Efficient screen space approach for Hardware Accelerated Surfel Rendering

G. Guennebaud, M. Paulin

CNRS - IRIT, Université Paul Sabatier , Toulouse, France
Email : guenneba@irit.fr, paulin@irit.fr

Abstract

At present, the best way to render textured point-sampled 3D objects is doubtless the use of elliptical weighted average (EWA) surface splatting. This technique provides a high quality rendering of complex point models with anisotropic texture filtering. In this paper we present a new multi-pass approach to perform EWA surface splatting on modern PC graphics hardware. The main advantage of our approach is to be bandwidth limited because we render only one vertex by sample. To achieve this goal we efficiently use the standard OpenGL point primitive. During the first pass, visibility splatting is performed by shifting surfel backward along the viewing rays and apply a parallax depth correction on each fragment. During the second rendering pass, screen space EWA filtering is computed for each vertex and performed for each fragment. Our algorithm is implemented using programmable vertex and fragment shaders of newer PC graphics hardware.

1 Introduction

Today, laser range and optical scanners generate a huge volume of point samples. Rendering or manipulating this mass of data is a main challenge for the community. In order to render this point cloud, a common way is to reconstruct triangle meshes from the samples using mesh reduction [7]. However, this approach has some drawbacks. First, for some application it can be useful to render data during, or immediately after acquisition. However, in this case the reconstruction phase leads to time latency between acquisition and rendering. Secondly, the reconstruction itself is not fine enough, and it can become incorrect when scanned object is too complex. Finally, mesh simplification algorithms can lead more geometric and texturing artefacts.

This explains the motivation of many recent efforts to propose a point sample rendering algorithm. Here, the object surface is defined by a dense set of sampled points without connectivity, commonly called surface elements or surfels. The main challenge of these algorithms is to directly render a large point set, i.e. to reconstruct a continuous image of the point cloud. To complete this task efficiently, a hierarchical data structure is useful for storing surfel set and rendering. This allows hierarchical visibility culling and multi-resolution rendering.

Today, to achieve interactive rendering performances, a hardware-accelerated approach is compulsory. Most point-rendering algorithms use graphics hardware acceleration. Otherwise, to render point models with complex surface texture, the EWA surface splatting algorithm [16] is doubtless the best method. It is the only approach that can support anti-aliasing with anisotropic texture filtering. EWA surface splatting was first introduced by Zwicker et al. [16] in a screen space formulation and software implementation. Recently, Liu Ren et al. [13] extended EWA surface splatting with an object space formulation that allows a hardware-accelerated implementation. However, this technique, as the most part of techniques using hardware, represents each surfel by a quad. Since to render one surfel we need to project four vertices, the bandwidth is wasted. Moreover, in the object space EWA surface splatting, the same computation is performed four times by the GPU.

In this paper, our motivation is to limit the bandwidth use, and consequently increase rendering speed, while keeping a high rendering quality. To perform this, we limit ourselves on the standard OpenGL point primitive. Anisotropic texture filtering can be performed by the screen space EWA surface splatting algorithm that can be implemented efficiently with Vertex and Fragment programs avail-

able on recent graphic hardware (Radeon 9x00 and GeForceFX).

2 Previous Work

The concept of using points as a rendering primitive has been introduced first by Levoy and Whitted [9]. In this pioneering report, they discuss on fundamental issues such as surface reconstruction and visibility. Built on these ideas, many point-sample rendering techniques have been proposed. To perform a classification, many criterions are commonly used, such as screen space versus object space reconstruction, software or hardware, speedup, etc. Since point-based representation is halfway geometrical description and pure image-based representation, there are two ways to apprehend point-based rendering.

For the first techniques presented here, point set is considered as a geometrical surface description. Then, the challenge is to reconstruct a real surface based on this point cloud. To do this, Kalaiah and Varshney [8] capture the local differential geometry at each point sample and use it for resampling and hardware-accelerated rendering with smooth shading effects. Alexa et al.[1] present a technique that upsamples the point set on-the-fly during rendering to achieve the desired screen-space density of points and to avoid holes. These two techniques allow big magnification without loss in shading and outline quality. However, none of these can handle textured models, so their application domains are very limited. Rusinkiewicz and Levoy [14] developed the QSplat system designed to display very large point-samples models resulting from 3D scanner. Stamminger and Drettakis [15] use standard OpenGL point primitives to render point-sampled procedural geometry. They do not consider point samples as a surface description but they assume that the geometry of the object is completely known (meshes, height-fields or procedural objects), which is not always the case. Moreover, texture filtering is not implemented.

