# Surfels: Surface Elements as Rendering Primitives

Hanspeter Pfister * Matthias Zwicker † Jeroen van Baar* Markus Gross†

Figure 1: *Surfel rendering examples.*

## Abstract

Surface elements (surfels) are a powerful paradigm to efficiently render complex geometric objects at interactive frame rates. Unlike classical surface discretizations, i.e., triangles or quadrilateral meshes, surfels are point primitives without explicit connectivity. Surfel attributes comprise depth, texture color, normal, and others. As a pre-process, an octree-based surfel representation of a geometric object is computed. During sampling, surfel positions and normals are optionally perturbed, and different levels of texture colors are prefiltered and stored per surfel. During rendering, a hierarchical forward warping algorithm projects surfels to a z-buffer. A novel method called visibility splatting determines visible surfels and holes in the z-buffer. Visible surfels are shaded using texture filtering, Phong illumination, and environment mapping using per-surfel normals. Several methods of image reconstruction, including supersampling, offer flexible speed-quality tradeoffs. Due to the simplicity of the operations, the surfel rendering pipeline is amenable for hardware implementation. Surfel objects offer complex shape, low rendering cost and high image quality, which makes them specifically suited for low-cost, real-time graphics, such as games.

**CR Categories:** I.3.3 [Computer Graphics]: Picture/Image Generation – Viewing Algorithms; I.3.6 [Computer Graphics]: Methodology and Techniques – Graphics Data Structures and Data Types.

**Keywords:** Rendering Systems, Texture Mapping.

*MERL, Cambridge, MA. Email: [pfister,jeroen]@merl.com
†ETH Zürich, Switzerland. Email: [zwicker,gross]@inf.ethz.ch

## 1 Introduction

3D computer graphics has finally become ubiquitous at the consumer level. There is a proliferation of affordable 3D graphics hardware accelerators, from high-end PC workstations to low-priced gamestations. Undoubtedly, key to this success is interactive computer games that have emerged as the "killer application" for 3D graphics. However, interactive computer graphics has still not reached the level of realism that allows a true immersion into a virtual world. For example, typical foreground characters in real-time games are extremely minimalistic polygon models that often exhibit faceting artifacts, such as angular silhouettes.

Various sophisticated modeling techniques, such as implicit surfaces, NURBS, or subdivision surfaces, allow the creation of 3D graphics models with increasingly complex shapes. Higher order modeling primitives, however, are eventually decomposed into triangles before being rendered by the graphics subsystem. The triangle as a rendering primitive seems to meet the right balance between descriptive power and computational burden [7]. To render realistic, organic-looking models requires highly complex shapes with ever more triangles, or, as Alvy Ray Smith puts it: "Reality is 80 million polygons" [26]. Processing many small triangles leads to bandwidth bottlenecks and excessive floating point and rasterization requirements [7].

To increase the apparent visual complexity of objects, texture mapping was introduced by Catmull [3] and successfully applied by others [13]. Textures convey more detail inside a polygon, thereby allowing larger and fewer triangles to be used. Today's graphics engines are highly tailored for high texture mapping performance. However, texture maps have to follow the underlying geometry of the polygon model and work best on flat or slightly curved surfaces. Realistic surfaces frequently require a large number of textures that have to be applied in multiple passes during rasterization. And phenomena such as smoke, fire, or water are difficult to render using textured triangles.

In this paper we propose a new method of rendering objects with rich shapes and textures at interactive frame rates. Our rendering architecture is based on simple surface elements (*surfels*) as rendering primitives. Surfels are point samples of a graphics model. In a preprocessing step, we sample the surfaces of complex geometric models along three orthographic views. At the same time, we perform computation-intensive calculations such as texture, bump, or displacement mapping. By moving rasterization and texturing from

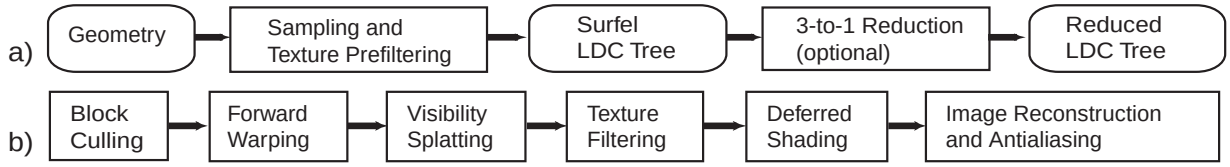


Figure 2: *Algorithm overview: a) Preprocessing. b) Rendering of the hierarchical LDC tree.*

the core rendering pipeline to the preprocessing step, we dramatically reduce the rendering cost.

From a modeling point of view, the surfel representation provides a mere discretization of the geometry and hence reduces the object representation to the essentials needed for rendering. By contrast, triangle primitives implicitly store connectivity information, such as vertex valence or adjacency – data not necessarily available or needed for rendering. In a sense, a surfel relates to what Levoy and Whitted call the *lingua franca* of rendering in their pioneering report from 1985 [18].

Storing normals, prefiltered textures, and other per surfel data enables us to build high quality rendering algorithms. Shading and transformations applied per surfel result in Phong illumination, bump, and displacement mapping, as well as other advanced rendering features. Our data structure provides a multiresolution object representation, and a hierarchical forward warping algorithm allows us to estimate the surfel density in the output image for speed-quality tradeoffs.

The surfel rendering pipeline complements the existing graphics pipeline and does not intend to replace it. It is positioned between conventional geometry-based approaches and image-based rendering and trades memory overhead for rendering performance and quality. The focus of this work has been interactive 3D applications, not high-end applications such as feature films or CAD/CAM. Surfels are not well suited to represent flat surfaces, such as walls or scene backgrounds, where large, textured polygons provide better image quality at lower rendering cost. However, surfels work well for models with rich, organic shapes or high surface details and for applications where preprocessing is not an issue. These qualities make them ideal for interactive games.

## 2 Related Work

The use of points as rendering primitives has a long history in computer graphics. As far back as 1974, Catmull [3] observed that geometric subdivision may ultimately lead to points. Particles were subsequently used for objects that could not be rendered with geometry, such as clouds, explosions, and fire [23]. More recently, image-based rendering has become popular because its rendering time is proportional to the number of pixels in the source and output images and not the scene complexity.

Visually complex objects have been represented by dynamically generated image sprites [25], which are quick to draw and largely retain the visual characteristics of the object. A similar approach was used in the Talisman rendering system [27] to maintain high and approximately constant frame rates. However, mapping objects onto planar polygons leads to visibility errors and does not allow for parallax and disocclusion effects. To address these problems, several methods add per-pixel depth information to images, variously called layered impostors [24], sprites with depth, or layered depth images [25], just to name a few. Still, none of these techniques provide a complete object model that can be illuminated and rendered from arbitrary points of view.

