VSVR: a very simple volume rendering implementation with 3D textures

THOMAS LEWINER

Department of Mathematics — Pontifícia Universidade Católica — Rio de Janeiro — Brazil
http://www.mat.puc-rio.br/˜tomlew.

Abstract. This paper presents a volume rendering implementation using 3D textures as an approximation for ray casting. The implementation available with this paper seeks simplicity: it uses only the 3D texture extension of `OpenGL` 1.1 and fits into a limited amount of code. This code intends to provide an introductive material for newbies and a good starting point for more complex implementations of volume rendering.
Keywords: ***Volume Rendering. 3D texture. Ray casting.***

Figure 1: *Tridimensional objects rendered as volume: (left) an implicit function close to its singularity, (middle) a tomography of two cylinders of an engine and (right) an x–ray scan of a human skull.*

1 Introduction

Although graphics hardware is designed for triangle meshes, direct volume rendering already achieved real time, accurate 3D visualization. Since this technique processes directly tridimensional scalar fields, it fits perfectly for a wide range of applications from modeling to simulation, with a particular emphasis on medical imaging.

This paper presents a very concise and simple implementation of volume rendering. It uses the usual approximation of ray casting by 3D textures [6]. As opposed to the many available source codes, our implementation aims at being didactic for the reader interested in developing its own volume rendering application. The code uses only basic `OpenGL` features, namely clipping planes and 3D textures, which are available since `OpenGL` 1.1 [8]. Moreover, it simplifies the usual 3D texture pipeline by using hardware clipping instead of explicitly computing slice intersections with the data cube. The example application provides interactive visualization of fair quality (Figure 1). Its source fits into a small amount of code and is available with this paper.

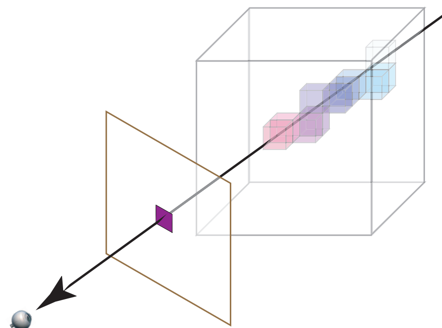

Figure 2: *The ray casting integral sums the color and opacity properties of each data voxel that intersects the ray.*

2 Ray casting with textured slices

Ray casting techniques [6] simulate the propagation of the light traversing a colored translucent volume. For example, in a computational tomography of a head, each piece of brain corresponds to a scintillation of the X radiation. The brain can then be rendered simulating the light propagation across each piece of brain, associating its scintillation with

Preprint MAT. 16/06, communicated on June 26th, 2006 to the Department of Mathematics, PUC — Rio de Janeiro, Brazil.

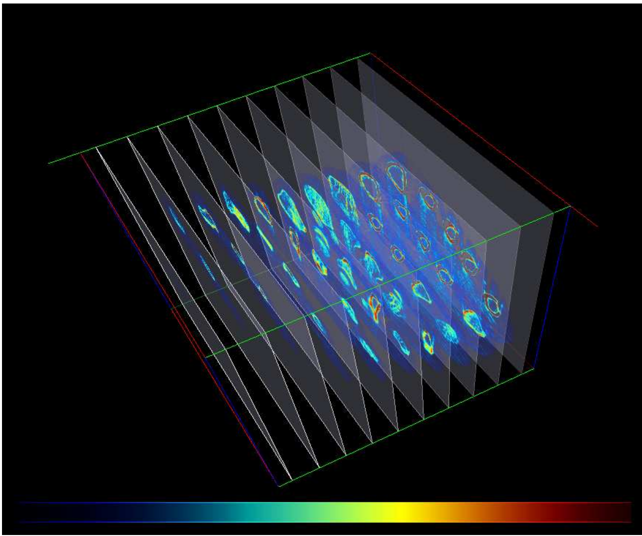

Figure 3: *Axis aligned slices may fail to render parts of the volume, even taking the closest axis to the view direction.*

