Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2003)
G.-P. Bonneau, S. Hahmann, C. D. Hansen (Editors)

Smart Hardware-Accelerated Volume Rendering

Stefan Roettger[1], Stefan Guthe[2], Daniel Weiskopf[1], Thomas Ertl[1], Wolfgang Strasser[2]

[1] VIS Group, University of Stuttgart †
[2] WSI-GRIS, University of Tübingen

Abstract

For volume rendering of regular grids the display of view-plane aligned slices has proven to yield both good quality and performance. In this paper we demonstrate how to merge the most important extensions of the original 3D slicing approach, namely the pre-integration technique, volumetric clipping, and advanced lighting. Our approach allows the suppression of clipping artifacts and achieves high quality while offering the flexibility to explore volume data sets interactively with arbitrary clip objects. We also outline how to utilize the proposed volumetric clipping approach for the display of segmented data sets. Moreover, we increase the rendering quality by implementing efficient over-sampling with the pixel shader of consumer graphics accelerators. We give prove that at least 4-times over-sampling is needed to reconstruct the ray integral with sufficient accuracy even with pre-integration. As an alternative to this brute-force over-sampling approach we propose a hardware-accelerated ray caster which is able to perform over-sampling only where needed and which is able to gain additional speed by early ray termination and space leaping.

CR Category: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.

1. Introduction

The basic principle of volume rendering was described by Kajiya[10] as early as in 1984. Since then the availability of graphics hardware has lead to the establishment of the so-called 3D slicing method[1, 3]. The ray integral for each pixel, which Kajiya reconstructed by ray tracing, is calculated with graphics hardware by rendering view-port aligned slices through the volume. In this way the emission and absorption along each viewing ray through the volume is computed by sampling and blending the volume for each numerical integration step[5]. The main advantage is that the entire task can be performed by the graphics hardware and is limited only by the fill rate of the graphics accelerator.

A volume is commonly given as a scalar function sampled on a uniform grid. The main source for this kind of volume representation are CT (Computer Tomography) or MRI (Magnetic Resonance Imaging) scanners, which are widely used in medical imaging. In order to associate opacity and emission to the scalar values the volume density optical model of Williams et al.[25, 13] is usually applied. It defines the opacity and the chromaticity vector to be functions of the scalar values. When slicing a volume in a back-to-front fashion, the opacity and chromaticity are stored in an one-dimensional look-up table (often referred to as the transfer function) which is used to transform the scalar value at each rendered pixel into an RGBA vector. Then the ray integral can be calculated by simple alpha-compositing the slices.

This basic principle of hardware-accelerated volume rendering of regular grids has been extended in several ways to increase the quality of the renderings. In the following we give a list of the well established extensions and their purposes:

- **Ambient and Diffuse Lighting**: The emission of the volume is attenuated by a lighting operation[22, 24] to give more visual clues regarding the curvature of details in the volume.
- **Pre-Integration**: The slicing of the volume results in the so-called ring artifacts which are due to insufficient over-sampling of high frequencies in the transfer function. This issue is resolved by rendering slabs instead of slices. The ray integral of the ray segments inside each slab is a function of the scalar values at the entry and exit points of the ray, and the thickness of the slab (see also Figure 1). By pre-integrating[12, 19] the transfer function for each combination of scalar values the ray integral can be looked up easily for each rendered pixel using 2D dependent texturing[6].

† University of Stuttgart, Computer Science Institute, VIS Group, Breitwiesenstrasse 20-22, 70565 Stuttgart, Germany; E-mail: Stefan.Roettger@informatik.uni-stuttgart.de .

- **Hardware-Accelerated Pre-Integration**: Since pre-integration involves a fair amount of numerical operations, a change of the transfer function leads to a delay of the volume visualization while the 2D dependent texture has to be recomputed. If self-attenuation is neglected, interactive update rates of up to 10 Hertz[6] are achieved for a transfer function with 256 entries. However, this introduces artifacts in regions where the transfer function is very opaque. A better solution is to speed up the computation of the exact ray integral by using graphics hardware[18, 7]. The quantization artifacts that normally arise with an 8-bit frame buffer are overcome by using the floating point render target of modern graphics accelerators such as the ATI Radeon 9700[2] or NVIDIA GeForce FX[16]. A 256-step hardware-accelerated pre-integration with full floating point precision takes about 50 milliseconds on the ATI Radeon 9700.
- **Material Properties**: Besides the pre-integration of the transfer function, one can also pre-integrate material properties which describe the fraction of ambient, diffuse, and specular lighting which has to be applied to each rendered slab[14].
- **Volumetric Clipping**: To explore the interior of a volume conveniently volumetric clipping[23] is used to cut off parts of the volume which otherwise would hide important information. For that purpose the clip geometry is defined by an iso-surface of an additional clip volume (which need not necessarily be of the same size as the scalar volume). During rendering the opacity of each pixel is set to zero in the pixel shader if the corresponding clip value exceeds a certain threshold.

In summary, the described extensions have led to high quality volume rendering using the graphics hardware. However, the combination of the pre-integration technique with volumetric clipping has been an unsolved problem. Hence our first goal is to demonstrate the combination of these advanced techniques. To increase the accuracy even further, we propose a hardware-accelerated ray caster which is able to adapt the sampling frequency to the reconstruction error of the ray integral.

