Rendering by surfels

Ireneusz Tobor Christophe Schlick Laurent Grisoni

[grisoni|schlick|tobor]@labri.u-bordeaux.fr
LaBRI, Université de Bordeaux 1, France

Abstract

This paper studies a technique for real-time rendering of very large environments in which the geometric primitive is not the usual triangle but the "surfel" (surface element). A surfel is simply a 3D discrete sample of a surface, associated with a 2D "sprite" corresponding to its rendering on the screen. The principle of this technique is somewhat similar to "point sampling rendering" proposed by Grossman and Dally in 1998. In this paper we present an approach that takes full advantage of common hardware abilities for surfel manipulation. In other words, every step involved in our algorithm is decomposed into a set of elementary operations that can be done by hardware. A nice consequence is that one can easily mix surfel objects and triangle objets in a scene (or even switch from one representation to the other for a given object, according to the viewing parameters) and still benefit from hardware rendering of the whole scene. From that point of view, surfel objects can be considered as a new kind of impostors.

Keywords: Image Based Rendering, Point Sample Rendering, Impostors, Complex Environments, Level of Details.

1 Motivation

The rendering techniques implemented in almost every current 3D graphics hardware are based on algorithm developped in the middle of the seventies: approximation of the 3D objects by a set of triangles, projection of the triangles on the screen combined with incremental scan-line for shading and Z-buffer for hidden parts removal. When the first 3D graphics hardware architectures were developped, about 15 years ago, implementing such techniques was obviously the optimal choice, on a quality versus cost criterion, according to existing computer resources. But the situation has greatly changed since the beginning of 3D graphics hardware: the memory size of a middle class graphics workstation has approximatively been scaled by a factor 250 (say, from 1 Mbyte to 256 Mbyte) while its screen resolution has only been scaled by about a factor 2.5 (say, from 1024×768 to 1600×1200). The huge memory increase has made it possible to manage scenes with dramatically more triangles but as the resolution increase has been about 100 times less, the average size of the projected triangles has also dramatically decreased. Consequently, the main advantage of scan-line rendering (i.e. incremental computation of the triangle inner points) has mostly vanished, for scenes with high complexity.

Starting from that observation, a new trend called *Image-Based Rendering* (IBR, for short) has emerged within Computer Graphics, in the middle on the 90s. The principle of IBR is to replace the use of polygons by images, for the rendering of 3D scenes. The community working on IBR has rapidly grown and the number of contributions has exploded in the last three years. It is out of the scope of this paper to provide an exhaustive and detailled survey, as a large number of algorithms can be gathered under the IBR term: view interpolation [CW93, Che95], plenoptic modeling [LF94, MB95], flat impostors [MS95], delta trees [DMBF96], lightfields [LH96], lumigraphs [GSC96], mesh impostors [SDB97], nailboards [Sch97], layered depth images [SG98], point sampling rendering [GD98], multi-layered impostors [DSSD99], etc.

In our opinion, IBR techniques can roughly be divided in two main families. In the first one, let call it *plenoptic techniques*, authors try to find a general solution to the problem of rendering a scene from any viewpoint while you have only a finite set of pictures of it. The main problem to solve in plenoptic techniques is to fill the holes (i.e. the 3D points for which no information is available). The resulting algorithms are very different from usual 3D graphics, and so can only be implemented by software, at least with current graphics hardware. In the second familly, let call it *impostor techniques*, the goal is to define specific IBR techniques that could benefit from the capabilities of existing 3D graphics hardware. The general idea of the techniques proposed so far is to use textured polygons to replace the complex parts on the scene. The main problem to solve in impostors techniques is to avoid visible parallax artifacts.

In this paper, we study a technique that belongs to this second familly. The general idea is to replace a complex object by a set of *surfels* (surfel elements), that are discrete 3D samples of the object associated with a 2D "sprite" that corresponds to its rendering on the screen[1]. As a surfel object is a true 3D impostor, parallax artifacts are automaticaly solved.

The remainder of the paper is organized as follows. Section 2 presents the surfel representation. Section 3 details the surfelization (i.e. conversion of an 3D object into a set of surfels). Section 4 proposes the surfel collector, a specific data stucture to store surfels. Normal quantization is also studied in detail. Section 5 presents the results and Section 6 concludes.

