-sphere radius r. Note that in [DVS03] this area measure is further refined for nodes by approximating the actual occupancy of points in the subtree. For object-space elliptical disks as in [Paj03, PSG04] the extent is computed as $A = \pi \cdot r \cdot \kappa r$ with r being the major ellipse axis length and κ 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 $f = \boldsymbol{n} \cdot (\boldsymbol{v} - \boldsymbol{p})/|\boldsymbol{v} - \boldsymbol{p}|$. For non-leaf nodes in the LOD hierarchy, this should be corrected for the bounding normal-cone semi-angle θ (see [Paj03, PSG04]).

As outlined in Section 5 we compare three different point-rendering modes: (1) OpenGL point, (2) depth-sprite or (3) triangle based splatting primitives. Correspondingly we adjust the view-dependent error metric as follows:

Points: Renders a screen-aligned disk for each point. Thus the extent is computed as the disk size $A = \pi \cdot r^2$ 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 $A = \pi \cdot r^2$ with the bounding sphere defining the disk radius r. It also uses the orientation factor $f = \cos(\gamma - \theta)$, the cosine of the angle γ between the normal $\boldsymbol{n}$ and the view-direction $|\boldsymbol{v} - \boldsymbol{p}|$ minus the normal-cone semi-angle θ (and clamped appropriately).

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

Therefore, given a screen-space error tolerance τ and viewpoint $\boldsymbol{v}$, a point ($\boldsymbol{p}$, $\boldsymbol{n}$, r, θ, κ) can be rendered if $f \cdot A \cdot |\boldsymbol{v} - \boldsymbol{p}|^{-2} \le \tau$, and must be split into smaller 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 θ, is larger than 90°.

4.2. Hierarchical LOD

In hierarchical LOD and rendering systems, the basic view-dependent LOD-selection algorithm consists of a depth- or breadth-first traversal of the multiresolution hierarchy as illustrated in Figure 4. The basic traversal not only incorporates the view-dependent LOD-metric as outlined in Section 4.1 to decide wether a node is rendered or further refined, but also back-tracking based on view-frustum and back-face culling.

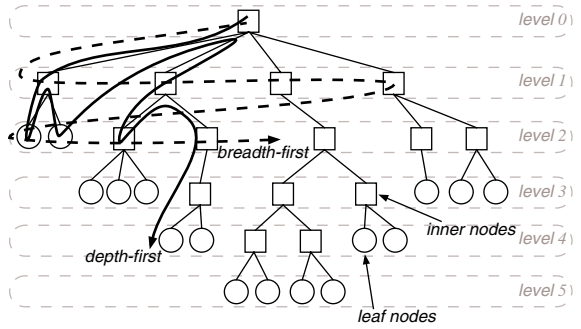

Figure 4: *Example LOD-hierarchy organization and traversal orders.*

In [RL00] a top-down traversal based on an error threshold τ 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.

Alternatively a depth- or breadth-first traversal of the LOD hierarchy is used in [BWK02]. Note that the traversal in this approach incorporates a fast piece-wise 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 [BWK02]. However, since GPU performance is increasingly faster than CPU, this approach may be slower on the CPU than leaving the transformation up to the GPU. 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 limits culling efficiency as the data has to be scanned at least up to the level and index of the highest-resolution point selected for rendering.

The explicit octree in [Paj03, PSG04] 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 Figure 4.

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 spatial extent 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 ε 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 [DVS03] 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 $rmin_h$, that may occasionally render nodes for which it was determined already to display the parent node, see also [DVS03]. Note that in the preprocess ε can be set to 1.0 to construct the sequential point trees. To determine all points to be rendered they are ordered according to *rmax*. For a given viewpoint $\boldsymbol{v}$ and error threshold τ, 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 $dmin = \sqrt{\tau} \cdot (|\boldsymbol{p}_0 - \boldsymbol{v}| + r_0)$ 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 Figure 5; with *lo* and *hi* being the smallest, respectively largest index for which $rmin_{lo} \le dmin$ and $rmax_{lo} \ge dmax$. The entire range of points from $\boldsymbol{p}_{lo}$ to $\boldsymbol{p}_{hi}$ is then processed by the graphics hardware. Using a vertex-program the actual per-point LOD-test $rmin_i \le \sqrt{\tau} \cdot |\boldsymbol{p}_i - \boldsymbol{v}| \wedge rmax_i \ge \sqrt{\tau} \cdot |\boldsymbol{p}_i - \boldsymbol{v}|$ is performed and points failing this test are ignored.

