Direct Point Rendering on GPU

Hiroaki Kawata[1] and Takashi Kanai[2]

[1] Keio University, Faculty of Environmental Information,
5322 Endo, Fujisawa, Kanagwa 252-8520, Japan
[2] RIKEN, Integrated Volume-CAD System Research Program,
2-1 Hirosawa, Wako-shi, Saitama 351-0198, Japan

Abstract. In this paper, we propose a method for directly rendering point sets which only have positional information by using recent graphics processors (GPUs). Almost all the algorithms in our method are processed on GPU. Our point-based rendering algorithms apply an image buffer which has lower-resolution image than a frame buffer. Normal vectors are computed and various types of noises are reduced on such an image buffer. Our approach then produces high-quality images even for noisy point clouds especially acquired by 3D scanning devices. Our approach also uses splats in the actual rendering process. However, the number of points to be rendered in our method is in general less than the number of input points due to the use of selected points on an image buffer, which allows our approach to be processed faster than the previous approaches of GPU-based point rendering.

1 Introduction

In recent years, point-based surface representation is becoming more and more important and drawing increasing attention thanks to recent advances of 3D scanning technology. Though a large number of points can be acquired by using 3D scanning devices, it is difficult to handle large meshes constructed from these point primitives. Therefore, approaches to point-based modeling or rendering, which directly handle point primitives instead of constructing meshes or high-order surfaces, have been a focus of constant attention.

Among these point-based approaches, point-based rendering is suitable for visualizing a large number of point primitives. In the process of rendering a mesh with millions of triangles, an overhead to rasterizing a triangle is too high because the area of a triangle projected to a screen buffer is often smaller than that of a pixel. This fact marks a watershed to alter a point-based rendering instead of a surface rendering based on triangles. Moreover, the process of point-based rendering can be accelerated using recent graphics processors (GPUs).

In this paper, we propose a direct rendering approach of point primitives using GPU. "*Direct*" in our case means that points are rendered using *only* their position information. Our approach is effective especially for rendering a large number of points acquired from 3D scanning devices. This is because that those acquired points mainly consist of only 3D positions and colors. On the other hand, a normal vector is also needed to compute shading effects when rendering. These normal vectors are usually calculated from 3D positions of points (e.g. [1, 2]) as a pre-process. If we define a normal vector for each point, large space equivalent to the same size as 3D positions will be required to store such normal vectors. This would cause a large bottleneck when rendering points on PCs with a little DRAM memory.

The idea of our approach is based on the method proposed by Kawata and Kanai [3]. However, our algorithm is mostly executed on GPU, and is essentially different from the algorithm in [3] when implemented on GPU. The main features of our approach are described as follows:

Ad-hoc normal vector calculation. Normal vectors needed for shading effects are calculated in the rendering process on GPU. This saves the calculation time of such vectors in the pre-processing stage.

Noise reduction. Noise reduction of points is applied on GPU. In general, points acquired from 3D scanning devices involve several types of noises. Our approach is effective for rendering such noisy points.

Fast rendering algorithm. Splats are used in the final rendering stage. Texture filtering needed for this stage is applied for only selected points on an image buffer. The number of such selected points are in general less than input points. Our algorithm then renders points quickly compared to the previous approaches of GPU-based point rendering.

2 Related Work

Details of the approaches of point-based computer graphics are described in [4, 5]. Here we mainly describe related researches of point-based rendering. Point based rendering was first introduced by Levoy and Whitted [6]. One important process is to fill holes between neighboring points. There are two types of approaches for such hole-filling; screen-based approach and object-based approach. Our approach is basically of the former approach.

On the other hand, most of GPU-based point-rendering approaches focus on the implementation of the filtering phase when overlapping splats. Ren et al. [7] proposed an object-space anisotropic EWA filtering on GPU. In [7], a rectangle is used as a rendering primitive which has the problem of rendering speed, because four vertices in each splat have to be transferred to GPU. Guennebaud et al. [8] and Botsch et al. [9] independently addressed this issue by using point sprites. Zwicker et al. [10] proposed a high-quality object-space EWA texture filtering on GPU. Botsch et al. [11] realized a per-pixel lighting (Phong shading) by using a normal vector field for each splat. Other approaches include a method to transfer points effectively to GPU by using linear point lists [12], and an implementation of point-selection algorithm [13].

