Towards Supporting Volumetric Data in FurryBall GPU Renderer

Michal Benátský*
Supervised by: Jiří Bittner†

Department of Computer Graphics and Interaction
Faculty of Electrical Engineering
Czech Technical University in Prague

Abstract

This paper describes an implementation of volumetric rendering for FurryBall gpu renderer. Since FurryBall is directly integrated into Autodesk Maya, our volumetric renderer supports all types of fluids, which can be simulated in Maya. We discuss the issues of integrating volumetric rendering into the FurryBall renderer. We show that our implementation of volumetric rendering is up to 3 orders of magnitude faster than Mental Ray on the tested scenes.

Keywords: volumetric rendering, fluids, modeling tools

1 Introduction

Fast rendering preview integrated directly into 3D modeling tools, would significantly enhance the artists comfort and productivity. Most 3D modellers however do not provide high quality realtime feedback and rendering of scenes can take hours even days.

GPU based multi pass rasterization can be used to generate fast realistic previews and with some limitations it can also be used to generate final high quality results. One of such renderers is FurryBall [10], which is a GPU based rasterization renderer, directly integrated into Autodesk Maya.

In this paper we show how to extend the renderer for dealing with fluids. In particular we discuss how to integrate our volumetric renderer with the rasterization of transparent objects, shadows, hair systems, and how to deal with volumetric grid containers intersections. We present results obtained on several test scenes representing simulated fluids and show that these results are consistent with Mental Ray while obtaining a significant speedups.

2 Related Work

Rendering volumetric data is well studied area of computer graphics. For a comprehensive overview of realtime volumetric rendering techniques please refer to Hadwiger et al. [3].

*benatmic@fel.cvut.cz
†bittner@fel.cvut.cz

Figure 1: Church model render in FurryBall, model courtesy of Art and Animation studio

Pixar's RenderMan based on Reyes architecture [1] is known as industry's standard and is very widely used. Render Ants [13] shows that it is possible to move all stages of Reyes to the gpu. V-Ray is one of the leading renderers in the field of GPGPU raytracing and it is fully integrated into Autodesk Maya and 3ds Max. Another well known render is iray from Mental Images, creators of Mental Ray raytracer.

Apart from FurryBall we are aware of only a few commercially available renderers based on multi pass rasterization. The most similar is pixar's lpics [8] for realtime preview of lighting on 3D scene. Another related renderer is Mach Studio, which is a realtime GPU rasterization renderer. However these renderers aren't fully integrated into 3D modeling software such as Maya or 3Ds Max.

The rest of the paper is organised as follows: The basics of volumetric rendering and our implementation of rendering volumetric data will be described in the next section. In section 4 we will describe the integration of volume rendering into existing renderer based on multi pass rasterization on the gpu and the integration with Autodesk Maya fluid rendering as well. In section 5 we will present results of our work.

3 Volumetric Rendering

In this section we first describe the theoretical background of volume rendering and then outline our implementation.

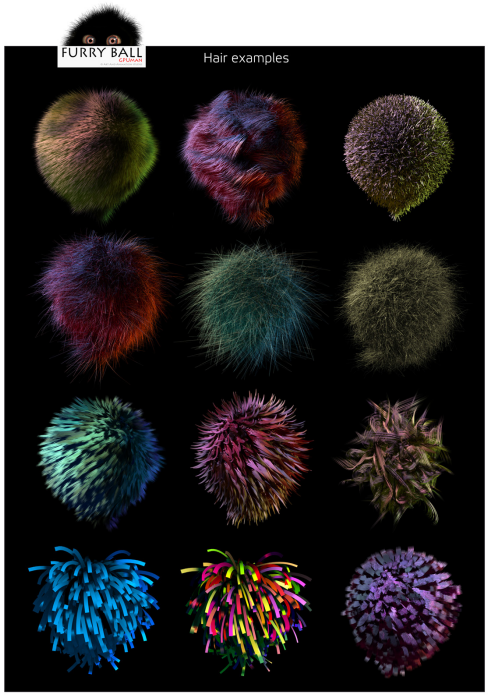

Figure 2: Fur rendering with FurryBall, model courtesy of Art and Animation studio

3.1 Theoretical background

The physical basis for volume rendering relies on geometric optics, in which is light assumed to travel along straight line unless interaction with participating media takes place [3]. The following types of interaction are typically taken into account.

Figure 3: Ray in volume with emission, absorption and scattering.

Emission Volume emits light and increasing the radiative energy. Practical example could be fire, which emits light by converting heat into light.

Absorption Volume absorbs light by converting radiative energy into heat.

Scattering Light can be scattered by volume, changing the direction of light propagation.