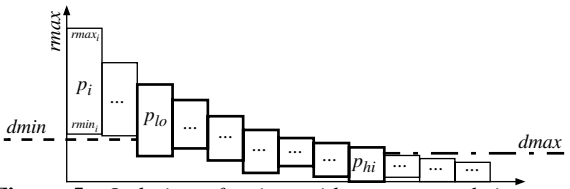

Figure 5: *Ordering of points with respect to their rmax values and selection of LOD-range.*

Note that this sequential LOD-range selection does not support any visibility culling on the CPU before submitting the entire conservative range of points to the GPU. Furthermore, in [DVS03] 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 techniques to successfully implement a point-splatting pipeline that exploits hardware acceleration. Although there are many different approaches in the literature, we have considered the most significant but still sufficiently different ones in our comparison. We briefly outline the different techniques below and discuss the most significant components to be considered when implementing a real-time point-renderer. The qualitative and quantitative performance differences of using various point-primitives are more accurately given in the experimental results 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. [RL00, DVS03, BWK02]) 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 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 be the actual screen-projected 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 [BK03] which can be considered 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 visually higher-quality renderings. With NV_SPRITES a single coordinate is specified per point and the graphics card rasterization unit generates a quadrilateral with texture coordinates. As presented in [BK03], with some work 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 α-value in the fragment program.

Triangles: The third hardware supported class of primitives are triangles and polygons. In [RPZ02] and [PSG04] polygonal faces are used with an α-texture which provides a disk or elliptical shape as desired (using α-tests). In fact, the α-texture can describe any desired blending kernel mapped onto the elliptical point splat primitive. The system presented in [RL00] also allows the use of oriented solid polygonal disks which tend to run significantly slower as they are made of many vertices. The use of more complex primitives than simple points has the advantage that α-texture mapping and blending kernels can be used to obtain smoothly blended points and more realistic rendered surfaces.

In summary, each point can be represented and rendered in different ways. However, 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 [RL00] 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 the front-facing points closest to the viewpoint are combined. If there exist front-facing points farther away, occluded from the viewpoint, it must be assured that these are not blended with the closest visible points.

This can be achieved by carefully selecting just the closest overlapping points. Commonly a two-pass ε-*z*-buffer rendering approach [RPZ02, PSG03, BK03, PSG04] 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 frag-

ment against some ε offset of the z value from the first pass.

Hence when rendering opaque point primitives with no blending, only a single pass over the data is performed, but when polygons or sprites are used with smooth blending 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 mainly for object-based blended splatting techniques where it is difficult to guarantee that overlapping blending kernel weights, once projected to image space, partition unity accurately on a per-pixel basis. Conservatively low blending kernels avoid overflow of blending weights in the α-channel and achieve correctly (proportionally) blended, but under-weighted colors in an intermediate image.

A first attempt to compensate this color defect is to download the final frame buffer to the CPU and normalize on a per-pixel basis the weighted color using the accumulated blending weights stored in the α-channel. Although this may seem to be a low-end approach, it performs fairly well and the cost of the normalization is significantly less than any of the other steps in the point-rendering pipeline.

Nevertheless, this can be solved more efficiently using the graphics hardware. In [PSM03, PSM04] a similar problem was addressed using the NVIDIA register combiners to perform the normalization. This has since been replaced by more efficient fragment programming techniques in [PSG03, PSG04] and independently in [BK03, DVS03].

5.4. Geometry Caching

The next set of optimizations available to achieve high frame rates deals with how to compact the data for an optimized submission to the graphics card. The common trend is to assemble an array of points, and use it in indexed mode by submitting indices of the selected points. This presents a large overall increase in performance as buffer traffic is reduced compared to submission of individual vertices.

