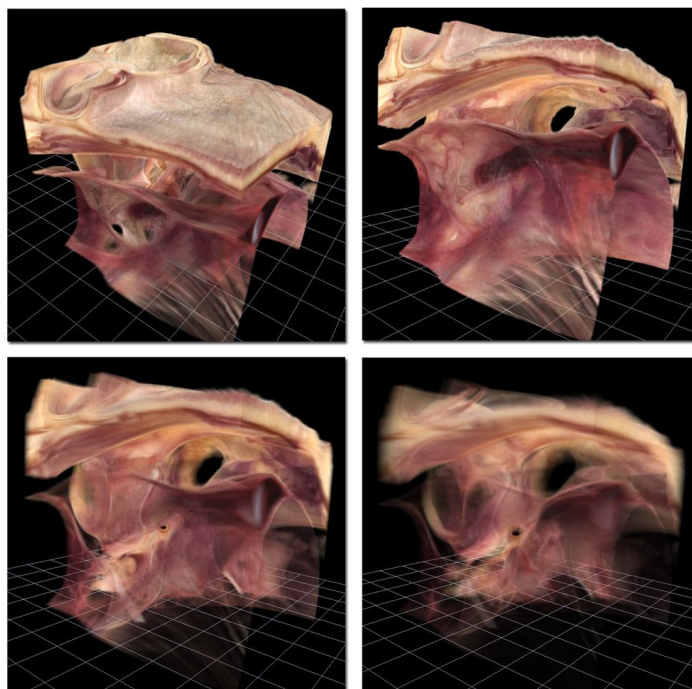


# Peter Triers Blog

---

## GPU raycasting tutorial

### GPU raycasting

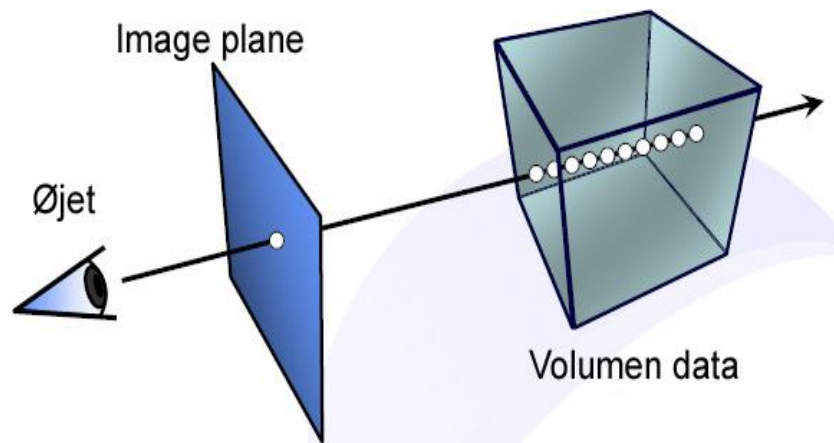


This page will try to explain how to implement a GPU based raycasting render, using open GL and Nvidia's CG. This tutorial assumes some experience with OpenGL and vertex-fragment shaders.

First of all why do we need this algorithm? Because it is a smart way to achieve high quality volume rendering and the raycasting algorithm is well suited for the modern GPU's. Especially the new 8800 series because of the unified shading system.

The reason behind this tutorial is to help people getting started with GPU raycasting because there is some technical difficulties that has to be addressed in order to render volumetric data like in the picture above.

The main core of the algorithm is to send one ray per screen pixel and trace this ray through the volume. This is possible to implement in a fragment program and the rendering can be done in realtime. The technique is pretty flexible for instance effect like shadows can be implemented with a few lines of code.

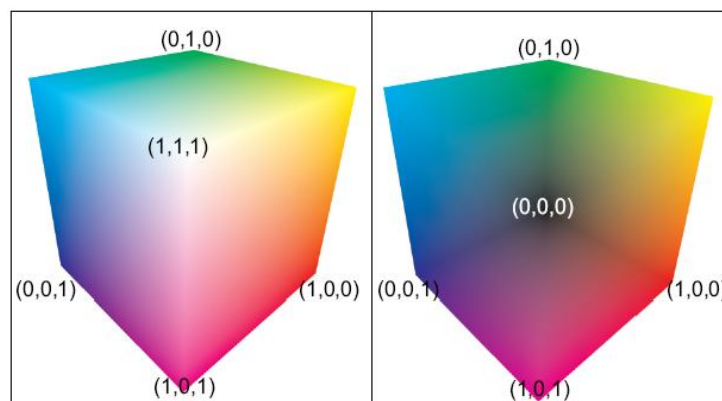


Heres a conceptual image of the raycasting algorithm where one ray pr pixel is spawned and traced through the volume.

