Vol. 8, No. 3: 1–21

Efficient Splatting Using Modern Graphics Hardware

Daqing Xue and Roger Crawfis
The Ohio State University

Abstract. Interactive volume rendering for data set sizes larger than one million samples requires either dedicated hardware, such as three-dimensional texture mapping, or a sparse representation and rendering algorithm. Consumer graphics cards have seen a rapid explosion of performance and capabilities over the past few years. This paper presents a splatting algorithm for direct volume rendering that utilizes the new capabilities of vertex programs and the OpenGL imaging extensions. This paper presents three techniques: immediate mode rendering, vertex shader rendering, and point convolution rendering, to implement splatting on a PC equipped with an NVIDIA GeForce4 display card. Per-splat and per-voxel render time analysis is conducted for these techniques. The results show that vertex-shader rendering offers an order of magnitude speed-up over immediate mode rendering and that interactive volume rendering is becoming feasible on these consumer-level graphics cards for complex volumes with millions of voxels.

1. Introduction

Over the years, many computer graphics researchers have developed accelerated splatting algorithms [Laur and Hanrahan 91], [Crawfis and Max 93] using translucent polygons, two-dimensional texture mapping hardware, and optimized software rasterization [Huang et al. 00]. Interactive rates are only achievable with rather small data sets. In recent years, with the development of advanced graphics cards, it is worthwhile to develop a more efficient splat-

1086-7651/03 $0.50 per page

ting algorithm for interactive volume rendering. These cards offer greater flexibilities and are able to run many tasks on the GPU instead of the CPU. An alternative technique is to render the volume using three-dimensional texture mapping hardware [Cullip and Neumann 93]. The entire volume is loaded into the display card's memory and viewed as a solid texture. However, the volume must typically be smaller than 256^3, due to the GPU memory limitations. In OpenGL, hardware splatting with two-diimensional texture mapping hardware is usually achieved in the following way:

1. create a texture image from the reconstruction kernel function as the splat footprint;
2. generate a quadrilateral that is centered about the voxel location for each voxel in the volume;
3. sort all voxels along the view direction;
4. reorient each voxel quadrilateral to be perpendicular to the viewing ray;
5. render the quadrilaterals with texture mapping in a back-to-front order.

Difficulties in obtaining a back-to-front sort are examined and solved by using redundant presorted lists [Max 93]. However, due to the very high rasterization performance of modern display cards, such as the GeForce4 by NVIDIA, the GPU is idle most of the time in an immediate mode rendering pipeline. Recently, in order to reduce such GPU stalls, Lindholm et al. [Lindholm et al. 01] used the vertex programs provided by a GeForce3 display card to perform transformation and lighting on the GPU to implement temporal-varying bump mapping in a very efficient way. In this paper, we present a vertex shader algorithm for splatting to reduce GPU stalls on a GeForce4 card. In addition, we develop a point convolution rendering algorithm, rather than texture mapping, to perform splatting using the OpenGL imaging extension. Both generate correct images for arbitrary viewing directions. We have implemented three rendering methods with hardware acceleration: immediate mode rendering, vertex shader rendering, and point convolution rendering. The immediate mode rendering is an implementation of [Crawfis and Max 93] and is used as a comparison for our two new algorithms. A detailed timing analysis is provided for each of these techniques. Texture splats were introduced in 1993 [Crawfis and Max 93] and have been well studied and implemented. However, there is still no clear description of the texture splat implementation. This paper investigates the texture splatting implementation using a vertex shader. The remainder of this paper is organized as follows. Section 2 examines the optical models used in our study. Section 3 describes the ideas and the implementation details of the three rendering techniques. Result images and a detailed timing analysis are presented in Section 4. In Section 5, we offer conclusions from our study.

2. Optical Models

We use two optical models in our study, a low-albedo optical model and an X-Ray absorption model. The first model takes into account the light extinction property of the material, in which front objects attenuate the back-light passing through them, as well as contribute additional energy scattered toward the viewer. The other model assumes that the material only absorbs the light energy.

2.1. Low-Albedo Optical Model

The low-albedo model is the approximation of the volume rendering integral (VRI) [Blinn 82], [Max 95]. The VRI analytically computes, the amount of light received at location x at the image plane:

$$I(x) = \int_0^L C(s)\mu(s)e^{-\int_0^s \mu(t)dt}ds. \tag{1}$$

Here, L is the length of the light ray, C is the reflected or emitted energy, and μ is the extinction coefficient of the modeled material [Max 95]. In most cases, this integral cannot be computed analytically or efficiently. Practical volume rendering algorithms discretize the VRI into a series of sequential intervals, i, of width δ, and use a Taylor series approximation of the exponential term. Equation (1) is then reduced to the following composition equation in [Meißner et al. 00]:

$$I(x) = \sum_{i=0}^{L/\Delta x} C(s_i)\alpha(s_i)\prod_{j=0}^{i-1}(1-\alpha(s_j)). \tag{2}$$

Furthermore, we can express Equation (2) as the familiar back-to-front compositing equation in [Porter and Duff 84]:

$$I_f(x) = \alpha_{\text{new}}(x)I_{\text{new}}(x) + (1-\alpha_{\text{new}}(x))I_f(x). \tag{3}$$

Here, $I_f(x)$ is the amount of light at location x in the frame buffer, and $\alpha_{\text{new}}(x)$, $I_{\text{new}}(x)$ are the new incoming opacities and light intensities, respectively. In OpenGL, we use the blending function

$$glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA)$$

to achieve the compositing for a back-to-front rendering order.