2 Principle of surfels

The first use of 3D discrete samples as a rendering primitive in Computer Graphics can be attributed to Reeves in 1983, with his famous concept of *particle systems* [Ree83]. A particle is simply a point in the 3D Euclidian space, combined with some additional information, such as color, density, shading or scattering coefficients. The main advantage of particle systems is that the rendering step become trivial: project each particle on the screen, check for visibility with Z-buffer, and finally shade the corresponding pixel by using the color stored in the particle. Particle systems have mainly been used as a convenient modeling primitive to generate and animate specific objects that are hard to manage by conventional geometric models: fire and explosion [Ree83], waterfalls [Sim90], fluids [MP89]. Original particles systems, based on what we propose to call *isotropic particles*, were developped for modeling and rendering volumetric models. Szelisky and Tonnesen [ST92] adapted

[1] Note that, after having written this paper, we discovered that a couple of papers are going to be presented this year at Siggraph[PZvBG00, RL00], and propose some ideas similar to the technique presented here. The main difference between these papers and ours is that we consider surfels objects only as 3D impostors. Consequently we seek for high-speed hardware rendering with an automated LOD scheme, rather than for high-quality software optimization as in the other two papers.

particle systems to surfacic models, by introducing, the concept of *oriented particles*, based on work done by Reynolds [Rey87]. In this technique, each particle represents a sample of an underlying surface, and is associated to a 3D frame that represents the local behaviour (normal vector, tangent plane) of this surface.

The possibility to use particles to render solid objects was first investigated by Levoy in 1985 [LW85]. The idea he proposed was to sample a solid object into a set of 3D points with a sufficient density to give the visual impression of a solid object when indivual points are projected on the screen. A similar idea was proposed by Max [MO95] to render trees. The most complete work today on that topic was done by Grossman and Dally [GD98].

As with plenoptic modeling, the most challenging problem with discrete 3D points in to fill the holes that may appear between the points. Grossman and Dally proposed a very clever solution, using a so-called ”push-and-pull” mechanism. Unfortunately, this algorithm cannot benefit from current 3D graphics hardware, which means that the whole rendering has to be done by software, which dramatically reduces performance (about 5 frames per second for a scene with one single object). As we want to use hardware, and only hardware, we propose to use a cruder, but much more efficient, solution in which each point is rendered using a fixed pixmap (also called ”sprite” in the hardware graphics terminology) on the screen. Note that there exists another common and somewhat similar technique, called ”splatting”, mainly used in the rendering of 3D density grids [LH91]. Splatting consists in rendering each 3D points by a polygon with a textured transparency; as it is a true 3D rendering process, splatting is obviously much more computation demanding as rendering of sprites.

We call *surfel*, the combination of a 3D point with a rendering 2D sprite. Before detailling in the next sections, the process we use to manage this new 3D graphics primitive, we can briefly make some remarks about the advantages of such a surfel-based representation:

- Every geometric model (polygonal meshes, spline patches, implicit surfaces, CSG trees, etc) can be easily converted into surfels. We call this process *surfelization*; it is basically a 3D rasterization process. Section 3 presents two surfelization algorithms, one based on a software 3D rasterization, and another based on a set of hardware Z-buffer shots.

- Several level-of-details can be easily provided for each object by changing the resolution of the grid before surfelization process. According to the position of the camera, the most interesting representation level may be loaded.

- Another interesting feature is that, for a given scene, surfel-based objects can be mixed with classical geometric models, such as triangle meshes or spline patches, and still benefit from hardware rendering. When defining a scene by the usual *scene-graph* mechanism [SC92], surfel grids can be easily included by defining a new kind of geometric node. Thus, one can define a level-of-detail mechanism in which one can switch from the surfel-based representation to the triangle-based or spline-based one, when a very detailled view is needed. From that point of view, surfel objects can be considered as a new type of impostors.

- Finally, as rendering of each individual surfel is done in constant time, whatever its position, its orientation or its optical properties, a guaranteed frame rate can be provided simply by fixing the number of surfels to render per frame, according to the hardware characteristics.

3 Surfelization

Figure 1 presents the simple test scene that will be used throughout the paper. It is a sphere flake constructed using the SPD tools [SPD], composed of 164001 triangular faces, that can be seen on the wireframe view of Figure 2.

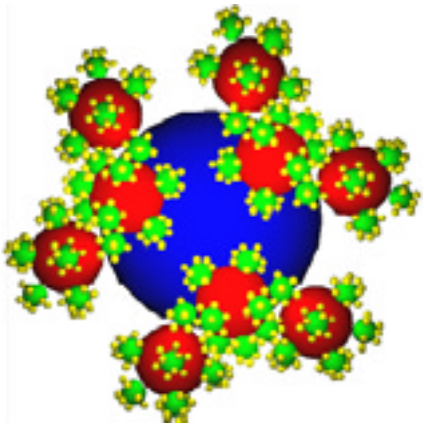

Figure 1: Shaded version of our test scene.