In the second class of rendering technique, irregularly spaced point set is reconstructed as a continuous texture. This was initially done by Grossman and Dally [5] who proposed a screen space hole filling with a pull-push algorithm that is prone to blocky artefacts. Pfister et al. [11] built on this work and added hierarchical data structure and texture filtering. Zwicker et al [16] introduced elliptical weighted average surface splatting that allows anisotropic texture filtering. Their software system has been recently improved by Liu Ren et al. [13] with an object space formulation of the EWA splatting and a hardware-accelerated implementation.

Apart from pure point based rendering, many system combine polygon and point primitives to render in real-time complex scenes. This idea has been first investigated in [2, 3] and recently extended by Coconu and Hege [4]. This last technique uses an octree-based spatial representation, containing both triangles and sampled points. The best suited for rendering is chosen dynamically in accordance with screen space projection criteria. Surfels are rendered with fuzzy splats (Gaussian with alpha blending) that perform coarse view independent texture filtering that can't allow big magnification or semi-transparent model. Anyway, with hybrid method we introduce connectivity information that diminishes the advantages of pure point based models. Moreover, as we explain in introduction, simple and valid polygonal representation of arbitrary models is not always available.

3 EWA framework

3.1 Screen Space EWA Surface Splatting

In this section we briefly review the screen space EWA splatting framework. For convenience, we use same notation that in [16] where you can get more details. Interested readers may find the object space derivation in [13].

Let P_k be the set of points that defines a 3D surface. Let us note that these points have no connectivity and can be irregularly spread in space. For each point three coefficients are assigned, w_k^r, w_k^g, w_k^b which represent a chromatic value. For convenience, in further explanation we consider only a scalar component w_k. We begin by defining a continuous texture function f_c on the surface represented by the set of points. To do this, we associate to each point a radially symmetric basis function r_k from which surfel position and orientation can be computed. These basis functions are reconstruction filters defined on locally parameterised domains. Let Q be a point with local coordinates u anywhere on the surface. Then, the continuous

function $f_c(u)$ is defined as the weighted sum :

$$f_c(u) = \sum_{k \in N} w_k r_k (u - u_k) \tag{1}$$

where u_k is the local coordinate of point P_k.

At rendering time, the texture function $f_c(u)$ is warped to screen space using a local affine mapping of the perspective projection at each point. In order to avoid aliasing artefacts, the output function must respect the Nyquist criterion of the screen pixel grid. Then the continuous output signal of the warping of $f_c(u)$ is band-limited by convolving it with a prefilter h, yielding the output function $g_c(x)$ where x are screen space coordinates. This output function can be written as a weighted sum of screen space resampling filters $\rho_k(x)$:

$$g_c(x) = \sum_{k \in N} w_k \rho_k (x) \tag{2}$$

where

$$\rho_k(x) = \left(r'_k \otimes h\right)(x - m_k(u_k)) \tag{3}$$

Here, m_k denotes the local affine approximation of the projective mapping $x = m(u)$ for the point u_k. This approximation is given by the Taylor expansion of m at u_k :

$$m_k(u) = m(u_k) + J_k(u - u_k) \tag{4}$$

where J_k is the Jacobian $J_k = \frac{\partial m}{\partial u}(u_k)$.

Like Heckbert [6], elliptical Gaussians are chosen both for the basis functions and the low-pass filter. Gaussians are closed under affine mappings and convolution. Then, the resampling kernel ρ_k can be expressed as a single elliptical Gaussian which allows fast evaluation at rendering time.