2. Accurate Volumetric Clipping

In order to combine pre-integration with volumetric clipping basically three steps are necessary. Let C_f and C_b be the scalar values of the clip volume at the entry and exit points of the ray segment in the range $[0,1]$. We also assume that the volume corresponding to clip values smaller than 0.5 should be clipped away. If both parameters C_f and C_b are above 0.5, then the slab is completely visible and no special treatment is necessary. Considering the case $C_f < 0.5$ and $C_b > 0.5$ as depicted in Figure 1, only the dark blue part of the volume has to be rendered. In this case we first have to set the front scalar value S_f to the value S'_f at entry point into the clipped region. Thus we perform a look-up into the pre-integration table with the parameters (S'_f, S_b) rather than with (S_f, S_b). In the general case S_f is replaced by S'_f according to

$$r = \left[\frac{[0.5 - C_f]}{C_b - C_f}\right], \quad S'_f = (1-r)S_f + rS_b \ .$$

The brackets denote clamping to the range $[0,1]$. For the case $C_f > 0.5$ and $C_b < 0.5$, the front scalar value need not be adjusted, which is expressed by $r = 0$. The same holds for the case $C_f > 0.5$ and $C_b > 0.5$. Similarly, the parameter S_b is replaced by S'_b as follows:

$$g = 1 - \left[\frac{[0.5 - C_b]}{C_f - C_b}\right], \quad S'_b = (1-g)S_f + gS_b \ .$$

If both clip values are below 0.5, the ray segment is clipped entirely, and thus the scalar values do not matter.

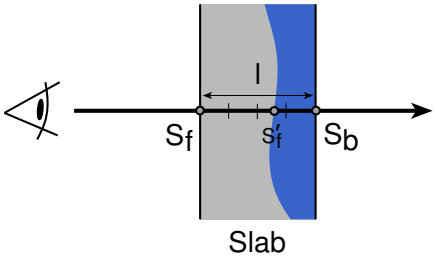


Figure 1: *Using slabs instead of slices for pre-integrated volume rendering as introduced by Engel et al.[6]. The scalar values at the entry and the exit point of the viewing ray are denoted by S_f and S_b, respectively. The thickness of the slab is denoted by l. The dark blue region remains after volumetric clipping. For this purpose, S_f has to be replaced by S'_f.*

The second problem we have to care about is the reduced length l' of the clipped ray segment. The numerical pre-integration depends on the three parameters S_f, S_b and l (see Figure 1). Using the optical model of Williams and Max[25] with the chromaticity vector κ and the scalar optical density ρ, the ray integral for each ray segment is given as follows:

$$\begin{aligned} S_l(x) &= S_f + \frac{x}{l}(S_b - S_f) \\ C(S_f, S_b, l) &= \int_0^l e^{-\int_0^t \rho(S_l(u))\mathrm{d}u} \kappa(S_l(t))\rho(S_l(t))\mathrm{d}t \\ \theta(S_f, S_b, l) &= e^{-\int_0^l \rho(S_l(t))\mathrm{d}t}, \quad \alpha = 1 - \theta \end{aligned}$$

Now we have to distinguish between two different types of transfer functions. For a transfer function that defines a set of isosurfaces the reduced ray segment length l' has no effect on the integrated chromaticity C and the integrated opacity α. For the other common case of a transfer function defined by a lookup table the three-dimensional integration problem can be reduced to two dimensions.

The pre-integration is performed for the constant ray segment length l. Let the visible fraction of the slab be denoted by $b = l'/l$. Then the transparency θ' of the clipped ray segment is the pre-integrated transparency θ raised to the b-th power because

$$\int_0^{l'} \rho(S'_l(t'))\mathrm{d}t' = b\int_0^{l} \rho(S_l(t))\mathrm{d}t \quad ,$$

$$\theta' = e^{-b\int_0^l \rho(S_l(t))\mathrm{d}t} = \left(e^{-\int_0^l \rho(S_l(t))\mathrm{d}t}\right)^b = \theta^b \quad .$$

In practice however, a first order approximation is sufficient if the thickness of the slabs is reasonable small.

If self-attenuation is neglected, the emission of the clipped ray segment is given by $C' = bC$. The computation of the correct emission cannot be expressed in terms of a two-dimensional pre-integration. For accurate results a three-dimensional pre-integration table has to be used, but high order polynomial approximations[7] do a remarkably good job to get rid of the huge three-dimensional pre-integration table. Aside from that issue, the neglection of self-attenuation is often a very good assumption in practice.

The third and last problem is the lighting of the slab. Sampling the gradient at a single point results in severe ring artifacts[23]. A better solution is to sample the gradient at the entry and exit points and to adjust the gradients in the same fashion as the scalar values. Then the final light intensity of the slab is the average of both adjusted light intensities. The operations involved here can be expressed as a linear interpolation of the gradients. We denote the interpolation factor by a.

Instead of calculating the factors for the adjustment of the scalar values, the emission, the opacity, and the gradients in the pixel shader we pre-compute the factors for all combinations of the clip values C_f and C_b and store them in a 2D dependent texture. The content of each R/G/B/A channel of the dependent texture as depicted in Figure 2 corresponds to the adjustment factors with the same name.

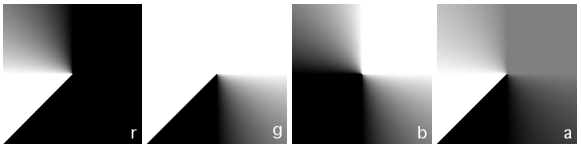


Figure 2: *2D dependent texture containing the adjustment factors used for accurate volumetric clipping (R/G/B/A channels are depicted from left to right).*

Using the pixel shader version 2.0[15], both scalar values can be adjusted by a single linear interpolation ("lrp" instruction). The entire pixel shader performing pre-integration, ambient and diffuse lighting, and accurate volumetric clipping is given in Appendix 8.2.

A comparison of the rendering quality between our approach and a naive application of the method by Weiskopf et al.[23] to pre-integration is depicted in Figure 3 (see Appendix 8.1 and 8.2 for the pixel shaders). A sphere has been cut out of the Bucky Ball data set so that the holes of the carbon rings are visible as green spots. Both images have been rendered with only 32 slices. Whereas the slices are clearly visible on the left, our method reproduces the clipped volume accurately. This means that our method is also exact in the viewing direction.