In order to generate the nessesary rays we use a clever trick by using OpenGL abillity to render geometry. How can this help us you might say? Well listen up my young apprentice. First we define a ray:

- A ray is just a origin point  $o$  and a direction vector  $dir$ .
- A ray describes a line in 3d space by using the formula  $P(t) = o + dir * t$
- So to generate a ray we need to find the origin point and the direction vector.

This can be done be rendering a cube where the colors represent coordinates, and let OpenGL's interpolation take care of the rest. The way to do this is to render the front and the backside of a unitcube that is illustrated just below.



If we subtract the backface (on the right) from the front we get at a direction vector for each pixel. This is the direction of our ray. The origin is just the frontface values of the cube. So we have to do two renderpasses one for the front and one for the back. To render the backside we enable frontface culling. In my implementation i use a OpenGL framebuffer object (fbo) to store the result from the rendering of the backside, and use the frontface rendering to generate the fragments that starts the raycasting process. If you are unfamiliar with framebuffer objects check out this [link](#).

So to calculate the raycasting we need to create the ray and then step through the volumetexture. This is all done in a single fragmentprogram and calculated on the GPU. The fragment program is fairly simple, the only real issue is to calculate texturecoordinates used to index the backface buffer in order to get the ray exit point. These texture coordinates is refered to as normalized device coordinate, and in the implementation we find a corresponding pixel in the backface buffer by this calculation.

```
float2 texc = ((IN.Pos.xy / IN.Pos.w) + 1) / 2;
```

Where  $IN.Pos$  is the modelviewprojected position. This calulation gives us the fragments screen position

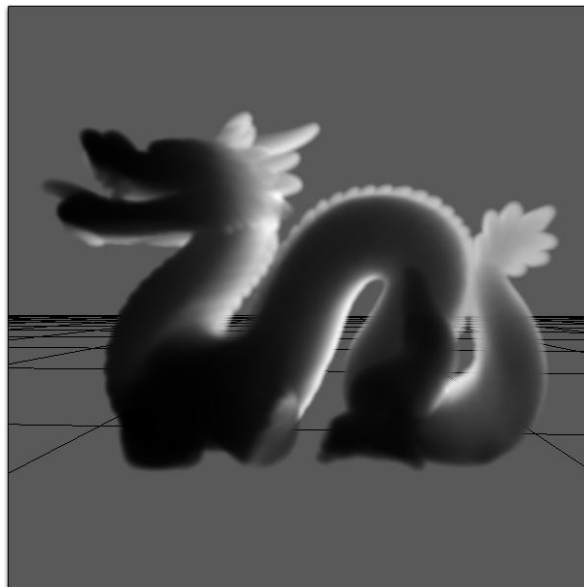
in the interval between  $[0,1]$ . Then the ray exitposition is found by using *texc* to index in the backface buffer like this:

```
float4 exit_position = tex2D(backface_buffer, texc);
```

Now we create the ray and use the shader model 3.0 looping capabilities to create a for loop. This loop will step through the volume with a certain stepsize *delta* and we can accumulate opacity and color value according to the nature of our volume data set. In the demo implementation the ray terminates when it leaves the volume or when the accumulated opacity reaches a high enough value. But there are many possibilities. For instance if we terminate the ray when a certain opacity threshold is first encountered then the result will be some kind of iso surface rendering.

Here is a link to the [raycasting shader](#).

The raycasting technique can be used for many types of rendering problems where polygonbased rendering have a hard time. For instance effects like smoke and glass can be rendered in realtime. Maybe i will do a tut on these subjects in the near future.