In our approach, splats are used to render points as done by most of the approaches described above. In addition, both the normal vector computation and noise reduction processes are also implemented on GPU.

3 Rendering Algorithm Using GPU

Figure 1 illustrates an overview of our direct point rendering algorithm on GPU. The input for our algorithm is a set of points $\mathcal{P}$: $p_i \in \mathcal{P}$ ($i = 1 \ldots n$, n is the number of points). Although only a 3D position $\mathbf{p}_i$ is required for each point, in some cases it is also possible to attach color information $\mathbf{c}_i$.

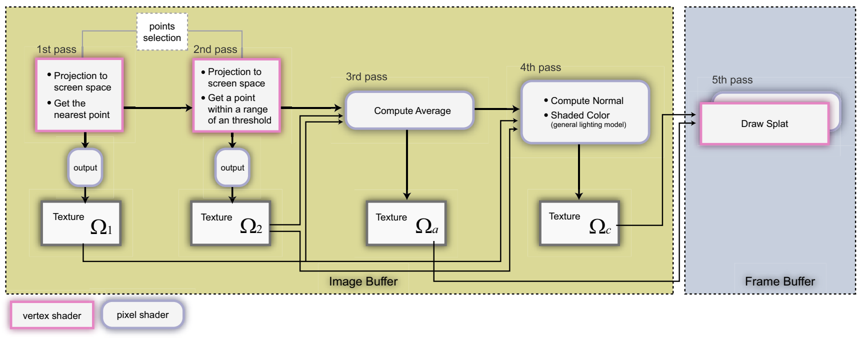

Fig. 1. Overview of our direct point rendering algorithm on GPU

Our approach adopts a totally five-pass algorithm. In the process of filling holes, we use an *image buffer* as proposed in [3] which has a lower resolution than a frame buffer. In our approach, an image buffer is used from the first to the fourth passes. In the fifth pass, we magnify a buffer to the resolution of an actual frame buffer and apply splat rendering in this pass. We allocate several floating-point textures Ω in VRAM memory on GPU. They have the same resolution as an image buffer and are used as inputs/outputs for each pass of our whole algorithm.

3.1 Setting the Resolution of an Image Buffer

In the first and second passes, we project each point to an image buffer[1](details are described in Section 3.2).

In these processes, an appropriate resolution of an image buffer has to be determined for filling holes. This resolution is dynamically determined by the resolution of a point set and by camera parameters (a view position, field of view, the size of view frustum, etc.).

Let the width and height of a frame buffer and an image buffer be w_s, h_s, w_i, h_i, respectively. w_i, h_i are then calculated by the following equations derived from the perspective projection:

$$w_i = w_s/\rho, \quad h_i = h_s/\rho, \tag{1}$$

[1] For the image buffer, we use a floating-point buffer on GPU. In case of DirectX, IEEE-format `D3DFMT_A32B32G32R32F` is available. In case of OpenGL, pbuffer can be used.

$$\rho = \frac{\sigma}{\tan(f/2)} \cdot \frac{1}{\lambda} \cdot w_s, \tag{2}$$

$$\lambda = \frac{z_f \cdot z_n}{z_f - z_n} \cdot |\mathbf{v} - \tilde{\mathbf{p}}| + 1. \tag{3}$$

where f denotes a view angle (fov), $\mathbf{v}$ denotes a view position, z_f, z_n denote the distance from a view position to the far plane and the near plane respectively, and $\tilde{\mathbf{p}}$ denotes a barycentric point of a point set.

σ in Equation (2) is *the resolution of a point set*, that is, a parameter which estimates an interval between neighbor points. σ can be calculated as follows: First, a $k-$neighbor graph of a point set is defined. For each point, the distance to the closest point is calculated. σ is set as the minimum of these distances. Since the definition of a $k-$neighbor graph and the calculation of the closest point is view-independent, we can calculate σ as a pre-process when we input a point set. Using the above equations, the value of ρ is usually greater or equal to 1, and the value of w_i, h_i is then smaller or equal to w_s, h_s.