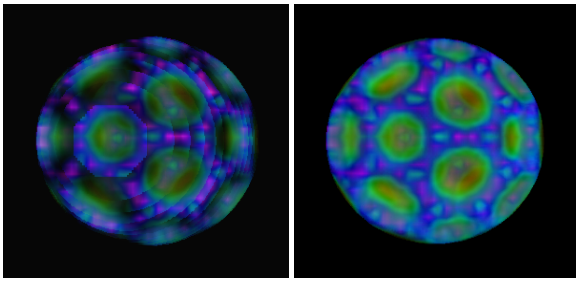


Figure 3: *Comparison of volumetric clipping quality.* ***Left****: Naive approach.* ***Right****: Accurate method.*

The presented volumetric clipping algorithm can also be extended to render segmented volumes[21]. In conjunction with the pre-integration technique this was an unsolved problem as well. If each segment is defined as a clip volume, this problem can be reduced to volumetric clipping.

For the case of two segmented volumes a clip volume is set up to include one entire segment. For the second segment we use an additional transfer function. For the part of the slab which is clipped away we perform another dependent texture lookup into the second pre-integration table and blend the two partial slabs depending on their orientation. For each additional segment this procedure has to be repeated.

3. Over-Sampling

In the last section we have described the combination of the pre-integration technique with volumetric clipping to improve the quality of volume visualizations. Nevertheless, even the mentioned advanced techniques are no guarantee for artifact-free renderings.

A fundamental assumption of volume rendering is that the scalar values of the volume are interpolated tri-linearly. The pre-integration technique has eliminated many of the rendering artifacts that arise from slicing the volume (compare top right and bottom left image of Figure 4), but it assumes a linear progression of the scalar values inside the slabs. This assumption does not match the actual cubic behaviour of the scalar values. Furthermore, the scalar function may have a sharp bend if a voxel boundary is crossed inside a slab. This

non-linear effect is amplified by lighting in regions with high second derivatives.

Considering pre-integration alone the exact reconstruction of the ray integral is a four-dimensional problem (for constant ray segment length l). Including the effect of lighting it can only be solved efficiently with over-sampling. The top row of Figure 4 illustrates the artifacts that may occur without over-sampling. Using 2-times over-sampling helps a lot, but there is still a large difference when stepping up to 4-times over-sampling. Fortunately 8-times over-sampling leads to insignificant improvements only, so that 4-times over-sampling is already a good choice for highest quality in practice.

Current graphics hardware allows to perform multiple blending steps at once in the pixel shader. We have implemented a pixel shader program that blends four slabs internally before it writes the result back into the frame buffer (see Appendix 8.3). As a side effect, the four slabs (see Figure 1) are blended with the high internal precision of the pixel shader.

For a window size of 512^2 pixels the rendering time is between 1 and 1.7 seconds for this multi-stepped approach. Because of the increased cache coherence we achieve a speed up of 70%-120% over the single-stepped version depending on the data set and viewing parameters. We conclude that 2-times over-sampling comes almost for free when using multi-stepping. For very high quality requirements one has to accept about twice the rendering time as normal.

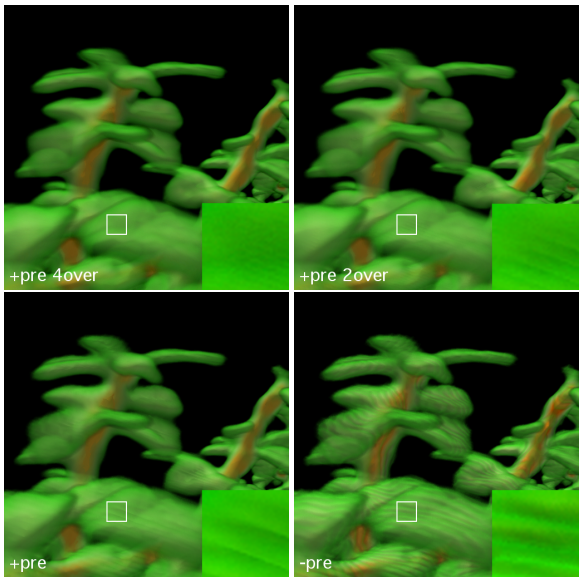


Figure 4: *A leave of the Bonsai[20]. The degradation of rendering quality is displayed from top left to bottom right: with pre-integration and 4-times over-sampling, with pre-integration and 2-times over-sampling, with pre-integration and without over-sampling, and neither with pre-integration nor over-sampling. In the bottom right corner of each image the zoom of a critical region is depicted which should show a smooth color transition. Due to slicing artifacts in the bottom right image artificial bands are visible. These remain even with 2-times oversampling but almost disappear with 4-times oversampling.*

4. Hardware-Accelerated Ray Casting

The previously discussed over-sampling approach suppresses artifacts caused by neglecting tri-linear interpolation, the crossing of voxel boundaries, and the non-linear behaviour of lighting. While this brute-force approach is practically artifact-free, it does not exploit any spatial coherence in the data set to increase the rendering performance. In comparison to brute-force over-sampling, a ray caster[4, 11] is able to adapt the sampling rate to the actual information in the data set. This often leads to a drastically reduced number of sample points.

Just like the flexibility of current graphics hardware allows hardware-accelerated ray tracing[17] it also allows to perform ray casting completely in hardware. In the following we describe a smart hardware-accelerated ray caster which applies an error-driven sampling scheme and additionally performs early ray termination.

The pre-integration technique is based on a piece-wise linear approximation of the ray integral, thus all higher order frequencies are neglected. In the optimal case one would choose the step length such that the difference of the exact ray integral and the linear approximation is lower than a pre-defined threshold. But this solution is infeasible since it requires the pre-integration of the entire data set.

Instead of solving the global problem we resort to solving the local problem, which basically requires the computation of the local approximation error of the scalar function and the visibility of this error, which is determined by the transfer function. A natural strategy to compute the step length should be based on at least the second derivative of the scalar function and the pre-integrated emission and opacity of the transfer function. This approach, which we call *adaptive pre-integration*, automatically subsumes the well-known acceleration technique of space leaping[5].

