Thirteenth Eurographics Workshop on Rendering (2002)
P. Debevec and S. Gibson (Editors)

Hardware-Accelerated Point-Based Rendering of Complex Scenes

Liviu Coconu and Hans-Christian Hege

Zuse Institute Berlin (ZIB), Germany

Abstract

High quality point rendering methods have been developed in the last years. A common drawback of these approaches is the lack of hardware support. We propose a novel point rendering technique that yields good image quality while fully making use of hardware acceleration.
Previous research revealed various advantages and drawbacks of point rendering over traditional rendering. Thus, a guideline in our algorithm design has been to allow both primitive types simultaneously and dynamically choose the best suited for rendering. An octree-based spatial representation, containing both triangles and sampled points, is used for level-of-detail and visibility calculations. Points in each block are stored in a generalized layered depth image. McMillan's algorithm is extended and hierarchically applied in the octree to warp overlapping Gaussian fuzzy splats in occlusion-compatible order and hence z-buffer tests are avoided. We show how to use off-the-shelf hardware to draw elliptical Gaussian splats oriented according to normals and to perform texture filtering. The result is a hybrid polygon-point system with increased efficiency compared to previous approaches.

1. Introduction

Although graphics hardware technology is one of the most rapidly advancing areas in computer industry, the complexity of models and scenes to be rendered seems to grow even faster. Applications aiming at interactive rendering of large virtual environments or extremely complex models must deal with the problem of unnecessarily processing large amounts of geometry that only cover a few screen pixels. Geometrical complexity should be adapted to the object's screen size.

Considerable research effort has been devoted in the past years to this subject, resulting in a variety of approaches, most of them being based on level-of-detail (LOD) control. One intensively investigated approach is mesh simplification. An appealing alternative to pure polygon based rendering turned out to be the point rendering paradigm. Representing object surfaces as sets of points without connectivity allows for easier simplification and generation of LOD representations. Recent papers have evidenced advantages of points for fast rendering of very complex models [19]. Others [17, 25] have shown that point models also allow for high quality rendering. While current hardware is highly tailored for polygon rendering, there is unfortunately only limited support for point primitives. For this reason, rendering methods offering high quality results must be implemented in software [7, 17, 25].

Point-based rendering can be more efficient than traditional rendering for complex-shaped models if triangles occupy a small screen area. If the projected screen area grows, traditional rendering becomes more efficient. The idea of combining the two paradigms has been investigated [23, 4, 2], resulting in systems that benefit from the advantages of both primitives and a wide range of available LOD representations. This research clearly shows that such systems could greatly benefit from efficient, but also high quality, point rendering.

The current paper is the result of an analysis whether is possible to use hardware acceleration for point rendering such that image quality similar to previous techniques is achieved while rendering speed is boosted. Our main contribution is a novel rendering algorithm based on fuzzy elliptical Gaussian splats, which can be efficiently rendered using recently introduced hardware features (point sprites, programmable geometry and texture pipelines). The main challenges are to correctly render overlapping fuzzy splats on z-buffer hardware in one rendering step and to perform texture filtering. We ensure correct blending order by using a per-object octree representation similar to a LDI tree [21, 3] which we call 'loose' LDI tree, constructed in a pre-process. We then extend McMillan's algorithm [15, 16, 14] to warp this hierarchy of overlapping splats in approximate back-to-front order and discuss the limitations and problems that may occur. Another issue that we address is the integration of point and triangles primitives in one rendering system. We record both points and triangles in the LOD hierarchy. During rendering, we switch to triangles if the point density is too low to ensure a hole-free surface. This leads to the situation in which parts of one object are rendered with points and others with triangles.

We use hardware also to perform mipmap texture filtering similar to Pfister et al. [17], and to draw elliptical Gaussians oriented by the per-point normal with low overhead. This results in similar image quality, but uses hardware acceleration and thus saves CPU resources and allows CPU/GPU parallelism.

The method is applicable to any 3D geometric model which can be decomposed into triangles. It supports instance sharing, which makes it particularly attractive for demanding applications like urban walkthroughs and interactive landscape visualization. The hybrid approach ensures efficient rendering of both magnified and minified objects by employing in each case the best suited rendering primitive. A wide range of LOD representations is available, which all benefit from hardware acceleration and thus ensure a good workload distribution between host processor and graphics hardware. At the same time, the simple and robust point rendering algorithm guarantees good image quality also for point primitives. Other types of drawing primitives could also be integrated, as well as scene occlusion culling using the already computed occlusion-compatible ordering.

2. Related Work

Over the recent past years, there have been many papers on point- or image-based rendering. The pioneering paper of Levoy and Whitted [10] proposes the use of points to model and render 3D continuous surfaces with textures. Splatting with fuzzy points and an adapted A-Buffer algorithm implemented in software are used for image reconstruction.

Grossman and Dally [6, 7] propose an object representation consisting of a dense set of surface point samples which contain color, depth and normal information. Max [13] and Gortler [5] represent scenes as collections of parallel projections with multiple depths layers – so called layered depth images (LDI). Shade et al. [21] used LDIs as intermediate representation and developed a basic method for rendering these, employing an ordering algorithm proposed by McMillan [15, 16, 14] to splat points in back-to-front order as well as incremental warping.

Several extensions of the LDI concept have been proposed. Lischinski [12] introduce the so-called layered depth cube, consisting of three orthogonal LDIs. Chang [3] introduced the LDI tree combining a hierarchical space partitioning scheme with the LDI concept. During rendering the LDI tree is traversed to the level that matches the output resolution.

Rusinkiewicz and Levoy [19] combine a multi-resolution hierarchy based on bounding spheres with a point rendering system. They mention fuzzy splatting as an alternative for better reconstruction and implemented it using multi-pass rendering.

Pfister et al. [17] perform sampling of geometrical complex textured objects into LDIs from 3 orthogonal views and create a spatial octree-based representation called LDC tree. Optionally, on each level the 3 LDIs are merged into a single one and the duplicated points are removed. The image reconstruction relies on so-called visibility splatting for removing invisible points and detecting holes, which are then filled using different reconstruction kernels to guarantee high quality texture filtering. The goal is the development of a point rendering pipeline which can be easily implemented in hardware.