A further optimization is to store the original geometry, or the necessary parts of it, in video memory (using NV_array_range or ARB_vertex_buffer extensions) and just submit the array of indices over the bus each frame. This approach, followed by [RPZ02, BK03, DVS03], increases the performance significantly at the expense of video memory. Furthermore, there are two limitations on current graphics card, which are: (1) addressable cached elements, i.e. in NVIDIA cards, is limited to a million per list; and (2) there is a limit of allocatable memory – 32MB on our test system – which depending on the point storage size can hold less than one million points. Thus in prior approaches the entire model could not exceed a complexity limited by these constraints.

We have successfully overcome this size limitation by adding a *cached geometry manager* (CGM) [LPT04] that updates the content of graphics card video memory dynamically for each frame with the currently visible points. This exploits coherence between consecutive frames as the set of visible points changes minimally. For fly-bys around point-objects this turns out to be very efficient and we have been able to render models larger than double or more the amount of available video memory. The CGM follows a least-recently-used (LRU) strategy to replace cached data. Once the LOD traversal has compiled a list of visible points, the CGM updates the cache contents to accommodate for any new points and modifies the index list to the proper indices. Finally these indices are sent to the graphics card as a regular indexed array.

6. Experiments

6.1. LOD-Framework and Environment

To perform comparative and objective experiments, the different point rendering techniques outlined in Section 5 have all been integrated into a single common view-dependent LOD rendering framework. In particular, our point rendering system *Confetti* [PSG04] has been extended to incorporate the following features:

- Rendering primitives: resizeable GL points, orientable NV_SPRITES and triangles with smooth α-texture blending.
- One-pass rendering algorithm for opaque resizeable points, and a two-pass rendering for blended primitives.
- A fragment shader based per-pixel normalization algorithm for blended primitives.
- All geometry is using OpenGL arrays and can use full caching in video memory, partial caching using an LRU cache manager or non-cached objects.
- Use of sequential point tree multiresolution representation for rendering.

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

6.2. Meaningful Experiments

We observed that in many previous publications 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 adequate 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 using similar techniques.

In order to have statistically meaningful tests, each test with different parameters has been performed as a rotation around the object for 1000 frames and with a rotation step of 1 degree, and averaged over time. The viewer is located so the object occupies around 50% of the viewport area for the 512x512 window resolution (see Figure 9). We have performed the tests with two different resolutions to investigate the fill-rate bottleneck of the different configurations. Additionally, we have tested 2 different LOD thresholds.