2.2. X-Ray Optical Model

The low-albedo optical model [Max 93] requires sorting the voxels along the viewing direction. We can use an x-ray model [Max 95] to render a volume and avoid sorting the voxels [Crawfis 96]. The x-ray model computes $I(x)$ as

$$I(x) = \int_0^L C(s)\mu(s)ds. \tag{4}$$

Here, L is the length of the viewing ray, C is the reflected or emitted energy, and μ is the absorption coefficient of the material. We can discretize Equation (4) and write its Riemann sum using a composition formula in OpenGL:

$$I_f(x) = \alpha_{\text{new}}(x)I_{\text{new}}(x) + I_f(x). \tag{5}$$

Here, $I_f(x)$ is the amount of light at location x in the frame buffer, and $\alpha_{\text{new}}(x)$, $I_{\text{new}}(x)$ are the new incoming opacities and light intensities, respectively. We use glBlendFunc(GL_SRC_ALPHA, GL_ONE) to perform the accumulation. From Equation (5), voxels can be rendered regardless of their viewing order, since all voxels contribute directly to the final pixel using an x-ray's integration model.

3. Splatting Techniques Using Hardware

3.1. Immediate Mode Rendering

To implement the textured splats algorithm of Crawfis and Max [Crawfis and Max 93] using the low-albedo optical model, two requirements must be met. First, the splat has to be rotated such that the normal of the texture mapped quadrilateral always points toward the eye. Second, the voxels must be rendered in a back-to-front order such that the correct image can be reconstructed according to the low-albedo model. Mueller et al. [Mueller et al. 99] have shown that for opaque surfaces an improvement over this technique is to split the reconstruction kernels by image-aligned slice planes. Our focus here is on a more efficient implementation of the texture splats algorithm in [Crawfis and Max 93]. We repeat their immediate-mode algorithm for volume rendering here, and have efficiently reimplemented it as a base comparison for further performance refinements. Immediate mode rendering is performed in the steps given in Algorithm 1 for volume rendering using textured splats:

This is an improved algorithm over [Crawfis and Max 93] because in their original implementation, Crawfis and Max rebuilt the quadrilateral geometry in world space for each view. Figure 1 shows an image rendered with oriented splats, where the scale of each splat is reduced for illustrative purposes.

Algorithm 1.

Obtain the new view parameters;
Generate a sorted voxel list along the new view direction;
For each voxel in a back-to-front order of the list, do
    Transform the voxel center position for the new view;
    Generate a quad centered at the voxel location;
    Rotate this quad such that its normal is parallel to the perspective view direction;
    Map the texture of the footprint onto the quad;
    Render the quad and composite it into the frame buffer;
End for.

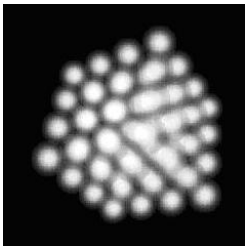


Figure 1. A cube of $4 \times 4 \times 4$ voxels is rendered from an oblique view with reduced-scale splats. The quadrilateral normals are parallel to the view direction.

3.2. Splatting Using a Vertex Shader Program

Vertex programs provide a powerful instruction set [Wynn 01a] which can be used to perform fairly complex vertex transformations on the GPU. A user-defined vertex program [Lindholm et al. 01] takes over the transformation stage of the OpenGL pipeline when the GL_NV_vertex_program is enabled. It thus bypasses the conventional transformation and lighting (T&L) pipeline. Figure 2 shows the pipeline for a general vertex shader. In our study, the

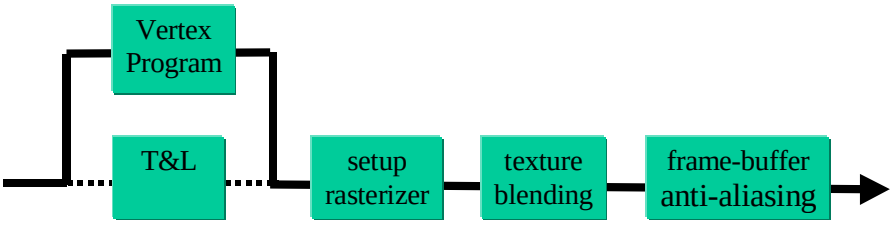


Figure 2. The pipeline of the vertex shader.

vertex program only deals with per-vertex transformations. No per-vertex lighting is considered, nor used, in our volume renderer.

3.2.1. Billboard

The running time of a vertex program depends mostly on its length or number of instructions. To use as few instructions as possible, we use a fast billboard technique [Lighthouse] to create the four coordinates of the quadrilateral centered around the voxel.

Suppose we have a model view matrix M for some view as follows:

$$M = \begin{bmatrix} \multicolumn{3}{c}{N} & \\ \boxed{\begin{matrix} m_0 & m_4 & m_8 \\ m_1 & m_5 & m_9 \\ m_2 & m_6 & m_{10} \end{matrix}} & \begin{matrix} m_{12} \\ m_{13} \\ m_{14} \end{matrix} \\ \begin{matrix} m_3 & m_7 & m_{11} \end{matrix} & m_{15} \end{bmatrix}$$

N is the top-left 3×3 submatrix of M. We can extract N from M and obtain its inverse matrix, N^{-1}. The inverse is simply the transpose of N, since N is always an orthogonal matrix (there are only rotations in our camera controls).

$$N = \begin{bmatrix} m_0 & m_4 & m_8 \\ m_1 & m_5 & m_9 \\ m_2 & m_6 & m_{10} \end{bmatrix} \quad N^{-1} = \begin{bmatrix} m_0 & m_1 & m_2 \\ m_4 & m_5 & m_6 \\ m_8 & m_9 & m_{10} \end{bmatrix}$$

From N^{-1}, the two vectors, right and up, are defined as follows:

$$\begin{aligned} right &= [m_0, m_4, m_8] \\ up &= [m_1, m_5, m_9]. \end{aligned}$$

Then, the four coordinates of our view-aligned quadrilateral (Figure 3) in world space are defined as:

$$a = center + (-right - up) * quad\_size/2 \tag{6}$$

$$b = center + (+right - up) * quad\_size/2 \tag{7}$$

$$c = center + (+right + up) * quad\_size/2 \tag{8}$$

$$d = center + (-right + up) * quad\_size/2. \tag{9}$$

Here, *center* is the voxel location and *quad_size* is the side length of the quadrilateral in world space. For parallel projection, the splat size on the screen is given by:

$$splat\_size = \frac{quad\_size * image\_size}{w}. \tag{10}$$

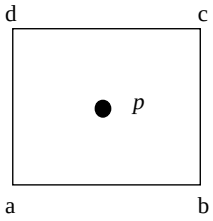


Figure 3. A $quad\_size \times quad\_size$ quadrilateral centered at the voxel p.