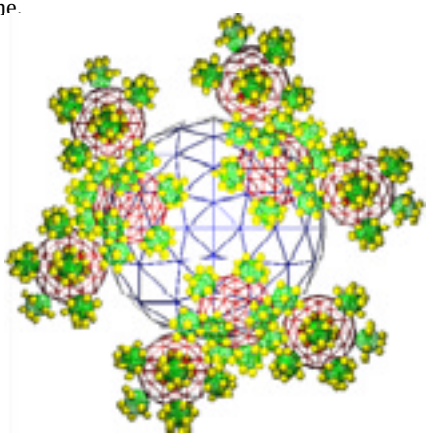

Figure 2: Wireframe version of our test scene.

A surfel-based representation of an object is actually a 3D grid where each cell stores a surfel. The following data required for rendering is associated to each surfel:

- The position of the surfel (this is implicitely given by the coordinates of the cell)

- The orientation of the surfel (this is provided by a normal vector which is actually quantized as explained below).

- The optical properties of the surfel (this can be provided either by a simple color or by an index to a table storing the different materials of the scene)

Consequently, surfelization (which means converting a geometric model into its equivalent surfel-based representation) is basically a rasterization process which extracts the desired informations and stores them in the corresponding grid cell. We have implemented two different algorithms for surfelization.

The first one is totally implemented by software. It consists in sampling the object at the grid resolution. In the case of implicit

surfaces or CSG trees, for instance, we simply evaluate the in/out function at each cell. In the case of polygonal meshes, each polygon is rasterized using classical scanline techniques adapted to 3D grids.

The second algorithm is much more interesting as it tries to benefit from existing 3D graphics hardware to speed-up the surfelization. Its principle is to repeatedly use hardware Z-buffer rendering to get all the information we need. For a given camera position, we use a first hardware Z-buffer with orthogonal projection to provide an image of the scene. This is called the *color buffer* and is usually based on Gouraud shading (Figure 1). An alternative solution, that we call *material buffer* consists in giving each pixel a constant color (without shading) according to the material of the object it belongs to. This material index will later be used for the rendering of surfels.

With Z-buffer the hardware also provides the *depth buffer* (Figure 3) which stores the camera to object distance for each pixel. By combining this information with the pixel coordinates, we can easily retrieve the 3D coordinates of each corresponding point. In a second step, we associate to each vertex of the mesh, a pseudo-color obtained by mapping the unit normal vector to the RGB domain (i.e. $[0,1]^3$). By applying a second hardware Gouraud shading with this configuration, we obtain what we call an *orientation buffer* (Figure 4) in which each pixel has got a pseudo-color corresponding to its normal vector. Consequently, for each pixel, we easily obtain the coordinates of the corresponding point in the scene, its color or material index and its orientation, which are all the informations required by the surfelization step. The coordinates are then quantized to the 3D grid resolution in order to find the grid cell where the surfel will be stored. Of course, this process only generates surfels that are visible from the current camera position. To generate a full surfelization of the scene, a multi-shot scheme based on a regular subdivision of the sphere, already described in [MO95, GD98], can be used. Note that the parts of an object that may be hidden in some concavities can also be captured by this harware process. The trick is simply to fetch the Z-buffer regularly, without waiting for the whole object to be rendered.

Depending on the rasterization, it might happen that two surfels fall into the same voxel of the underlying grid. In such a case, the surfel collector (see section 4) handles the collisions, and creates an averaging surfel. When the surfels are too different (i.e. normal pointing in different directions or different material indices), two surfels are stored in the cell.

In every case, for each object, several surfel-based representations, using different grid resolutions, are created. In our current implementation we store all the versions and use them to provide an efficient LOD mechanism during rendering. Note that smooth transition between an object at a given level of detail to the view at an immediately coarser level is guaranteed by the fact that neighboring surfels, when viewed from far enough, mix together on the screen, resulting in the same effect as the display of a unique "average" surfel.

4 Storage of surfels

The 3D grid data structure that appears as a straightforward solution for storing the result of the surfelization process is not used actually. As the surfels only exist at the surface of the objects, this grid is usually extremely sparse, so it would involve considerable memory waste. A classical data structure to efficiently store sparse grids is the *hash table*, which is particularly well-adapted to static data, as it is the case here. We propose a hash table structure, called *surfel collector*, that is optimized according to our specific needs.