For solving complex light transport problem are commonly used simplified models. In "Absorption only" model volume absorb incident light. No light is emitted or scattered. "Emission only" model presents volume which is completely transparent, but can emit light. "Emission-Absorption" model is most common in volume rendering. Volume emits light and absorb incident light. "Single Scattering" model counts with single scattering from light that comes from external light source (not from the volume). "Multiple Scattering" model has goal to evaluate the complete illumination model for volumes.

Autodesk Maya uses "Emission-Absorption" model. Light scattering is approximated by shadow diffusion. Our renderer is implemented into Maya, so we use Emission-Absorption as well.

Emission-Absorption model can be described by Volume-Rendering Integral, which integrates radiance along the direction of light flow from the starting point $s = s_0$ to the endpoint $s = D$.

$$I(D) = I_0 e^{-\int_{s_0}^{D} \kappa(t)dt} + \int_{s_0}^{s} q(s) e^{-\int_{s_0}^{D} \kappa(t)dt} ds \qquad (1)$$

The term I_0 represents the light from the background. $I(D)$ is the radiance leaving the volume at $s = D$. κ is the *absorption coefficient* and q is *emission*. First term of the equation 1 describes the light from the background attenuated by volume. Second term represents the integral contribution of the source terms attenuated by the volume along the remaining distance to the camera.

$$\tau(s_1, s_2) = \int_{s_2}^{s_1} \kappa(s) ds \qquad (2)$$

τ defines *optical depth* between positions s_1 and s_2, which defines how long can light travel before it is absorbed. Smaller values defines material which is near to be transparent and higher values defines nearly opaque material. Transparency for a material between s_1 and s_2 is:

$$T(s_1, s_2) = e^{-\tau(s_1, s_2)} = e^{-\int_{s_1}^{s_2} \kappa(t)dt} \qquad (3)$$

Once we define transparency, we can rewrite volume rendering integral into:

$$I(D) = I_0 T(s_0, D) + \int_{s_0}^{s} q(s) T(s_0, D) ds \qquad (4)$$

Volume rendering integral cannot be solved analytically, that's why we use discretisation. Common approach is to split integration domain into n intervals: $s_0 < s_1 < ... < s_{n-1} < s_n$. Transparency and color contribution of the ith interval is:

$$T_i = T(s_{i-1}, s_i), c_i = \int_{s_{i-1}}^{s_i} q(s) T(s, s_i) ds \qquad (5)$$

The radiance at the exit point is:

$$I(D) = I(s_n) = I(s_{n-1})T_n + c_n = (I(s_{n-2})T_{n-1} + c_{n-1})T_n + c_n = \ldots$$

which can be rewritten as

$$I(D) = \sum_{i=0}^{n} c_i \prod_{j=i+1}^{n} T_j \tag{6}$$

with $c_0 = I(s_0)$. Which leads to recursive front to back equations

$$C_i = C_{i+1} + T_{i+1}C_i$$
$$T_i = T_{i+1}(1-\alpha_i)$$

and back to front equation:

$$C_i = C_{i-1}(1-\alpha_i) + C_i$$
$$T_i = T_{i-1}(1-\alpha_i)$$

α is opacity and $\alpha = (1-T)$.

Volume raycasting Volume raycasting [5] traces ray from the camera into the volume and solves volume rendering integral along these rays. The biggest advantage of volume raycasting is that rays are completely independent on each other and can be processed in parallel.

3.2 Implementation

We implemented two techniques for rendering the volumes. The first method is very simple and visualises the volume data as sprites (see Figure 4) and the second uses volume raycasting.

Particle rendering - splatting We represent every voxel of the volume as a vertex that is processed by a geometry shader to create a billboard. When the billboard is rasterized, the pixel shader samples the color and value ramps (transfer functions) to get appropriate color and opacity for every pixel. This part was not expected to produce high quality results, it was implemented only for the verification of received data from Maya and especially for the fast previews.

Volume raycasting Our second method uses volume raycasting written in HLSL on the pixel shader. It was first design decision to use direct compute, which is GPGPU part of the DirectX 11 API, but the nature of volume raycasting is allowing to implement it on the pixel shader, because of no need of synchronisations.

The rays are created based on the camera parameters. We support variable focal points and off-axis stereo camera. Ray origin and direction are converted into texture space using inverted world matrix. Converting to texture space allows faster raymarching and also it allows to perform a fast AABB test, because fluid container is a unit cube in texture space. This tells us minimum and maximum distance where to sample on the ray.