Here, *splat_size* is the desired splat size (in pixels) on the screen, *image_size* is the created image size, and w is the clip window size in clip space for parallel projection. We use Equation (10) to determine the splat size for our rendering timing analysis.

3.2.2. Vertex Shader Program

A vertex shader typically includes two kinds of vertex programs and auxiliary NVIDIA calls. The Vertex Program (VP) is invoked during the execution of each glVertex and glDrawElements calls. The other program is the Vertex State Program (VSP) that is executed only once per view from the user application. The VSP saves parameters in GeForce4's program parameter registers that are subsequentially used by the VP. Auxiliary NVIDIA calls pass parameters into the NVIDIA program registers on the GPU that are used by both the VSP and the VP. The vertex program, vertex state program, and auxiliary NVIDIA calls cooperate to perform the proper transformations and splat orientations in our vertex shader. This method uses a vertex array with four entries per voxel. Each entry contains the vertex index (from 0 to 3) on a quadrilateral and the voxel center in world space. These are transformed into clip space coordinates of the texture-mapped quadrilateral. We show the implementation details of our vertex shader. To make them easy to understand, we also show the partial listings of the GLUT-based OpenGL display function.

Auxiliary NVIDIA calls. Auxiliary NVIDIA calls initialize the program registers used in the VSP and the VP. All program registers are four-component vector registers. We track the model view matrix and the concatenation of the model view matrix and the projection matrix into the program registers c[0]–c[3], and c[4]–c[7], respectively, whenever the view changes. We also set the coefficients of the *right* and *up* vectors from Equations (6)–(9) in the registers c[16]–c[19]. The texture coordinates are passed into registers c[24]–c[27]. The reconstruction kernel radius is passed into the register c[8].

```
//Track the Modelview matrix into vector registers c[0]-c[3].
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 0, GL_MODELVIEW, GL_IDENTITY_NV);

// Track the concatenation of the modelview and projection matrix
// in registers c[4]-c[7].
glTrackMatrixNV( GL_VERTEX_PROGRAM_NV, 4, GL_MODELVIEW_PROJECTION_NV,
  GL_IDENTITY_NV );

// multiplier for billboard
// c[16].x - c[19].x are the coefficients of right vector
// in Equations (6) - (9)
// c[16].y - c[19].y are the coefficients of up vector
// in Equations (6) - (9)
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 16,  -1.0, -1.0, 1.0, 0.0 );
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 17,  1.0, -1.0, 1.0, 0.0 );
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 18,  1.0, 1.0, 1.0, 0.0 );
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 19,  -1.0, 1.0, 1.0, 0.0 );

// c[24].xy - c[27].xy store the texture coordinates for the quads.
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 24,  0, 0, 0, 0 );
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 25,  1, 0, 0, 0 );
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 26,  1, 1, 0, 0 );
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 27,  0, 1, 0, 0 );

// store the splat size from Equation (10) in register c[8]
float r = splat_size;
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 8,  r, r, r, r );
```

Vertex state program. The VSP is performed once per view from the GLUT display callback function. It sets the view-dependent parameters in the proper vertex program registers. In our study, it calculates the offsets of the four vertices on a quadrilateral. The offsets are then stored in the registers c[20]–c[23] used by the VP.

```
!!VSP1.0
# Fast billboard: calculate the right and up vector
# Input: -- set by auxiliary NVIDIA calls
#   c[0]...c[3] contains the modelview matrix.
#   c[4]...c[7] contains the concatenation of modelview and projection matrix
#   c[16]...c[19] contains the multiplier coefficent.
# Output: -- used by the VP
#    c[20]...c[23] contains the billboard offset of a voxel

# Move RIGHT and UP into R0, and R1. Mask w, so w remains 0.
    MOV R0.xyz,c[0];
    MOV R1.xyz,c[1];
# Multiply right and up vectors by their coefficients in Equations (6) - (9).
    MUL R3,R0,c[16].x;
    MAD c[20],R1,c[16].y,R3;
    MUL R3,R0,c[17].x;
    MAD c[21],R1,c[17].y,R3;
    MUL R3,R0,c[18].x;
    MAD c[22],R1,c[18].y,R3;
```

```
    MUL R3,R0,c[19].x;
    MAD c[23],R1,c[19].y,R3;
    END
```