Conceptually, the process of ray casting can be divided in three main tasks, which are the ray setup, the sampling of the ray, and the numerical integration of the samples. In order to exploit the massive parallelism of the graphics hardware, we compute the integration for all rays in parallel. For each step the front faces of the bounding box are rendered to process all visible rays in a front-to-back fashion (see Figure 5).

The ray position is determined by a floating point render target which contains the current ray parameter. For a given sampling distance l the ray parameter is updated accordingly

and written back into the floating point render target. Simultaneously, the integrated chromaticity and opacity are written to additional floating point render targets (we use two target for the RG and BG channels). A so-called importance volume is used to extract the maximum isotropic sampling distance as previously defined.

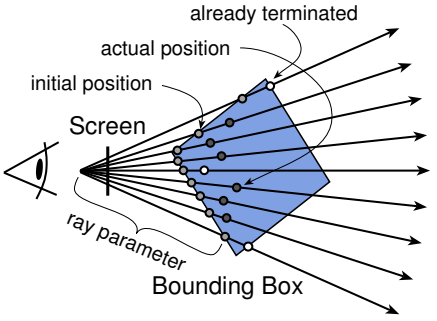


Figure 5: *Ray casting scheme: All viewing rays are processed simultaneously. For each integration step the pre-integration technique is used. After each step the colors are blended and the ray parameter corresponding to the next sampling position is written back into alternating render targets (ping-pong rendering).*

A ray is terminated if it either leaves the volume or if the integrated opacity is approximately one. If a ray is terminated we have to prevent further computation. For this purpose, we use the Z-buffer since it is tested before any pixel shader computation. This also includes texture reads. If the ray ends we set the Z-buffer such that the corresponding pixel is not processed anymore. This requires an additional rendering pass.

To find out whether all rays have been terminated we apply an asynchronous occlusion query. In total, we require $2n-1$ passes with n being the maximum number of samples for the rays in the generated image. The first pass is additionally setting up the initial ray parameters. In order to allow re-implementation of the ray caster the pixel shader 2.0 code is included in the Appendices 8.4, 8.5, and 8.6.

In contrast to the brute-force slicing approach the total number of samples depends on the transfer function and on the coherence of the volume data. Also considering early ray termination, the number of samples (see Figure 6) is always less than for the slicing approach, but the same reconstruction quality is achieved. In addition to this, all computations including blending are performed with full floating point precision, so that artifacts due to frame buffer quantization cannot occur (compare Figure 7).

5. Results

All performance measurements have been conducted on a PC equipped with an ATI Radeon 9700 graphics accelerator. Figure 8 shows a Bonsai[20] of size 256^3 with and without accurate volumetric clipping using 4-times over-sampling. Rendering times are approximately 2 and 1.5 seconds, respectively. The rendering times for the ray-casted bonsai with opaque and semi-transparent transfer functions are between 1 and 3 seconds per frame (Figures 6 and 9). The smaller NegHIP data set (Figure 7) with a size of 128^3 voxels took 2.2 seconds with a semi-transparent transfer function and 0.7 seconds with an opaque transfer function.

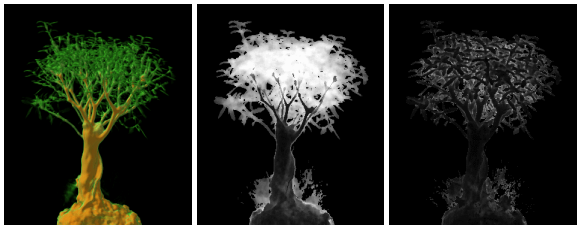


Figure 6: *Number of sampling steps for different transfer functions: On the left the original image is depicted and at the center the corresponding number of sampling steps is shown (White corresponds to 512 samples). On the right a more opaque transfer function was chosen to illustrate the impact of early ray termination.*

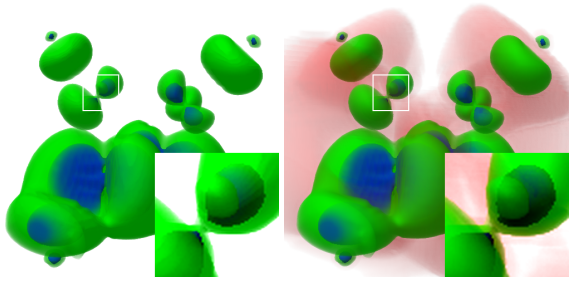


Figure 7: *Quality comparison between slicing with pre-integration and 4-times over-sampling (left) and ray casting with full floating point accuracy and adaptive pre-integration (right). On the left highly transparent areas are neglected due to 8 bit frame buffer quantization.*

In comparison to traditional slicing the quality of ray casting is always higher. Depending on the transfer function ray casting may even be faster. By increasing the error threshold, the rendering time can easily be reduced to 0.1 seconds per frame. The artifacts introduced here are much less visible than for the analogue case of reducing the number of slices.

With the upcoming of graphics accelerators such as the NVIDIA GeForce FX, we expect the performance of a hardware-accelerated ray caster to increase much more than the performance of regular multi-stepped slicing. The reasons are as follows: First, an additional pass is no longer

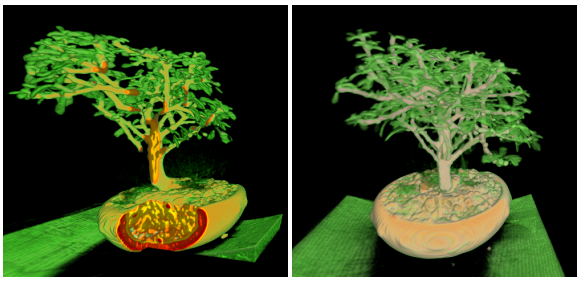


Figure 8: *Bonsai with and without volumetric clipping.*

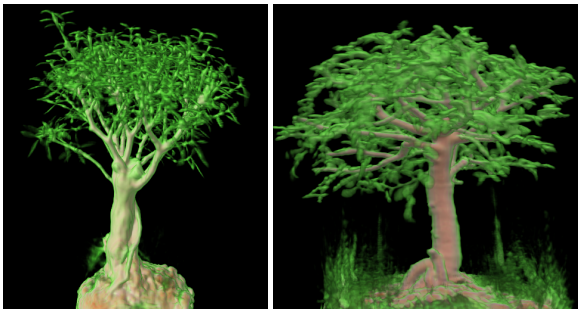


Figure 9: *Hardware-accelerated ray casting.*

needed for ray termination. Secondly, multiple steps can be performed at once because of a higher maximum instruction count. As buffer writes between the passes consume most of the rendering time, this overhead can be reduced dramatically with the NVIDIA GeForce FX.