Wand et al. [23] proposed a splatting approach combined with level of detail control to render impressively complex scenes. They use a variable number of random points sampled on the fly and rendered in sampling order. However, the texture filtering problem is not specifically addressed and the splats are opaque squares.

Techniques specifically aiming at high quality point rendering are presented by Zwicker et al. [25] and Schaufler et al. [20]. Zwicker et al. [25] extend the resampling framework of Heckbert [9] to irregular point grids to filter high quality textures.

Recent papers[4, 2] investigate more closely the integration of point and geometry rendering in one system and propose data structures which allow fine-grained transitions between different primitives driven by each primitive's rendering costs. This would ideally result in a totally balanced system, but there is also an overhead associated with primitive selection. Stamminger and Drettakis [22] propose an elegant and efficient sampling scheme for complex and procedural objects. A very recent paper of Ren et al. [18] proposes a hardware-accelerated, multi-pass implementation of EWA surface splatting [25].

Our rendering method was inspired by several of these techniques , especially the work of Pfister et al.[17]. In contrast, we specifically aim at efficient usage of existing graphics hardware and our main target are interactive applications like virtual walkthroughs.

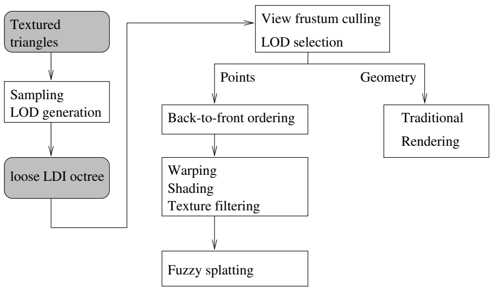

Figure 1: *Schematical overview of the algorithm.*

3. Method Overview

The proposed method consists of the following steps (Fig. 1): in a preprocessing step, an octree-based LOD hierarchy containing the original triangles and sampled points is built. During rendering, this spatial structure is used to select the level of detail and draw the scene parts (i.e. octree blocks) in appropriate resolution, using either triangles or points. The point rendering technique will be the main focus of the rest of the paper.

4. Preprocessing

The off-line preprocessing phase of the algorithm takes an arbitrary triangulated model as input, chooses a set of sample points on the model's surface and computes a LOD hierarchy of both points and triangles.

Conceptually, an object is sampled from three orthogonal view directions corresponding to the axes of its bounding box. Sampling and LOD generation from points is similar in many respects to Pfister et al.[17]. Thus in the following we will only note the differences.

Instead of a ray tracer, our current implementation uses a geometry rasterizer for point sampling. Each triangle contained in a leaf octree node is orthogonally projected on the bounding cube face and rasterized at the fixed grid resolution. Position, normal, material and texture information are computed by linear interpolation between the vertices and recorded for each point sample. This simple rasterization technique results in a short preprocessing time. A ray tracer could potentially offer better results and can be easily used instead.

In this way, we first obtain 3 LDIs, similar to the system of Pfister et al. [17]. In contrast to their work, for reasons that will be made clear in the rendering section, we always merge the three LDIs into a single data structure which we call *loose LDI* (LDDI). This is similar to a LDI, but the original sampled (x, y, z)-coordinates of each sample are preserved and explicitly stored per sample. As such, the LLDI structure is only a spatial classification of the contained samples. Besides points, all triangles contained in a leaf octree cell are stored at that node. After the geometry is sampled into the leaf octree nodes, we further construct the rest of the octree bottom-up, as Fig. 2 shows.

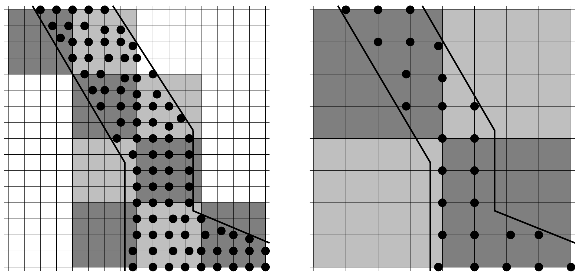

Figure 2: *Level-of-detail generation by sub-sampling the original loose LDI (shown in 2D). The sampling points do not strictly match the LDI grid.*

During rasterization, the surface attributes are stored at each sampled point. Special attention is required by the texture sampling, since it is performed at preprocessing time in a view-independent fashion and at a fixed resolution. Similar to Pfister et al. [17], we perform view independent EWA (Elliptical Weighted Average, described by Heckbert [9, 8]) texture pre-filtering by projecting Gaussian kernels in texture space and store several mipmap levels per sample to deal with minification. Due to the hybrid nature of our rendering algorithm, we don't have to take magnification into account, since in this case the original geometry is rendered with traditional texture mapping.

The sampling process and data structure described above refer to a single three-dimensional object. For scenes containing many objects, we preserve instantiation by encoding the scene as a collection of nodes representing instances of objects. Each node contains a pointer to the corresponding sampled object and a transformation matrix which can be then processed at rendering time without significant overhead. This storage scheme also offers the possibility to render scenes with moving objects.

5. Image Reconstruction

The previously discussed hierarchical structure is used for rendering. As triangles are straightforward to render on current hardware, we focus in the following on point rendering issues.

The simplest reconstruction method is splatting with opaque splats. In this case, the image quality drops rapidly as the splat size increases. More sophisticated reconstruction methods have been proposed which offer better image and especially texture quality [17, 25] using Gaussian kernels for the reconstruction. However, these methods would require special hardware features like an accumulation buffer and are currently implemented in software.

A good compromise would be to draw each point as a Gaussian splat (with an alpha channel falling off with a Gaussian) and to use hardware alpha blending for compositing the splats. The main difficulty when rendering fuzzy splats on current hardware is the well known conflict between alpha-blending and z-buffer tests: transparent surfaces must be drawn back-to-front. Again, the general solution is the A-buffer algorithm of Carpenter [1]. Rusinkiewicz and Levoy [19] mention fuzzy splatting as an alternative for better image quality and implemented it using double-pass rendering and z-offsetting. However, this won't produce correct results for surfaces with large depth discontinuities and requires the scene to be rendered two times, which is expensive for large scenes.