Some image-based approaches represent objects without explicitly storing any geometry or depth. Methods such as view interpolation and Quicktime VR [5] or plenoptic modeling [21] create new views from a collection of 2D images. Lightfield [17] or lumigraph [9] techniques describe the radiance of a scene or object as a function of position and direction in a four- or higher-dimensional space, but at the price of considerable storage overhead. All these methods use view-dependent samples to represent an object or scene. However, view-dependent samples are ineffective for dynamic scenes with motion of objects, changes in material properties, and changes in position and intensities of light sources.

The main idea of representing objects with surfels is to describe them in a view-independent, object-centered rather than image-centered fashion. As such, surfel rendering is positioned between geometry rendering and image-based rendering. In volume graphics [16], synthetic objects are implicitly represented with surface voxels, typically stored on a regular grid. However, the extra third dimension of volumes comes at the price of higher storage requirements and longer rendering times. In [8], Perlin studies "surflets," a flavor of wavelets that can be used to describe free-form implicit surfaces. Surflets have less storage overhead than volumes, but rendering them requires lengthy ray casting.

Our research was inspired by the following work: Animatek's Caviar player [1] provides interactive frame rates for surface voxel models on a Pentium class PC, but uses simplistic projection and illumination methods. Levoy and Whitted [18] use points to model objects for the special case of continuous, differentiable surfaces. They address the problem of texture filtering in detail. Max uses point samples obtained from orthographic views to model and render trees [20]. Dally et al. [6] introduced the delta tree as an object-centered approach to image-based rendering. The movement of the viewpoint in their method, however, is still confined to particular locations. More recently, Grossman and Dally [12] describe a point sample representation for fast rendering of complex objects. Chang et al. [4] presented the LDI tree, a hierarchical space-partitioning data structure for image-based rendering.

We extend and integrate these ideas and present a complete point sample rendering system comprising an efficient hierarchical representation, high quality texture filtering, accurate visibility calculations, and image reconstruction with flexible speed-quality tradeoffs. Our surfel rendering pipeline provides high quality rendering of exceedingly complex models and is amenable for hardware implementation.

## 3 Conceptual Overview

Similar to the method proposed by Levoy and Whitted [18], our surfel approach consists of two main steps: sampling and surfel rendering. Sampling of geometry and texture is done during preprocessing, which may include other view-independent methods such as bump and displacement mapping. Figure 2 gives a conceptual overview of the algorithm.

The sampling process (Section 5) converts geometric objects and their textures to surfels. We use ray casting to create three orthogonal layered depth images (LDIs) [25]. The LDIs store multiple surfels along each ray, one for each ray-surface intersection point. Lischinski and Rappaport [19] call this arrangement of three orthogonal LDIs a *layered depth cube (LDC)*. An important and novel aspect of our sampling method is the distinction between sampling of *shape*, or geometry, and *shade*, or texture color. A surfel stores both shape, such as surface position and orientation, and shade, such as multiple levels of prefiltered texture colors. Because of the similarities to traditional texture mipmaps we call this hierarchical color information a *surfel mipmap*.

From the LDC we create an efficient hierarchical data structure for rendering. Chang et al.[4] introduce the LDI tree, an octree with an LDI attached to each octree node. We use the same hierarchical space-partitioning structure, but store an LDC at each node of the octree (Section 6). Each LDC node in the octree is called a *block*. We call the resulting data structure the *LDC tree*. In a step called *3-to-1 reduction* we optionally reduce the LDCs to single LDIs on a block-by-block basis for faster rendering.

The rendering pipeline (Section 7) hierarchically projects blocks to screen space using perspective projection. The rendering is accelerated by block culling [12] and fast incremental forward warping. We estimate the projected surfel density in the output image to control rendering speed and quality of the image reconstruction. A conventional z-buffer together with a novel method called *visibility splatting* solves the visibility problem. Texture colors of visible surfels are filtered using linear interpolation between appropriate levels of the surfel mipmap. Each visible surfel is shaded using, for example, Phong illumination and reflection mapping. The final stage performs image reconstruction from visible surfels, including hole filling and antialiasing. In general, the resolution of the output image and the resolution of the z-buffer do not have to be the same.

# 4 Definition of a Surfel

We found the term surfel as an abbreviation for *surface element* or *surface voxel* in the volume rendering and discrete topology literature. Herman [15] defines a surfel as an oriented $(n-1)$-dimensional object in $R^n$. For $n = 3$, this corresponds to an oriented unit square (voxel face) and is consistent with thinking of voxels as little cubes. However, for our discussion we find it more useful to define surfels as follows:

**A surfel** is a zero-dimensional n-tuple with shape and shade attributes that locally approximate an object surface.

We consider the alternative term, point sample, to be too general, since voxels and pixels are point samples as well.

# 5 Sampling

The goal during sampling is to find an optimal surfel representation of the geometry with minimum redundancy. Most sampling methods perform object discretization as a function of geometric parameters of the surface, such as curvature or silhouettes. This *object space* discretization typically leads to too many or too few primitives for rendering. In a surfel representation, object sampling is aligned to *image space* and matches the expected output resolution of the image.

## 5.1 LDC Sampling

We sample geometric models from three sides of a cube into three orthogonal LDIs, called a *layered depth cube (LDC)* [19] or *block*. Figure 3 shows an LDC and two LDIs using a 2D drawing. Ray casting records all intersections, including intersections with back-facing surfaces. At each intersection point, a surfel is created with floating point depth and other shape and shade properties. Perturbation of the surface normal or of the geometry for bump and displacement mapping can be performed on the geometry before sampling or during ray casting using procedural shaders.

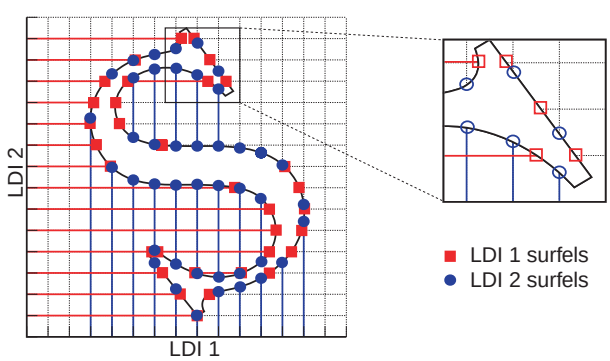


Figure 3: *Layered depth cube sampling (shown in 2D).*

Alternatively, we could sample an object from predetermined directions on a surrounding convex hull using orthographic depth images [6, 12]. However, combining multiple reference images and eliminating the redundant information is a difficult problem [21], and sampling geometry with reference images works best for smooth and convex objects. In addition, LDC sampling allows us to easily build a hierarchical data structure, which would be difficult to do from dozens of depth images.

