Hardware-accelerated Point Generation and Rendering of Point-based Impostors

J. Andreas Bærentzen

September 15, 2004

Abstract

This paper presents a novel scheme for generating points from triangle models. The method is fast and lends itself well to implementation using graphics hardware. The triangle to point conversion is done by rendering the models, and the rendering may be performed procedurally or by a black box API.

I describe the technique in detail and discuss how the generated point sets can easily be used as impostors for the original triangle models used to create the points. Since the points reside solely in GPU memory, these impostors are fairly efficient.

Source code is available online.

1 Introduction

The main advantage of points over triangles lies in the simplicity of the point primitive. This means that the rendering pipeline for points is simpler than for triangles. Consequently, point rendering is potentially faster than triangle rendering when the average triangle size approaches that of a pixel.

Points were originally proposed as a primitive for surface rendering by Levoy and Whitted [10]. Recently, the work by Grossman and Dally [7] seems to mark the beginning of a renewed interest in the point as a rendering primitive, and the last few years have seen a number of techniques pertaining to rendering and modelling objects using points. I suggest the reader consults [9] for a recent survey.

While there have been many papers on point rendering and manipulation of point models, techniques for the generation of point models have received less attention. In this paper, a simple and efficient technique for converting triangle models to point sets is presented. The basic idea is to render the object a number of times from a set of directions. For each direction, all the visual layers of the object are rendered using a technique called *depth peeling* [5]. From the non–background pixels of each layer, point information is extracted.

1.1 Related Work

Most techniques for point generation mimic rendering, but generate points instead of pixels. For instance, Grossman and Dally project an object orthographically from 32 directions [7]. However, instead of sampling the projection on a square lattice they use a lattice of equilateral triangles (i.e. the pixels of the projection are the vertices of equilateral triangles).

Pfister et al. [12] use a ray tracer to generate their sampling. The object is rendered from three perpendicular directions, and all intersections along a ray are recorded. Thus, a *layered depth images* (LDI) [15, 11] is generated from each of the three directions. A layered depth image is an image where each pixel contains a list of depth values. An LDI can be converted to a point cloud simply by transforming the pixel space coordinates (x,y,depth) back into object space.

Tobor et al. [17] propose a hardware based approach which is similar to mine, but which uses only one z-buffer. An object is rendered from multiple directions, and, in the first pass, depth and color are extracted. Subsequently, normal information is extracted by storing normals as colors. To get all the layers of an object, the z-buffer is fetched regularly. However, "regularly" has to be after every triangle, unless the scene is partitioned into sets of triangles which are guaranteed not to overlap. Assuming that one reads back the depth buffer after every triangle (or just very frequently) pipeline stalling results since the pipeline is flushed before performing a read. Moreover, it is problematic to read the framebuffer too frequently since the graphics pipeline is optimized for sending rather than reading data.

Not all authors use rendering methods to generate points, though. Stamminger and Drettakis propose a hierarchical sampling scheme called the $\sqrt{5}$ scheme [16]. The scheme is based on sampling a parameter domain. Another example is the work by Michael Wand et al. who developed a probabilistic sampling strategy [18].

2 Method

To generate points from a triangle mesh model, a technique called depth peeling is used. Depth peeling can be used to render all the visual layers of a model. The first visual layer is what one normally sees. The second visual layer is the set of surfaces occluded only by the first layer. This set cannot be rendered directly using modern graphics hardware due to the absence of two z-buffers. However, given two z-buffers this is almost as simple. Having rendered the first layer, the normal z-buffer is copied to the second z-buffer and rendered again. During this rendering the 1st z-buffer is used (as usual) to reject a fragment if its z value is greater than the stored z value. The 2nd z-buffer is used to reject a fragment if its z value is smaller or equal to the stored value. Finally, the 1st z-buffer is updated as usual with the depth of fragments that pass the depth test whereas second z-buffer is read-only.