Let $G_V(x)$ be a 2D elliptical Gaussian with variance matrix $V \in R^{2\times 2}$. $G_V(x)$ is defined as :

$$G_V(x) = \frac{1}{2\pi |V|^{\frac{1}{2}}} e^{-\frac{x^T V^{-1} x}{2}} \tag{5}$$

A typical choice for the variance matrix of prefilter h is the identity matrix I. Let V_k^r be the variance matrix of the basis functions r_k. Then the resampling kernel ρ_k can be written as a single Gaussian with a variance matrix that combines the warped basis function and the low-pass filter :

$$\rho_k(x) = \frac{1}{|J_k^{-1}|} G_{J_k V_k^r J_k^T + I}(x - m_k(u_k)) \tag{6}$$

which is called the screen space EWA resampling filter. Finally, substituting this into 2, the continuous output function is the weighted sum :

$$g_c(x) = \sum_{k \in N} w_k \frac{1}{|J_k^{-1}|} G_{J_k V_k^r J_k^T + I}(x - m_k(u_k))$$

3.2 Determining the resampling kernel

To evaluate the expression 6, we should determine the two parameters V_k^r and J_k. V_k^r is only function of the model's sampling and can be computed at the preprocess time. If the maximum distance between the kth surfel and its neighbours is h_k then we take as V_k^r :

$$V_k^r = \begin{pmatrix} h_k^2 & 0 \\ 0 & h_k^2 \end{pmatrix} \tag{7}$$

Of course, this approach supposes a uniform sampling of the model, which is impossible to obtain in practice. To compute the Jacobian J_k we use the technique described in [13] which is easier to implement in Vertex Programs. This lead to the following expression :

$$\begin{array}{c} J_k = \eta \begin{bmatrix} S_x O_z - S_z O_x & T_x O_z - T_z O_x \\ S_y O_z - S_z O_y & T_y O_z - T_z O_y \end{bmatrix} \\ \eta = \frac{v_h}{2tan\left(\frac{fov}{2}\right)} \frac{1}{O_z^2} \end{array} \tag{8}$$

where v_h is the viewport height, fov is the field of view, $O = (O_x, O_y, O_z)$ is the surfel's position in camera space, $S = (S_x, S_y, S_z$ and $T = (T_x, T_y, T_z)$ are the basis vectors defining the local surface parameterisation in camera space.

4 Algorithm overview

The global algorithm of our method is shown figure 1. First, all visible surfels are extracted from the data structure (see section 8) and rendered a first time by the hardware in order to compute the depth buffer (section 5) before the EWA splatting pass (section 6). As explain in section 7, after these two passes, we have to perform normalization on each pixel to force a partition of unity in screen space. As shown on the figure, after the extraction of visible surfels, the rendering is fully performed by the hardware. The dashed-line denotes the possibility to store samples into the memory of GPU via vertex buffer object.

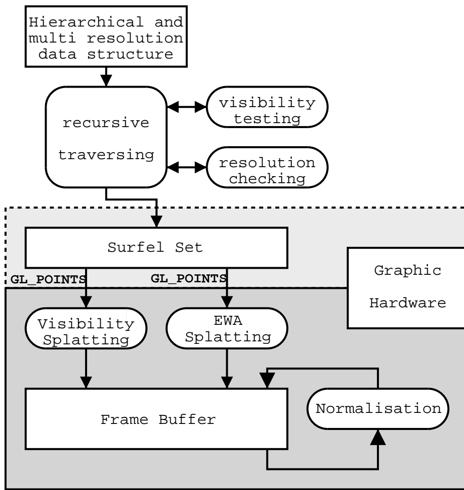


Figure 1: Schematic overview of the algorithm.

5 Visibility Splatting

Visibility splatting algorithm has been known for a long time [12]. Its purpose is to obtain a correct depth buffer of the current view (i.e. without any holes). Usually, this is realized by rendering an opaque quad for each surfel. The resulting depth image is used as a filter to identify visible surfels closest to the viewer : only fragments in the foreground are kept and accumulated during the EWA splatting pass. However, to prevent the discarding of visible splat contribution, the depth image should be translated away from the viewpoint by a small threshold. As proposed in [13], to prevent occlusion artefacts the depth image should be translated along the viewing rays rather than the camera space z-axis.

However, as we have already said, the use of quads for representing surfels uselessly consumes AGP bus bandwidth and vertex computation. Our approach is to use only one vertex by surfel. So, we can evaluate for each surfel, its projection size in screen space and use this value as the point size of the GL_POINTS primitive. However, the result of the projection of a vertex with standard OpenGL point primitive and a point size of n is, in viewport space, a square centred to the projection of the surfel with a size of n pixels. So, whatever the orientation is, the result will be the same. Moreover, the depth of each resulting fragment is the same.