Vertex program. The VP is invoked for each input vertex. In our VP, v[0] contains the index of a vertex on a quadrilateral, v[1] contains the voxel position (the quadrilateral center), and v[3] contains the vertex color. They are set in the GLUT display callback function. The registers c[4]–c[7] store the concatenation of the model view matrix and projection matrix. The registers c[20]–c[24] store the billboard offsets of a voxel which are passed down from the VSP. Our VP calculates the new position of a vertex on a quadrilateral in world space and then transforms it into clip space. The vertex color and texture coordinates are also passed into o[COL0] and o[TEX0], respectively, for the down-stream pipeline stage.

```
!!VP1.0
# This is the vertex program to do transformation for billboard.

# Input:  -- from the VSP and auxiliary NVIDIA calls
#  v[0].x contains the vertex index of four corners of a quad, from 0 to 3
#  v[1] contains the input vertex position, from NVIDIA vertex array
#  v[3] contains the vertex color, from NVIDIA vertex array
#  c[4]...c[7] contains the concatenation of the modelview
#  and projection matrices.
#  c[20]...c[23] contains the billboard offset of a voxel
#  c[24]...c[27] contains texture coordinates
#  c[8] contains the splat size
#  R0,R1are temporary registers
# Output:
#  o[HPOS] contains the vertex coordinates in clip space
#  o[COL0] contains the color of the vertex
#  o[TEX0].xy contains the texture coordinates

# HPOS,COL0,TEX0 are the indices of the output registers which are
#   defined by the down-stream pipeline stage.

# Determine which vertex on the quad we are suppose to output.
    ARL A0.x,v[0].x;
    MOV R0,c[A0.x+20];
    MAD R1, R0, c[8], v[1];
# Transform the given vertex from world space into the clip space.
    DP4 o[HPOS].x,R1,c[4];
    DP4 o[HPOS].y,R1,c[5];
    DP4 o[HPOS].z,R1,c[6];
    DP4 o[HPOS].w,R1,c[7];
# Pass through the color.
    MOV o[COL0],v[3];
# Pass through the texture coords.
    MOV o[TEX0].xy,c[A0.x+24];
    END
```

GLUT display callback function. The GLUT display callback function is invoked for each redisplay, and specifies the input vertex arrays, performs the VSP, and invokes the VP. We first enable the NVIDIA vertex program. The VSP is called here to set the *right* and *up* vectors in the proper program registers. glVertexAttribPointerNV associates the vertex index on a quadrilateral, center position, and the color with the input registers v[0], v[1], and v[3], respectively. Then, glDrawElements is issued during which geometry transformation and splat orientation are performed by invoking the VP on the GPU.

```
 // Enable NVIDIA vertex program
glEnable( GL_VERTEX_PROGRAM_NV );
glBindProgramNV( GL_VERTEX_PROGRAM_NV, vpid );
glExecuteProgramNV(GL_VERTEX_STATE_PROGRAM_NV, vspid, NULL_DATA);

//Turn on our NVIDIA attribute arrays
glEnableClientState(GL_VERTEX_ATTRIB_ARRAY0_NV);
glEnableClientState(GL_VERTEX_ATTRIB_ARRAY1_NV);
glEnableClientState(GL_VERTEX_ATTRIB_ARRAY3_NV);

// Associate the attribute arrays with the vertex array
glVertexAttribPointerNV(0, 1, GL_FLOAT, sizeof(BB_Vertex),
    &current_vertex_list[0].index);
glVertexAttribPointerNV(1, 3, GL_FLOAT, sizeof(BB_Vertex),
    &current_vertex_list[0].x);
glVertexAttribPointerNV(3, 4, GL_FLOAT, sizeof(BB_Vertex),
    &current_vertex_list[0].r);

// Issue the drawing operation during which the vertex program is invoked
glDrawElements(GL_QUADS, 4*vertex_num, GL_UNSIGNED_INT, current_vertex_indices);

glDisableClientState(GL_VERTEX_ATTRIB_ARRAY0_NV);
glDisableClientState(GL_VERTEX_ATTRIB_ARRAY1_NV);
glDisableClientState(GL_VERTEX_ATTRIB_ARRAY3_NV);

glDisable( GL_VERTEX_PROGRAM_NV );
```

3.3. Voxel Rendering Order

To obtain a correctly composited image from the low-albedo optical model, the voxels must be depth-sorted for each view. We investigated several methods to sort the voxels. First, we examine sorting on the fly for each new view, where we rebuild the index list into the vertex array each time the view changes. Using a quick-sort algorithm, its time complexity is $n * \log(n)$, where n is the number of voxels in a volume. A bucket-sort algorithm gives a much better

Volume Size	Sorting Time (in seconds	
	Quick Sort	Bucket Sort
$64 \times 64 \times 64$	0.625	0.11
$100 \times 100 \times 100$	2.55	0.41
$128 \times 128 \times 128$	5.45	0.88

Table 1. Sorting/rendering time for different volume sizes on an Intel P4 processor of 2.0 Ghz.

sorting speed since its time complexity is proportional to n. Table 1 shows the sorting time for volumes with different sizes. Since the sorting time was not a significant cost for the immediate-mode render, this approach worked well. However, as we drive the splat rasterization time down, the time required for sorting becomes significant. Since most volume data sets have more than 64^3 voxels, sorting for each view, or even for each nth view, was deemed impractical for rendering the volume interactively.

An alternative way to deal with rendering order is to use the x-ray model instead of the low-albedo optical model described in Section 2. Voxel rendering order is no longer necessary to generate a correct image.

Finally, in rough sorting of the voxel list, all voxels are presorted for several orthogonal viewing directions. When an arbitrary view comes, the presorted voxel list with the closest sorted view direction is selected and voxels are rendered using this order.

3.4. Point Convolution Rendering

Considering that the voxels are the samples of the volume in R^3, the continuous volume, $V(x, y, z)$, can be reconstructed from the discrete sample values with the reconstruction kernel. The value of $V(x, y, z)$ can be calculated from Equation (11):

$$V(x, y, z) = \sum_{(i,j,k) \in Vol} f(i, j, k)h(i - x, j - y, k - z) \tag{11}$$

Here, $f(i, j, k)$ is the discrete voxel value at the location (i, j, k), and h is our reconstruction kernel. To integrate the volume along an arbitrary viewing direction w, using an x-ray model, we first transform the volume into eye space:

$$V(u, v, w) = \sum_{(s,t,r) \in Vol} \tilde{f}((s, t, r)h(s - u, t - v, r - w). \tag{12}$$

Here, $\tilde{f}(s, t, r) = \tilde{f}((i, j, k)M^T) = f(i, j, k)$ and M is the transformation matrix to eye space. We calculate the image $I(u, v)$, as follows:

$$\begin{aligned} I(u,v) &= \int V(u,v,w)dw \\ &= \int \sum_{(s,t,r)\in Vol} \tilde{f}((s,t,r)h(s-u,t-v,r-w)dw \\ &= \sum_{(s,t,r)\in Vol} \tilde{f}((s,t,r) \int h(s-u,t-v,r-w)dw. \end{aligned} \tag{13}$$