To render fuzzy splats, we use a simpler and more efficient solution which requires only one rendering step: exploiting our spatial classification, we draw the splats in paint order, without using the z-buffer. We extend the idea of Shade et al. [21] which uses McMillan's algorithm [15, 16, 14] to render a single LDI.

The rendering process can be described by the following simple algorithm:

```
for $object := back$ to $front$ do
    $TraverseOctree(object.octree\_root)$
od

proc $TraverseOctree(octree\_block) \equiv$
  if $(visible)$
    then
        if $(estimated\_max\_distance < threshold)$
          then $splat\_points(octree\_block)$
          else if $(!leaf\_block(octree\_block))$
            then
                for $child\_bl := back\_bl$ to $front\_bl$ do
                    $TraverseOctree(child\_bl)$
                od
            else
                $draw\_triangles(octree\_block)$
          fi
        fi
  fi
.
```

The level-of detail selection is rather straightforward and shortly described in the next section. Then, the generic occlusion-compatible traversal in the algorithm above is detailed in section 5.2, followed by shading, splatting and texture filtering issues.

5.1. Level-of-Detail Selection

The level-of-detail selection algorithm is derived from Pfister et al. [17]. We start with the root block in the octree, representing the coarsest resolution, and traverse the octree recursively top-down, performing view frustum culling. In contrast to them, we employ an occlusion-compatible traversal and switch to triangles when the estimated point density becomes insufficient. Related to this, the sampling resolution should be chosen such that the optimal balance between point and triangle rendering is met. Ideally the algorithm switches to points if the rendering time for triangles exceeds the time for points. A triangle/point rendering performance model similar to Cohen et al. [4] can be used for this purpose. In our current system this is a user-defined parameter. As a future work, we plan to implement an automatic procedure for computing the optimal sampling resolution.

In the case of large environments, like e.g. in landscape walkthroughs, many objects may occupy a small screen area for which even the coarsest representation is too detailed, which results in waste of rendering time. The problem arises as the resolution of the root block of the octree is the coarsest resolution which we can represent. We handle this by further sub-sampling the coarsest LLDI at rendering time (picking fewer points from the LLDI in a random fashion).

5.2. Paint-Order Warping

The method we use to ensure that the splats are drawn in paint order is an extension of the algorithm presented by [15] and adopted by Shade et al.[21] for LDI rendering. The original algorithm warps a LDI from an arbitrary input camera position to a new camera position. In our case of ortographic projection, input camera can be considered at infinite distance. The algorithm is illustrated in Fig.3: the new camera position is projected on the LDI plane. The projection, called epipolar point, determines 4 quadrants (in the most general case) separated by 2 epipolar lines. The layered pixels are processed inwards and back to front. As demonstrated by Shade et al. [21], this algorithm ensures occlusion-compatible warping for primitives that do not exceed the LDI grid cells.

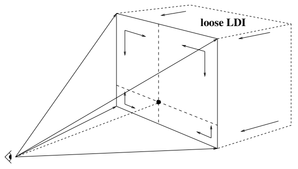

Figure 3: *Paint-order warping: each quadrant determined by the projection of the eye position in the LLDI plane is processed inwards. At each grid location, the samples are processed back-to-front with respect to the view direction.*

We first ignore our LLDI structure and consider a standard LDI. To enable correct image reconstruction with fuzzy splats, the splats must overlap, thus exceeding the grid cells. In this case, the algorithm does not necessarily work correctly, as shown in Fig. 4. Visibility artifacts appear for low angles between LDI plane normal and view direction, as surfels from a far surface may overlap previously drawn surfels from a near surface and alter its color. We circumvent

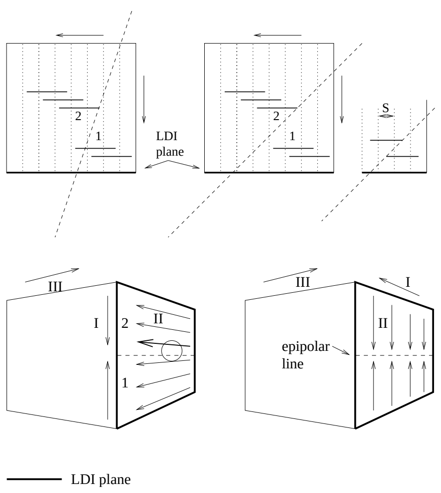

Figure 4: *Top: surface 1 is overwritten by surface 2 (left); this does not happen if the view angle is large (right), allowing a splat size of* 2*S. Bottom: artifacts result along the visible epipolar line (left), which can be avoided by choosing the other epipolar line for the external loop of the algorithm (right).*

this problem by maintaining a second LDI representation from another orthogonal view. The memory overhead is not high, as this is only an array of indices. For each view direction, there is a LDI representation with view angle greater than $\pi/4$. We use this property to derive the maximum splat size in the worst case, shown in the same Fig. 4. If the distance between different surfaces is greater or equal to the grid space S, the worst case yields a maximum splat size of $2S$ when drawing elliptical splats according to the normal. However, in practice, we use smaller splats and so compensate for the fact that samples do not strictly match the LDI grid. If the distance between different surfaces is less than S, they begin to be blended together.

A less obvious source of artifacts still exists. The original algorithm consists of three nested loops: split the LDI plane grid by one epipolar line and process each row inwards (I), split each row by the other epipolar line and process layered pixels inwards (II) and finally process each layered pixel back-to-front (III). An epipolar line is *visible* if it falls inside the LDI domain. As Fig. 4 bottom-left shows, artifacts can occur in the vicinity of the visible epipolar line if chosen for the most external loop of the algorithm (I), as samples from region 2 belonging to far surfaces can overwrite near surfaces in region 1. By using two LDI representation, we always have at most one visible epipolar line. Consequently, we modify the algorithm to always choose the invisible epipolar line for the external loop (I) (Fig. 4 bottom-right). This eliminates the problem and the previous splat size determination applies.