## 5.2 Adequate Sampling Resolution

Given a pixel spacing of $h_0$ for the full resolution LDC used for sampling, we can determine the resulting sampling density on the surface. Suppose we construct a Delaunay triangulation on the object surface using the generated surfels as triangle vertices. As was observed in [19], the imaginary triangle mesh generated by this sampling process has a maximum sidelength $s_{max}$ of $\sqrt{3}h_0$. The minimum sidelength $s_{min}$ is 0 when two or three sampling rays intersect at the same surface position.

Similarly to [12], we call the object *adequately sampled* if we can guarantee that at least one surfel is projected into the support of each ouptut pixel filter for orthographic projection and unit magnification. That condition is met if $s_{max}$, the maximum distance between adjacent surfels in object space, is less than the radius $r'_{rec}$ of the desired pixel reconstruction filter. Typically, we choose the LDI resolution to be slightly higher than this because of the effects of magnification and perspective projection. We will revisit these observations when estimating the number of projected surfels per pixel in Section 7.2.

## 5.3 Texture Prefiltering

A feature of surfel rendering is that textures are prefiltered and mapped to object space during preprocessing. We use *view-independent* texture filtering as in [12]. To prevent view-dependent texture aliasing we also apply per-surfel texture filtering during rendering (see Sections 7.4 and 7.6).

To determine the extent of the filter footprint in texture space, we center a circle at each surfel on its tangent plane, as shown in Figure 4a. We call these circles *tangent disks*. The tangent disks are

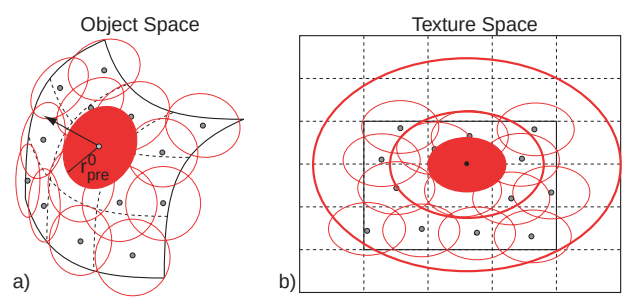


Figure 4: *Texture prefiltering with tangent disks.*

mapped to ellipses in texture space (see Figure 4b) using the predefined texture parameterization of the surface. An EWA filter [14] is applied to filter the texture and the resulting color is assigned to the surfel. To enable adequate texture reconstruction, the elliptical filter footprints in texture space must overlap each other. Consequently, we choose $r^0_{pre} = s_{max}$, the maximum distance between adjacent surfels in object space, as the radius for the tangent disks. This usually guarantees that the tangent disks intersect each other in object space and that their projections in texture space overlap.

Grossman and Dally [12] also use view-independent texture filtering and store one texture sample per surfel. Since we use a modified z-buffer algorithm to resolve visibility (Section 7.3), not all surfels may be available for image reconstruction, which leads to texture aliasing artifacts. Consequently, we store several (typically three or four) prefiltered texture samples per surfel. Tangent disks with dyadically larger radii $r^k_{pre} = s_{max}2^k$ are mapped to texture space and used to compute the prefiltered colors. Because of its similarity to mipmapping [13], we call this a *surfel mipmap*. Figure 4b shows the elliptical footprints in texture space of consecutively larger tangent disks.

# 6 Data Structure

We use the LDC tree, an efficient hierarchical data structure, to store the LDCs acquired during sampling. It allows us to quickly estimate the number of projected surfels per pixel and to trade rendering speed for higher image quality.

## 6.1 The LDC Tree

Chang et al. [4] use several reference depth images of a scene to construct the LDI tree. The depth image pixels are resampled onto multiple LDI tree nodes using splatting [29]. We avoid these interpolation steps by storing LDCs at each node in the octree that are subsampled versions of the highest resolution LDC.

The octree is recursively constructed bottom up, and its height is selected by the user. The highest resolution LDC — acquired during geometry sampling — is stored at the lowest level $n = 0$. If the highest resolution LDC has a pixel spacing of $h_0$, then the LDC at level $n$ has a pixel spacing of $h_n = h_0 2^n$. The LDC is subdivided into blocks with user-specified dimension $b$, i.e., the LDIs in a block have $b^2$ layered depth pixels. $b$ is the same for all levels of the tree. Figure 5a shows two levels of an LDC tree with $b = 4$ using a 2D drawing. In the figure, neighboring blocks are differently shaded,

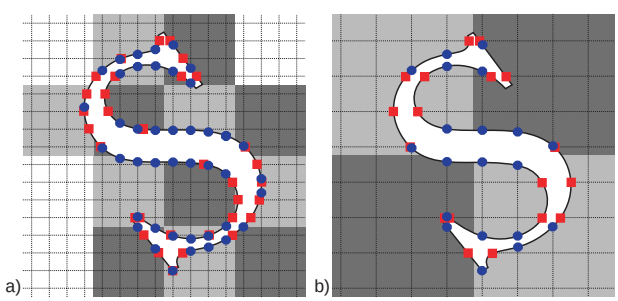


Figure 5: *Two levels of the LDC tree (shown in 2D).*

and empty blocks are white. Blocks on higher levels of the octree are constructed by subsampling their children by a factor of two. Figure 5b shows level $n = 1$ of the LDC tree. Note that surfels at higher levels of the octree reference surfels in the LDC of level 0, i.e., surfels that appear in several blocks of the hierarchy are stored only once and shared between blocks.

Empty blocks (shown as white squares in the figure) are not stored. Consequently, the block dimension $b$ is not related to the dimension of the highest resolution LDC and can be selected arbitrarily. Choosing $b = 1$ makes the LDC tree a fully volumetric octree representation. For a comparison between LDCs and volumes see [19].