If $h(u,v,w)$is the three-dimensional reconstruction kernel for a voxel at $(0,0,0)$, its two-dimensional footprint, $footprint_h(u,v)$, is given by:

$$footprint_h(u,v) = \int_{-\infty}^{\infty} h(u,v,w)dw. \tag{14}$$

If we restrict ourselves to rotationally symmetric kernels, Equation (13) can be reduced to:

$$I(u,v) = \sum_{(s,t,r)\in Vol} \tilde{f}((s,t,r)footprint_h(s-u,t-v). \tag{15}$$

Equation (15) is the familiar splatting formula. It can be rewritten as

$$\begin{aligned} I(u,v) &= \sum_{(s,t)\in Vol} \tilde{p}_w((s,t)footprint_h(s-u,t-v) \\ &= \tilde{p}_w(u,v) * footprint_h(u,v). \end{aligned} \tag{16}$$

Here, $*$ is the convoluting operation and $\tilde{p}_w(u,v)$ is the projection function of $\tilde{f}(u,v,w)$ along the w direction as follows:

$$\tilde{p}_w(u,v) = \sum_{(s,t,r)\in Vol} \tilde{f}((s,t,r)\delta(s-u,t-v). \tag{17}$$

Since the convolution operation is supported in OpenGL 1.2 or later, we can create the final image from the voxels directly using Equation (16).

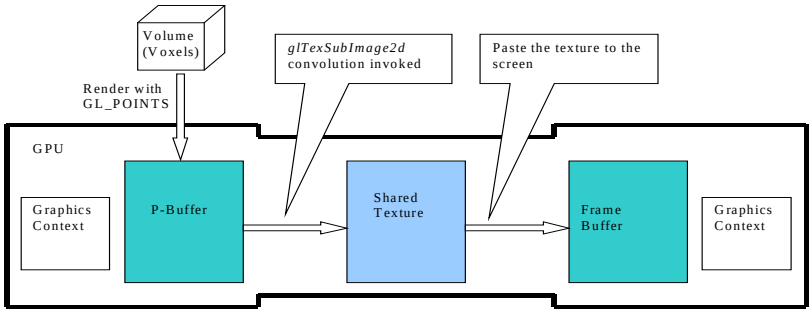
The P-buffer [Wynn 01b] offers the full graphics context capacity as the frame buffer, except that all drawing operations are performed off-screen. Our point convolution rendering applies additive blending of point primitives into the P-buffer to obtain the projection of all of the voxel centers in Equation (17). The color at each point is set to some monochrome color due to the x-ray model. The voxels are then projected onto the P-buffer and added into the frame buffer. Then, a texture is copied from the P-buffer to a shared texture object between the P-buffer and the frame buffer, during which convolution is invoked to generate the final image. Rather than mapping the kernel texture for each quadrilateral centered at the voxel location, convolution with the

Algorithm 2.

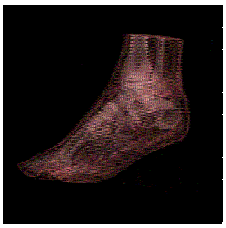
1) Transform all voxels into the new view;
2) Render all voxels to an off-screen P-buffer using the GL_POINTS primitive with additive blending;
3) Enable the convolution operation for the P-buffer;
4) A shared texture between the P-buffer and the frame buffer is bound and filled with the P-buffer via *glTexSubImage2D*;
5) The shared texture is pasted onto the frame buffer.

kernel is performed during the pixel copy operation. Algorithm 2 gives the required steps.

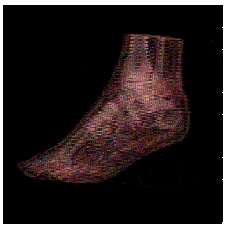
Figure 4(a) shows the pipeline of this method. The transformed voxels are first projected on the P-buffer with GL_POINTS primitive. Figure 4(b) shows the image in the P-buffer after projecting the voxels of a foot data set onto the P-buffer using GL_POINTS. The voxel intensities are magnified to make the image visible and comparable with Figure 4(c). A texture image is then



(a)



(b)



(c)

Figure 4. The pipeline for point convolution. (a) is the diagram of point convolution; (b) is the image in the P-buffer with voxel centers rendered as points into the P-buffer before convolution; (c) is the texture image created.

copied from the P-buffer using *glTexSubImage2D* command as in Step 4 in Algorithm 2. Since the flag GL_CONVOLUTION is enabled, the convolution operation is applied for each pixel and thus each texel value in the texture image is the output of the convolution between its corresponding pixel in the P-buffer and the kernel filter. Figure 4(c) shows the image copied from the P-buffer with convolution for each pixel. This texture image is then pasted to the on-screen frame buffer to generate the final image. Since the P-buffer is on the graphics card memory and the texture is shared between the P-buffer and on-screen frame buffer, there is no latency due to the bandwidth limitation between the CPU and the GPU.

Rendering with GL_POINTS is extremely fast using hardware-supported OpenGL. We can render the points in any order, since the x-ray model is a simple integration. The additive blending accomplishes this if two voxels project to the same point.

4. Results and Discussion

All presented results are generated on a Dell Precision 530 workstation configured as listed in Table 2. For the point convolution, parallel projection must be applied since the convolution is performed after all voxels have been projected onto the P-buffer and we use the same kernel radius for all samples regardless of their distances to the eye position. There is no such limitation for the other two methods. The parallel projection is used here for consistency in comparing the three methods. We present several timing tests to measure the bottlenecks and limitation of each method.