3.2 Points Selection for Noise Reduction and Averaging

In the first and second passes described in Figure 1, the selection of a point set to an image buffer is processed by using vertex shader. These processes are done to reduce noises of a point set. Note that only a point can be stored for each pixel on current GPUs. We then add rendering passes to store multiple points. In our case, a two-pass rendering algorithm is applied.

For each pass, we apply perspective projection for each point and store a projected point to an image buffer. This process is done by both vertex shader and pixel shader on GPU. In the vertex shader, the perspective projection to a camera coordinate for each point is applied. In the pixel shader, a projected point is simply written to an image buffer. After the process of the pixel shader, the contents of an image buffer are copied to a floating-point texture Ω. In this case, a projected point is stored in a corresponding pixel of a texture as a floating point value.

Figure 2 shows the procedure for selective storage of points to an image buffer. When multiple points are projected to the same pixel, only the closest

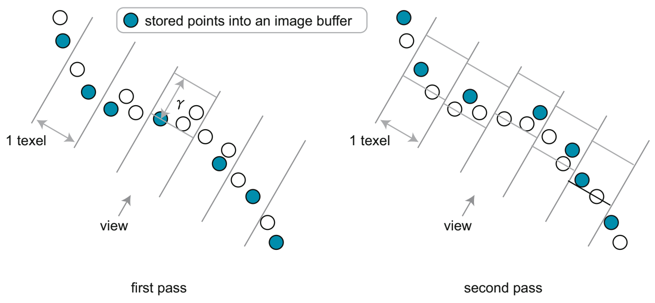

Fig. 2. Overview of selective storage process

point to a view position is selected and stored to a pixel of a texture Ω_1 in the first pass (Figure 2 left).

In the second pass, we also apply perspective projection for each point and store the projected one to an image buffer. In this case, we select a point whose depth from a point selected in the first pass is in the range of γ, and which is the farthest point within this range (Figure 2 right). There are two purposes for this selective storage: One is to omit isolated points (called *outliners*), and the other is to reduce bumpy noises by averaging.

To implement this selective storage on GPU, we first look up a point on Ω_1 and investigate whether the depth of a projected point is within the range of γ or not in the vertex shader of the second pass. If a point is out of range, we omit this point. By setting γ, we can omit later processes for outliners or for points of back faces. γ is a view-independent parameter, then a constant value can be set in the pre-processing stage. According to our experiments, we found that 0.5-1.0% over a diagonal length of a bounding box surrounding at an object is suitable for our results.

Next, the farthest point within the range of a threshold γ is selected by setting "write if the depth of a pixel is larger than that which has already been written"[2]in the depth test on GPU. A selected point by this test is stored in a pixel of a texture Ω_2.

Even if a point has both its position and a color, both results can be written to separate buffers in each pass by using MRT (*Multiple Renger Target*). Moreover, we also use *Vertex Texturing* to look up a point stored in a texture Ω_1 in the vertex shader of the second pass. These two functionalities are supported from Shader Model 3.0, and can be used by only *n*VIDIA GeForce 6 series GPU.

In the third pass, we look up two corresponding points on floating-point textures Ω_1 and Ω_2 in the pixel shader to compute their average value. An average value is then stored to a texture Ω_a. This value is used only as a position for drawing splats in the final pass described in Section 3.4.

3.3 Computation of Normal Vectors

In this sub-section, we describe our novel approach to compute normal vectors on GPU. The computation of normal vectors is processed in the pixel shader of the fourth pass in Figure 1.

In the process of computing normal vectors, information on neighboring points for each point is needed. To rapidly acquire neighboring points, we utilize textures Ω_1 and Ω_2 created in the selective storage process described before. Figure 3 shows the principle of computing normal vectors. In the left figure of Figure 3, a normal vector at a point p is computed by using its neighboring point p_i (18 points at a maximum).

We approximate here the proximity of a point p defined on the camera coordinate system to a plane,

$$f(x,y) = z = Ax + By + C, \tag{4}$$

[2] In case of DirectX, we can use `D3DCMP_GREATER` flag for this process.