After rendering, the 1st z-buffer is copied to the 2nd z-buffer, and we are ready to render the third layer. Simply iterating this process until nothing is rendered produces all layers in the scene. For each layer a depth buffer and at least one color buffer are obtained. The resulting representation is a layered depth image.

Of course, it is possible to keep the second z-buffer if one wishes to render the same layer more than once. In fact, it is often useful to render a layer at least two times: First to extract color information and then to extract normal information.

The only problem is that graphics cards generally do not have more than one z-buffer. However, this problem can be overcome using extensions for shadow mapping which, essentially, provide an extra z-buffer using texturing. This method has been used by Cass Everitt to implement depth peeling, and details regarding the implementation are provided in [5]. Everitt used depth peeling as a part of a scheme for order-independent transparency.

Having rendered all visual layers in an object, the resulting images are traversed and the pixel x y and depth values are combined to obtain a point in space. This technique has a number of virtues:

- Being hardware accelerated, depth peeling is likely to be faster than any software based technique for LDI generation.
- Any layer can be rendered any number of times. Thus one may extract any number of attributes for each point.
- It is not necessary to have access to the code that renders the original triangle model. For instance, a vertex program can be used to copy normals to colors. Thus, it is possible to generate an impostor from an object rendered by a function from a closed source API (as long as the API does not load its own vertex programs).

I compute three LDI representations using the above method. Each LDI corresponds to a viewing direction parallel to one of the X, Y, and Z axes. Two color buffers are generated for each layer. The first buffer actually contains color, and the second contains normals (stored as RGB values). For each viewing direction and for each (non-background) pixel in each layer, it is tested whether the normal is most parallel to the given viewing direction. If that is the case, the 3D position of the pixel is computed from its xy position and depth. Of course, the position is in screen coordinates, and it must be converted back to object coordinates. Finally, the <color, normal, position> triple is added to a vector of points. When all layers have been processed the LDI representation can be discarded, and the final output is a point cloud represented by the aforementioned vector.

Note that the method applies to *polygon–soups*, i.e. the models do not have to be manifold or watertight. It is, however, important that back face culling is not enabled. Otherwise, it would not be possible to capture all visual layers when rendering from a given direction.

2.1 Variations

If we do have access to the individual triangles of the model, some speedups are possible. One may partition the triangles into three sets where each set contains the triangles most perpendicular to a corresponding viewing direction. If the model consists of many triangles, the rendering is likely to be geometry limited, and this greatly enhances performance. It also means that one need not detect which points to throw away afterward – since the selection has been performed on a per triangle basis.

Unfortunately, this acceleration sometimes leads to a slightly decreased point density where two neighboring triangles belong to different sets. A simple fix is to assign a triangle to a given set if it is close to perpendicular to the projection direction.

It is possible to generate LDIs from more than three directions. I find that this improves the precision measurably, but it does not make a difference in the subsequent visualization as the precision is good enough using only three LDIs. Typically, the error is on the order of a millionth of a unit when the model is unit size. The error depends almost entirely on the depth buffer precision (which is often 24 bits). The XY position is known since fragments are sampled at their centers [14].

3 Implementation

Depthpeeling has been implemented using a class called `DepthPeeler`. `DepthPeeler` simply provides a second z-buffer, and it has a very simple interface:

```
class DepthPeeler
{
  // ...
public:
  DepthPeeler(int width, int height);
  void disable_depth_test2();
  void enable_depth_test2();
  void read_back_depth();
};
```

`enable_depth_test2` and `disable_depth_test2` allow the user to switch on and off the depth test. `read_back_depth()` copies the current depth buffer to the second depth buffer. The user of the class is responsible for copying any other information from the frame buffer.