Figure 3: Depth buffer of our test scene.

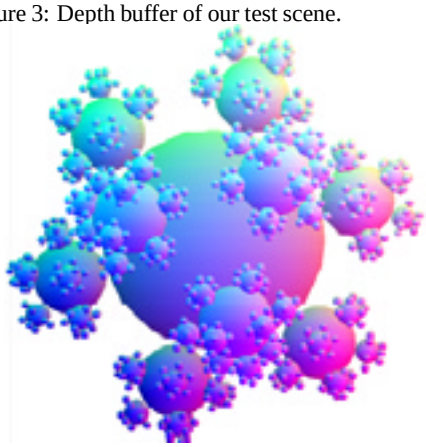

Figure 4: Orientation buffer of our test scene.

4.1 Surfel collector

The principle of the surfel collector is to quantize every information stored in the surfels and dispatch the corresponding bits among the hash table key and the hash table data. Once this principle is given, it can be tuned for any particular situation by changing the precision of the quantization and/or the length of the hash table. We describe here two possible implementations, a low (respectively high) quality collector where the storage of each surfel requires 4 (respectively 6) bytes of memory.

4.1.1 Low quality collector

The quantization used in this case is 24 bits for position quantization (which offers $256 \times 256 \times 256$ possible positions), 11 bits for orientation quantization (which offers 2048 possible orientations), and 9 bits for color or material index (which offers 512 possible materials). It means a total of 44 bits per surfel which are dispatched as follows:

- 12 bits are extracted to generate the hash key of a 4096 entries hash table. These bits are composed of the 4 less significant bits of each spatial coordinates, which guarantees a good shuffle for the surfels in the hash table.

- the 32 remaining bits are stored as a 4-byte word in the corresponding table entry. During the surfelization, table entries

are dynamically stored in a sorted linked lists. At the end, all lists are converted to static arrays.

4.1.2 High quality collector

The quantization used in this case is 30 bits for position quantization (which offers $1024 \times 1024 \times 1024$ possible positions), 15 bits for orientation quantization (which offers 32768 possible orientations), and 18 bits for color or material index (which offers 262144 possible materials). It means a total of 63 bits per surfel which are dispatched as follows:

- 15 bits are extracted to generate the hash key, of a 32768 entries hash table. Those bits are composed of the 4 less significant bits of the quantized spatial coordinates (in order to guarantee a good suffling), associated to the 3 sign bits of the quantized normals. See section 4.2 for details about normal quantization process.

- the 48 remaining bits are stored as a 6-byte word in the corresponding table entry.

This high-resolution version of the collector has the advantage, in comparison to the coarser one, that one can take advantage of the normal sign bits presence in the haskey, which can be used for a backface culling pre-calculus.

4.2 Orientation quantization

Floating point precision is far from being necessary in every CG applications. Quantizing floating point data to a given number of bits has been used since the early ages of computers. Quantizing point coordinates does not present any difficulty but efficient quantization of orientations (i.e. normal vectors) is a bit harder. Grossman and Dally [GD98] briefly mentionned orientation quantization, but did not describe their solution in detail. Deering [Dee95] addressed the general problem of orientation quantization and presented a method that takes full advantage of the symmetries involved. His method presents the drawback to have a linear complexity in term of possible quantized values: all possible dot products between the normal and the quantized values are evaluated, and the maximum defines the correct sample. Moreover, this technique involves expensive trigonometric functions.

Our choice is slightly different: we consider the octahedron joining the six points on the coordinate axis at a unit distance from the origin. Starting from this octahedron, we first reduce the sampling problem to the first octant by taking advantage of the sphere symmetry (i.e. we first extract the 3 sign bits of the normal). The sampling is done by recursively subdividing the triangle into 4 new smaller ones, and fairing the newly created mesh so that the new vertices were on the sphere (see Figure 5), which can straightforwardly be done by dividing the vector by its norm. When the whole approximating mesh is settled, the orientation samples are simply defined as the normal vectors of each triangle. Once the sampling is created, there are two problems to solve:

- First, how to store the precalculated normal values? Here, the hierarchical structure of the approximating mesh is used. Each triangle is assigned an integer value, which corresponds to its place in the quadtree hierarchical structure created during the mesh creation. In other words, for each subdivision level, each new triangle is assigned an integer number which is made by concatenating its father's index and two additionnal bits that define its location in the current level. Figure 6 presents the beginning of this construction. When all the mesh is processed, the normal vectors are stored in a one dimensional array ordered by these indices. In comparison to Deering's method, we do not need trigonometric functions, which produces faster evaluation.