Model	LOD rep	Pri	0.0% threshold									0.0001% threshold								
			LOD (ms)	Cache (ms)	Render time (ms)		FPS		Avg splats	#P * 1e6 / SEC (PPS)		LOD (ms)	Cache (ms)	Render time (ms)		FPS		Avg Splats	#P * 1e6 / SEC (PPS)	
					NC	C	NC	C		NC	C			NC	C	NC	C		NC	C
balljoint	Hier	P	6	*0*	3	0.5	105	151	64490	21 (6.8)	129 (9.8)	5	*0*	2	0.4	137	184	33759	17 (4.6)	84 (6.2)
		S	6	*0*	45	45	19	19	64490	1.4 (1.3)	1.4(1.3)	5	*0*	37	37	23.9	23.9	33759	0.9 (0.8)	0.9 (0.8)
		E	13	*0*	36	25	20.2	25.8	64477	1.8 (1.3)	2.6(1.7)	11	*0*	33	29	22.4	24.9	33718	1.0 (0.7)	1.2 (0.8)
	Seq	P	0	*0*	16	16	63.1	63.4	192031	12 (12)	12 (12)	0	*0*	15	15	65.1	64.8	188034	12 (12)	12 (12)
		S	0	*0*	63	63	15.7	15.7	192031	3.0 (3.0)	3.0 (3.0)	0	*0*	60	60	16.5	16.5	188034	3.1 (3.1)	3.1 (3.1)
	QSplat	P			14		78.1		73718	(5.4)										
		D			24		40.6		74612	(3.1)										
female	Hier	P	16	*0*	8	1	42.3	63.1	159959	20 (6.8)	160(10.1)	11	*0*	4	0.6	64.3	83.2	65173	16 (4.2)	109 (5.4)
		S	15	*0*	78	74	10.7	11.2	159959	2.0(1.7)	2.1 (1.8)	11	*0*	48	48	16.7	16.6	65173	1.4 (1.1)	1.4 (1.1)
		E	33	*0*	66	33	10.1	8.5	159890	2.4 (1.6)	4.8 (1.4)	25	*0*	57	35	12.1	11.2	65161	1.1 (0.8)	1.9 (1.0)
	Seq	P	0	*0*	34	34	28.9	29.1	424210	12 (12)	12 (12)	0	*0*	32	32	30.8	30.8	399546	12 (12)	12 (12)
		S	0	*0*	123	122	8.2	8.2	424210	3.4 (3.5)	3.4 (3.5)	0	*0*	104	104	9.6	9.6	399546	3.8 (3.8)	3.8 (3.8)
	QSplat	P			27		36.5		171114	(6.2)										
		D			59		16.9		189541	(3.2)										
david head	Hier	P	95	135	50	36	6.8	3.7	841696	16 (5.7)	23 (3.2)	17	12	4	0.6	46.6	33.5	63001	16 (2.9)	105 (2.1)
		S	95	155	370	317	2.1	1.7	841696	2.3 (1.8)	2.6 (1.5)	17	12	35	18	18.9	21.1	63001	1.8 (1.2)	3.5 (1.3)
		E	122	-	139	-	3.8	-	275192	6.0 (1.0)	-	42	-	72	-	8.7	-	62996	0.8 (0.5)	-
	Seq	P	0	-	119	-	8.3	-	1457280	12 (12)	-	0	-	39	-	25.4	-	485930	12 (12)	-
		S	0	-	259	-	3.8	-	1457280	5.6 (5.6)	-	0	-	106	-	9.4	-	485930	4.6 (4.6)	-
	QSplat	P			220		4.5		1092577	(4.9)										
		D			375		2.6		1068218	(2.8)										
david 2mm	Hier	P	98	58	27	14	7.9	5.8	476618	18 (3.8)	34 (2.8)	7	5	2	0.5	112	82.2	26407	13 (2.9)	52 (2.2)
		S	98	80	205	153	3.3	3.0	476618	2.3 (1.6)	3.1 (1.4)	7	5	13	8	49.9	49.8	26407	2.0 (1.3)	3.3 (1.3)
		E	151	-	148	-	3.3	-	275516	1.7 (0.9)	-	19	-	33	-	19.3	-	26408	0.8 (0.5)	-
	Seq	P	0	-	463	-	2.1	-	5757742	12 (12)	-	0	-	10	-	98.6	-	122826	12 (12)	-
		S	0	-	1200	-	0.8	-	5757742	4.8 (4.7)	-	0	-	29	-	34.5	-	122826	4.2 (4.2)	-
	QSplat	P			73		13.7		585716	(8.0)										
		D			171		5.8		571793	(3.3)										

Table 1: *Averaged timing experiments on LOD selection, dynamic caching and rendering (in seconds); observed frames per second; average number of points rendered each frame; and points rendered per second (in millions); with (C) and without (NC) caching vertices in video memory; for a 0.0% and 0.0001% viewport are screen-space error tolerance. Models are represented as octree hierarchy, sequential point trees or QSplat format; and rendered as OpenGL points (P), depth-sprites (S), oriented ellipses (E) or oriented disks (D).*

6.3. Results

In Table 1 we report the various results we measured in our experiments. As the CPU is performing these tasks exclusively, the *LOD* selection and *cache* manager times are obtained by timing the appropriate function calls. The *render* time, however, was measured as the difference $t_{render} = t_{frame} - (t_{LOD} + t_{cache})$ between the real frame-to-frame time (from one glutSwapBuffers to another glutSwapBuffers function call) and the sum of *LOD* and *cache* times. While slightly conservative, as it includes a few minor setup functions, it is much more realistic than simply timing the functions that perform the OpenGL calls (i.e. issuing glDrawElements calls costs nothing but surely takes some time on the graphics card to complete the rendering). The frame-per-second *FPS* was obtained by measuring the real time difference between two separate frames (i.e. from glutSwapBuffers to glutSwapBuffers).