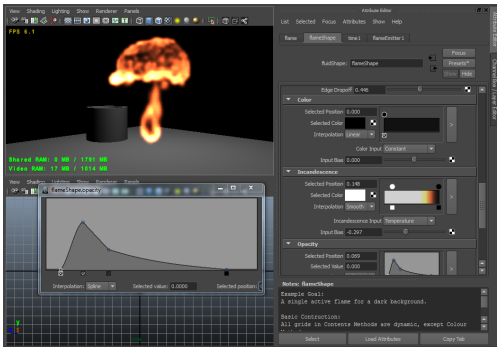

Figure 4: Fluid rendered using splatting in FurryBall

The sampling has a fixed step, but depth jitter can be used. Sampling is performed in parallel in the world space and in the texture space. The world space position is important for connecting to existing and optimized FurryBall shaders, especially for getting attenuation, light color (which can be defined as a texture) and shadow.

Note that our current implementation of volumetric ray casting fully replaced the splatting based approach as it can render a more accurate preview even faster (when using under sampling).

4 Integration with FurryBall and Autodesk Maya

FurryBall is a GPU based rasterization plug-in for Autodesk Maya [10]. FurryBall's initial purpose was a fast preview for artists, before raytracing a CG movie. As its development continued, the rasterization possibilities were able to suit most of the artist's needs and therefore it was extended from a simple previewer into a complete production renderer. FurryBall is written in C++, using DirectX 11 API and Open Maya API. It uses also Python and Maya Embedded Language (MEL). See Figure 1 and 2 for examples of images rendered with FurryBall.

4.1 Integration with FurryBall

Camera settings FurryBall supports all Autodesk Maya's camera settings, which means that implemented volume raycaster must support all this settings too and build rays correctly. We solved this problem by reconstructing near and far plane in world space using viewport, view and projection matrix and passing them to the shader. Then we interpolate between corner values and find correct ray origin and direction.

Opaque objects FurryBall allows to render all polygonal meshes and apply various textures such as color textures, bump textures and displacements textures. It also supports Maya layered textures and realtime adaptive DirectX 11 HW tessellation with displacement textures.

Volume raycasting can be combined with opaque geometry easily by stopping sampling at z-buffer value. Problematic part if this approach is when the size of the final render is not equal with the size of fluid render pass. This is supported because gaseous phenomena basically doesn't have hard edges and can be rendered in lower resolution. It is possible to have fluid render pass at 1/2 resolution of final render without visible quality loss. However this produces artifacts if geometry is inside volume, if no lower resolutions are possible to use even for final rendering.

Transparent objects Transparent objects in FurryBall are rendered with depth peeling [2] , with the possibility to set up the number of layers.

Integration of volumetric objects with transparent geometry is problematic, especially when geometry intersects fluid container, there is need to blend objects into the fluid in correct sample.

Two approaches were considered. First use customised depth peeling which take into account fluid rendering. That would solve the problem of sorting and also would solve problem of transparent object and fluid intersection by splitting fluid sampling into more parts.

Another considered approach was using order independent transparency presented by AMD at GDC 2010 [7], which uses unordered access writes into the output memory and outputs linked list for every rendered fragments containing informations about depth and color. This linked list is then sorted per fragment on the compute shader, blended with each other in correct order and then with the final render. This should be combined with fluid rendering by passing ordered lists to the fluid rendering pass.

Shadows for meshes FurryBall integrates various shadow mapping techniques. Basic shadow mapping techniques, including basic shadow mapping for spot and point lights and cascade shadow maps for directional lights [6]. [11]. For computing simplified soft shadows it uses PCF or more advanced PCSS [6], which can either use regular, Perlin or Poisson disk sampling.

It is not hard to combine our volumetric rendering with shadow mapping. Ray sampling in raycaster knows world position of the sample, so it can ask shadow map for shadows.

Reflections and planar mirrors General reflections and refractions and computed using environment mapping. Planar mirror reflections use rendering into a texture from the reflected camera position.

Both of these methods are not problematic to combine with described volume raycasting as rays can be created from the viewport and camera matrices.

Hair systems FurryBall has its own hair rendering system. It is based on constructing billboards or regular geometry along curves on geometry shader. This might be slower than a solution with lines, but it offers much more control and possibilities. Such as vegetation rendering using textured fur. Curves can be fully independent of Maya or can be connected to Maya hair system and benefit from Maya hair simulation.

Fluid integration with the hair rendering is the same as with regular geometry and rays stops at z-buffer value.

Shadow for hair and fur Shadows for hair can be computed by common shadow mapping techniques, which, however, do not provide sufficient quality. Tiny objects like hair appear much better when being shadowed with a transition function. Therefore FurryBall implements Deep Opacity Maps [12] and Fourier Opacity maps [4] for these cases.