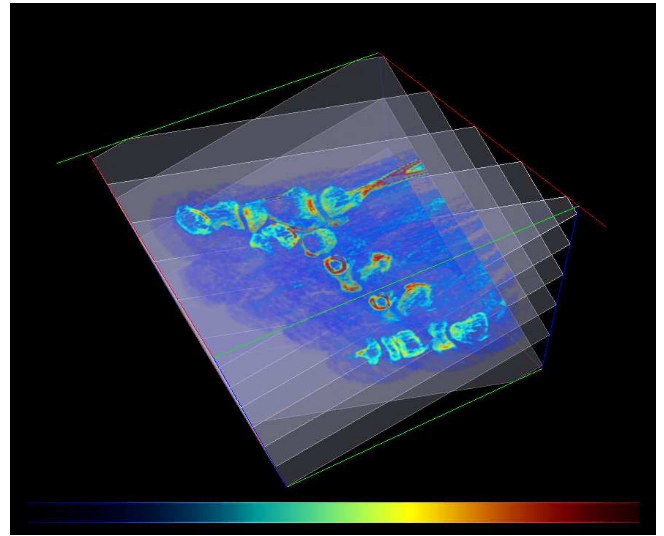

Figure 4: *Slices perpendicular to the view direction optimize the covering of screen pixels.*

a translucent color. In the following, the scintillation of each piece of brain will be called the *data cube*, and its association with a translucent color the *transfer function*.

Additive reprojection. Each translucent volume element contributes to the final color of the ray when it reaches the screen. This final color is usually computed by summing these contributions (Figure 2). However, instead of simulating the ray reflection and refraction as for ray tracing, ray casting force light raysto be straight. This approximation allows faster display through the use of 3D textures.

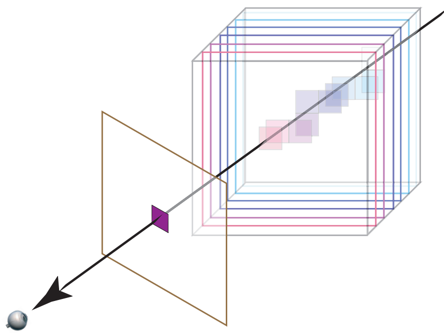

Figure 5: *The ray casting integral can be approximated by summing over parallel 2D slices. The pixel color of each 2D slice is interpolated from the data cube.*

Textured slices. For a given view point, the volume is rendered by shooting rays from the view point to each screen pixel. With `OpenGL` textures, the color of all the pixels can be computed simultaneously by superimposing 2D slices textured by the data properties (Figure 5): The texture color at a point $\mathbf{p}$ of a 2D slice is obtained by applying the transfer function to the voxel of the data cube containing $\mathbf{p}$.

Axis–aligned slices. Rendering with slices parallel to the axis allow using only 2D textures and require very few computations. However, as the view direction becomes tangent to the slice, some screen pixels are not covered by any slice. Even taking the closest axis to the view direction, it may fail to render part of the volume (Figure 3).

View–aligned slices. This can be corrected by generating slices always perpendicular to the view direction: it maximizes the number of slices covering each screen pixel (compare Figure 4 with Figure 3). This technique requires computing non–rectangular slices in order to fit into the data cube, and generating the texture of this slice. The next section describes how we perform these operations using `OpenGL` clipping planes and 3D textures.

3 Implementation

In order to avoid computing the intersection of the slices with the data cube, we render rectangular slices, clipped by the support planes of the cube faces. Since on screen, the rectangular slices must cover the data cube, we compute the rectangle of these slices as the bounding box of the data cube on screen (Figure 6). This bounding box computation requires projecting only the 8 vertices of the data cube. The slices are then generated parallel to the screen by shifting that bounding box in depth. The induced depth ordering allows using only basic commands for transparency.