We will consider the projection of a standard OpenGL point as a screen-space bounding-square of the real splat shape centred to the surfel projection. During rasterisation, each generated fragment is incorrect in two ways : it could not handle the real projected surfel's shape, and else its depth is incorrect and must be recomputed. These corrections can be easily implemented with ray-casting. However, they can be performed more efficiently by clipping the surfel's tangent plane (section 5.1) and recomputing the correct depth as explained in section 5.2

5.1 Surfel's plane clipping

Ideally, a surfel is represented in object space by a tangent disk (with a radius of h) or a tangent quad. For better efficiency, we chose to approximate these classical representations by a tangent plane bounded by a frustum of a pyramid as shown in figure 2.

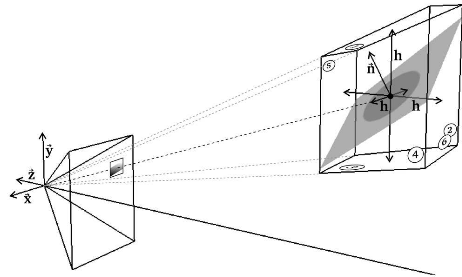


Figure 2: The frustum of pyramid which bounded the surfel's tangent plane.

The standard OpenGL point primitive easily does the clipping of the surfel's tangent plane by the first four frustum's planes. We just compute the projected size in viewport space :

$$OpenGLpointsize = \frac{2h}{z} \frac{height}{2tan\left(\frac{fov}{2}\right)} \tag{9}$$

To perform the clipping by the last two frustum's plane, we just compute the minimum and maximum depth value and kill all fragments that are not in this range. To do this, we need to compute the real depth of each fragment as explained in the next section.

5.2 Per Fragment Depth Correction

Let the current surfel with position P^c and normal N^c. The superscript c denotes these vectors are expressed in camera space. Given a point Q^c onto the surfel's tangent plane, let Q^p be its projection onto the near plane and Q^v its coordinates in the viewport (figure 3). Then, our aim is to compute as fast as possible Q^c_z from Q^v. As notation, we use capital letter for vector and small letter for scalar quantities. Subscript denotes components of a vector.

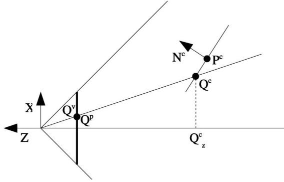


Figure 3: Shown in two dimension, one surfel (with position P^c and normal N^c) and one point Q^c on the tangent plane of the surfel (Q^p is its projection on the virtual screen and Q^v its coordinate in the viewport space)

Since Q^c is onto the tangent plane and the viewing ray, we have :

$$Q^c_z = \frac{P^c \cdot N^c}{Q^p \cdot N^c} Q^p_z \tag{10}$$

Our view frustum is defined by the four values r, t, n, f as shown in figure 4.

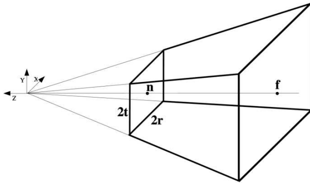


Figure 4: The view frustum defined by four parameters : r, t, n and f (resp. right, top, near and far).

We can write Q^p from Q^v as :

$$Q^p = \begin{bmatrix} Q^v_x \frac{2r}{v_w} - r \\ Q^v_y \frac{2t}{v_h} - t \\ -n \end{bmatrix} \tag{11}$$

where v_w (resp. v_h) is the viewport width (resp. viewport height). Then :

$$Q^p \cdot N^c = Q^v \cdot \begin{bmatrix} \frac{2r}{v_w} N^c_x \\ \frac{2t}{v_h} N^c_y \end{bmatrix} - \begin{bmatrix} r \\ t \\ n \end{bmatrix} \cdot N^c \tag{12}$$

and :

$$\frac{1}{Q^c_z} = \frac{N^c \cdot \begin{bmatrix} \frac{r}{n} \\ \frac{t}{n} \\ 1 \end{bmatrix} - Q^v \cdot \begin{bmatrix} N^c_x \frac{2r}{nv_w} \\ N^c_y \frac{2t}{nv_h} \end{bmatrix}}{P^c \cdot N^c} \tag{13}$$