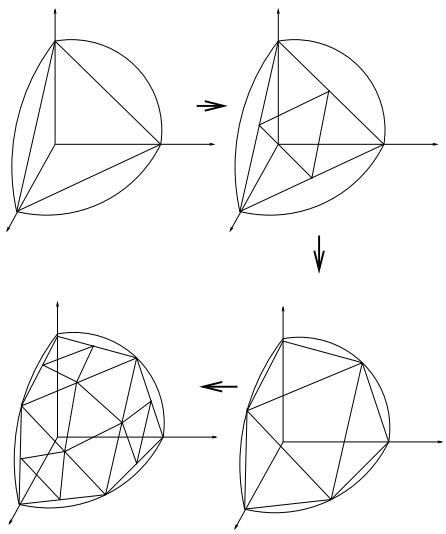

Figure 5: Normal sampling scheme, based on a octahedron. This scheme can be reproduced recursively at will in order to produce as fine a sampling as needed.

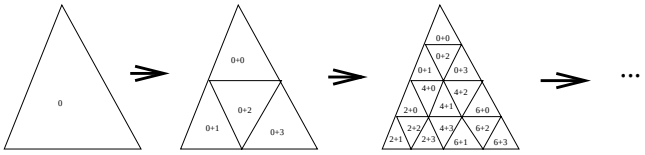

Figure 6: triangle index calculus process.

- Second, how to assign a quantized orientation to a floating point normal vector? This problem can be formulated in another way: given a non-quantized normal, how to find the triangle the normal belongs to. The hierarchical structure of the approximating mesh is again taken into account in order to achieve this search. The trick is first to look for the coarsest solid angle of the hierarchy the normal belongs to. This, from an algorithmic point of view, corresponds to the question to know if a point lies within a pyramid, and is a classical question in computational geometry. Once this coarsest triangle is determined, the search can recursively go on to the triangle's sons, until we reach the finest definition of the approximation mesh. The normal vector is then assigned the quantized orientation corresponding to the triangle. The complexity of this sampling algorithm is $O(logn)$, n being the number of triangles constituting the sampling of the octant, which is to be compared to the $O(n)$ complexity of Deering's sampling algorithm.

It can also be noticed that, if the user really wants to save memory, our method can still be used without storing the normal vector look-up table. As the algorithm recovering the quantized orientation from its associated index is rather efficient, recomputing them on-the-fly is not that expensive.

4.3 Advantages of surfel collector

One nice feature of the surfel collector data structure is that it is really straigthforward to implement, and easily adaptable to any

given particular application. It should be noted that our current implementation quantizes the (x,y,z) coordinates of the surfel position which leads to an implicit underlying cartesian grid. But one may also imagine to quantize cylindrical or spherical coordinates; the hash table does not mind about the meaning of the bits it crunches. Another advantage of this collector is point shuffling. When points are sequentially extracted from the collector, they are shuffled all over the sampling space. As a result, a sequential extraction will generate a display that does not follow any obvious logical organisation. This structure hence naturally fits real-time applications, where the amount of time per image rendering is limited: a simple algorithm displaying as many points as possible, until the time is over, will allow the user to see the general shape of the scene, no matter how complex it is.

5 Results

Figures 7 to 12 show the results we obtain by converting and rendering our test scene as surfels. Figures 7 and 8 shows the view obtained from the same camera position and the same picture size (i.e. 512x512) as Figure 1, when representing each surfel as a one single pixel. Figure 7 (resp. 8) uses a 128^3 (resp. 256^3) quantization for coordinates. Figure 9 (resp. 10) shows the same view as Figure 7 (resp. 8) by using a 4x4 (resp. 2x2) square sprite per surfel. When using OpenGL [Ope], this is simply obtained by a call to the function `glPointSize`. Thanks to the hardware, the rendering time is *exactly the same* as with the 1x1 pixel view. Note that the shading of the squares is obtained by applying the hardware shading model with the quantized normal vector. In our current implementation, selecting the size of sprite is done on a per-object basis, simply by comparing the size of the projected bounding box of an object with the surfel density. Of course, better adaptive schemes may be employed.

Figure 11 (resp. 12) shows again the same view with a 4x4 (resp. 2x2) sprite, but this time, the hardware antialiasing capability has been activated. The rendering time we obtain is almost equivalent to the previous ones. Depending on the material, this antialiasing is done either by drawing a flat disk instead of a square (this is the case on the pictures shown here), either by using a true antialised disk including partial tranparency. Note that partial transparency is a bit more expensive as it requires multipass rendering to avoid "false ordering" artifacts [RL00].