## 6.2 3-to-1 Reduction

To reduce storage and rendering time it is often useful to optionally reduce the LDCs to one LDI on a block-by-block basis. Because this typically corresponds to a three-fold increase in warping speed, we call this step *3-to-1 reduction*. First, surfels are resampled to integer grid locations of ray intersections as shown in Figure 6. Currently we use nearest neighbor interpolation, although a more

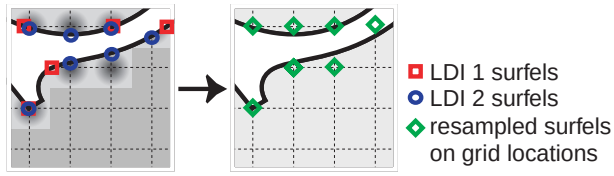


Figure 6: *3-to-1 reduction example.*

sophisticated filter, e.g., splatting as in [4], could easily be implemented. The resampled surfels of the block are then stored in a single LDI.

The reduction and resampling process degrades the quality of the surfel representation, both for shape and for shade. Resampled surfels from the same surface may have very different texture colors and normals. This may cause color and shading artifacts that are worsened during object motion. In practice, however, we did not encounter severe artifacts due to 3-to-1 reduction. Because our rendering pipeline handles LDCs and LDIs the same way, we could store blocks with thin structures as LDCs, while all other blocks could be reduced to single LDIs.

As in Section 5.2, we can determine bounds on the surfel density on the surface after 3-to-1 reduction. Given a sampling LDI with pixel spacing $h_0$, the maximum distance between adjacent surfels on the object surface is $s_{max} = \sqrt{3}h_0$, as in the original LDC tree. The minimum distance between surfels increases to $s_{min} = h_0$ due to the elimination of redundant surfels, making the imaginary Delaunay triangulation on the surface more uniform.

# 7 The Rendering Pipeline

The rendering pipeline takes the surfel LDC tree and renders it using hierarchical visibility culling and forward warping of blocks. Hierarchical rendering also allows us to estimate the number of projected surfels per output pixel. For maximum rendering efficiency, we project approximately one surfel per pixel and use the same resolution for the z-buffer as in the output image. For maximum image quality, we project multiple surfels per pixel, use a finer resolution of the z-buffer, and high quality image reconstruction.

## 7.1 Block Culling

We traverse the LDC tree from top (the lowest resolution blocks) to bottom (the highest resolution blocks). For each block, we first perform view-frustum culling using the block bounding box. Next, we use *visibility cones*, as described in [11], to perform the equivalent of backface culling of blocks. Using the surfel normals, we precompute a visibility cone per block, which gives a fast, conservative visibility test: no surfel in the block is visible from any viewpoint within the cone. In contrast to [11], we perform all visibility tests hierarchically in the LDC tree, which makes them more efficient.

## 7.2 Block Warping

During rendering, the LDC tree is traversed top to bottom [4]. To choose the octree level to be projected, we conservatively estimate for each block the number of surfels per pixel. We can choose one surfel per pixel for fast rendering or multiple surfels per pixel for supersampling. For each block at tree level $n$, the number of surfels per pixel is determined by $i^n_{max}$, the maximum distance between adjacent surfels in image space. We estimate $i^n_{max}$ by dividing the maximum length of the projected four major diagonals of the block bounding box by the block dimension $b$. This is correct for orthographic projection. However, the error introduced by using perspective projection is small because a block typically projects to a small number of pixels.

For each block, $i^n_{max}$ is compared to the radius $r'_{rec}$ of the desired pixel reconstruction filter. $r'_{rec}$ is typically $\frac{\sqrt{2}}{2}s_o$, where $s_o$

is the sidelength of an output pixel. If $i^n_{max}$ of the current block is larger than $r'_{rec}$ then its children are traversed. We project the block whose $i^n_{max}$ is smaller than $r'_{rec}$, rendering approximately one surfel per pixel. Note that the number of surfels per pixel can be increased by requiring that $i^n_{max}$ is a fraction of $r'_{rec}$. The resulting $i^n_{max}$ is stored as $i_{max}$ with each projected surfel for subsequent use in the visibility testing and the image reconstruction stages. The radius of the actual reconstruction filter is $r_{rec} = \max(r'_{rec}, i_{max})$ (see Section 7.6).

To warp a block to screen space we use the optimized incremental block warping by Grossman and Dally, presented in detail in [11]. Its high efficiency is achieved due to the regularity of LDCs. It uses only 6 additions, 3 multiplications, and 1 reciprocal per sample. The LDIs in each LDC block are warped independently, which allows us to render an LDC tree where some or all blocks have been reduced to single LDIs after 3-to-1 reduction.

## 7.3 Visibility Testing

Perspective projection, high z-buffer resolution, and magnification may lead to undersampling or holes in the z-buffer. A z-buffer pixel is a *hole* if it does not contain a visible surfel or background pixel after projection. Holes have to be marked for image reconstruction. Each pixel of the z-buffer stores a pointer to the closest surfel and the current minimum depth. Surfel depths are projected to the z-buffer using nearest neighbor interpolation.

To correctly resolve visibility in light of holes, we scan-convert the orthographic projection of the surfel tangent disks into the z-buffer. The tangent disks have a radius of $r^n_t = s_{max}2^n$, where $s_{max}$ is the maximum distance between adjacent surfels in object space and $n$ is the level of the block. We call this approach *visibility splatting*, shown in Figure 7. Visibility splatting effectively separates visibility calculations and reconstruction of the image, which produces high quality images and is amenable to hardware implementation [22].

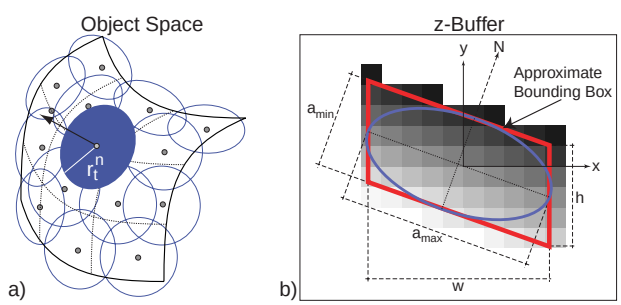


Figure 7: *Visibility splatting.*

After orthographic projection, the tangent disks form an ellipse around the surfel, as shown in Figure 7b. We approximate the ellipse with a partially axis-aligned bounding box, shown in red. The bounding box parallelogram can be easily scan-converted, and each z-buffer pixel is filled with the appropriate depth (indicated by the shaded squares in the figure), depending on the surfel normal $N$. This scan conversion requires only simple setup calculations, no interpolation of colors, and no perspective divide.