With standard OpenGL frustum, the depth value is computed as follow :

$$\begin{aligned} depth &= \tfrac{f+n}{f-n} + \tfrac{2fn}{f-n}\tfrac{1}{z} \\ &= g_1 + \tfrac{g_2}{z} \end{aligned} \tag{14}$$

Then, using equations 13 and 14 and rewriting we can express the depth :

$$\begin{aligned} depth &= g_1 + \tfrac{g_2}{Q^c_z} \\ &= a - Q^v \cdot B \end{aligned} \tag{15}$$

with :

$$\begin{aligned} a = &\quad g1 + \tfrac{g_2}{P^c \cdot N^c} N^c \cdot \begin{bmatrix} r/n \\ t/n \\ 1 \end{bmatrix} \\ B = &\quad \tfrac{g_2}{P^c \cdot N^c} \begin{bmatrix} N^c_x \frac{2r}{v_w} \\ N^c_y \frac{2t}{v_h} \end{bmatrix} \end{aligned} \tag{16}$$

The resulting depth buffers of standard OpenGL point with and without our correction are compared on figure 5. As shown, standard OpenGL point primitive increases the size of the sphere with aliased edge. This is corrected by the frustum clipping, and the depth correction provides a more smoothed depth buffer.

5.3 Implementation details

In our implementation, a and B are computed in a vertex program and sent to the fragment program in a four components vector as : $[B_x, B_y, 0, a]$. Then the correct depth value of a fragment can be computed with only two instructions (1 DP3 and 1 ADD). Hence, we verify the membership of this value in the correct range (1 MAD) and if necessary, the fragment is killed (instruction KIL).

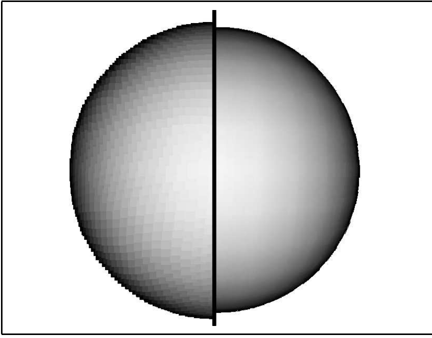


Figure 5: Left, the depth buffer of a sphere with standard OpenGL point primitive. Right, the depth buffer of the same sampled sphere with frustum clipping and depth recomputation. The infinity depth is intentionally set at white to show edge.

6 EWA Splatting

This section corresponds to the second pass of our algorithm. Each surfel of position P_k is rendered by the graphic pipeline which computes its projected position in viewport space P_k^v, centers the resampling kernel at P_k^v and evaluates it for each pixel. However, the Gaussian resampling kernel is computed only for a limited range of the exponent $\beta(x) = \frac{1}{2}x^T x$. Hence, we choose a cutoff radius c, such $\beta(x) < c$ (typically $c = 1$). Once again we consider the result of a standard OpenGL point as a viewport space bounding square. Then, we take as the OpenGL point size :

$$OpenGLpointsize = \frac{2\sqrt{2c}h_k}{z} \frac{v_h}{2tan\left(\frac{fov}{2}\right)} \quad (17)$$

To efficiently compute the equation 6 for each generated fragment, the variance matrix (cf. equation 8) and the center of the kernel P_k^v are computed for each surfel in the vertex program.

Let us note that is useless to compute the real depth of each fragment for this pass if we choose reasonable value for the depth epsilon of the visibility pass and the cutoff radius. Else, with large cutoff radius, some visible fragments may be discarded. A bad solution will be to increase the depth epsilon since large value for the depth epsilon may lead to the blending of several surfaces. A better solution will be to test only the depth of the surfel's projection center rather than all fragments, but it is not currently possible.

Implementation details

For each sample, the vertex program performs following operations :

- warps position and normal to camera space
- computes the resampling kernel (section 3.2)
  - computes the base of the local parameterisation (s, t)
  - computes the Jacobian J
  - computes the inverse of variance matrix
- warps position to viewport space
- evaluates the OpenGL point size
- interpolates between mip-map levels
- performs lighting and multiplies the result by $\frac{outputScaleFactor}{2\pi|J^{-1}||Var|^{\frac{1}{2}}}$.