The performance of a software ray caster[11] is approximately the same as for our hardware-accelerated approach. However, the software ray caster achieves its best performance only without pre-integration, so the resulting quality is not comparable.

6. Conclusion

We have combined volumetric clipping with the pre-integration technique to achieve high-quality volume visualizations. Since pre-integration does not completely solve the problem of the reconstruction of the ray integral, we presented a hardware-accelerated ray caster. By adaptive pre-integration of the viewing rays the reconstruction error can be guaranteed to be below the threshold contrast sensitivity[9]. The ray caster also applies pre-integration, space leaping and early ray termination. In the future we plan to speed up the ray caster by using hierarchical approaches[8] that will allow the visualization of very large data sets.

7. Acknowledgements

We would like to thank Mike Doggett of ATI for his valuable support, and especially for providing an ATI Radeon 9700 graphics accelerator. We would also like to thank Michael Wand for interesting discussions on the topic of importance volumes.

8. Appendix

8.1. Naive Volumetric Clipping:

```
ps_2_0                  // ps 2.0
dcl_volume s0           // 3D RGBA volume
dcl_2d s1               // 2D pre-integration table (RGBA)
dcl_volume s2           // 3D clip volume (Luminance)
dcl t0.xyz              // texcoord front
dcl t1.xyz              // texcoord back
dcl t2.xyz              // normalized eye vector
def c0,0.5,1.0,2.0,0.0  // constant definitions
texld   r0,t0,s0        // get front scalar value
texld   r1,t1,s0        // get back scalar value
texld   r4,t0,s2        // get front clip value
mov     r2.r,r0.a       // move front scalar value to r2.r
mov     r2.g,r1.a       // move back scalar value to r2.g
texld   r2,r2,s1        // pre-integration lookup
mul     r2.rgb,r2,r2.a  // post-multiply alpha
mad     r0.rgb,c0.b,r0,-c0.g // expand normal
dp3_sat r3,t2,r0        // calculate light intensity
mul     r2.rgb,r2,r3    // multiply emission with intensity
add     r4.r,r4.r,-c0.r // depending on comparison with 0.5..
cmp     r2.a,r4.r,r2.a,c0.a // ..set opacity to zero
mov     oC0,r2          // move to output register
```

8.2. Accurate Volumetric Clipping:

```
ps_2_0                  // ps 2.0
dcl_volume s0           // 3D volume
dcl_2d s1               // 2D pre-integration table
dcl_volume s2           // 3D clip volume
dcl_2d s3               // clip coefficient table (RGBA)
dcl t0.xyz              // texcoord front
dcl t1.xyz              // texcoord back
dcl t2.xyz              // normalized eye vector
def c0,0.5,1.0,2.0,0.0  // constant definitions
texld   r0,t0,s0        // get front scalar value
texld   r1,t1,s0        // get back scalar value
texld   r4,t0,s2        // get front clip value
texld   r5,t1,s2        // get back clip value
mov     r4.g,r5.r       // move back clip value to r4.g
texld   r4,r4,s3        // get clip coefficients
lrp     r2,r4,r1.a,r0.a // adjust scalar values
texld   r2,r2,s1        // pre-integration lookup
mul     r2.rgb,r2,r2.a  // post-multiply alpha
mad     r0.rgb,c0.b,r0,-c0.g // expand front normal
dp3_sat r3,t2,r0        // calculate front light intensity
mad     r1.rgb,c0.b,r1,-c0.g // expand back normal
dp3_sat r3.g,t2,r1      // calculate back light intensity
add_sat r3,c0.r,r3      // bias light intensities
lrp     r5.r,r4.a,r3.g,r3.r // adjust light intensities
mul     r2.rgb,r2,r5.r  // multiply emission with intensity
mul     r2,r2,r4.b      // attenuate emission and opacity
mov     oC0,r2          // move to output register
```