The *avg splats* reports the average number of actually selected, visible and drawn points per frame (i.e. not including any culled points). The *#points/sec* (in millions) (*PPS*) reports the pure point rendering rate as *avg splats* divided by *render* time (thus measuring the splat primitive performance by not including the *Confetti* system's LOD selection cost), as well as the overall splatting performance in parenthesis (*Confetti*'s overall system performance).

All measurements are divided into running the system without geometry cache (NC) and caching enabled (C).

The different models were tested using three different multiresolution representations and several point splatting primitives. We used a simple (linearized) octree hierarchy (Hier) and sequential point trees (Seq) within *Confetti*, as well as the genuine QSplat system [RL00]. The different splatting primitives encompassed OpenGL points (P), depth-sprites (S), oriented elliptical disks (E) and oriented circular polygonal disks (D).

From Table 1 we can observe a number of expected but also other interesting results. Practically, the most important measure is the achievable frame rate. We can observe that (1) points as anticipated are significantly faster than sprites and ellipses, (2) blended depth-sprites are slightly slower in most cases than the α-textured triangles (ellipses), (3) the cache only improves rendering when the entire model fits into video memory, and (4) in most cases the per-point LOD selection and rendering wins over the sequential point trees (which are only faster for the David head model).

With respect to (3) we have to note here that the test machine has an extremely fast graphics card which makes any complex and dynamic cache use infeasible. However, on medium or slow graphics cards we can expect that clever dynamic cache usage may indeed cause an overall improvement of the rendering speed.

With respect to (4) we observe that the LOD selection of sequential point trees is virtually zero, however, a significantly larger number of points is submitted to the GPU which in turn reduces the observed frame rate considerably.

We can also see that the pure OpenGL points/sec rendering rate is faster for the per-point LOD selection and rendering algorithms than for the sequential point trees. This makes sense if one considers that the sequential point trees perform a more complex vertex program on each processed point. Using depth-sprites, the result turns back in favor of the sequential point trees for the rendering rate (but not for the observable frame rate at low error tolerance).

Another observation is that α-textured triangles representing elliptical disks combined with per-point LOD selection and rendering are competitive in raw point rendering speed with sequential point tree depth-sprites, and even superior in the overall frame rate at low error tolerance.

In Figures 6 and 7 we can see the different rendering quality achievable in *Confetti* with resizeable points or blended depth-sprites and triangles. In this test, the α-textured triangles suffer from a quantization effect of very low α-blending weights which are summed together and then normalized, which in the end causes the observed artifacts. Figure 8 shows the different rendering quality offered by *QSplat* by points or oriented disks.

7. Conclusion

The contributions of this paper lie in an overview and comparison of the most recent point-based multiresolution hierarchies and view-dependent LOD selection algorithms, and exhaustive experiments and realistic side-by-side evaluation of different hardware accelerated point-rendering techniques.

It shows, within one single common application framework, what the different implementations can offer in terms of rendering quality and performance. This allows a prospective customer of point-based rendering technology to make informed decisions on what algorithms to consider for the particular point-rendering task at hand.

A first 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 (PPS) rendering rate. For example, hierarchical point rendering achieves a mind boggling 160M pps rendering rate for female. However, normalized for the observable frame rate this reduces actually to a realistic 10M pps throughput. Also notice that sequential point trees exhibit a whopping 12M PPS throughput, winning over the 10M of hierarchical LOD selection. However, the frame rate of 63 FPS is double for hierarchical compared to the 29 FPS of sequential points.

While the pure PPS rate is generally high for sequential point trees they only work efficiently in FPS if the object(s) fits in cache and are viewed at a far distance (when many nodes are cut-off). This is due to the complex GPU vertex programs, overly conservative number of points processed and no visibility culling.

Another conclusion is that simple opaque points are indeed very fast, and they also offer good rendering quality for most interactive display applications.

Caching on the other hand, has to be improved for the dynamic case when the object data does not fully fit in video memory.

8. 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.

9. References

[BK03] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern GPUs. In *Proceedings Pacific Graphics 2003*, pages 335–343. IEEE, Computer Society Press, 2003.

[BWK02] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proceedings Eurographics Workshop on Rendering*, pages 53–64, 2002.