Since each component in the frame buffer is clamped to one, we use a global $outputScaleFactor$ constant to make sure that the sum of each contribution is less than 1. A typical choice for $outputScaleFactor$ is 0.7. Let $X \in R^2$ be the position of the current fragment in viewport space. Then, the fragment program performs the following operations :

- computes the exponent : $\beta(X - P_k^v)$
- kills the fragment if $\beta(X - P_k^v) > c$
- multiplies the fragment color by $e^{-\beta(X-p_k^v)}$

7 Normalization

As done by Zwicker et al.[16], after all surfels were splatted the result must be normalized. Reasons are the irregular sampling of point models and the truncation of the Gaussian kernel. Each pixel is normalised by the sum of the accumulated contributions :

$$g(x) = \sum_{k \in N} w_k \frac{\rho_k(x)}{\sum_{j \in N} \rho_j(x)} \quad (18)$$

This is easily done with Fragment Program as a third pass. The resulting frame buffer is copied directly into a texture of the GPU and rendered as a simple quad. During the previous pass the alpha component is used to store and compute the sum of the accumulated contribution.

8 Hierarchical rendering

To improve performance, it is useful to associate our rendering technique with a hierarchical data

structure that allows hierarchical culling and progressive rendering. We chose a simple octree traversing from the lowest to highest resolution.

To test the visibility of a block we have implemented view-frustum culling and back-face culling via visibility-cones [5]. We could also use other hierarchical data structures such as a bounding sphere hierarchy [14] or a LDC tree [11]. In fact, all surfels are stored into multiple large buffer (typically one by resolution level) and each node stores only the start and the end position in the correct buffer similar to [13]. This minimizes the switching of vertex buffers during rendering and enhances performance. Then, when adding a node into the list of visible block we reconstruct a large buffer simply by comparing block's buffer and index.

9 Results

We implemented our algorithm with standard OpenGL ARB_Vertex_Program and ARB_Fragment_Program extensions [10] supported by Radeon 9x00 from ATI and GeForceFX family from NVidia. However, we have only tested our implementation on a GeForceFX 5800 with an AMD Athlon 2800+. In comparison with object space approach, we considerably have decreased the vertex computation cost (only one vertex by sample against four) and globally vertex programs have fewer instructions (table 1). Although our fragment programs are very simple (less than five instructions), the cost of rasterisation and per fragment computation is more expensive, especially for viewing direction tangential surfels where many fragments are rasterised needlessly while OpenGL point primitive handles an axis aligned square.

	Visibility Splatting	EWA Splatting	Normalization
Our approach	29/5	45/5	- / 3
Object Space approach[13]	13/-	74/-	-

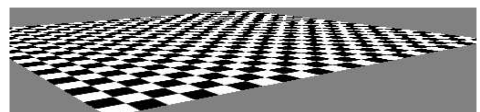
Table 1: Comparison of the number of instructions needed for each pass. The first number corresponds to vertex programs and the second to fragment programs

The table 2 shows rendering performance of our algorithm on two models (Figure 7) for two frame buffer resolution and when object level culling is disabled. Although our fragment programs are very simple, we observe a slowdown of 2.5 in comparison with the case where fragment programs are disabled. However, we can expect better result with nvidia's driver revision since triangle primitive is not so much penalized by simple fragment programs. To evaluate the cost of vertex programs, we have measured that the GeForceFX 5800 is able to render 60 millions of small GL_POINTS primitives per second. To achieve this, we use ARB_vertex_buffer_object and point size lesser than five. The second row of the table 2 shows that 30 millions of primitives are sent per second when only vertex programs are enabled.

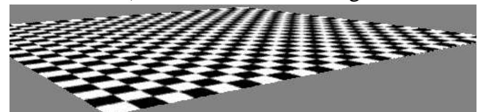
# surfels	head 205865	turtle 418287
Our approach	32.5/24.9 fps	17.1/12.9 fps
Our approach with fragment programs disabled	69.7/67.8 fps	36.8/36.2 fps
Our object space EWA surface splatting implementation	16.7/15.6 fps	8.9/7.9 fps

Table 2: Rendering performance of our system on a NVidia GeForceFX 5800. The first (resp. second) number corresponds to a frame buffer resolution of 512x512 (resp. 1024x1024).