Table 3 and Figure 5 capture the time per splat using different projected splat sizes. All three techniques render a 100^3 cube with splat sizes ranging from 1×1 to 70×70 pixels. The maximum size of the convolution kernel supported by the GeForce4 is 11×11 pixels. An image resolution of 512×512 was used for all the tests. The results are listed in Table 3.

The time to interactively render a volume consists of three parts: the geometry transformation time, the time to issue OpenGL functions calls and pass the data down to the GPU, and the time to perform the rasterization. From Table 3 and Figure 5, when the splat size is less than 20×20, the rasteri-

CPU	2.0 GHz Intel P4 × 1
Main Memory	2 GB
Graphics Card	GeForce Ti 4600 with 128 MB
OS	Microsoft XP Pro

Table 2. The specification of the workstation.

Splat size (pixel × pixel)	Immediate Mode (microsec)	Vertex Shader (microsec)	Point Convolution (microsec)
1×1	3.33	0.318	0.378
3×3	3.36	0.320	0.398
5×5	3.35	0.328	0.435
7×7	3.35	0.324	0.497
9×9	3.33	0.320	0.567
10×10	3.34	0.328	0.615
11×11	3.34	0.328	0.660
15×15	3.36	0.383	-
20×20	3.36	0.522	-
30×30	3.35	1.14	-
40×40	3.35	2.06	-
50×50	3.37	3.14	-
60×60	4.56	4.52	-
70×70	6.17	6.10	-

Table 3. Per-splat rendering time for the three methods.

zation time is negligible. The other two parts dominate the total rendering time. We observe in Figure 5 that the vertex shader reduces its rendering time to one-tenth that of immediate mode rendering by reducing the issuing time using a vertex array and the transformation time via a vertex program. On the contrary, when the splat size is greater than 50 × 50, most of the rendering time is occupied by texture rasterization and the time for voxel transformation only accounts for a very small part. The GPU is always busy rasterizing. The benefit from reducing the GPU stalls by the vertex shader is almost offset completely. We observe this fact in Figure 5 where the curves for immediate mode rendering and vertex shader rendering almost converge when the splat size is greater than 50 × 50.

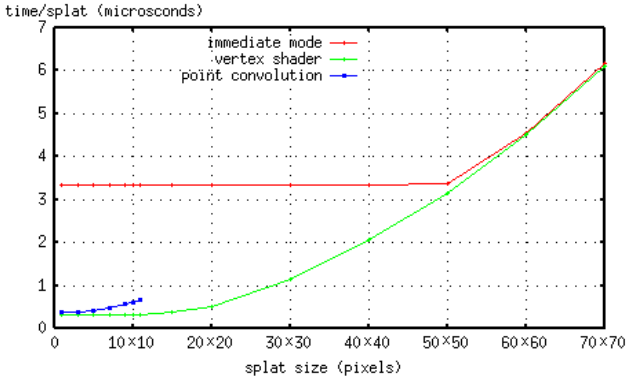


Figure 5. The plot from Table 3.

Number of Voxels	Immediate mode (microsec)	Vertex shader (microsec)	Point convolution (microsec)
10000	3.58	0.610	43.9
20000	3.56	0.415	22.1
50000	3.35	0.364	8.86
100000	3.32	0.360	4.48
200000	3.34	0.344	2.28
500000	3.33	0.325	0.936
800000	3.34	0.320	0.603
1000000	3.34	0.319	0.506
2000000	3.31	0.316	0.273
3000000	3.30	0.314	0.197

Table 4. Per-voxel rendering times of the three methods.

A footprint size of 50 × 50 would be used for the smallest of volumes and is not really appropriate for preclassified splatting. It is presented here to help illustrate the GPU stall that is prevalent when performing immediate mode rendering. At more than 50 × 50 footprint sizes, the GPU finally has enough work to occupy it while the CPU sends down the next splat. Note that without a dedicated splat primitive, such as the one implemented using vertex shaders here, issuing more than one splat using a vertex array is not possible due to the per-splat reorientation.

Table 4 and Figure 6 illustrate the time for rendering each voxel given a fixed splat resolution, but an increasing volume resolution or number of voxels. These experiments help illustrate the overhead associated with each of the methods. All results are generated using a splat size of 7 × 7 pixels.

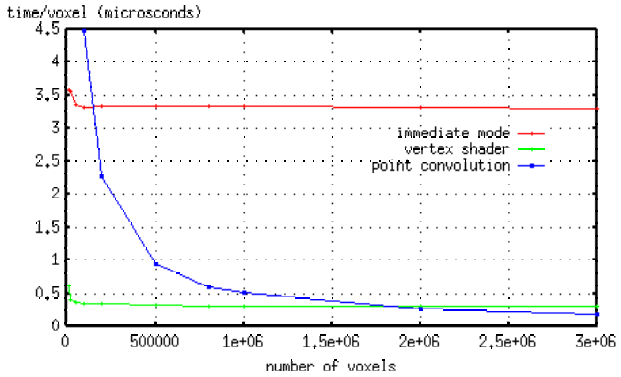


Figure 6. The plot from Table 4. Note: To make the figure clearer, the first three rendering times for the point convolution method are not drawn in the figure to reduce the range.

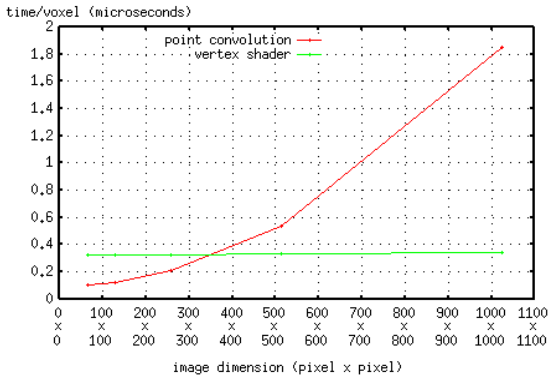


Figure 7. Per-voxel rendering time for vertex shader rendering and point convolution rendering using different window sizes. A splat size of 7×7 pixels is applied for all window sizes.