[CAZ01] Jonathan D. Cohen, Daniel G. Aliaga, and Weiqiang Zhang. Hybrid simplification: Combining multi-resolution polygon and point rendering. In *Proceedings IEEE Visualization 2001*, pages 37–44, 2001.

[CH02] Liviu Coconu and Hans-Christian Hege. Hardware-oriented point-based rendering of complex scenes. In *Proceedings Eurographics Workshop on Rendering*, pages 43–52, 2002.

[CN01] Baoquan Chen and Minh Xuan Nguyen. POP: A hybrid point and polygon rendering system for large data. In *Proceedings IEEE Visualization 2001*, pages 45–52, 2001.

[DH02] Tamal K. Dey and James Hudson. PMR: Point to mesh rendering, a feature-based approach. In *Proceedings IEEE Visualization 2002*, pages 155–162. Computer Society Press, 2002.

[DVS03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. In *Proceedings ACM SIGGRAPH 03*, pages 657–662. ACM Press, 2003.

[GD98] J.P. Grossman and William J. Dally. Point sample rendering. In *Proceedings Eurographics Rendering Workshop 98*, pages 181–192. Eurographics, 1998.

[Gro01] Markus H. Gross. Are points the better graphics primitives? Computer Graphics Forum 20(3), 2001. Plenary Talk Eurographics 2001.

[KV01] Aravind Kalaiah and Amitabh Varshney. Differential point rendering. In *Proceedings Rendering Techniques*, pages –. Springer-Verlag, 2001.

[KV03] Aravind Kalaiah and Amitabh Varshney. Modeling and rendering points with local geometry. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):30–42, January-March 2003.

[LPT04] Roberto Lario, Renato Pajarola, and Francisco Tirado. Cached geometry manager for view-dependent lod rendering. Technical Report UCI-ICS-04-07, Department of Computer Science, University of California Irvine, 2004.

[LW85] Marc Levoy and Turner Whitted. The use of points as display primitives. Technical Report TR 85-022, Department of Computer Science, University of North Carolina at Chapel Hill, 1985.

[MN03] Niloy J. Mitra and An Nguyen. Estimating surface normals in noisy point cloud data. In *Symposium on Computational Geometry*, pages 322–328. ACM, 2003.

[Paj03] Renato Pajarola. Efficient level-of-details for point based rendering. In *Proceedings IASTED Invernational Conference on Computer Graphics and Imaging (CGIM 2003)*, 2003.

[PG01] Mark Pauly and Markus Gross. Spectral processing of point-sampled geometry. In *Proceedings ACM SIGGRAPH 2001*, pages 379–386. ACM Press, 2001.

[PGK02] Mark Pauly, Markus Gross, and Leif P. Kobbelt. Efficient simplification of point-sampled surfaces. In *Proceedings*

IEEE Visualization 2002, pages 163–170. Computer Society Press, 2002.

[PKKG03] Mark Pauly, Richard Keiser, Leif Kobbelt, and Markus Gross. Shape modeling with point-sampled geometry. In *Proceedings ACM SIGGRAPH 2003*, pages 641–650. ACM Press, 2003.

[PSG03] Renato Pajarola, Miguel Sainz, and Patrick Guidotti. Object-space point blending and splatting. In *ACM SIGGRAPH Sketches & Applications Catalogue*, 2003.

[PSG04] Renato Pajarola, Miguel Sainz, and Patrick Guidotti. Confetti: Object-space point blending and splatting. *IEEE Transactions on Visualization and Computer Graphics*, pages –, 2004.

[PSM03] Renato Pajarola, Miguel Sainz, and Yu Meng. 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.

[PSM04] Renato Pajarola, Miguel Sainz, and Yu Meng. DMesh: Fast depth-image meshing and warping. *International Journal of Image and Graphics (IJIG)*, pages –, 2004.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings SIGGRAPH 2000*, pages 335–342. ACM SIGGRAPH, 2000.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings SIGGRAPH 2000*, pages 343–352. ACM SIGGRAPH, 2000.

[RPZ02] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings EUROGRAPHICS 2002*, pages –, 2002. also in Computer Graphics Forum 21(3).

[SPSM04] Miguel Sainz, Renato Pajarola, Antonio Susin, and Albert Mercade. SPOC: Simple point-based object capturing. *IEEE Computer Graphics & Applications*, pages –, July-August 2004.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *Proceedings SIGGRAPH 2001*, pages 371–378. ACM SIGGRAPH, 2001.