A 24 bit depth buffer is used. The depth texture is a rectangular texture of the internal format `GL_DEPTH_COMPONENT24`. The texture parameters are set up for depth comparison which means that the depth in the depth texture is compared against the R texture coordinate where R codes the depth of the fragment. The result of the comparison is an alpha value of 1 if the fragment should pass and 0 otherwise. For each fragment a very simple program is used. The program simply uses the fragment xyz position as texture coordinates. The result of the texture lookup is an alpha value which is copied to the alpha value of the fragment whose color is otherwise unchanged. Later an alpha test is used to reject fragments whose alpha value is 0. The implementation uses only standard OpenGL [14], `TEXTURE_RECTANGLE_NV`, and `FRAGMENT_PROGRAM_ARB`.

Using the `DepthPeeler`, it is very simple to implement point generation as explained above. The method has been implemented using OpenGL on a PC equipped with a GeforceFX 5900 graphics card[1]

4 Examples and Discussion

Figure 1: The Happy Buddha model. The original triangle model is on the left, and the point model which consists of 6360009 points (to ensure high fidelity) is shown on the right. Note though that with so many points, the triangle model is actually faster to render (6.7 fps vs. 2.1 fps). Both the mesh model and the point model were rendered using antialiasing.

The method has been applied with good results. Figure 1 shows two images of the *happy buddha*. The one on the left was rendered using triangles and the other

[1]Source code for both point generation and point rendering (Section 5) has been made available online at the address listed at the end of this paper.

one using points. Although the two images are not pixel–identical, they are almost indistinguishable.

I have timed the point generation program, and the results are shown in Table 1. Clearly the timings depend a lot on the depth complexity. For most of the other models, the LDI viewport size (and hence the number of generated points) used during point generation seems to be the other most important factor. However, in the case of the buddha model, the difference between the small viewport and the large one is less pronounced. This is probably due to the fact that the model is so large. For very large models, we conclude that the geometry stage (transformation of triangle vertices) dominates the point generation process.

Note that this method (like other LDI based methods, such as [12]) produces a regular sampling which does not take local variations in object detail into account. This issue could be dealt with by infusing points with more curvature information and letting the point size be determined by this information. However, since the sampling is still regular, the point cloud would have to be thinned. These issues are discussed by Kalaiah and Varshney [8]. A technique for estimating curvature from points is mentioned in Fleishman et al. [6].

Because depth peeling is implemented using graphics hardware, the method is fairly efficient – even though the model is rendered a number of times. It is also an advantage that the rendering pipeline is used for point generation. Because of this, the point set can be generated from a procedural model or a model rendered by a black box API.

5 Rendering Point Impostors

As an application of the method, I present a scheme for rendering impostors of complex geometry such as trees. Point based imposters have been used previously as a component of systems for rendering large scenes [19], and the notion is somewhat related to layered impostors [13].

Stamminger et al. suggested that one might create a set of points by sampling random locations on an object [16, 4]. This creates a list of points spread out over the object, and any prefix of this list is a shorter list of points that are also randomly distributed over the object. Thus, by removing a point, we, effectively, go down a level of detail. However, since the absence of a single point is not noticeable, the transition is very smooth as pointed out in [16].

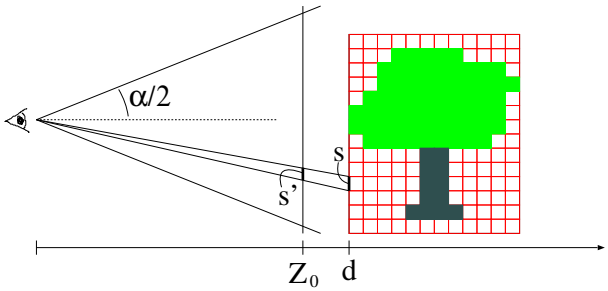

Figure 2: This figure illustrates s', s, Z_0, d, and the viewing angle α.

Instead of randomly sampling, I create a random permutation of the points. There are many ways to create such a permutation, and it is obviously desirable that the points are very evenly spread out over the object. A number of strategies were tried, for instance iteratively choosing the point farthest from the set of points already selected (an algorithm known as Hochbaum-Shmoys [2]). However, simply creating a random permutation using the C++ Standard Template Library function `random_shuffle` produces a result that is visually indistinguishable from the more sophisticated methods.