The direction of the minor axis $a_{min}$ of the projected ellipse is parallel to the projection of the surfel normal $N$. The major axis $a_{max}$ is orthogonal to $a_{min}$. The length of $a_{max}$ is the projection of the tangent disk radius $r^n_t$, which is approximated by $i_{max}$. This approximation takes the orientation and magnification of the LDC tree during projection into account. Next, we calculate the coordinate axis that is most parallel to $a_{min}$ (the y-axis in Figure 7). The short side of the bounding box is axis aligned with this coordinate axis to simplify scan conversion. Its height $h$ is computed by intersecting the ellipse with the coordinate axis. The width $w$ of the bounding box is determined by projecting the vertex at the intersection of the major axis and the ellipse onto the second axis (the $x$-axis in Figure 7).

$\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ are the partial derivatives of the surfel depth $z$ with respect to the screen $x$ and $y$ direction. They are constant because of the orthographic projection and can be calculated from the unit normal $N$. During scan conversion, the depth at each pixel inside the bounding box is calculated using $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$. In addition, we add a small threshold to each projected $z$ value. The threshold prevents surfels that lie on the foreground surface to be accidentally discarded. Pixels that have a larger $z$ than the $z$ values of the splatted tangent disk are marked as holes.

If the surface is extremely bent, the tangential planes do not cover it completely, potentially leaving tears and holes. In addition, extreme perspective projection makes orthographic projection a bad approximation to the actual projected tangent disk. In practice, however, we did not see this as a major problem. If the projected tangent disk is a circle, i.e., if $N$ is almost parallel to the viewing direction, the bounding box parallelogram is a bad approximation. In this case, we use a square bounding box instead.

Using a somewhat related approach, Grossman and Dally [12] use a hierarchical z-buffer for visibility testing. Each surfel is projected and the hole size around the surfel is estimated. The radius of the hole determines the level of the hierarchical z-buffer where the z-depth of the surfel will be set. This can be regarded as visibility splatting using a hierarchical z-buffer. The advantage is that the visibility splat is performed with a single depth test in the hierarchical z-buffer. However, the visibility splat is always square, essentially representing a tangential disk that is parallel to the image plane. In addition, it is not necessarily centered around the projected surfel. To recover from those drawbacks, [12] introduces weights indicating coverage of surfels. But this makes the reconstruction process more expensive and does not guarantee complete coverage of hidden surfaces.

## 7.4 Texture Filtering

As explained in Section 5.3, each surfel in the LDC tree stores several prefiltered texture colors of the surfel mipmap. During rendering, the surfel color is linearly interpolated from the surfel mipmap colors depending on the object minification and surface orientation. Figure 8a shows all visible surfels of a sampled surface projected to the z-buffer. The ellipses around the centers of the surfels mark the projection of the footprints of the highest resolution texture prefilter (Section 5.3). Note that during prefiltering, we try to guarantee that the footprints cover the surface completely. In figure 8b the number of samples per z-buffer pixel is limited to one by applying z-buffer depth tests. In order to fill the gaps appearing in the coverage of the surface with texture footprints, the footprints of the remaining surfels have to be enlarged. If surfels are discarded in a given z-buffer pixel, we can assume that the z-buffer pixels in the 3x3 neighborhood around it are not holes. Thus the gaps can be filled if the texture footprint of each surfel covers at least the area of a z-buffer pixel. Consequently, the ellipse of the projected footprint has to have a minor radius of $\sqrt{2}s_z$ in the worst case, where $s_z$ is the z-buffer pixel spacing. But we ignore that worst case and use $\frac{\sqrt{2}}{2}s_z$, implying that surfels are projected to z-buffer pixel centers.

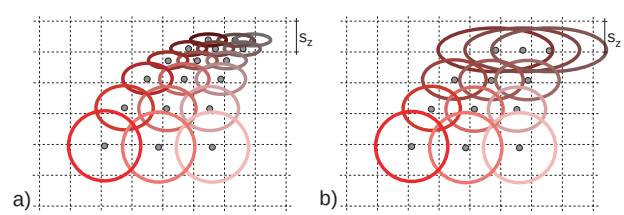


Figure 8: *Projected surfel mipmaps.*

Figure 8b shows the scaled texture footprints as ellipses around projected surfels.

To select the appropriate surfel mipmap level, we use traditional *view-dependent* texture filtering, as shown in Figure 9. A circle with

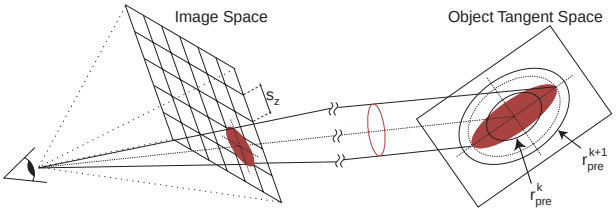


Figure 9: *Projected pixel coverage.*

radius $\frac{\sqrt{2}}{2}s_z$ is projected through a pixel onto the tangent plane of the surface from the direction of the view, producing an ellipse in the tangent plane. In this calculation, the projection of the circle is approximated with an orthographic projection. Similar to isotropic texture mapping, the major axis of the projected tangent space ellipse is used to determine the surfel mipmap level. The surfel color is computed by linear interpolation between the closest two mipmap levels with prefilter radii $r^k_{pre}$ and $r^{k+1}_{pre}$, respectively.

### 7.5 Shading

The illumination model is usually applied before visibility testing. However, deferred shading after visibility testing avoids unnecessary work. Grossman and Dally [12] perform shading calculations in object space to avoid transformation of normals to camera space. However, we already transform the normals to camera space during visibility splatting (Section 7.3). With the transformed normals at hand, we use cube reflectance and environment maps [28] to calculate a per-surfel Phong illumination model. Shading with per-surfel normals results in high quality specular highlights.

### 7.6 Image Reconstruction and Antialiasing

Reconstructing a continuous surface from projected surfels is fundamentally a scattered data interpolation problem. In contrast to other approaches, such as splatting [29], we separate visibility calculations from image reconstruction [22]. Z-buffer pixels with holes are marked during visibility splatting. These hole pixels are not used during image reconstruction because they do not contain any visible samples. Figure 10 shows the z-buffer after rendering of an object and the image reconstruction process.