The first two methods approximate the splatting by blending reconstruction kernel textures for each voxel in a back-to-front order. If there are only a small number of voxels (less than 10000) in a volume, the overhead of other calls such as initialization cannot be neglected, and thus rendering time per voxel is a little longer. As the number of voxels increases, the time for the initialization and set-up is negligible. Per-voxel rendering times reach a stable level. Again, we do not consider the time required to sort the voxels in this experiment.

The third method, point convolution, is a different case. The time to rasterize a point in OpenGL is negligible on most modern graphics cards. The per-voxel rendering time decreases proportionally to the increase of voxels. This is because the convolution is invoked during the execution of glTexSubImage2D which copies the P-buffer to the shared texture. Since the convolution is a per-pixel operation, the consuming time is determined by the size of P-buffer, or in this case, by the window size. The P-buffer is created with the same size as the application window. Figure 7 demonstrates this fact. When the window size increases, per-voxel rendering time of the point convolution method follows, since there are more pixels to be convolved and convolution is a time-consuming task. However, the per-voxel rendering time of the vertex shader method is not deterred due to the very high fill rate of the GeForce4.

In Table 5, the frame times on the four data sets for both the x-ray model and the low-albedo model are reported. The frame time for the low-albedo is a little higher than that for the x-ray model. This is due to the overhead to switch the rough sorting voxel list in the low-albedo model. From Table 5, we observe that the frame times for vertex shader rendering are much faster

Data Set	Size	Relevant Voxels	Frame time (in seconds)				
			Immediate Mode		Vertex Shader		Point Convolution*
			x-ray	Low-albedo	x-ray	Low-albedo	x-ray
NEGHIP	64^3	82,316 (31.4%)	0.275	0.295	0.0193	0.0255	0.519
Foot	128^3	197,865 (9.4%)	0.660	0.682	0.0470	0.0660	0.525
CT Head	128^3	673,014 (32.1%)	2.25	2.31	0.166	0.215	0.549
UNC Brain	$256^2 \times 145$	2,493,047 (26.2%)	8.35	8.63	0.585	0.844	0.641
Nerve	$512^2 \times 76$	4,621,098 (23.2%)	14.9	15.8	1.09	1.51	0.656

* Point convolution only uses the x-ray model.

Table 5. Average frame times (in seconds) for ten random views of the four data sets for the x-ray and low-albedo models. The image size is 512^2. All images are rendered with a splat size of 9×9 pixels.

than immediate mode rendering. The frame rate using the vertex shader is a little more than ten times that of immediate mode as presented in [Crawfis and Max 93]. Interactive volume rendering on consumer-level graphics cards is quickly becoming feasible.

Table 5 also shows that the point convolution offers an acceptable frame rate. Its frame time varies only slightly for a fixed image size and a fixed kernel size on the four data sets. The major rendering time is occupied by the pixel convolution which is totally determined by the image size and kernel size. For a fixed window-size application, the rendering time of point convolution is not affected greatly by the number of voxels. It provides a new method to implement hardware-accelerated volume rendering using an x-ray model for large data sets.

Figure 8 shows the images created from the NEGHIP (64^3), the foot (128^3), and the CT head (128^3) data sets. All result images are generated with preclassification transfer functions. The transfer functions are chosen case by case. The images ((a), (d), (g)) in the left column are rendered with the low-albedo models by the vertex shader. The images in the middle and right columns are rendered with the x-ray model by the vertex shader and point convolution methods, respectively.

5. Conclusions

Based on our results and analysis in Section 4, we summarize the pros and cons of the three rendering techniques in Table 6. The vertex shader offers both fast rendering speed and good image quality on the fairly complex volume (with million voxels). However, the vertex shader uses four times the amount

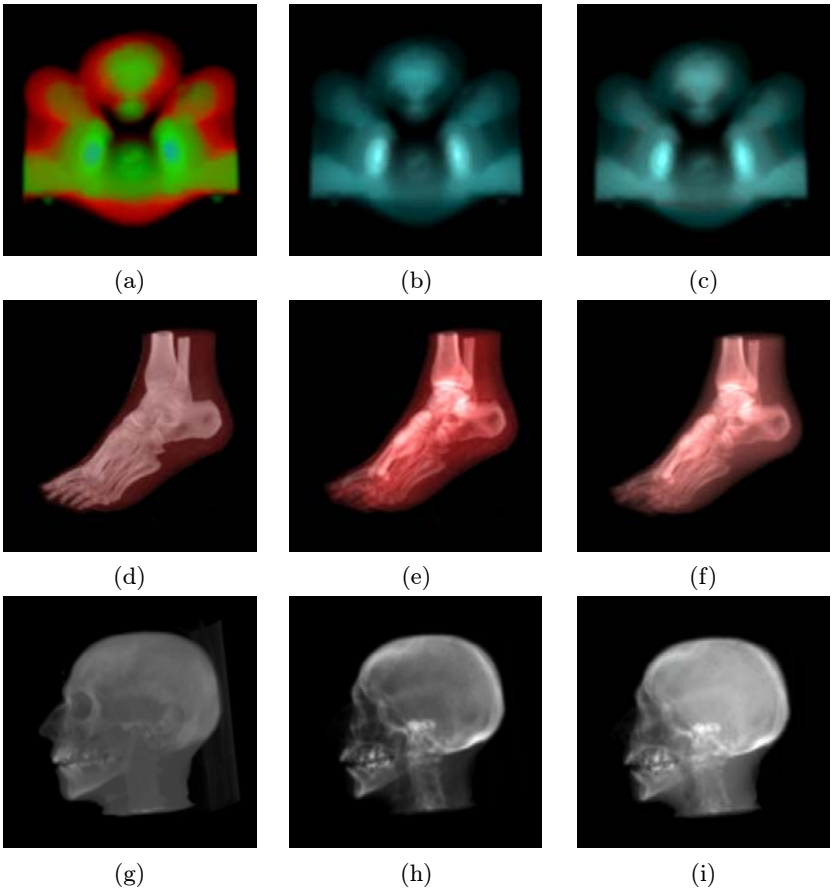


Figure 8. The images in the left column are rendered with the low-albedo model on the preclassified voxels by the vertex shader method; the middle and right columns are rendered with the x-ray model by the vertex shader and the point convolution methods, respectively. All images are created with a splat size of 9×9 pixels. (a), (b), and (c) are the images created from the NEGHIP (64^3) data set; (d), (e), and (f) are the images created from the foot (128^3) data set; (g), (h), and (i) are the images created from a CT head (128^3) data set.