The number of points to use is determined by the distance from the viewer to the model. Specifically, a number of points N out of the total number N_0 is selected so that N/N_0 equals the ratio of the projected area of a pixel in the LDI to a pixel in the frame buffer. Specifically,

$$N = N_0 (s')^2$$

where s' is the projected side length of a pixel in the LDI representation. Let Z_0 be the distance from the eye to the plane where a unit in world space corresponds to the side length of a pixel, and let d be the distance from the eye point to the impostor. Now,

$$\frac{s}{d} = \frac{s'}{Z_0} \iff (s')^2 = \left(\frac{sZ_0}{d}\right)^2 .$$

By substitution

$$N = N_0 (s')^2 = N_0 \left(sZ_0/d\right)^2 . \qquad (1)$$

From basic trigonometry, we know that

$$Z_0 = \frac{W}{2\tan(\alpha/2)} , \qquad (2)$$

where W is the window height and α is the viewing angle. When $N > N_0$ we switch to the triangle model. (1) and (2) are illustrated in Figure 2.

Model	LDI size	Loading	Triangles	X	Y	Z	Generation	Points	Total
Man	100	0.002714	656	8	4	2	0.030281	8690	0.123
Cow	100	0.027424	5804	5	4	4	0.069103	8891	0.205
Tree	100	0.928199	67450	21	16	37	0.853004	27829	1.89
Bunny	100	0.303191	69451	3	4	4	0.530025	20395	0.946
Happy Buddha	100	5.5545	1087716	6	10	6	7.44342	12762	13.238
Man	1000	0.002818	656	8	4	3	1.76825	863837	2.225
Cow	1000	0.025001	5804	5	4	5	2.15941	875494	2.58
Tree	1000	0.977727	67450	25	20	46	22.2428	2763228	24.305
Bunny	1000	0.299279	69451	4	5	4	5.13714	2018092	6.213
Happy Buddha	1000	5.62718	1087716	9	10	6	10.864	1244814	17.201
		secs.			layers		secs.		secs.

Table 1: This table shows timings for point generation. LDI size refers to the maximum side length of any of the viewports used when generating the X, Y, Z LDIs. Except for "Man" and "Tree", the models are all well known standard models in computer graphics. The "Man" model is shown in Figure 4 (right), and the trees are shown in Figure 3.

An important advantage of the method is the fact that there is no popping. The only place where visible popping is possible is when we switch to the triangle model. To avoid popping in this case, the triangle model is faded in, and then the point model is faded out. This fading is done as a function of distance rather than time and it removes the last traces of popping.

The technique discussed above is based on rendering either points or triangles (except when blending the two representations). A slightly different approach would be to combine point and triangle rendering as advocated by some authors. For instance, Chen et al. [1] build a tree data structure from a triangle mesh. Parts of the tree may be drawn as points and parts as triangles. While this may be useful for very large models, it is unlikely that the traversal of a hybrid hierarchy can be performed on the graphics card. Currently, only the traversal of a pure point hierarchy on the graphics card has been proposed [3]. Hence, in the context of impostors, I suggest that the best way to combine point and polygon rendering would be to divide the model into coarse features which are best represented using polygons and small details which are approximated using points.

5.1 Implementation Issues

A salient feature of the method is the fact that the all levels of detail are represented by a single list of points which is stored in GPU memory. The ordering should be randomized only once on the CPU and then sent to the graphics card (e.g. using vertex buffer objects). A random subset of N points may then be rendered simply by rendering the first N points in the list stored on the graphics card.

Note that when rendering points which do not represent smooth surfaces, it is very important to enable point smoothing since aliasing artifacts can otherwise be very noticeable.

5.2 Examples and Discussion

The point impostor methods was used to render trees in a simple terrain rendering application. A screen dump from this application is shown in in Figure 3.