Figure 13 to 16 show another view of the scene, obtained by getting closer to the left part of the sphereflake. Figure 13 shows the result obtained with shaded polygons, Figures 14 to 16 is the result with 1x1 pixel, 5x5 square and 4x4 disk respectively. Of course with such a close view, the surfel representation should theoretically not be used and one should switch to the polygonal representation. But we wanted to show that even so, the result is quite acceptable, especially if you consider this image as a part of an animated sequence.

Finally, to test the usability of the surfel representation in large environments, we have generated an animation showing 4 sphereflakes flying around the camera. As each sphere is composed of about 164000 polygons, The polygonal version of the scene is composed of 656000 triangles. It can be rendered with a framerate of 0.5 frames per second. The surfel version uses a surfel collector with 5 level-of-details automatically generated and can be rendered at 10 frames per second on the same workstation (an Onyx2 computer, with InfiniteReality videocard). The shereflake model doesn't take advantage of the LOD algorithms, that are known to work well on spheres: we chose not to use those techniques because in the general case, such a scheme would involve tricky implementation in the animation working frame, which is to be compared to the immediate surfel application.

Figure 7: Visualisation using surfels of screen size 1 based on a 128^3 grid.

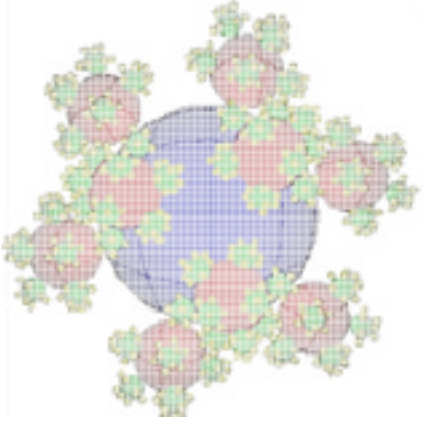

Figure 8: Visualisation using surfels of screen size 1 based on a 256^3 grid.

6 Conclusion and future work

In this paper, we studied a technique for real-time rendering of very large environments in which the geometric primitive is not the usual triangle but the "surfel" (surface element), a 3D discrete sample of a surface, associated with a 2D "sprite" corresponding to its rendering on the screen. The main feature of this technique is to account of the capabilities of current graphics hardware as every step is decomposed into a set of elementary operations that can be done by hardware. A nice consequence is that one can easily mix surfel objects and triangle objets in a scene and still benefit from hardware rendering of the whole scene. From that point of view, surfel objects can be considered as a new kind of impostors. We presented compact and efficient data structure for surfels, as well as a quantization normal technique, that allows efficient retrieval, without the use of look-up table.

We are currently investigating several extensions of the technique. In order to provide surfels with full LOD ability, it would be of high interest to have a structure that stores a "wavelet-like" implementation of surfels, that is, a recursive structure being composed of a coarse version and additional details, permitting to reconstruct finer surfels when needed. In addition to that, it is important to find a general solution to deal with the mixing of surfels for LOD representation, and anti-aliasing respect.

Currently, the surfelization is done on a per object basis. So, during the rendering, one switches between full polygon and full

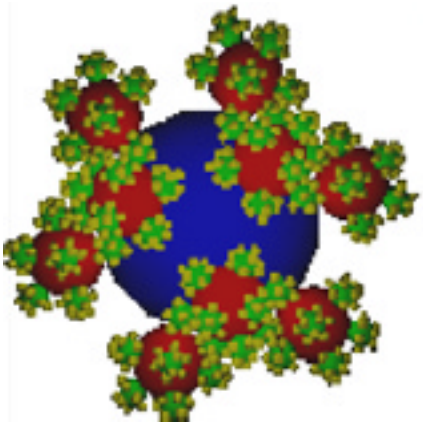

Figure 9: Visualisation using surfels of screen size 4 based on a 128^3 grid.