We apply the same algorithm to traverse in back-to-front order the children of an octree block in the rendering algorithm. Each block is regarded as a LDI with a 2x2 grid and 2 elements on each layered pixel. The aforementioned observations also apply in this case, as blocks may overlap. In this way, we ensure back to front order for the whole object by applying the same algorithm hierarchically.

For scenes containing several objects, we must ensure that the objects are also processed in occlusion compatible order. Currently we do this by representing the scene as a quadtree and traversing this quadtree in occlusion-compatible order. However, such an order may not exist or may be difficult to compute if objects intersect each oder. Our warping algorithm computes an occlusion-compatible order for the splats, but it does not necessarily guarantee the correct z-order for very closed splats (which are blended together). For this reason, the z-buffer must be turned off, otherwise artifacts would occur when rendering continuous surfaces. It suffices to turn off only z-buffer updates and still allow z-buffer tests. In this way, if necessary, the visibility can be solved by an additional rendering step: after fuzzy splatting, each object is rendered again, only in the z-buffer using standard splatting and (possibly) a coarser resolution. It is, of course, much preferrable to avoid this by properly modelling the scene and sampling intersecting objects together. For certain non-compact objects like trees, we can turn the z-buffer on, since artifacts are only visible on continuous surfaces. An efficient and general solution would be an extension of z-buffer hardware to allow z-offseting in one step: the updated z-value should be allowed to differ from the value used for the z-test.

Regarding the effective warping, one attractive feature of layered depth images is incremental warping, using the uniform grid, as presented in detail by Grossman [6]. In contrast, our current system relies on the high vertex processing speed of modern graphics hardware, thereby freeing host processor resources to balance the workload. This also allows to use the non-strict LLDI representation. Thus, we perform warping in hardware, keeping our LLDI data in 1D arrays (vertex buffers) which can be sent to the graphics engine together with index buffers reflecting the back-to-front order computed per frame as above (see the implementation section for details). However, as processor speed increases, incremental warping might be a viable alternative. We plan a thorough comparative analysis of the two warping methods in the future.

5.3. Shading

Since we store normals at each sample, we can apply arbitrarily complex local light models and other shading techniques. The only restriction is to use the same illumination for both triangles and points to ensure similar appearance. In our current implementation we use either a simplified Phong model or a diffuse illumination model for both points and triangles.

5.4. Splatting

The level-of-detail selection algorithm presented above guarantees that the maximum distance between the projections of two adjacent points, s_{max}, is always smaller than a specified threshold d_t, expressed in pixels. We can directly use d_t as splat size for opaque discs to guarantee a hole-free reconstruction. However, opaque splats produce a poor image quality, especially for textured objects, which dramatically drops if the splat size exceeds one pixel. We use elliptical Gaussians which offer improved texture quality (see next section) and avoid silhouette artifacts like thickening.

For fuzzy Gaussians, the splat size should be chosen such that $\alpha = 1/2$ at $d_t/2$ distance from center, which guarantees opaque surfaces in the worst case (Fig. 5). This is true for a A-buffer approach, where visibility and blending are separated, like in Zwicker et al. [25]. We rely on back-to-front sorting for visibility and draw the splats without any other test using alpha blending in the following manner:

$$col_{screen} = col_{splat} * \alpha_{splat} + col_{screen} * (1 - \alpha_{splat}). \quad (1)$$

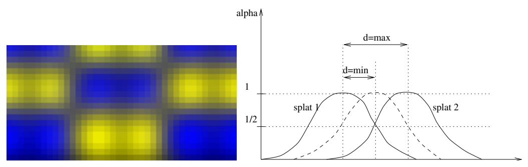

Figure 5: *Fuzzy splatting: visual appearance (left), α values of two superposed fuzzy splats (right).*

Compared to the ideal A-buffer compositing of two splats, alpha-blending does not guarantee perfect opacity: a small amount of background color also contributes to the result. However, this usually does not lead to objectable visual artifacts in practice if slightly more opaque splats are drawn; the opacity is also improved because usually several other samples from the same surface will contribute to the pixel. Transparent surfaces are not handled correctly. The color transition between adjacent points will also not be ideal, but still much better compared to opaque splats.

5.5. Texture Filtering

Using opaque splats produces visibly aliased textures, as each pixel receives color from only one splat and the equivalent texture function is limited in frequency by the splat size. In the high quality texture filtering method of Zwicker et al. [25], the color of each pixel is computed as a weighted average of several samples. Although fuzzy splats cannot correctly reproduce this behavior, they still produce significantly better results than opaque splats, as each pixel can receive color from more than one sample. We use the pre-filtered texture mipmap levels to perform simple view dependent texture filtering as detailed in [17]. The difference to their system is that we also consider the perspective transformation when selecting the two mipmap levels. The mipmap selection and interpolation can be implemented with hardware support (see Section 6).

The main drawback of this texture filtering scheme is the order-dependent property of alpha blending: the final image depends on the order in which splats are warped. If splats are warped along a line in a direction, textures will appear to be slightly displaced in the opposite direction, as an effect of the incorrect weighting and normalization. Formally, in the ideal reconstruction of Zwicker et al. [25], the color of a pixel p is computed as weighted average of the neighbored samples p_k:

$$c = \frac{\sum_{k=1}^{n} w_k c_k}{\sum_{k=1}^{n} w_k}, w_k = G_k(p_k - p). \quad (2)$$

where G_k are Gaussian kernels. For our alpha blending scheme, we obtain:

$$c = B \prod_{k=1}^{n} (1 - w_k) + \sum_{k=1}^{n} c_k w_k \prod_{j=k+1}^{n} (1 - w_j) \quad (3)$$

whereby B is the background color. This simply means that that later drawn splats contribute more to the pixel color. However, our warping algorithm induces a well-define splatting order and thus reduces this effect. Texture artifacts (a small displacement due to change in the ordering) will still be visible along the epipolar line and at block boundaries for flat surfaces textured with high frequency, regular patterns (see video). But, in practice, this kind of surfaces are anyway more efficiently rendered with triangles.