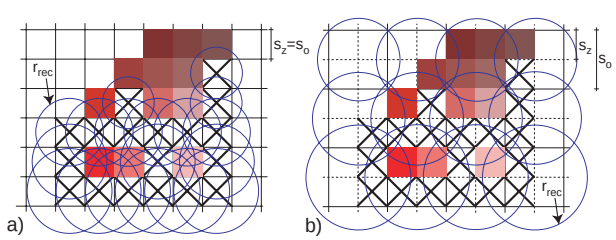


Figure 10: *Image reconstruction.*

The simplest and fastest approach, shown in Figure 10a, is to choose the size of an output pixel $s_o$ to be the same as the z-buffer pixel size $s_z$. Surfels are mapped to pixel centers using nearest neighbor interpolation, shown with color squares in the figure. Holes are marked with a black X. Recall that during forward warping each surfel stores $i_{max}$, an estimate of the maximum distance between adjacent projected surfels of a block. $i_{max}$ is a good estimate for the minimum radius of a pixel filter that contains at least one surfel. To interpolate the holes, we use a radially symmetric Gauss filter with a radius $r_{rec}$ slightly larger than $i_{max}$ positioned at hole pixel centers. Alternatively, to fill the holes we implemented the pull-push algorithm used by Grossman and Dally [12] and described by Gortler et al.[9].

A high quality alternative is to use supersampling, shown in Figure 10b. The output image pixel size $s_o$ is any multiple of the z-buffer pixel size $s_z$. Dotted lines in the figure indicate image-buffer subpixels. Rendering for supersampling proceeds exactly the same as before. During image reconstruction we put a Gaussian filter at the centers of all output pixels to filter the subpixel colors. The radius of the filter is $r_{rec} = \max(r'_{rec}, i_{max})$. Thus $r_{rec}$ is at least as large as $r'_{rec} = \frac{\sqrt{2}}{2}s_o$, but it can be increased if $i_{max}$ indicates a low density of surfels in the output image.

It is instructive to see how the color of an output pixel is determined for regular rendering and for supersampling in the absence of holes. For regular rendering, the pixel color is found by nearest neighbor interpolation from the closest visible surfel in the z-buffer. The color of that surfel is computed by linear interpolation between two surfel mipmap levels. Thus the output pixel color is calculated from two prefiltered texture samples. In the case of supersampling, one output pixel contains the filtered colors of one surfel per z-buffer subpixel. Thus, up to eight prefiltered texture samples may contribute to an output pixel for $2 \times 2$ supersampling. This produces image quality similar to trilinear mipmapping.

Levoy and Whitted [18] and Chang et al. [4] use an algorithm very similar to Carpenter's A-Buffer [2] with per-pixel bins and compositing of surfel colors. However, to compute the correct per pixel coverage in the A-buffer requires projecting all visible surfels. Max [20] uses an output LDI and an A-buffer for high quality anti-aliasing, but he reports rendering times of 5 minutes per frame. Our method with hierarchical density estimation, visibility splatting, and surfel mipmap texture filtering offers more flexible speed-quality tradeoffs.

## 8 Implementation and Results

We implemented sampling using the Blue Moon Rendering Tools (BMRT) ray tracer [10]. We use a sampling resolution of $512^2$ for the LDC for $480^2$ expected output resolution. At each intersection point, a Renderman shader performs view-independent calculations, such as texture filtering, displacement mapping, and bump mapping, and prints the resulting surfels to a file. Pre-processing for a typical object with 6 LOD surfel mipmaps takes about one hour.

A fundamental limitation of LDC sampling is that thin structures that are smaller than the sampling grid cannot be correctly represented and rendered. For example, spokes, thin wires, or hair are hard to capture. The rendering artifacts are more pronounced after 3-to-1 reduction because additional surfels are deleted. However, we had no problems sampling geometry as thin as the legs and wings of the wasp shown in Figure 1 and Figure 12.

The surfel attributes acquired during sampling include a surface normal, specular color, shininess, and three texture mipmap levels. Material properties are stored as an index into a table. Our system does currently not support transparency. Instead of storing a normal we store an index to a quantized normal table for reflection and environment map shading [28]. Table 1 shows the minimum storage requirements per surfel. We currently store RGB colors as 32-bit integers for a total of 20 Bytes per surfel.

| Data | Storage |
|---|---|
| 3 texture mipmap levels | $3 \times 32$ bits |
| Index into normal table | 16 bit |
| LDI depth value | 32 bit |
| Index into material table | 16 bit |
| Total per sample: | 20 Bytes |

Table 1: *Typical storage requirements per surfel.*

Table 2 lists the surfel objects that we used for performance analysis with their geometric model size, number of surfels, and file size

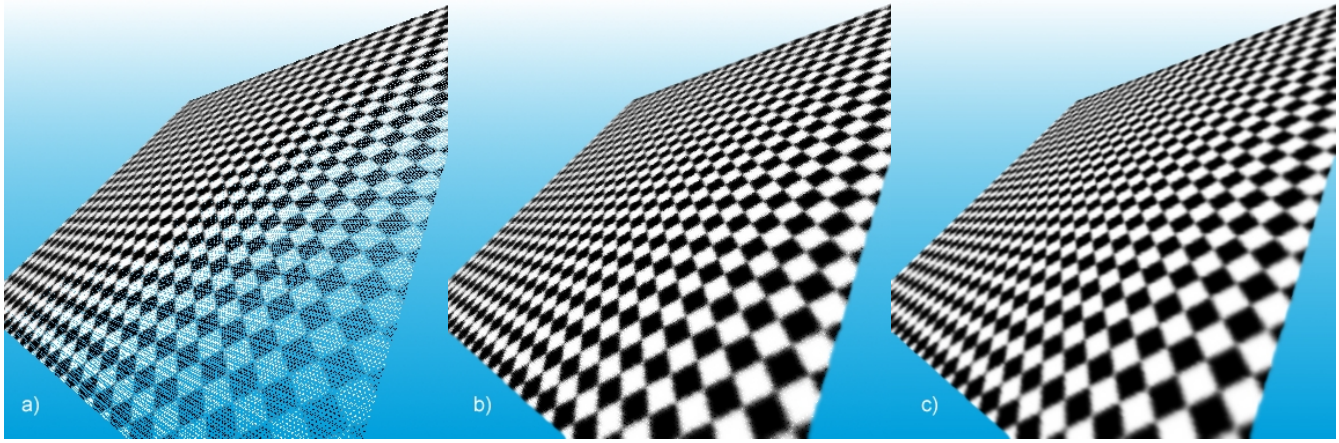


Figure 11: *Tilted checker plane. Reconstruction filter: a) Nearest neighbor. b) Gaussian filter. c) Supersampling.*

before and after 3-to-1 reduction. All models use three LODs and three surfel mipmap levels. The size of the LDC tree is about a factor of 1.3 larger than the LDC acquired during sampling. This

| Data | # Polys | 3 LDIs | 3-to-1 Reduced |
|---|---|---|---|
| Salamander | 81 k | 112 k / 5 MB | 70 k / 3 MB |
| Wasp | 128 k | 369 k / 15 MB | 204 k / 8 MB |
| Cab | 155 k | 744 k / 28 MB | 539 k / 20 MB |

Table 2: *Geometric model sizes and storage requirements (# surfels / file size) for full and 3-to-1 reduced LDC trees.*

overhead is due to the octree data structure, mainly because of the pointers from the lower resolution blocks to surfels of the sampled LDC. We currently do not optimize or compress the LDC tree.