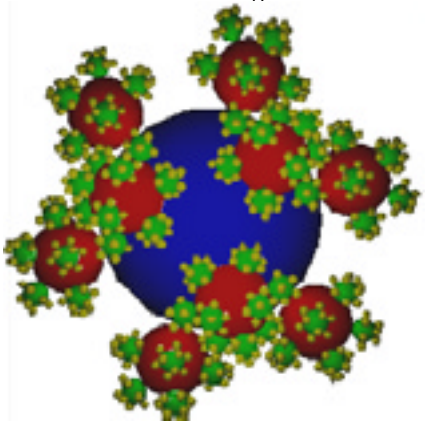

Figure 10: Visualisation using surfels of screen size 2 based on a 256^3 grid.

surfel representation of an object, according to the viewing parameters. One can imagine, a hierarchical surfelization principle, where only parts of an object can be surfelized (e.g. surfels for the small spheres, polygons for the large ones, in the sphereflake test scene). Similarly, clusters of objects could easily be surfelized as a whole, and of course this can be generalized to hierarchies of clusters. Another direction that we are investigating is to use surfels as a modeling primitive, in the same spirit as particle systems [Ree83, MP89, ST92, WH94].

References

[Che95] E. Chen. Quicktime - an image-based approach to virtual environment navigation. *Proc. of SIGGRAPH 95*, pages 29–37, 1995.

[CW93] E. Chen and L. Williams. View interpolation for image synthesis. *Proc. of SIGGRAPH 93*, pages 279–285, 1993.

[Dee95] M. Deering. Geometry compression. In *Proc. of Siggraph'95*, pages 13–20, 1995.

[DMBF96] W. Dally, L. MacMillan, G. Bishop, and H. Fuchs. The delta tree: An object centered approach to ibr. *Tech. Report Artificial Inteligence 1604*, 1996.

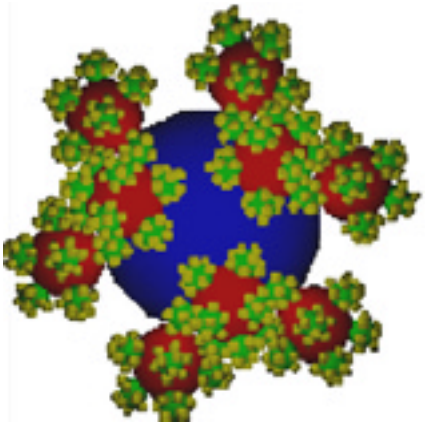

Figure 11: Visualisation using surfels of screen size 4 based on a 128^3 grid, combined with OpenGl anti-aliasing ability.

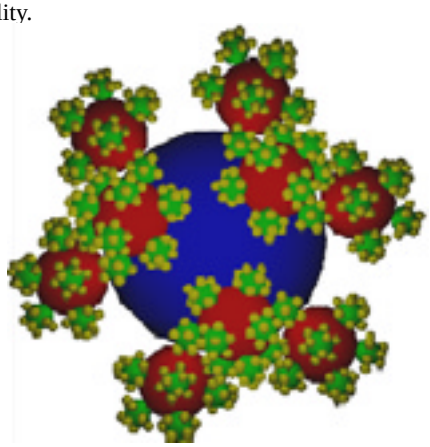

Figure 12: Visualisation using surfels of screen size 2 based on a 256^3 grid, combined with OpenGl anti-aliasing ability.

[DSSD99] X. Decoret, G. Schaufler, F. Sillion, and J. Dorsey. Multi-layered impostors for accelerated rendering. *Proc. of EUROGRAPHICS'99*, pages 231–242, 1999.

[GD98] J. P. Grossman and W. J. Dally. Point sample rendering. In *Proceedings of the 9th Eurographics Workshop on Rendering*, pages 181–192, 1998.

[GSC96] S. Gortler, R. Szeliski, and M. Cohen. The lumigraph. *Proc. of SIGGRAPH 96*, pages 43–54, 1996.

[LF94] S. Laveau and O. Faugeras. 3d scene representation as a collection of images and matrices. *Tech. Report 2205*, 1994.

[LH91] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. In *Proc. of Siggraph'91*, pages 285–288, 1991.

[LH96] M. Levoy and P. Hanrahan. Light field rendering. *Proc. of SIGGRAPH 96*, pages 31–42, 1996.

[LW85] M. Levoy and T. Whitted. The use of points as a display primitive. *Tech. Report 85-022*, 1985.

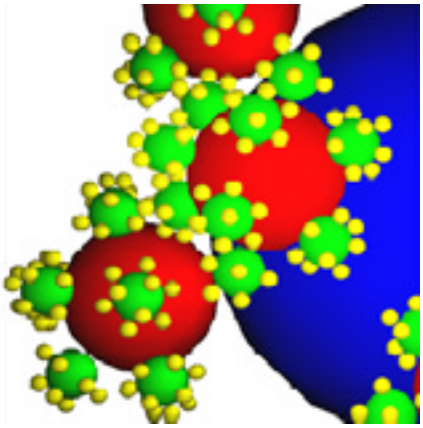

Figure 13: Close view on shaded original object.