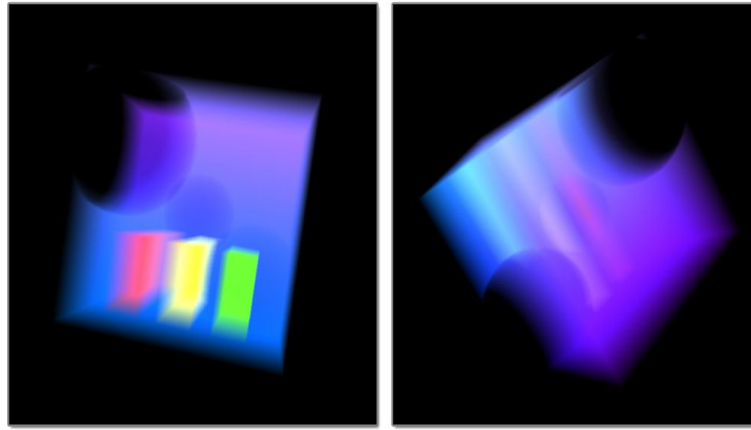


The infamous Stanford dragon rendered using GPU raycasting.

To get you started with this cool rendering algorithm, i have made a simple GPU raycasting implementation that hopefully will clear up the details. That is just the kind of guy i am 😊

The demo contains a windows executable and source code that implements the GPU raycasting algorithm. To compile you need [Nvidias cg](#) and [Glew](#).

The demo just shows a volume of colors where some spheres have been subtracted. The stepsize of the ray can be adjusted by pressing the “w” and “e” buttons. Notice this demo might be hard on your gfx card, just try to press “w” a lot this will result in a big stepsize and the raycasting will update more rapidly.



To download windows demo and sourcecode click [here](#).

As a last comment: this is just a tutorial that will get you started with GPU raycasting, the technique was invented back in 2003 by these guys:

J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In Proceedings of IEEE Visualization 2003, pages 287–292, 2003.

The GPU raycasting is an active research area and if you want to learn more about this, here is a couple of references:

A more recent article that explain a shader model 3.0 implementation can be found at this url:

<http://www.vis.uni-stuttgart.de/ger/research/fields/current/spvolren/>

VRVis has posted a lot of really good papers on the subject so that has been my greatest resource. Visit them at this address: <http://medvis.vrvis.at/home/>

If you find some errors or make some cool improvements please let me know [trier@daimi.au.dk](mailto:trier@daimi.au.dk). Have fun!!

- 
- 

## • Pages

- [Billeder af Ida](#)
- [Billeder af Ida feb 2007](#)
- [CAVI stereo movie player](#)
- [GPU raycasting tutorial](#)
- [Ida marts 2007](#)
- [Master Thesis the visible ear](#)
- [My drawings.](#)
- [Nu med nyheder på dansk.](#)
- [Old Projects](#)
- [Pressure boy](#)

- [Shader framework](#)
- [Tank Island project page.](#)
- [The Visible Ear Project](#)
- [Water boy](#)

## • Archives

- [April 2009](#)
- [August 2008](#)
- [October 2007](#)
- [September 2007](#)
- [July 2007](#)
- [June 2007](#)
- [April 2007](#)
- [March 2007](#)
- [February 2007](#)
- [January 2007](#)
- [November 2006](#)
- [October 2006](#)
- [September 2006](#)
- [August 2006](#)
- [July 2006](#)
- [June 2006](#)

## • Categories

- [Uncategorized](#) (46)

## • Blogroll

- [Planet DAIMI](#)
- [Christian Schneider](#)
- [Carsten Noe](#)
- [Lars Hvam](#)
- [Thomas Mølhave](#)
- [Thereses blog](#)

## • Computer games

- [Gamespot](#)
- [Rpgdpt](#)

## • Dev

- [CG-Alexandra](#)
- [Gamedev](#)
- [Opendgl](#)
- [Spiludvikling](#)

## • Stuff

- [Bedstefars side](#)
- [Chuck Norris top 100 facts](#)
- [CV](#)
- [Daimi webmail](#)
- [gmail](#)
- [Google](#)
- [Ni](#)
- [Tegnebordet](#)
- [The Konzacks](#)
- [Vesterhavshytten](#)
- [Web gallery](#)

## • Meta

- [Login](#)
- [Valid XHTML](#)
- [XFN](#)
- [WordPress](#)

---

Peter Triers Blog is proudly powered by [WordPress](#)  
[Entries \(RSS\)](#) and [Comments \(RSS\)](#).

---

This blog is protected by [Dave's](#) [Spam Karma 2](#): **19605** Spams eaten and counting...