Self shadowing and casting shadows for fluids is using Fourier Opacity maps, which allows us to save multiple hair systems and fluid containers into one map and cast shadows one to another and on the geometry.

Fluid containers intersections Fluid container intersections is common case in Autodesk Maya rendering. It is used for create sky, where one container simulates blue atmosphere and second contains clouds. Problem of fluid intersections is that samples has to be blended together correctly.

More approaches was considered. First one was very similar to depth peeling, where volume sampling is done in layers and layers are blended together. This wasn't accepted as too slow. We decided that FurryBall will sample two fluids together in one pass, if they intersects each other. This solution is faster and more accurate than making slices.

4.2 Integration with autodesk Maya

Maya offers very complex fluid solver, which is based on Navier-Strokes equations. The solver can handle both 2D and 3D grids and simulates density, temperature, speed, fuel and pressure. It also supports user defined gradients and constants in the simulation.

For fluid shading the most important are the *ramps* for colors and other values. Ramps define transfer functions by piece-wise linear functions. The user can connect one of the functions (like density or temperature) with one of the shading ramps (color, incandescence and opacity). It is also possible to connect gradients. Every ramp also has a specified bias. Color ramp defines the RGB transfer function which is affected by each light. Incandescence is also color transfer function which defines emitted light from

the volume. And at last is there opacity ramp which defines opacity transfer function.

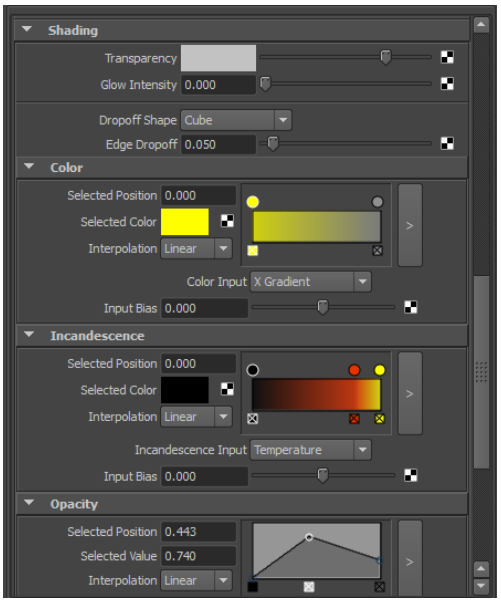

Figure 5: Fluid shading setup in Autodesk Maya

The final appearance depends on several additional settings. The user can define RGB transparency, which affects final opacity and color and also can choose some of the dropoff functions. User also define the number of samples per voxel and choose whether the fluid in container will receive/cast shadows. Maya light linking is also available (linking lights with scene objects to determine which one is lighted with which light). User chooses volume resolution and size, which affects final density, regular scale doesn't.

Fluid can be affected by procedural noises. Perlin [9], Billow, Volume wave and Wispy noises are employed. Lots of setting is possible. Coordinates can be fixed with volume grid or can flow fluid simulation by using velocity vectors. In FurryBall fluid renderer is currently Perlin and Billow noise implemented.

Fluids can receive and cast shadows. Self shadowing of fluid can be turned off for better performance of rendering. For better performance is also possible set fluid to do not interact with all lights in scene (or linked lights) but only one defined via fluid node gui.

Fluid node also enables to choose number of sample per voxel and interpolation method (linear or Smooth - cubic).

Our fluid renderer reads all ramps through the Maya API, samples them into a 2D texture. The attached functions such as density or temperature are loaded into 3D textures. The gradients are not loaded as they are computed directly in a shader.

	FurryBall	Mental Ray	speed up
no shadows	11ms	12000ms	1009 ×
shadows	36ms	18000ms	534 ×
shadows and noise	96ms	26000ms	279 ×
fire	96ms	6000ms	62 ×

Table 1: Rendering performance of FurryBall compared to Mental Ray. For rendered images see Figure 6. Image resolution was 800x600px. Scene with percolator contained 12000 triangles and fluid container with resolution $10\times10\times10$. And scene with fire contained 300 triangles and fluid container with $35\times35\times35$ resolution.

5 Results

FurryBall is able to render all of the maya fluids and provide same or very similar results as built-in renderers. In this section we will present several images rendered with FurryBall with their render times. Table 1 shows the rendering performance of FurryBall compared to Mental Ray, which we used as reference, on 4 different scenes. Figure 6 shows scenes referred in table 1. Figure 7 shows clouds created using perlin noise and Figure 8 shows scene with homogenous volume and two spot lights.