6. Implementation and Results

The algorithm was implemented in the C++ language and integrated in the data visualization system Amira [24]. Amira provides a flexible and extensible framework and supports a large variety of file formats. We have implemented two modules, one for sampling and one for rendering. Our test platform was a 2.0 Ghz Pentium 4 with 1 GB memory and a ATI Radeon 8500 graphics card.

6.1. Sampling

The sampling module takes a geometrical triangulated 3D model as input and produces a hierarchical LOD structure as previously described. Table 1 shows sampling times for our test models.

Using geometry rasterization, the sampling times are reduced in comparison to other approaches; Pfister et al. [17] for example report a pre-processing time of 1 hour for typical objects similar to our tree. A raytracer would probably offer better sampling results for very small triangles (we intend to test one in the future), but our sampling scheme could be also improved.

Object	Tris	Sampl. Res.	Sampl. Points	Sampl. Tris	Time (m:s)
harley	45k	256^3	135k	56k	0:05
lobus	1000k	256^3	230k	1000k	1:20
tex.tree	250k	256^3	350k	250k	0:50
tex.surf	128	512^3	266k	352	0:35
Pisa	337k	320^3	230k	350k	0:20

Table 1: *Sampling times for different objects. The number of recorded triangles is slightly larger than the original number of triangles because triangles at octree block separation are duplicated to avoid cracks when applying block culling. This, however, can be avoided by relaxing the view frustum culling such that additional neighbored blocks are rendered for blocks intersecting the frustum limits.*

6.2. Rendering

The main advantage of our technique is efficient implementation on current off-the-shelf graphics hardware. A recent innovation are the programmable vertex and pixel processing pipelines which were already implemented in hardware (see Lindholm et al. [11]). We used the Microsoft DirectX 8 API to program a vertex shader (or vertex program in the OpenGL context) which performs warping, shading as well as texture mipmap level selection and interpolation. A pixel shader is used to draw elliptical Gaussians by manipulating an alpha texture.

Special attention is payed to sending primitives efficiently to the graphics hardware. We store the LLDI in a compact, space-efficient form as described in [21]. The samples are stored in a one-dimensional array and the LLDI representation is an array of offsets in this array. The second LLDI representation (see Section 5.2) requires an additional index level. The initial idea was to pass the whole array of samples as a vertex buffer to the hardware in one call and then for each frame an index buffer reflecting the occlusion-compatible order. This would have minimized the CPU bus traffic, as the GPU can use DMA accesses to read data from video or AGP memory for indexed primitives. Unfortunately, points are an exception under DirectX with respect to indexing: indexed point primitives are not supported. Consequently, in the current implementation we have to do the indexing in software and send more data over the CPU bus. We will be able to use indexing in OpenGL, as soon as the new features are incorporated.

6.2.1. Vertex Processing

The current vertex format is shown in Table 2. The necessary storage space per sample is 48 bytes per sample for specular lighting. For diffuse light only, we can pack the normals in a four-bytes integer and we obtain 32 bytes. We can even do material indexing if we have few materials per octree block. Otherwise, the overhead due to the increased number of state changes will undo the gain obtained by reducing the vertex size.

Element	Size
position	3x32 bits
normal	3x32bits
ambient color	32 bits
diffuse color	32 bits
specular color+coef.	32 bits
3 texture mipmap levels	3x32 bits

Table 2: *Vertex buffer format.*

For each sample, the vertex shader performs following operations:

- warp position using the composite world-view-projection matrix
- perform lighting in model space
- interpolate between mip-map levels (2 or 3). The coefficients are set per octree block as vertex shader constants
- compute the inverse of the cosine of the angle between normal and view direction $\beta = 1/cos(\theta)$
- compute the n_x and n_y normal components in camera space and normalize this 2D vector.

The last two computations are necessary to render elliptical Gaussian splats according to the sample normals, as explained in the next section. When using point sprites, the output texture coordinate registers are overwritten by internally generated values, so we only can use the specular color output register (denoted as oD1 in DirectX) to pass these values down to the pixel shader. Because color components must be in the range 0.0 - 1.0, we scale the above values appropriately.

6.2.2. Point Splatting

Each point is splatted using a new feature of the graphics hardware called ‘point sprites’. It allows to apply a texture on a square described by a single vertex and a size. We use a texture with alpha channel falling off with a circular Gaussian and use alpha test to skip pixels with α less than a threshold (0.1 - 0.2). A particularly interesting implementation issue is how elliptical Gaussians are drawn. In the vertex shader, we compute for each sample the inverse of the cosine of the angle between normal and view direction β and the planar normal components in camera space n_x and n_y. We can make the splats elliptical by manipulating the texture coordinates when rasterizing the square splat in the pixel shader: $(t_x^{'}, t_y^{'})^t = \beta\,(t_x, t_y)R_{n_x,n_y}$, where R_{n_x,n_y} is the rotation matrix determined by the normal components. This rotates the texture and then shrinks it along the normal projection according to the coefficient β (Fig. 6).

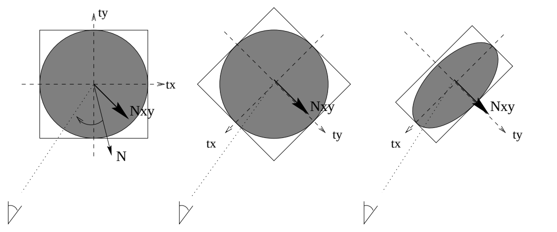

Figure 6: *Using texture mapping to draw elliptical Gaussians. First, texture coordinates are rotated such that they are aligned to the projected normal. Then, a deformation is applied to obtain an ellipse approximation.*