Figure 14: Close view on the same area as on Figure 13, but using surfels of screen size 1 and grid of resolution 256^3.

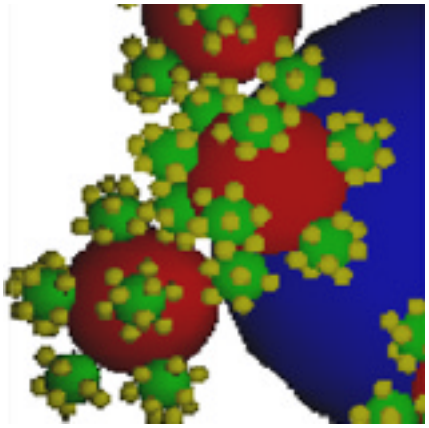

Figure 15: Close view on the same area as on Figure 13, but using surfels of screen size 5 and grid of resolution 256^3.

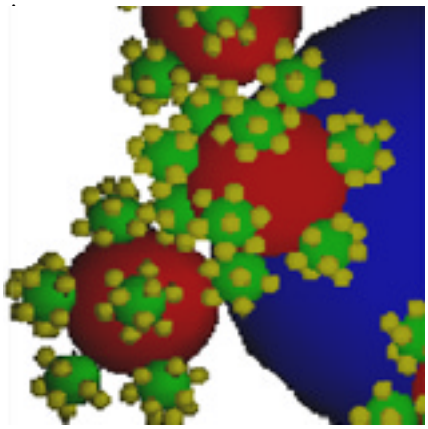

Figure 16: Close view on the same area as on Figure 13, but using surfels of screen size 4 and grid of resolution 256^3, combined with OpenGl antialiasing ability.

[MB95] L. MacMillan and G. Bishop. Plenoptic modeling: an image based rendering system. *Proc. of SIGGRAPH 95*, pages 39–46, 1995.

[MO95] N. Max and K. Ohsaki. Rendering trees from precomputed z-buffer views. *Proc. of 6th Eurographics Workshop on Rendering*, pages 45–54, 1995.

[MP89] G. Miller and A. Pearce. Globular dynamics: A connected particle system for animating viscous fluids. In *Proc. of Siggraph'89 (course 30 notes)*, pages 1–23, 1989.

[MS95] P. Maciel and P. Shirley. Visual navigation of large environnements using textured clusters. *Proc. of Symposium on Interactive 3D Graphics*, pages 95–102, 1995.

[Ope] Opengl, http://www.opengl.org.

[PZvBG00] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Proc. of Siggraph'2000*, 2000.

[Ree83] W. Reeves. Particle systems: A technique for modeling a class of fuzzy objects. In *Proc. of Siggraph'83*, pages 359–376, 1983.

[Rey87] C. Reynolds. Flocks, herds and schools: A distributed behaviour model. In *Proc. of Siggraph'87*, pages 25–34, 1987.

[RL00] S. Rusinkiewicz and M. Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proc. of Siggraph'2000*, 2000.

[SC92] P. Strauss and R. Carey. An object-oriented 3d graphics toolkit. In *Proc. of Siggraph'92*, pages 341–349, 1992.

[Sch97] G. Schaufler. Nailboards: a rendering primitive for image caching in dynamic scenes. *Proc. of 8th Eurographics Workshop on Rendering*, pages 129–136, 1997.

[SDB97] F. Sillion, G. Drettakis, and B. Bodelet. Efficient impostor manipulation for real-time visualization of ur-

ban scenery. *Proc. of EUROGRAPHICS'97*, pages 207–218, 1997.

[SG98] J. Shade and S. Gortler. Layered depth images. *Proc. of SIGGRAPH 98*, pages 231–242, 1998.

[Sim90] K. Sims. Particle animation and rendering using data parallel computation. In *Proc. of Siggraph'90*, pages 405–413, 1990.

[SPD] Spd tools, http://www.acm.org/pubs/tog/resources/spd/overview.html.

[ST92] R. Szeliski and D. Tonnesen. Surface modeling with oriented particle systems. In *Proc. of Siggraph'92*, pages 185–194, 1992.

[WH94] A. Witkin and P. Heckbert. Using particles to sample and control implicit surfaces. In *Proc. of Siggraph'94*, pages 269–278, 1994.