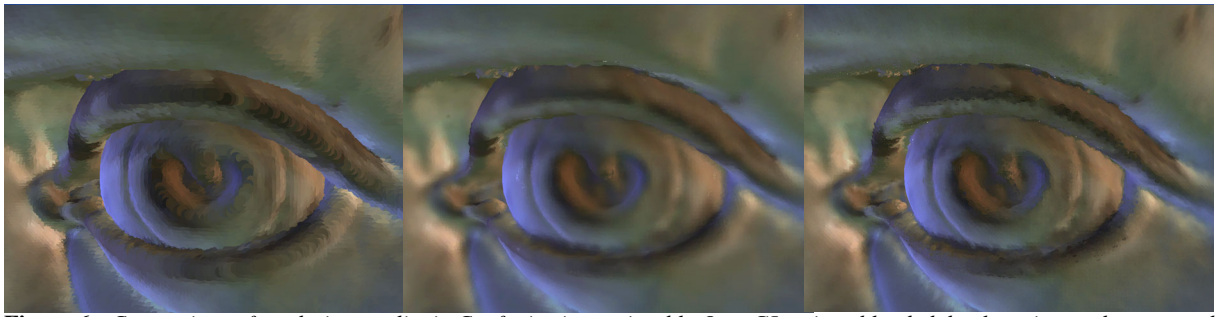

Figure 6: *Comparison of rendering quality in Confetti using resizeable OpenGL points, blended depth-sprites and α-textured triangles (left to right).*

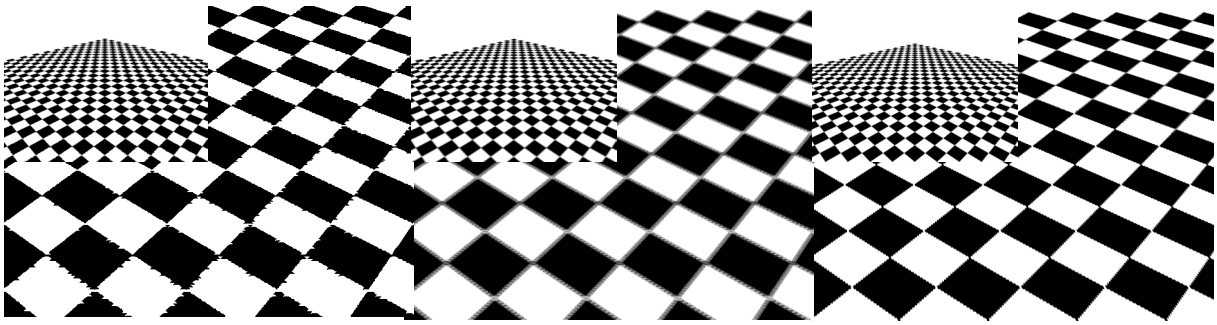

Figure 7: *Demonstration of antialiasing properties using resizeable OpenGL points, blended depth-sprites and α-textured triangles (left to right).*

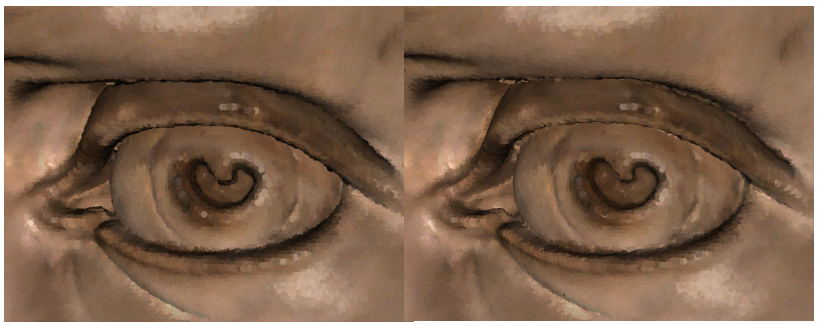

Figure 8: *Comparison of rendering quality in QSplat using OpenGL points and oriented disks (left to right).*

Figure 9: *Viewport configuration of experimental tests.*