However, due to current technical limitations (the texture coordinate registers are overwritten), we cannot pass enough data to implement this manipulation. Instead, we use the following approximation at each pixel:

$$\alpha = \alpha_{circGauss}(1 - \|b\ (t_x, t_y)^t (n_x, -n_y)\|). \tag{4}$$

$$b = (1 - \frac{1}{e^2})(\beta - \frac{1}{\beta})\sqrt{2\,ln 2}. \tag{5}$$

which represents a circular Gaussian multiplied with a linear approximation of an elliptical Gaussian (the minus sign before n_y accounts for opposite direction of the y axis in texture coordinate system). This yields satisfactory results in practice. We also limit the β coefficient (the 'ellipsoidity') to avoid holes in the surface and actually compute b in the vertex shader. The pixel shader performs the following:

- receive in the specular input register the vector $(n_x, -n_y, 0, b)$
- compute the dot product $(n_x, -n_y).(t_x, t_y)$, after converting texture coordinates (t_x, t_y) to signed floats relative to the square center
- compute $\alpha_{linear} = (1 - \|b(t_x, t_y)^t (n_x, -n_y)\|)$
- compute output alpha value as the product of α_{linear} with the value sampled from the circular Gaussian texture at the location (t_x, t_y)
- transmit output color received from vertex shader.

6.2.3. Smooth LOD Transitions

Switching from triangles to points on a per-octree block basis results in visible popping. We implement a smooth transition using alpha blending: as the object part moves back, we still draw the triangle representation for a while and render the points over it, modulating the alpha value of the splats with a coefficient that varies from 0 to 1. For non-compact objects rendered with points (e.g. trees) we can have continuous LOD control, using a nice feature of our algorithm. During transition to a coarser level, we eliminate progressively and randomly points from the current level, taking care to converge to the set of points of the coarser representation. The overhead is low, since we anyway loop through the LLDI to select indices. We draw a LLDI sample if it is marked as belonging also to the next coarser level or if it meets a criterium which progressively decreases the number of selected points. The technique could be also extended to compact objects, if a proper compactness criterium is added such that no holes appear.

6.3. Results

To demonstrate the image reconstruction for textured objects, we sampled and rendered two objects: a textured height field and the classic checkerboard plane. The sampling resolution was 512^3 and the results are presented in Fig. 8 and compared to simple splatting rendering. It can be seen that paint-order fuzzy splatting offers significantly better texture quality, reducing color bleeding and aliasing. However, the EWA surface splatting approach of Zwicker et al. [25] offers higher quality textures and less frequency artifacts. The hardware-accelerated approach of Ren et al. [18] also achieves high image quality, but needs two rendering steps and uses 1 textured rectangle per sample. This significantly drops performance (3-4 times less splats per second than our implementation on the same hardware). We intend to continue the research and improve our texture filtering.

Fig. 7 and 9 top show the Harley Davidson and the Pisa Tower model, respectively, rendered with points. Drawing elliptical Gaussians substantially reduces silhouette artifacts (thickening). The surfaces look smoother and edges are antialiased. We also tested our method for non-compact objects like trees (Fig. 9), which also benefit from our elliptical primitives.

In Table 3, we show rendering times for different test models. We compare the rendering time with the following times: without ordering, with ordering and square splats and with an implementation that keeps data in video/AGP memory, computing the respective overheads. Obviously, doing indexing in software and pushing large amounts of data through the CPU introduces a significant overhead (more than 50%, except the tree model which uses diffuse illumination and thus a reduced vertex size), while the overhead due to ordering is around 30%. Our system will greatly benefit of indexed point primitives. The comparison with pure triangle rendering shows that some models are oversampled and do not actually benefit from switching from triangles to points (Harley), while other models (Lobus, Pisa) achieve significant speed-up. It is therefore necessary to provide an automatic selection of the sampling resolution to correctly balance the rendering times for triangles and points on a given platform, as Cohen et al. [4] do. We plan this as a future work.

As Table 3 shows, we achieve high raw splat rates even in the current CPU-bus-intensive implementation. On fast CPU/memory systems and with indexed point primitives, we expect to be limited only by the vertex processing speed of the hardware. A key efficiency issue is the fact that we only have to render the scene once.

model	#tris	#points	tris only (fps)	fuzzy splats (fps)	splat rate (10^6/s)	non-fuzzy splats video/AGP(fps)	+ord	+fuzz	+index
Harley	50k	130k	122	55	6.8	91	20%	5%	54%
Pisa	350k	230k	24	23.1	5.2	54	27%	9%	92%
Lobus	1000k	350k	8.3	16.7	5.6	38	33%	16%	63%
Tree	250k	200k	51	44.9	9	73	30%	19%	23%

Table 3: *Rendering performance for different models at 640x480 output resolution and splat size 2.*

scene size (Pisa)	#tris	splat=2 (fps)	splat=3 (fps)	splat=4 (fps)	splat rate 10^6/s	splat=3 non-fuzzy video/AGP(fps)
10x10	3.5×10^7	7.3	11.6	30.6	6.0	29.0
20x20	1.4×10^8	6.1	10.8	15.5	5.7	26.0
50x50	8.7×10^8	3.7	6.4	9.4	4.9	17.5
100x100	3.5×10^9	2.1	3.6	4.7	3.5	10.8
200x200	1.4×10^{10}	1.3	1.8	2.2	2.1	5.9

Table 4: *Rendering performance for different scenes and splat sizes at 500x400 output resolution.*

To analyze the rendering time for complex scenes, we performed several tests with scenes containing many replicated objects arranged in a grid. Having different objects would only imply a memory overhead. The viewpoint was chosen as shown in Fig. 7 right, resulting in the lowest frame rate for the scene. We tested different splat sizes (which means different point density thresholds and is approximately equivalent with testing for different image resolutions). As Table 4 and the attached video shows, we can render complex scenes interactively. The rendering time depends roughly logarithmically of the scene complexity. For many small objects, the overhead due to per object and per octree block calculations becomes high. Further enhancement of the algorithm in this respect can greatly improve performance.