Figure 1 shows different renderings of surfel objects, including environment mapping and displacement mapping. Figure 12 shows an example of hole detection and image reconstruction. Visibility splatting performs remarkably well in detecting holes. However, holes start to appear in the output image for extreme closeups when there are less than approximately one surfel per 30 square pixels.

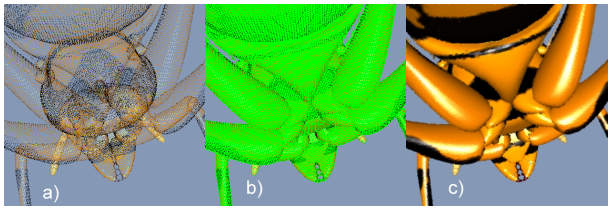


Figure 12: *Hole detection and image reconstruction. a) Surfel object with holes. b) Hole detection (hole pixels in green). c) Image reconstruction with a Gaussian filter.*

To compare image quality of different reconstruction filters, we rendered the surfel checker plane shown in Figure 11. There is an increasing number of surfels per pixel towards the top of the image, while holes appear towards the bottom for nearest neighbor reconstruction. However, a checker plane also demonstrates limitations of the surfel representation. Because textures are applied during sampling, periodic texture patterns are stored explicitly with the object instead of by reference. In addition, flat surfaces are much more efficiently rendered using image space rasterization, where attributes can be interpolated across triangle spans.

Table 3 shows rendering performance broken down into percentages per major rendering tasks. The frame rates were measured on a 700 MHz Pentium III system with 256 MB of SDRAM using an unoptimized C version of our program. All performance numbers are averaged over one minute of an animation that arbitrarily rotates

| Data | WRP | VIS | SHD | REC | CLR | fps |
|---|---|---|---|---|---|---|
| Output image: $256 \times 256$ | | | | | | |
| Salamander | 39% | 3% | 28% | 17% | 13% | 11.2 |
| Wasp | 61% | 4% | 21% | 8% | 8% | 6.0 |
| Cab | 91% | 2% | 5% | 1% | 1% | 2.5 |
| Output image: $480 \times 480$ | | | | | | |
| Salamander | 14% | 18% | 31% | 22% | 16% | 4.6 |
| Wasp 3to1 | 29% | 17% | 29% | 15% | 9% | 2.7 |
| Wasp 3LDI | 48% | 13% | 22% | 11% | 6% | 2.0 |
| Wasp SS | 15% | 22% | 28% | 18% | 16% | 1.3 |
| Cab | 74% | 7% | 11% | 5% | 3% | 1.4 |
| Output image: $1024 \times 1024$ | | | | | | |
| Salamander | 5% | 14% | 26% | 32% | 23% | 1.3 |
| Wasp | 13% | 19% | 25% | 26% | 17% | 1.0 |
| Cab | 16% | 36% | 24% | 16% | 8% | 0.6 |

Table 3: *Rendering times with breakdown for warping (WRP), visibility splatting (VIS), Phong shading (SHD), image reconstruction (REC), and framebuffer clear (CLR). Reconstruction with pull-push filter. All models, except Wasp 3LDI, are 3-to-1 reduced. Wasp SS indicates 2x2 supersampling.*

the object centered at the origin. The animation was run at three different image resolutions to measure the effects of magnification and holes.

Similar to image-based rendering, the performance drops almost linearly with increasing output resolution. For $256^2$ or object minification, the rendering is dominated by warping, especially for objects with many surfels. For $1024^2$, or large object magnification, visibility splatting and reconstruction dominate due to the increasing number of surface holes. The performance difference between a full LDC tree (Wasp 3LDI) and a reduced LDC tree (Wasp 3to1) is mainly in the warping stage because fewer surfels have to be projected. Performance decreases linearly with supersampling, as shown for 2x2 supersampling at $480^2$ resolution (Wasp SS). The same object at $1024^2$ output resolution with no supersampling performs almost identically, except for slower image reconstruction due to the increased number of hole pixels.

To compare our performance to standard polygon rendering, we rendered the wasp with 128k polygons and 2.3 MB for nine textures using a software-only Windows NT OpenGL viewing program. We used GL_LINEAR_MIPMAP_NEAREST for texture filtering to achieve similar quality as with our renderer. The average performance was 3 fps using the Microsoft OpenGL implementation (opengl32.lib) and 1.7 fps using Mesa OpenGL. Our unoptimized surfel renderer achieves 2.7 fps for the same model, which compares favorably with Mesa OpenGL. We believe that further optimization will greatly improve our performance.

Choosing the block size $b$ for the LDC tree nodes has an influence on block culling and warping performance. We found that a block size of $b = 16$ is optimal for a wide range of objects. However, the frame rates remain practically the same for different choices of $b$ due to the fact that warping accounts for only a fraction of the overall rendering time.

Because we use a z-buffer we can render overlapping surfel objects and integrate them with traditional polygon graphics, such as OpenGL. However, the current system supports only rigid body animations. Deformable objects are difficult to represent with surfels and the current LDC tree data structure. In addition, if the surfels do not approximate the object surface well, for example after 3-to-1 reduction or in areas of high curvature, some surface holes may appear during rendering.

## 9 Future Extensions

A major strength of surfel rendering is that in principal we can convert any kind of synthetic or scanned object to surfels. We would like to extend our sampling approach to include volume data, point clouds, and LDIs of non-synthetic objects. We believe that substantial compression of the LDC tree can be achieved using run length encoding or wavelet-based compression techniques. The performance of our software renderer can be substantially improved by using Pentium III SSE instructions. Using an occlusion compatible traversal of the LDC tree [21], one could implement order-independent transparency and true volume rendering.

Our major goal is the design of a hardware architecture for surfel rendering. Block warping is very simple, involving only two conditionals for z-buffer tests [11]. There are no clipping calculations. All framebuffer operations, such as visibility splatting and image reconstruction, can be implemented using standard rasterization and framebuffer techniques. The rendering pipeline uses no inverse calculations, such as looking up textures from texture maps, and runtime texture filtering is very simple. There is a high degree of data locality because the system loads shape and shade simultaneously and we expect high cache performance. It is also possible to enhance an existing OpenGL rendering pipeline to efficiently support surfel rendering.