8.3. 4Steps@Once:

```
ps_2_0                  // ps 2.0
dcl_volume s0           // 3D volume
dcl_2d s1               // 2D pre-integration table
dcl t0.xyz              // texcoord front
dcl t1.xyz              // texcoord back
dcl t2.xyz              // normalized eye vector
dcl t3.xyz              // second texcoord front
dcl t4.xyz              // second texcoord back
dcl t5.xyz              // central texcoord
```

```
def c0,0.5,1.0,2.0,0.0   // constant definitions
texld   r0,t0,s0         // get front scalar value
texld   r1,t1,s0         // get back scalar value
texld   r3,t3,s0         // get second front scalar value
texld   r4,t4,s0         // get second back scalar value
texld   r6,t5,s0         // get central scalar value
mad     r0.rgb,c0.b,r0,-c0.g // expand front normal
dp3_sat r0.r,t2,r0       // calculate front light intensity
add_sat r0.r,c0.r,r0.r   // bias light intensity
mad     r3.rgb,c0.b,r3,-c0.g // expand second front normal
dp3_sat r3.r,t2,r3       // calculate front light intensity
add_sat r3.r,c0.r,r3.r   // bias light intensity
mad     r6.rgb,c0.b,r6,-c0.g // expand central normal
dp3_sat r6.r,t2,r6       // calculate front light intensity
add_sat r6.r,c0.r,r6.r   // bias light intensity
mad     r4.rgb,c0.b,r4,-c0.g // expand second back normal
dp3_sat r4.r,t2,r4       // calculate front light intensity
add_sat r4.r,c0.r,r4.r   // bias light intensity
mov     r2.r,r4.a        // move second back scalar value
mov     r2.g,r1.a        // move back scalar value
texld   r2,r2,s1         // pre-integration lookup
mul     r2.rgb,r2,r2.a   // post-multiply alpha
add     r2.a,c0.g,-r2.a  // compute transparency
mul     r2.rgb,r2,r4.r   // multiply emission with intensity
mov     r5.r,r6.a        // move central scalar value
mov     r5.g,r4.a        // move second back scalar value
texld   r5,r5,s1         // pre-integration lookup
mul     r5.rgb,r5,r5.a   // post-multiply alpha
add     r5.a,c0.g,-r5.a  // compute transparency
mul     r5.rgb,r5,r6.r   // multiply emission with intensity
mad     r2.rgb,r2,r5.a,r5 // blend operation (1st)
mul     r2.a,r2.a,r5.a   // accumulate alpha
mov     r5.r,r3.a        // move second front value
mov     r5.g,r6.a        // move central scalar value
texld   r5,r5,s1         // pre-integration lookup
mul     r5.rgb,r5,r5.a   // post-multiply alpha
add     r5.a,c0.g,-r5.a  // compute transparency
mul     r5.rgb,r5,r3.r   // multiply emission with intensity
mad     r2.rgb,r2,r5.a,r5 // blend operation (2nd)
mul     r2.a,r2.a,r5.a   // accumulate alpha
mov     r5.r,r0.a        // move front scalar value
mov     r5.g,r3.a        // move second front scalar value
texld   r5,r5,s1         // pre-integration lookup
mul     r5.rgb,r5,r5.a   // post-multiply alpha
add     r5.a,c0.g,-r5.a  // compute transparency
mul     r5.rgb,r5,r0.r   // multiply emission with intensity
mad     r2.rgb,r2,r5.a,r5 // blend operation (3rd)
mul     r2.a,r2.a,r5.a   // accumulate alpha
add     r2.a,c0.g,-r2.a  // compute opacity
mov     oC0,r2           // move to output register
```

8.4. Ray Casting: Ray Setup and First Integration Step

```
ps_2_0                   // ps 2.0
def c0,0.0,2.0,63.75,1.0 // constant definitions
def c1,1.0,0.0,0.333,0.5 // (c2,c3) is the bounding box
dcl t0.xyz               // first intersection point with volume
dcl t1.xyz               // ray direction (not normalized)
dcl t2.xyz               // scaling of texture coordinates
dcl t3.xyz               // direction to light source
dcl_volume s0            // 3D volume
dcl_volume s1            // 3D table holding sampling distances
dcl_volume s2            // 3D pre-integration table
texld   r2,t0,s0         // sample volume at first intersection
texld   r5,t0,s1         // get distance to next sampling point
nrm     r0.xyz,t1        // normalize ray direction..
mul     r0.xyz,r0,t2     // ..and multiply with scaling factors
add     r4.xyz,c2,-t0    // calculate signed distances..
add     r3.xyz,c3,-t0    // ..to the bounding box
rcp     r6.x,r0.x        // calculate reciprocal..
rcp     r6.y,r0.y        // ..of each component..
rcp     r6.z,r0.z        // ..of ray direction
mul     r3.xyz,r3,r6     // calculate distance to bounding box..
mul     r4.xyz,r4,r6     // ..in ray direction
max     r6.xyz,r3,r4     // get closest..
min     r4.x,r6.x,r6.y   // ..non-negative..
min     r1.y,r4.x,r6.z   // ..intersection point
mov     r1.xzw,c0.x      // store intersection 4 ray termination
mad     r1.xz,r5.x,c1,r1.x // compute new sampling position
min     r1.x,r1.x,r1.y   // clamp to volume boundary
mad     r3.xyz,r1.x,r0,t0 // calculate second sampling point..
texld   r4,r3,s0         // ..and sample volume at this position
mad     r2.xyz,r2,c0.y,-c0.w // extract gradient
nrm     r8.xyz,r2        // normalize gradient
add     r6.w,r1.x,-r1.z  // get interval length of current step
lrp     r6.xy,c1,r2.w,r4.w // setup texcoords 4 pre-integration..
mad     r6.z,r6.w,c1.z,-c1.z // ..including ray segment length l
texld   r7,r6,s2         // pre-integration lookup
dp3_sat r8.w,r8,t3       // calculate light intensity
mul     r6.x,c0.z,r6.w   // change opacity to correctly..
add     r7.a,c0.w,-r7.a  // ..represent the ray segment length..
pow     r10.a,r7.a,r6.x  // ..which includes raising it..
add     r7.a,c0.w,-r10.a // ..to the power of l
mul     r7.rgb,c1.w,r7   // bias color 4 lighting
mad     r7.rgb,r8.w,r7,r7 // multiply emission with intensity
mul     r7.rgb,r7,r7.a   // post-multiply alpha
mov     r0,r7.b          // split color 4 floating point output..
mov     r0.g,r7.a        // ..into two render targets
mov     oC0,r7           // output color (RG)
mov     oC1,r0           // output color (BA)
mov     oC2,r1           // output ray parameter
```