7. Conclusion and Future Work

We described a rendering method for complex scenes containing textured models. As previous research pointed out, the key to an efficient system is to use the different rendering primitives in their maximum efficiency domains. Recent hybrid approaches [4, 2] achieve a fine-grained triangle-point trade-off by tightly integrating them in one data structure. In our system, the trade-off is performed on a per octree block basis. This can be regarded as trading off accuracy for processing overhead. The main contribution of this paper is a new hardware accelerated method for point rendering using elliptical Gaussians which is integrated with traditional rendering. This technique is suited for interactive applications, as it offers much better image quality than previous hardware accelerated systems (which use simple opaque splats) and preserves the speed. With indexed point primitives support, it can further benefit from a speed up of more than 50%. Moreover, it will directly benefit from further hardware improvements with respect to speed and programmability.

As future work, we will implement an automatic procedure for selecting the adequate sampling resolution of a triangulated model, given a point/triangle performance factor on a system (similar to Cohen et al. [4]). This will guarantee that the rendering performance of our system does not fall below pure triangle rendering. In order to improve the scene traversal and allow a very large numbers of objects, a mechanism must be provided that joins several small objects together and thus avoids the overhead of processing each object separately. A way of doing this is to extend the LLDI hierarchy to the whole scene and merge the coarsest object representations in it. We have already implemented and tested such a scheme which allows the interactive rendering of scenes containing up to 10^6 objects.

Furthermore we will investigate the possibilities to perform even smoother transitions between different levels of detail. Our current sampling scheme also admits significant improvements. Texture filtering deserves a more detailed research. We plan to alternatively implement incremental warping on LDI and compare with the current implementation. We also want to extend our system to other models like procedural objects. We expect these future improvements to allow the interactive rendering of much more visually appealing scenes.

References

1. Carpenter, L. The A-buffer, an antialiased hidden surface method. In *Computer Graphics (Proceedings of SIGGRAPH 84)* (Minneapolis, Minnesota, July 1984), vol. 18:3, ACM, pp. 103–108. 4
2. Chan, B., and Nguyen, M. X. Pop: A hybrid point and polygon rendering system for large data. *IEEE Visualization 2001* (2001). 1, 2, 9
3. Chang, C.-F., Bishop, G., and Lastra, A. LDI tree: A hierarchical representation for image-based rendering. *Proceedings of SIGGRAPH 99* (August 1999), 291–298. 2, 2
4. Cohen, J. D., Aliaga, D. G., and Zhang, W. Hybrid simplification: combining multi-resolution polygon and point rendering. *IEEE Visualization 2001* (October 2001), 37–44. 1, 2, 4, 8, 9, 9
5. Gortler, S., He, L. W., and Cohen, M. Rendering layered depth images. Tech. Rep. MSTRTR 97-09, Microsoft Research, 1997. 2
6. Grossman, J. Point sample rendering. Master's thesis, MIT, Dept. of Electrical Engineering and Computer Science, 1998. 2, 5
7. Grossman, J. P., and Dally, W. J. Point sample rendering. *Eurographics Rendering Workshop 1998* (June 1998), 181–192. 1, 2
8. Heckbert, P. S. Survey of texture mapping. *IEEE Computer Graphics & Applications 6*, 11 (November 1986), 56–67. 3
9. Heckbert, P. S. Fundamentals of texture mapping and image warping. Master's thesis, UCB/CSD 89/516, 1989. 2, 3
10. Levoy, M., and Whitted, T. The use of points as a display primitive. Tech. Rep. TR 85-022, University of North Carolina at Chapel Hill, 1985. 2
11. Lindholm, E., Kilgard, M. J., and Moreton, H. A user-programmable vertex engine. In *Proceedings of SIGGRAPH 2001* (August 2001), ACM, pp. 149–158. 7
12. Lischinski, D., and Rappoport, A. Image-based rendering for non-diffuse synthetic scenes. *Eurographics Rendering Workshop 1998* (June 1998), 301–314. 2
13. Max, N. Hierarchical rendering of trees from precomputed multi-layer Z-buffers. In *Eurographics Rendering Workshop 1996* (New York City, NY, 1996), X. Pueyo and P. Schröder, Eds., Eurographics, pp. 165–174. 2
14. McMillan, L. Computing visibility without depth. Tech. Rep. TR 95-047, University of North Carolina at Chapel Hill, 1995. 2, 2, 4
15. McMillan, L. A list-priority rendering algorithm for redisplaying projected surfaces. Tech. Rep. TR 95-005, University of North Carolina at Chapel Hill, 1995. 2, 2, 4, 4
16. McMillan, L., and Bishop, G. Plenoptic modeling: An image-based rendering system. In *Proceedings of SIGGRAPH 95* (August 1995), Computer Graphics Proceedings, Annual Conference Series, ACM, pp. 39–46. 2, 2, 4
17. Pfister, H., Zwicker, M., van Baar, J., and Gross, M. Surfels: Surface elements as rendering primitives. *Proceedings of SIGGRAPH 2000* (July 2000), 335–342. 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 6, 6
18. Ren, L., Pfister, H., and Zwicker, M. Object space EWA splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings of Eurographics* (2002). 2, 8
19. Rusinkiewicz, S., and Levoy, M. Qsplat: A multiresolution point rendering system for large meshes. *Proceedings of SIGGRAPH 2000* (July 2000), 343–352. 1, 2, 4
20. Schaufler, G., and Jensen, H. W. Ray tracing point sampled geometry. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering* (June 2000), 319–328. 2
21. Shade, J., Gortler, S. J., wei He, L., and Szeliski, R. Layered depth images. *Proceedings of SIGGRAPH 98* (July 1998), 231–242. 2, 2, 4, 4, 4, 7
22. Stamminger, M., and Drettakis, G. Interactive sampling and rendering for complex and procedural geometry. In *Rendering Techniques 2001 (Proceedings of the Eurographics Workshop on Rendering)* (2001). 2
23. Wand, M., Fischer, M., Peter, I., auf der Heide, F. M., and Strasser, W. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. *Proceedings of SIGGRAPH 2001* (August 2001), 361–370. 1, 2
24. *Amira – User's Guide and Reference Manual* as well as *Amira – Programmer's Guide*. Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB) and Indeed - Visual Concepts GmbH, Berlin, http://www.amiravis.com, 2001. 6
25. Zwicker, M., Pfister, H., van Baar, J., and Gross, M. Surface splatting. *Proceedings of SIGGRAPH 2001* (August 2001), 371–378. 1, 1, 2, 2, 2, 3, 6, 6, 6, 8