of memory to store the quadrilateral for each voxel in comparison with the point convolution. In addition, the vertex shader needs more memory to store the presorted voxel lists for the low-albedo model while there is no such requirement for the point convolution. The point convolution provides an acceptable rendering speed to render the very large data set and generates the x-ray style image. For all three methods, an image-aligned splatting with post-classification could improve the image quality significantly.

	Immediate Mode	Vertex Shader	Point Convolution
Projection Mode	Parallel/Perspective	Parallel/Perspective	Parallel
Optical Model	Low-albedo/x-ray	Low-albedo/x-ray	x-ray
Image Quality	Good	Good	Good, alias occurs when view direction is exactly perpendicular to the volume face
Image Color	Color for Low-albedo	Color for Low-albedo	Monochrome
Rendering Speed ($\leq$ 2 million voxels)	Slowest	Fastest	Fast
Rendering Speed ($>$ 2 million voxels)	Slowest	Fast	Fastest
Sorting	Requred for Low-albedo	Requied for Low-albedo	No
Memory	Number of voxels	4*Number of voxels	Number of voxels

Table 6. Pros and cons of three methods.

References

[Blinn 82] J. F. Blinn. "Light Reflection Functions for Simulation of Clouds and Dusty Surfaces." *Computer Graphics (Proc. SIGGRAPH '82)* 16:3 (1982), 21–29.

[Crawfis and Max 93] R. Crawfis and N. Max. "Texture Splats for 3D Vector and Scalar Field Visualization." In *Proc. Visualization '93*, pp. 261–266. Los Alamitos, CA: IEEE CS Press, 1993.

[Cullip and Neumann 93] T. Cullip and U. Neumann. "Accelerating Volume Reconstruction with 3D Texture Hardware." Technical Report, UNC TR93-0027, 1993.

[Crawfis 96] R. Crawfis. "Real-Time Slicing of Data Space." In *Proc. Visualization '96 (October 1996)*, pp. 271–277. Los Alamitos, CA: IEEE CS Press, 1996.

[Huang et al. 00] J. Huang, N. Shareef, K. Mueller, and R. Crawfis. "FastSplats: Optimized Splatting on Rectilinear Grids" In *IEEE Visualization 2000*, pp. 219-226, Los Alamitos, CA: IEEE Press, 2000.

[Laur and Hanrahan 91] D. Laur and P. Hanrahan. "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering." *Computer Graphics (Proc. SIGGRAPH '91)* 25:4 (1991), 285–288.

[Lighthouse] Lighthouse 3D. Available from World Wide Web (http://www.lighthouse3d.com/opengl/billboarding/).

[Lindholm et al. 01] E. Lindholm, M. J. Kilgard, and H. Moreton. "A User-Programmable Vertex Engine." In *Proceedings of SIGGRAPH 2001, Computer*

Graphics Proceedings, Annual Conference Series, edited by E. Fiume, pp. 12–17. Reading, MA: Addison-Wesley, 2001.

[Max 93] N. Max. "Sorting for Polyhedron Compositing." In *Focus on Scientific Visualization,* edited by H. Hagen, H. Muller, and G. Nielson, pp. 259–268. Berlin: Springer-Verlag, Berlin, 1993.

[Max 95] N. Max. "Optical Model for Direct Volume Rendering." *IEEE Trans. Vis. and Comp. Graph.* 1:2 (1995), 99–108.

[Meißner et al. 00] M. Meißner, J. Huang, D. Bartz, K. Muller, and R. Crawfis. "A Practical Evaluation of Popular Volume Rendering Algorithms." In *Proc. of Volume Vis. Sym.*, pp. 81–90. Los Alamitos, CA: IEEE CS Press, 2000.

[Mueller et al. 99] K. Mueller, J. Huang, N. Shareef, and R. Crawfis. "High-Quality Splatting on Rectilinear Grids With Efficient Culling of Occluded Voxels." *IEEE Trans. Vis. and Comp. Graph.* 5:2 (1999) 116–143.

[Porter and Duff 84] T. Porter and T. Duff. "Compositing Digital Images," *Computer Graphics (Proc. SIGGRAPH '84)* 18:3 (1984), 253–259.

[Wynn 01a] C. Wynn. "OpenGL Vertex Programming on Future-Generation GPUs." Available from World Wide Web (http://developer.nvidia.com/), 2001.

[Wynn 01b] C. Wynn. "Using P-Buffers for Off-Screen Rendering in OpenGL." Available from World Wide Web (http://developer.nvidia.com/), 2001.

Web Information:
The glut-based source code for three rendering methods is available at http://www.acm.org/jgt/papers/XueCrawfis03.

Daquing Xue, Department of Computer and Information Science, The Ohio State University, 395 Dreese Labs, 2015 Neil Avenue, Columbus, OH 43210-1277 (xue@cis.ohio-state.edu)

Roger Crawfis, Department of Computer and Information Science, The Ohio State University, 395 Dreese Labs, 2015 Neil Avenue, Columbus, OH 43210-1277 (crawfis@cis.ohio-state.edu)

Received August 30, 2002; accepted in revised form April 15, 2003.