8.5. Ray Casting: Ray Termination

```
ps_2_0                   // ps 2.0
def c0,-1.0,-1.0,0.0,0.0 // constant definitions
def c1,0.996,0.0,0.0,0.0 // threshold equals 1/256
dcl t4.xyzw              // position of pixel in screen space
dcl_2d s3                // previous ping-pong buffers holding..
dcl_2d s4                // ..floating point colors
dcl_2d s5                // buffer containing sample positions
texldp  r1,t4,s5         // get sampling position..
mov     r1.zw,c0.z       // ..and clear unused part
texldp  r8,t4,s4         // get opacity
add     r0,r1.x,-r1.y    // calc distance to volume boundary
cmp     r1,r0,c0,r1      // check if distance equals 0
add     r0,c1.x,-r8.g    // compare opacity against threshold..
cmp     r1,r0,r1,c0      // ..4 early ray termination
mov     r0,-r1           // leave shader if the ray..
texkill r0               // ..is not being terminated
texldp  r7,t4,s3         // get remaining parts of color
mov     oC0,r7           // output floating point color into..
mov     oC1,r8           // ..alternate ping-pong buffers
```

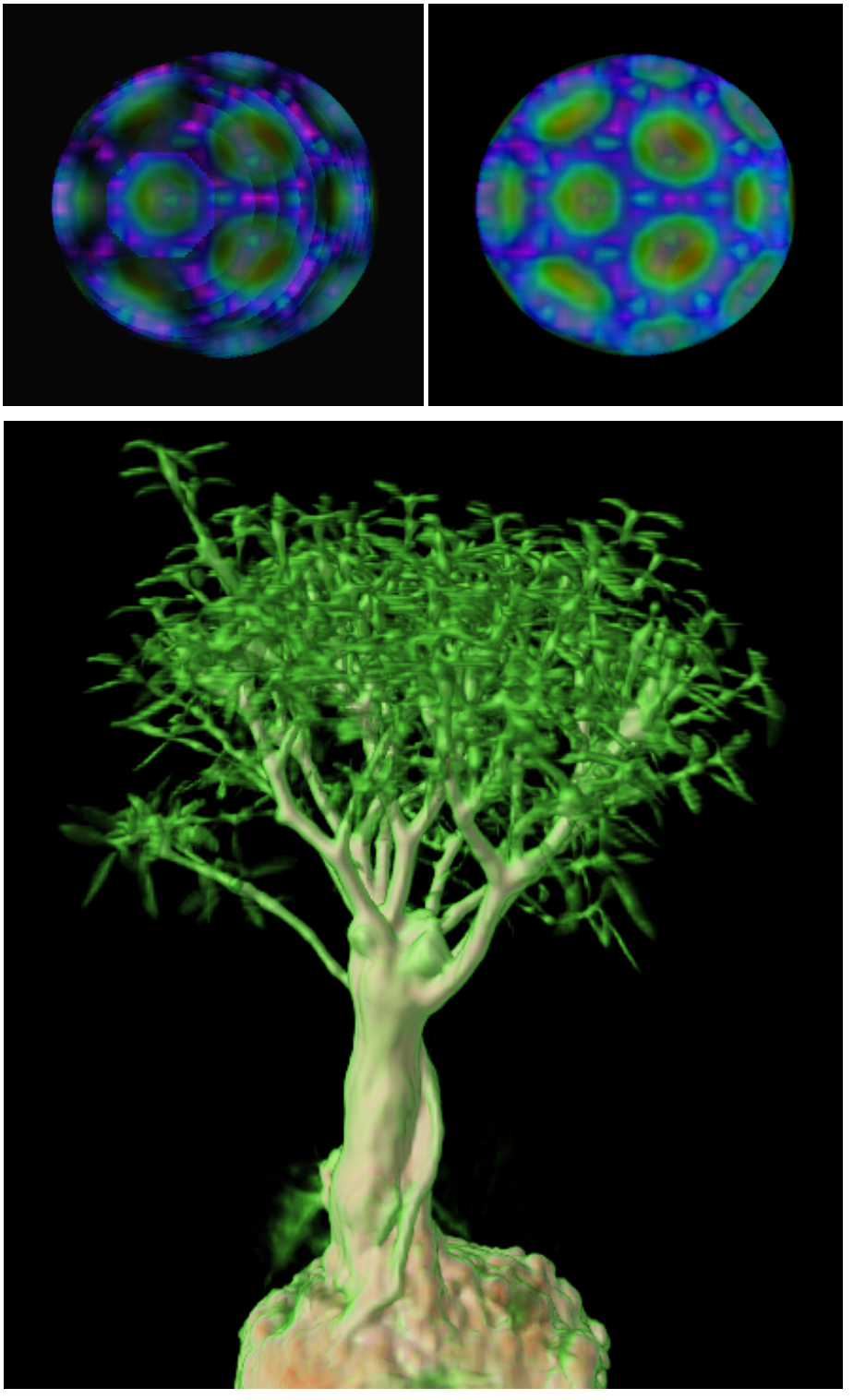
8.6. Ray Casting: One Additional Integration Step

```
ps_2_0                   // ps 2.0
def c0,0.0,2.0,63.75,1.0 // constant definitions
def c1,1.0,0.0,0.333,0.5 // (c2,c3) is the bounding box
dcl t0.xyz               // first intersection point with volume
dcl t1.xyz               // ray direction (not normalized)
dcl t2.xyz               // scaling of texture coordinates
dcl t3.xyz               // direction to light source
dcl t4.xyzw              // position of pixel in screen space
dcl_volume s0            // 3D volume
dcl_volume s1            // 3D table holding sampling distances
dcl_volume s2            // 3D pre-integration table
dcl_2d s3                // previous ping-pong buffers holding..
dcl_2d s4                // ..floating point colors
dcl_2d s5                // buffer containing sample positions
texldp  r1,t4,s5         // get sampling position
texldp  r7,t4,s3         // get color from two..
texldp  r8,t4,s4         // ..floating point render targets
mov     r7.z,r8.x        // combine the color channels of..
mov     r7.w,r8.y        // ..the two render targets
nrm     r0.xyz,t1        // normalize ray direction..
mul     r0.xyz,r0,t2     // ..and multiply with scaling factors
mad     r3.xyz,r1.x,r0,t0 // calculate first sampling point
texld   r2,r3,s0         // sample volume at first sampling point
texld   r5,r3,s1         // get distance to next sampling point
mad     r1.xz,r5.x,c1,r1.x // compute new sampling position
min     r1.x,r1.x,r1.y   // clamp to volume boundary
mad     r3.xyz,r1.x,r0,t0 // calculate second sampling point..
texld   r4,r3,s0         // ..and sample volume at this position
mad     r2.xyz,r2,c0.y,-c0.w // extract gradient
nrm     r8.xyz,r2        // normalize gradient
add     r6.w,r1.x,-r1.z  // get interval length of current step
```

```
lrp     r6.xy,c1,r2.w,r4.w // setup texcoords 4 pre-integration..
mad     r6.z,r6.w,c1.z,-c1.z // ..including ray segment length l
texld   r9,r6,s2         // pre-integration lookup
dp3_sat r8.w,r8,t3       // calculate light intensity
mul     r6.x,c0.z,r6.w   // change opacity to correctly..
add     r9.a,c0.w,-r9.a  // ..represent the ray segment length..
pow     r10.a,r9.a,r6.x  // ..which includes raising it..
add     r9.a,c0.w,-r10.a // ..to the power of l
mul     r9.rgb,c1.w,r9   // bias color 4 lighting
mad     r9.rgb,r8.w,r9,r9 // multiply emission with intensity
mul     r9.rgb,r9,r9.a   // post-multiply alpha
add     r8.a,c0.w,-r7.a  // blend new color..
mad     r7,r9,r8.a,r7    // ..with the old one
mov     r0,r7.b          // split color 4 floating point output..
mov     r0.g,r7.a        // ..into two render targets
mov     oC0,r7           // output color (RG)
mov     oC1,r0           // output color (BA)
mov     oC2,r1           // output ray position
```