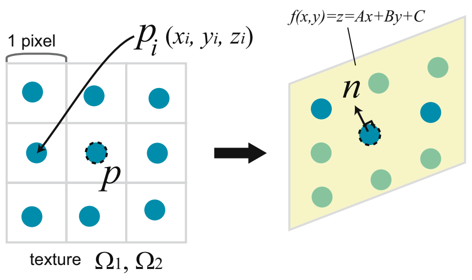

Fig. 3. Plane-fitting process and computation of normal vector

by using least-square fitting (Figure 3 right). We first look up neighboring points $p_i = \{x_i, y_i, z_i\}$ $(i = 1 \ldots N \leq 18)$ from Ω_1, Ω_2. For point p, we compute coefficients A, B, C by solving the following equation:

$$\sum_{i=1}^{N} \{z_i - (Ax_i + By_i + C)\}^2 \rightarrow min. \tag{5}$$

A, B, C are the solution of a 3×3 linear equation. Here we try to solve this equation by using both Cramer's formula and Gauss's elimination method. Although the latter is the numerical solution, we can expand the code and write it to the pixel shader directly. Compared to these two solutions, we found that the computation time and thus the number of instruction sets in the pixel shader is almost equal (Gauss's elimination: 336, Cramer's formula 340). By using such computed coefficients, a normal vector can be defined as $(A, B, -1)$. We then apply shading to compute a color by using a normal vector and a point from Ω_a and store a color to a texture Ω_c.

We approximate a plane in Equation (4) to the proximity of a point p. However, in this equation, we cannot define a plane parallel to a viewing direction. In our case, situations which a normal vector is completely perpendicular to the viewing direction rarely occur. We then think that it is a trivial problem for practical use. We also think that more general equations such as $Ax+By+Cz+D = 0$ can be used to fit to a plane. In this case, complicated numerical approaches such as Newton's method are required to compute coefficients, which tend to increase the number of instruction sets and thus the computation time.

3.4 Rendering with Splats

We now draw splats in the fifth pass with two floating-point textures Ω_a and Ω_c where positions and colors are contained respectively in Figure 1. Along with Botsch et al.'s approach [9], each splat is rendered with alpha-blending by using a *Point Sprite* with an attached *Gaussian kernel texture*. A Gaussian kernel texture is a 2D texture sample (point sprite) in which a Gaussian function is embedded. Each vertex of this texture has its own 2D texture coordinate (u_i, v_i) whose origin is

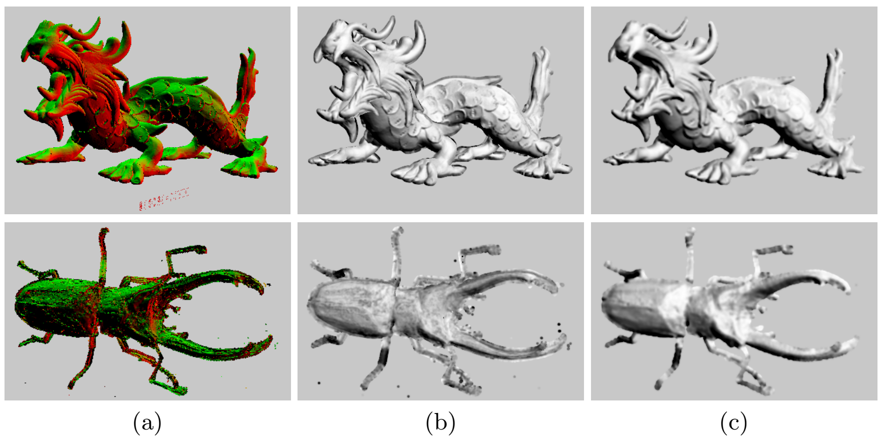

Fig. 4. Rendering results for actual range images. Top: Stanford Asian Dragon. Bottom: Beetle. (a) Range image. (b) Rendering results by approach in [9]. (c) Rendering results by our approach.

the center point. The rendering process is done in both vertex shader and pixel shader. The final color in a frame buffer is a weighted sum of splat colors. The resolution of textures Ω_a, Ω_c is smaller than that of a frame buffer. We therefore need to magnify them to the same size as a frame buffer in this pass.