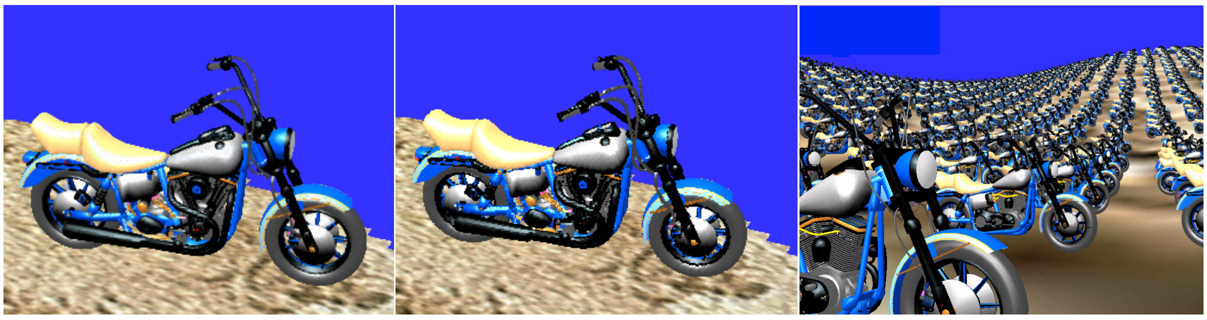

Figure 7: *Bike model rendered with ellipsoidal Gaussian splats (left), opaque splats (middle), scene with 50x50 models (right). Splat size=2.*

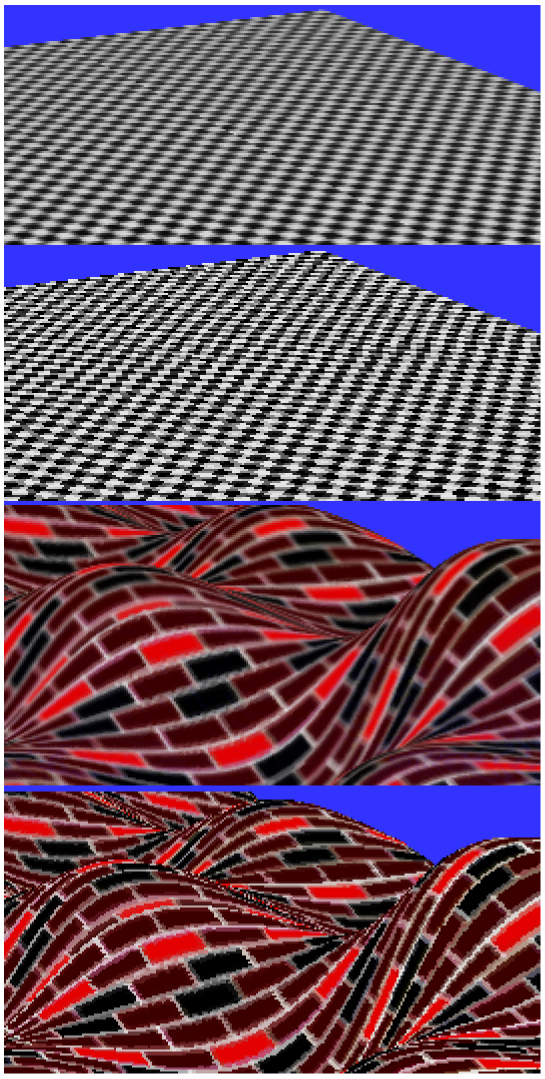

Figure 8: *Texture reconstruction. Top-down for each model: Gaussian elliptical splats and view-dependent mipmap filtering, opaque splats. Splat size = 3.*

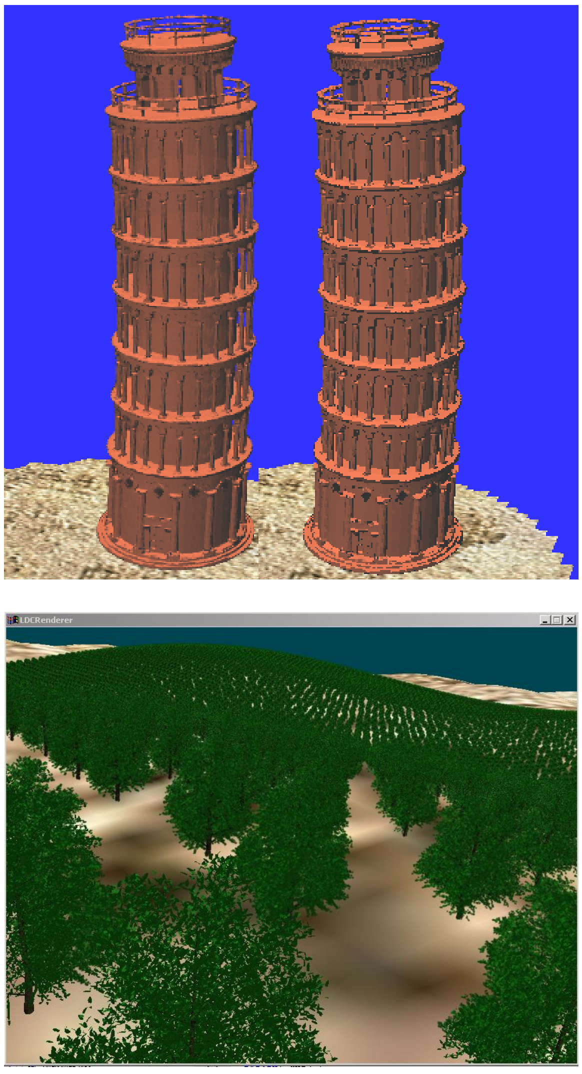

Figure 9: *Top: comparison of Gaussian elliptical splats (left) with opaque squares (right) at splat size=2. Bottom: 50x50 non-compact objects (trees).*