The clipping is performed directly through `OpenGL` clipping planes (Figure 6), which substitues the software computation of the slice geometry by hardware operations. Moreover, the clipping planes are defined by the input data cube, and can thus be set once for the `OpenGL` context, and then just enabled and disabled at each display operation.

The `OpenGL` architecture then computes directly in hardware the slices' textures through a trilinear interpolation in-

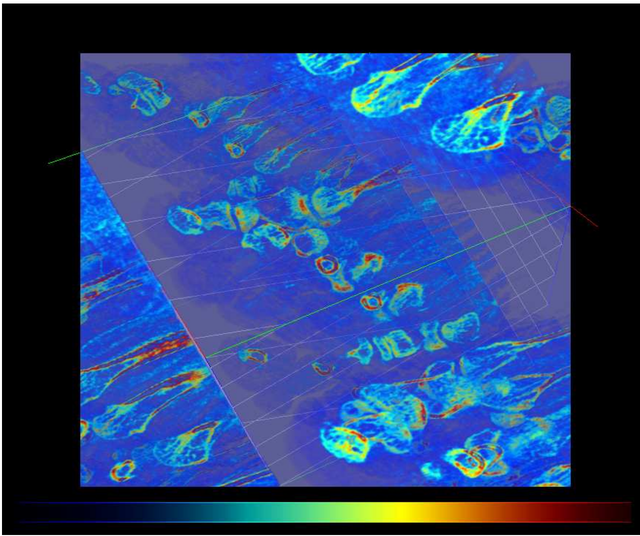

Figure 6: *Textured slices before clipping: the slices' size is the bounding box of the data cube on screen.*

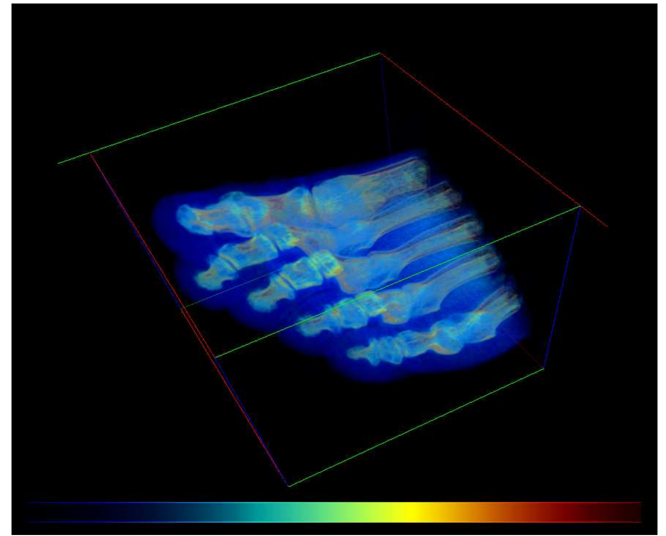

Figure 7: *Final rendering after clipping: the clipping planes are the support of the cube's faces.*

side the data cube. To do so, the data cube is sent to the graphic card as a 3D texture.

Summarizing, the rendering pipeline is composed of the following 4 steps (the functions refer to the source code extract of the appendix):

1. enable clipping planes (*function* ***gl_clip***).
2. load the 3D texture with the transfer function (*functions* ***tf_glload*** *and* ***tex_glload***).
3. compute the bounding box of the texture cube in screen coordinates (*beginning of* ***gl_redisplay***).
4. draw one rectangle for each slice by shifting the bounding box in depth (*end of* ***gl_redisplay***).

4 Extensions

A `C++` implementation is available with the paper. It focuses on the projection part of the 3D texture, as described in the previous sections, and provides a very simple interface for transfer function based on color maps and explicit opacity function. Further extensions can be incorporated to that code, for example handling of large textures, illuminations, pre–integration techniques and more complex transfer function. Further references on volume rendering can be found in [3].