4 Results

Figure 4 shows visual comparisons between our approach and an approach in [9] for actual range images. The upper part of Figure 4 denotes the results for a point set of Stanford Asian Dragon (3,609,600 points). This point set is acquired from a 3D scanning device, however it looks "good", namely, it has less bumpy noises. For such point sets, we found that the approach in [9] produces higher-quality images than our approach, because normal vectors can be computed correctly. In contrast, some blurring effects appear as a result of our approach. This is thought to be a certain loss of points in the selective storage process.

The bottom of Figure 4 shows results for a point set of Beetle (559,327 points). This point set has high bumpy noises and considerable outliners, and it is difficult to compute normal vectors correctly in this case. In the result of [9], the blob patterns appear, while our approach can produce better quality image even for such point sets, proving that our approach fully demonstrates it's ability for noisy point sets.

5 Conclusion and Future Work

In this paper, we have proposed a direct rendering approach for GPU to points which only have positional information. We have demonstrated three advantages of our approach by experiments: First, we keep the rendering quality as the previous GPU-based point rendering approaches, while involving normal vector computation for each frame. Secondly, our approach fully demonstrates it's ability for noisy point sets by the noise reduction process. Finally, the computation time is twice faster than previous approaches by using the selective storage process.

In future work, we wish to decrease flickering effects in interactive rendering, and to apply our approach for irregular point sets.

References

1. Hoppe, H., DeRose, T., Duchampy, T., McDonald, J., Stuetzle, W.: Surface reconstruction from unorganized points. Computer Graphics (Proc. SIGGRAPH 92) **26** (1992) 71–78
2. Jones, T., Durand, F., Zwicker, M.: Normal improvement for point rendering. IEEE Computer Graphics & Applications **24** (2004) 53–56
3. Kawata, H., Kanai, T.: Image-based point rendering for multiple-range images. In: Proc. 2nd International Conference on Information Technology & Applications (ICITA 2004), Macquarie Scientific Publishing, Sydney (2004) 478–483
4. Krivanek, J.: Representing and rendering surfaces with points. Technical Report DC-PSR-2003-03, Department of Computer Science and Engineering, Czech Technical University in Prague (2003)
5. Kobbelt, L.P., Botsch, M.: A survey of point-based techniques in computer graphics. J. Computers & Graphics **28** (2004) 801–814
6. Levoy, M., Whitted, T.: The use of points as a display primitive. Technical Report 85-022, Computer Science Department, University of North Carolina at Chapel Hill (1985)
7. Ren, L., Pfister, H., Zwicker, M.: Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. Computer Graphics Forum (Proc. Eurographics 2002) **21** (2002) 461–470
8. Guennebaud, G., Paulin, M.: Efficient screen space approach for hardware accelerated surfel rendering. In: Proc. 8th International Fall Workshop on Vision, Modeling and Visualization (VMV 2003), IOS Press, Amsterdam (2003) 1–10
9. Botsch, M., Kobbelt, L.P.: High-quality point-based rendering on modern GPUs. In: Proc. Pacific Graphics 2003, IEEE CS Press, Los Alamitos CA (2003) 335–343
10. Zwicker, M., Rasanen, J., Botsch, M., Dachsbacher, C., Pauly, M.: Perspective accurate splatting. In: Proc. Graphics Interface 2004, Morgan Kaufmann Publishers, San Francisco, CA (2004) 247–254
11. Botsch, M., Spernat, M., Kobbelt, L.P.: Phong splatting. In: Proc. Symposium on Point-Based Graphics 2004, Eurographics Association (2004) 25–32
12. Dachsbacher, C., Vogelgsang, C., Stamminger, M.: Sequential point trees. ACM Transactions on Graphics (Proc. SIGGRAPH 2003) **22** (2003) 657–662
13. Guennebaud, G., Barthe, L., Paulin, M.: Deferred splatting. Computer Graphics Forum (Proc. Eurographics 2004) **23** (2004) 653–660