In order to test anti-aliasing, we render a simple plane consisting of 64k surfels with a checkerboard texture(6).



a) Without EWA filtering.



b) With EWA filtering.

Figure 6: Checkerboard rendering using two different screen space surface splatting algorithms.

Figure 7 shows a head and a turtle rendered with our system. Figure 8 shows a detail of our head model with different parts of our algorithm disabled. On figure 8a all fragment programs are dis-

abled. On figure 8b, only depth recomputation and surfel's plane clipping are disabled. Finally, the figure 8c shows the same model with our complete multipass algorithm.

10 Conclusion and Future Work

We have described an efficient rendering method based on the EWA surface splatting algorithm. We have shown how to handle oriented plane and elliptical Gaussian with a simple OpenGL point primitive. Besides increased performances, our approach provides more flexibility. We will further extend our method to handle more complex surfel's representation than tangent plane. For example, we plan to add curvature information to sample (as done by Kalaiah and Varshney [8]) that allows simplification scheme for low textured model and high magnification with nice shading effect. We also intent to optimise our implementation and extend it to support deformable objects. Here, the main challenge is to develop a data structure that allows efficient visibility culling and progressive rendering on dynamic point clouds.

References

[1] M. Alexa, L. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. Silva. Point Set Surfaces. In *Proceedings of IEEE Visualisation,* pages 21-28. San Diego, CA, October 2001.

[2] B. Chen and M. X. Nguyen. POP : A Hybrid Point and Polygon Rendering System for Large Data. In *Proceedings of IEEE Visualisation,* pages 45-52. San Diego, CA, October 2001.

[3] J. Cohen, D. Aliaga, and W. Zhang. Hybrid Simplication : Combining Multi-Resolution Polygon and Point Rendering. In *Proceedings of IEEE Visualisation,* pages 37-44. San Diego, CA, October 2001.

[4] L. Coconu and Hans-Christian Hege. Hardware-Accelerated Point-Based Rendering of Complex Scenes. In *Proceedings of 13th Eurographics Workshop on Rendering,* pages 43-52. Pisa, IT, June 2002.

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

[6] P. Heckbert. *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California at Berkeley, Department Of Electrical Enguneering and Computer Science, June 1987.

[7] H. Hoppe, T. DeRose, T. Duchampt, J. McDonald, and W. Stuetzle. Surface Reconstruction from Unorganised Points. In *Computer Graphics,* SIGGRAPH 92 Proceedings, pages 71-78. Chicago, IL, July 1992.

[8] A. Kalaiah and A. Varshney. Differential Point Rendering. In *Proceedings of the 12th Eurographics Workshop on Rendering,* pages 138-150. London, UK, June 2001.

[9] 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.

[10] SGI OpenGL Extension Registry *http://oss.sgi.com/projects/ogl-sample/registry/*

[11] H. Pfister, M. Zwicker, J. Van Baar and M. Gross. Surfels : Surface Elements as Rendering Primitives.. In *Computer Graphics,* SIGGRAPH 2000 Proceedings, pages 335-342. Los Angeles, CA, July 2000.

[12] 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.

[13] L. Ren, H. Pfister and M. Zwicker. Object Space EWA Surface Splatting In *Proceedings of Eurographics 2002* Sept 2002.

[14] S. Rusinkiewicz and M. Levoy. QSplat : A Multiresolution Point Rendering for Complex and Procedural Geometry. In *Proceedings of the 12th Eurographics Workshop on Rendering,* pages 151-162. London, UK, June 2001.

[15] M. Stamminger and G. Drettakis. Interactive Sampling and Rendering for Complex and Procedural Geometry. In *Proceedings of the 12th Eurographics Workshop on Rendering,* pages 151-162. London, UK, June 2001.

[16] M. Zwicker, H. Pfister, J. Van Baar and M. Gross. Surface Splatting. In *Computer Graphics,* SIGGRAPH 2001 Proceedings, pages 371-378. Los Angeles, CA, July 2001.

Figure 7: A head and a turtle rendered with our system.

a) All fragment programs are disabled.

b) Only depth recomputation and surfel's plane clipping are disabled.

c) Rendered with our complete system.

Figure 8: Details of our head model rendered with different part of our system disabled.