Large textures. The texture memory limitation on graphics card prevents rendering large texture as one block. In order to by-pass this limitation, the data cube must be cut into blocks, each block being rendered separately. With the provided source code, this can be done by instantiating several `VSVR` classes. More complex techniques work on adapted slice geometry to avoid rendering empty regions [1].

Illumination. The structures of 3D solid objects, when rendered as volume, can be strengthen by shading processes [6, 7]. This process requires some further concepts of graphics programming. However, the proposed source code could serve as the vertex projection part of the rendering pipeline, which, combined with a finite difference computation for the gradient, can enter in usual fragment programs.

Pre–integration. The source code of the paper applies the transfer function directly to the 3D texture, and the `OpenGL` pipeline *then* interpolates the colors and opacity for each slice. This can result in sharp color transition if using a contrasted color map on a texture with high gradient. This artifact can be solved by using pre–integration techniques, which apply the transfer function *after* the texture interpolation [2, 4].

Transfer functions. Once the volume rendering pipeline is established, the main user control of the display resumes to controlling the transfer function. In particular, mutli–dimensional transfer functions [5] improve this control.

Web information

A `C++` implementation is available online at `http://www.mat.puc-rio.br/~tomlew`.

Acknowledgments

The engine and the skull of Figure 1 are courtesy of General Electric and of Siemens Medical Solutions. The foot model is a courtesy of Philips Research. All these models are available at `http://www.volvis.org/`.

References

[1] C. Bethune and J. Stewart. Adaptive Slice Geometry for Hardware-Assisted Volume Rendering. *Journal of Graphics Tools*, 10(1):55–70, 2005.

[2] K. Engel, M. Kraus and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Graphics hardware*, pages 9–16. ACM, 2001.

[3] K. Engel and T. Ertl. Interactive High-Quality Volume Rendering with Flexible Consumer Graphics Hardware. In *Eurographics State of The Art Report*. Blackwell, 2002.

[4] R. Espinha and W. Celes. High–Quality Hardware–Based Ray–Casting Volume Rendering Using Partial Pre–Integration. In *Sibgrapi*, pages 273–280. IEEE, 2005.

[5] J. Kniss, G. Kindlmann and C. Hansen. Multidimensional Transfer Functions for Interactive Volume Rendering. *Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[6] M. Levoy. Efficient ray tracing of volume data. *Transactions on Graphics*, 9(3):245–261, 1990.

[7] M. Meisner, U. Hoffmann and W. Straser. Enabling Classification and Shading for 3D Texture Mapping Based Volume Rendering. In *Visualization*, pages 1–32. IEEE, 1999.

[8] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification (Version 1.1). Silicon Graphics, 1998.

Main functions of the source code

void VSVR::gl_clip () const sets the clipping planes

```
{
 GLdouble plane[4] ;
 plane = { +1, 0, 0,  0   } ;
 glClipPlane( GL_CLIP_PLANE0, plane ) ;
 plane = { -1, 0, 0, ni() } ;
 glClipPlane( GL_CLIP_PLANE1, plane ) ;

 plane = { 0, +1, 0,  0   } ;
 glClipPlane( GL_CLIP_PLANE2, plane ) ;
 plane = { 0, -1, 0, nj() } ;
 glClipPlane( GL_CLIP_PLANE3, plane ) ;

 plane = { 0, 0, -1,  0   } ;
 glClipPlane( GL_CLIP_PLANE4, plane ) ;
 plane = { 0, 0, +1, nk() } ;
 glClipPlane( GL_CLIP_PLANE5, plane ) ;
}
```

void VSVR::tf_glload () const loads the transfer function

```
{
 int n = tf_size() ;
 glPixelTransferi(GL_MAP_COLOR, GL_TRUE);
 glPixelMapfv(GL_PIXEL_MAP_I_TO_R, n, _tf_red  );
 glPixelMapfv(GL_PIXEL_MAP_I_TO_G, n, _tf_green);
 glPixelMapfv(GL_PIXEL_MAP_I_TO_B, n, _tf_blue );
 glPixelMapfv(GL_PIXEL_MAP_I_TO_A, n, _tf_alpha);
}
```