An important issue is setting the distance d_0 where the application switches from the polygonal model to the point impostor. If d_0 is too small, many points are required. This entails a higher point rendering cost and a large memory overhead due to the size of the point cloud. On the other hand, if d_0 is too big, the full advantage of the point impostors is not reaped.

Thus, the setting of d_0 involves a tradeoff. This tradeoff is illustrated in Table 2 which compares sizes and rendering times for the polygonal models and the point impostors used in the terrain application. The models were rendered at distances ranging from 100 to 2000 units. The corresponding point renderings (of Tree no. 2) are shown in Figure 4 (left). In summary, the scheme is particularly simple, and it allows the point set to reside in GPU memory which is essential to ensuring fast rendering.

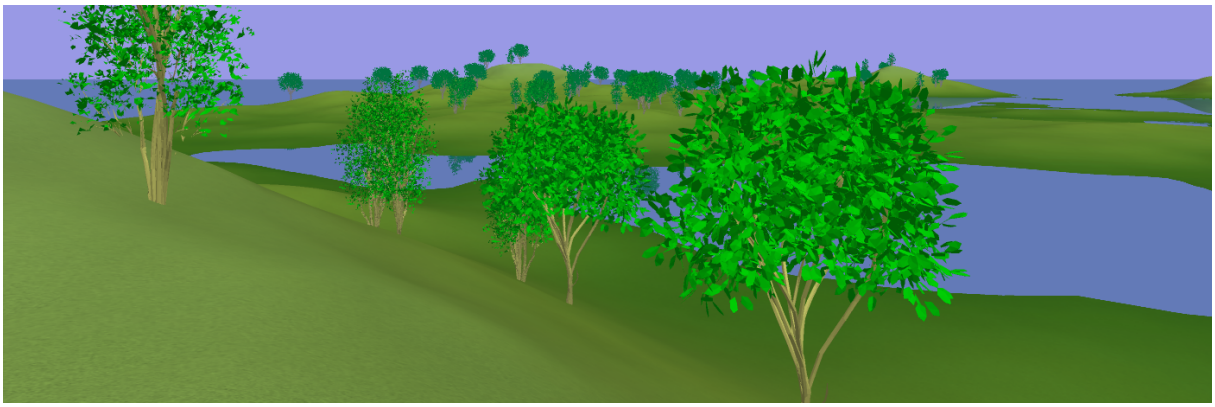

Figure 3: Point rendered tree impostors in a simple landscape. The four closest trees are rendered either using triangles or a blend of triangles and points. The farther trees are rendered exclusively using points.

Tree no. 1 (mesh size: 1.1 MB))				
	points			mesh
Distance	no.	size	FPS	FPS
2000	1269	0.05 MB	115	5.2
1000	5079	0.2 MB	57	5.2
500	20312	0.8 MB	19	5.2
100	57744	2.3 MB	7.5	5.2
Tree no. 2 (mesh size: 1.7 MB)				
	points			mesh
Distance	no.	size	FPS	FPS
2000	4929	0.2 MB	58	2.2
1000	19720	0.79 MB	20	2.2
500	78815	3.2 MB	5.5	2.2
100	118398	4.7 MB	3.7	2.2

Table 2: This table shows frame rates for rendering 50 instances of the two tree model at distances ranging from 100 to 2000 units. The sizes of the corresponding point clouds are also shown. While the frame rates for the mesh is constant (as expected), the frame rates for point rendering depend greatly on the distance and thus number of points used.

6 Web Information

The images from Figure 1, an animation of the scene in Figure 3, and C++ source code for both generation and point rendering are available at http://www.acm.org/jgt/papers/Baerentzen05

References

[1] Baoquan Chen and M. X. NGuyen. Pop: A hybrid point and polygon rendering system for large data. In *IEEE Visualization 01*, 2001.

[2] Hochbaum D. and Shmoys D. A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, 10(2):180–184, 1985.