## 10 Conclusions

Surfel rendering is ideal for models with very high shape and shade complexity. As we move rasterization and texturing from the core rendering pipeline to the preprocessing step, the rendering cost per pixel is dramatically reduced. Rendering performance is essentially determined by warping, shading, and image reconstruction — operations that can easily exploit vectorization, parallelism, and pipelining.

Our surfel rendering pipeline offers several speed-quality tradeoffs. By decoupling image reconstruction and texture filtering we achieve much higher image quality than comparable point sample approaches. We introduce visibility splatting, which is very effective at detecting holes and increases image reconstruction performance. Antialiasing with supersampling is naturally integrated in our system. Our results demonstrate that surfel rendering is capable of high image quality at interactive frame rates. Increasing processor performance and possible hardware support will bring it into the realm of real-time performance.

## 11 Acknowledgments

We would like to thank Ron Perry and Ray Jones for many helpful discussions, Collin Oosterbaan and Frits Post for their contributions to an earlier version of the system, and Adam Moravanszky and Simon Schirm for developing a surfel demo application. Thanks also to Matt Brown, Mark Callahan, and Klaus Müller for contributing code, and to Larry Gritz for his help with BMRT [10]. Finally, thanks to Alyn Rockwood, Sarah Frisken, and the reviewers for their constructive comments, and to Jennifer Roderick for proofreading the paper.

## References

[1] Animatek. Caviar Technology. Web page. http://www.animatek.com/.

[2] L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. In *Computer Graphics*, volume 18 of *SIGGRAPH '84 Proceedings*, pages 103–108. July 1984.

[3] E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.D. thesis, University of Utah, Salt Lake City, December 1974.

[4] C.F. Chang, G. Bishop, and A. Lastra. LDI Tree: A Hierarchical Representation for Image-Based Rendering. In *Computer Graphics*, SIGGRAPH '99 Proceedings, pages 291–298. Los Angeles, CA, August 1999.

[5] S. E. Chen. Quicktime VR – An Image-Based Approach to Virtual Environment Navigation. In *Computer Graphics*, SIGGRAPH '95 Proceedings, pages 29–38. Los Angeles, CA, August 1995.

[6] W. Dally, L. McMillan, G. Bishop, and H. Fuchs. The Delta Tree: An Object-Centered Approach to Image-Based Rendering. Technical Report AIM-1604, AI Lab, MIT, May 1996.

[7] M. Deering. Data Complexity for Virtual Reality: Where do all the Triangles Go? In *IEEE Virtual Reality Annual International Symposium (VRAIS)*, pages 357–363. Seattle, WA, September 1993.

[8] D. Ebert, F. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing & Modeling - A Procedural Approach*. AP Professional, second edition, 1994.

[9] S. Gortler, R. Grzeszczuk, R. Szeliski, and M. Cohen. The Lumigraph. In *Computer Graphics*, SIGGRAPH '96 Proceedings, pages 43–54. New Orleans, LS, August 1996.

[10] L. Gritz. Blue Moon Rendering Tools. Web page. http://www.bmrt.org/.

[11] J. P. Grossman. *Point Sample Rendering*. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, August 1998.

[12] J. P. Grossman and W. Dally. Point Sample Rendering. In *Rendering Techniques '98*, pages 181–192. Springer, Wien, Vienna, Austria, July 1998.

[13] P. Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics & Applications*, 6(11):56–67, November 1986.

[14] P. Heckbert. *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Science, June 17 1989.

[15] G. T. Herman. Discrete Multidimensional Jordan Surfaces. *CVGIP: Graphical Modeling and Image Processing*, 54(6):507–515, November 1992.

[16] A. Kaufman, D. Cohen, and R. Yagel. Volume Graphics. *Computer*, 26(7):51–64, July 1993.

[17] M. Levoy and P. Hanrahan. Light Field Rendering. In *Computer Graphics*, SIGGRAPH '96 Proceedings, pages 31–42. New Orleans, LS, August 1996.

[18] M. Levoy and T. Whitted. The Use of Points as Display Primitives. Technical Report TR 85-022, The University of North Carolina at Chapel Hill, Department of Computer Science, 1985.

[19] D. Lischinski and A. Rappoport. Image-Based Rendering for Non-Diffuse Synthetic Scenes. In *Rendering Techniques '98*, pages 301–314. Springer, Wien, Vienna, Austria, June 1998.

[20] N. Max. Hierarchical Rendering of Trees from Precomputed Multi-Layer Z-Buffers. In *Rendering Techniques '96*, pages 165–174. Springer, Wien, Porto, Portugal, June 1996.

[21] L. McMillan and G. Bishop. Plenoptic Modeling: An Image-Based Rendering System. In *Computer Graphics*, SIGGRAPH '95 Proceedings, pages 39–46. Los Angeles, CA, August 1995.

[22] V. Popescu and A. Lastra. High Quality 3D Image Warping by Separating Visibility from Reconstruction. Technical Report TR99-002, University of North Carolina, January 15 1999.

[23] W. T. Reeves. Particle Systems – A Technique for Modeling a Class of Fuzzy Objects. In *Computer Graphics*, volume 17 of *SIGGRAPH '83 Proceedings*, pages 359–376. July 1983.

[24] G. Schaufler. Per-Object Image Warping with Layered Impostors. In *Rendering Techniques '98*, pages 145–156. Springer, Wien, Vienna, Austria, June 1998.

[25] J. Shade, S. J. Gortler, L. He, and R. Szeliski. Layered Depth Images. In *Computer Graphics*, SIGGRAPH '98 Proceedings, pages 231–242. Orlando, FL, July 1998.

[26] A. R. Smith. Smooth Operator. *The Economist*, pages 73–74, March 6 1999. Science and Technology Section.

[27] J. Torborg and J. Kajiya. Talisman: Commodity Real-Time 3D Graphics for the PC. In *Computer Graphics*, SIGGRAPH '96 Proceedings, pages 353–364. New Orleans, LS, August 1996.

[28] D. Voorhies and J. Foran. Reflection Vector Shading Hardware. In *Computer Graphics*, Proceedings of SIGGRAPH 94, pages 163–166. July 1994.

[29] L. Westover. Footprint Evaluation for Volume Rendering. In *Computer Graphics*, Proceedings of SIGGRAPH 90, pages 367–376. August 1990.