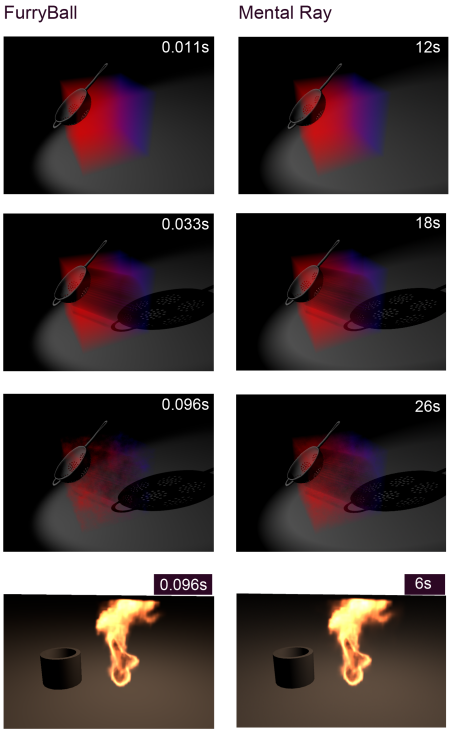

Figure 6: Images with render times. Left column: rendered with FurryBall. Right column: rendered with Mental Ray

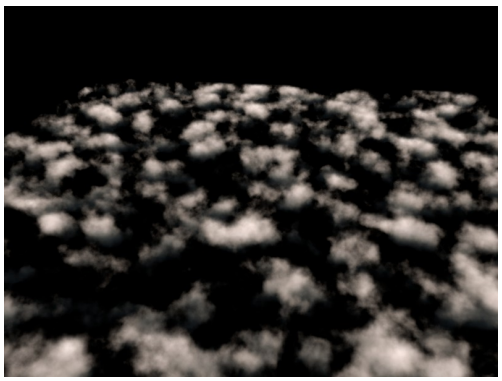

Figure 7: Procedural clouds rendered in FurryBall

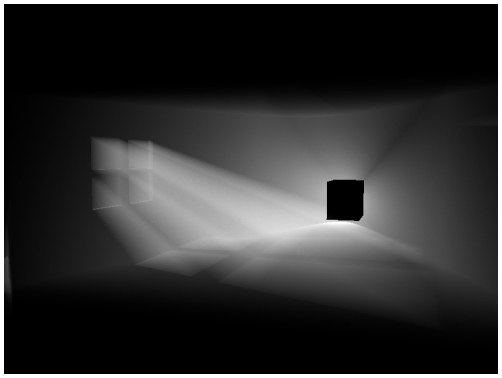

Figure 8: Volume with constant density, two lights are in scene

6 Conclusion

FurryBall became an interesting alternative to traditional renderers. It can be used for previsualization and realtime feedback before raytracing, but also as a final renderer, which can still provide results 50 - 300 times faster than traditional renderers. It is current being used for production of a new feature movie. Currently the fluid rendering plugin for FurryBall supports ray marching with emission and absorption and without restricting the user it produces the same or very similar results as reference built-in renderers.

References

[1] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, 21:95–102, August 1987.

[2] Cass Everitt. Interactive order-independent transparency, 2001.

[3] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[4] Jon Jansen and Louis Bavoil. Fourier opacity mapping. In Daniel G. Aliaga, Manuel M. Oliveira, Amitabh Varshney, and Chris Wyman, editors, *SI3D*, pages 165–172. ACM, 2010.

[5] Marc Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8:29–37, May 1988.

[6] Mahdi Mohammadbagher, Jan Kautz, Nicolas Holzschuch, and Cyril Soler. Screen-space percentage-closer soft shadows. 2010.

[7] Holger Grn Nick Thibieroz. Oit and gi using dx11 linked lists. AMD, 2010.

[8] Fabio Pellacini, Kiril Vidimce, Aaron E. Lefohn, Alex Mohr, Mark Leone, and John Warren. Lpics: a hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Trans. Graph*, 24(3):464–470, 2005.

[9] Ken Perlin. Improving noise. *ACM Trans. Graph.*, 21:681–682, July 2002.

[10] FurryBall renderer. http://furryball.aaa-studio.eu. Art and Animation studio, 2010.

[11] Erik Sintorn, Elmar Eisemann, and Ulf Assarsson. Sample-based visibility for soft shadows using alias-free shadow maps. December 10 2008.

[12] Cem Yuksel and John Keyser. Deep opacity maps. *Comput. Graph. Forum*, 27(2):675–680, 2008.

[13] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. Renderants: interactive reyes rendering on gpus. Number 5, pages 155:1–11, New York, NY, USA, 2009. ACM. SIGGRAPH Asia 2009.