[3] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Transactions on Graphics*, 22(3):657–662, 2003.

[4] O. Deussen, C. Colditz, M. Stamminger, and G. Drettakis. Interactive visualization of complex plant ecosystems. *Visualization, 2002. VIS 2002. IEEE*, pages 219–226, 2002.

[5] C. Everitt. Interactive order-independent transparency. Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, May 2001. Available at http://www.nvidia.com/., 2001.

[6] Shachar Fleishman, Daniel Cohen-Or, Marc Alexa, and Claudio T. Silva. Progressive point set surfaces. *ACM Transactions on Graphics*, 22(4):997–1011, 2003.

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

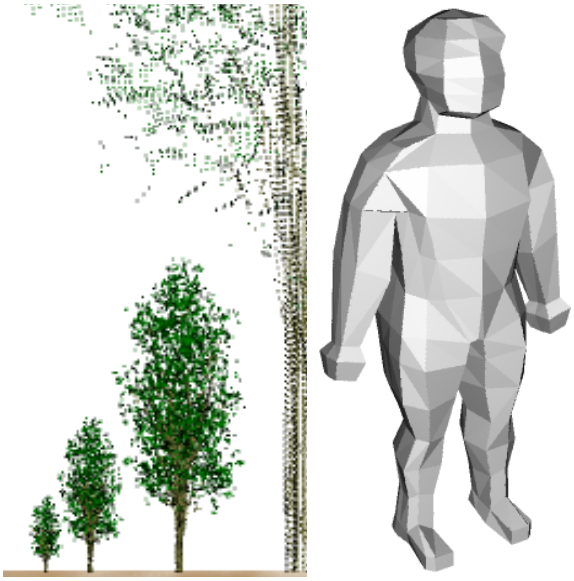

Figure 4: (left) The same tree rendered at distances of (form left to right) 2000, 1000, 500, and 100 units. At short distance, the point cloud looks sparse although all points are used. At greater distances, the cloud looks dense, contains much fewer points and renders in a fraction of the time. (right) Point rendering of the "Man" figure used in the timings shown in Table 1.

[8] Aravind Kalaiah and Amitabh Varshney. Differential point rendering. In K. Myskowski and S. Gortler, editors, *Rendering Techniques 2001*, 12th Eurographics workshop on Rendering. Springer Verlag, 2001.

[9] Jaroslav Krivanek. Representing and rendering surfaces with points. Technical Report DC-PSR-2003-03, Department of Computer Science and Engineering, Czech Technical University in Prague, 2003.

[10] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical report, University of North Carolina at Chapel Hill, 1985.

[11] Dani Lischinski and Ari Rappoport. Image-based rendering for non-diffuse synthetic scenes. In *Proceedings, Ninth Eurographics Workshop on Rendering*, pages 301–314, 1998.

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

[13] Gernot Schaufler. Per-object image warping with layered impostors. In *Eurographics Workshop on Rendering*, pages 145–156, June 29-July 1 1998.

[14] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 2.0)*. SGI, 2004.

[15] Jonathan Shade, Steven Gortler, Li wei He, and Richard Szeliski. Layered depth images. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242. ACM Press, 1998.

[16] Marc Stamminger and George Drettakis. Interactive sampling and rendering for complex and procedural geometry. In K. Myskowski and S. Gortler, editors, *Rendering Techniques 2001*, 12th Eurographics workshop on Rendering. Springer Verlag, 2001.

[17] I. Tobor, C. Schlick, and L. Grisoni. Rendering by surfels. In *Proceedings of Graphicon*, 2000.

[18] M. Wand, M. Fischer, I. Peter, F.M. auf der Heide, and W. Strasser. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. *Computer Graphics Proceedings. SIGGRAPH 2001*, pages 361–70, 2001.

[19] Michael Wimmer, Peter Wonka, and François Sillion. Point-based impostors for real-time visualization. In *Eurographics Workshop on Rendering*, pages 163–176, 2001.