References

1. Kurt Akeley. RealityEngine Graphics. *Computer Graphics (SIGGRAPH '93 Proceedings)*, 27:109–116, 1993.
2. ATI. Programmability Features of Graphics Hardware. *SIGGRAPH '02 Course Notes*, 2002.
3. Brian Cabral, Nancy Cam, and Jim Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proc. Symposium on Volume Visualization '94*, pages 91–98. ACM SIGGRAPH, 1994.
4. John Danskin and Pat Hanrahan. Fast Algorithms for Volume Ray Tracing. In *Proc. Symposium on Volume Visualization '92*, pages 91–98. ACM, 1992.
5. R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. *Computer Graphics*, 22(4):65–74, 1988.
6. K. Engel, M. Kraus, and Th. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Eurographics Workshop on Graphics Hardware '01*, pages 9–16. ACM SIGGRAPH, 2001.
7. S. Guthe, S. Roettger, A. Schieber, W. Strasser, and Th. Ertl. High-Quality Unstructured Volume Rendering on the PC Platform. In *Proc. EG/SIGGRAPH Graphics Hardware Workshop '02*, pages 119–125, 2002.
8. S. Guthe, M. Wand, J. Gonser, and W. Strasser. Interactive Rendering of Large Volume Data Sets. In *Proc. Visualization '02*, pages 53–60. IEEE Computer Society Press, 2002.
9. R.F. Hess and E.R. Howell. The Threshold Contrast Sensitivity Function in Strabismic Amblyopia. *Vision Research*, 17:1049–1055, 1977.
10. James T. Kajiya. Ray Tracing Volume Densities. In *Proc. SIGGRAPH '84*, pages 165–174. ACM, 1984.
11. Günther Knittel. The ULTRAVIS System. In *Proc. Visualization Symposium '00*, pages 71–79, 2000.
12. N. L. Max, P. Hanrahan, and R. Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24(5):27–33, 1990.
13. Nelson L. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.
14. M. Meissner, S. Guthe, and W. Strasser. Interactive Lighting Models and Pre-Integration for Volume Rendering on PC Graphics Accelerators. In *Proc. Graphics Interface '02*, pages 209–218, 2002.
15. Microsoft. Next Generation 3-D Gaming Technology Using DirectX 9.0. *Proc. Game Developers Conference '02*, 2002.
16. NVIDIA. CineFX Architecture. *Proc. SIGGRAPH '02*, 2002.
17. T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002.
18. S. Roettger and Th. Ertl. A Two-Step Approach for Interactive Pre-Integrated Volume Rendering of Unstructur ed Grids. In *Proc. IEEE Symposium on Volume Visualization '02*, pages 23–28. ACM Press, 2002.
19. S. Roettger, M. Kraus, and Th. Ertl. Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection. In *Proc. Visualization '00*, pages 109–116. IEEE, 2000.
20. S. Roettger and B. Thomandl. Three little Bonsais that did survive CT scanning and contrast agent but not the graphix lab conditions. *http://wwwvis.informatik.uni-stuttgart.de/~roettger/data/Bonsai*, 2000.
21. U. Tiede, T. Schiemann, and K. H. Hohne. High-Quality Rendering of Attributed Volume Data. In *Proc. Visualization '98*, pages 255–262. IEEE, 1998.
22. Allen Van Gelder and Kwansik Kim. Direct Volume Rendering with Shading via Three-Dimensional Textures. In *Proc. IEEE Symposium on Volume Visualization '96*, pages 23–30, 1996.
23. D. Weiskopf, K. Engel, and Th. Ertl. Volume Clipping via Per-Fragment Operations in Texture-Based Volume Visualization. In *Proc. Visualization '02*, pages 93–100, 2002.
24. R. Westermann and Th. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Computer Graphics*, Annual Conference Series, pages 169–177. ACM, 1998.
25. P. L. Williams and N. L. Max. A Volume Density Optical Model. In *Computer Graphics (Workshop on Volume Visualization '92)*, pages 61–68. ACM, 1992.



Top left: Naive volume clipping approach (rendered Bucky Ball shows slicing artifacts). **Top right**: Accurate volume clipping method. **Bottom**: Hardware-accelerated ray casting (Bonsai rendered with ATI Radeon 9700).