void VSVR::tex_glload () loads the 3D texture

```
{
 // init the 3D texture
 glEnable(GL_TEXTURE_3D_EXT);
 glGenTextures(1, &tex_glid);
 glBindTexture(GL_TEXTURE_3D_EXT, tex_glid);

 // texture environment setup
 glTexParameteri( GL_TEXTURE_3D_EXT,
          GL_TEXTURE_MIN_FILTER, GL_LINEAR );
 glTexParameteri( GL_TEXTURE_3D_EXT,
          GL_TEXTURE_MAG_FILTER, GL_LINEAR );
 glTexParameteri( GL_TEXTURE_3D_EXT,
          GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE );
 glTexParameteri( GL_TEXTURE_3D_EXT,
          GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
 glTexParameteri( GL_TEXTURE_3D_EXT,
          GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );

 // load the texture image
 glTexImage3DEXT(
   GL_TEXTURE_3D_EXT, // target
   0,                 // level
   GL_RGBA,           // color storage
   (int) tex_ni(),    // width
   (int) tex_nj(),    // height
   (int) tex_nk(),    // depth
   0,                 // border
   GL_COLOR_INDEX,    // format
   GL_FLOAT,          // type
   _texture       ); // allocated texture buffer

 glPixelTransferi(GL_MAP_COLOR, GL_FALSE);
}
```

void VSVR::gl_redisplay (int* nslices *) const display : *nslices* is the number of slices

```
{
 // gets the direction of the observer
 double x,y,z ;
 double  model[16], proj[16] ;
 int     view[4];
 glGetDoublev (GL_MODELVIEW_MATRIX , model);
 glGetDoublev (GL_PROJECTION_MATRIX, proj );
 glGetIntegerv(GL_VIEWPORT         , view );

 //--------------------------------------//
 // bounding box of the data cube on screen
 double xmin = ymin = zmin =  FLT_MAX ;
 double xmax = ymax = zmax = -FLT_MAX ;
 for( int i = 0; i < 8; ++i )
 {
  float bbx = (i&1) ? ni() : 0 ;
  float bby = (i&2) ? nj() : 0 ;
  float bbz = (i&4) ? nk() : 0 ;
  gluProject( bbx, bby, bbz,
          model, proj, view, &x, &y, &z );

  if( x < xmin ) xmin = x ;
  if( x > xmax ) xmax = x ;
  if( y < ymin ) ymin = y ;
  if( y > ymax ) ymax = y ;
  if( z < zmin ) zmin = z ;
  if( z > zmax ) zmax = z ;
 }

 //--------------------------------------//
 // draw each slice shifting the bounding box
```

```
 float dz = (zmax-zmin) / nslices ;
 float sz = zmax - dz/2 ;

 glColor4f( 1,1,1,1 );
 glBegin( GL_QUADS ) ;
 for(int n = nslices-1; n >= 0; --n, sz -= dz)
 {
  gluUnProject( xmin,ymin,sz,
           model, proj, view, &x, &y, &z );
  glTexCoord3d( x/ni(), y/nj(), z/nk() );
  glVertex3d  ( x,y,z ) ;

  gluUnProject( xmax,ymin,sz,
           model, proj, view, &x, &y, &z );
  glTexCoord3d( x/ni(), y/nj(), z/nk() );
  glVertex3d  ( x,y,z ) ;

  gluUnProject( xmax,ymax,sz,
           model, proj, view, &x, &y, &z );
  glTexCoord3d( x/ni(), y/nj(), z/nk() );
  glVertex3d  ( x,y,z ) ;

  gluUnProject( xmin,ymax,sz,
           model, proj, view, &x, &y, &z );
  glTexCoord3d( x/ni(), y/nj(), z/nk() );
  glVertex3d  ( x,y,z ) ;
 }
 glEnd() ; // GL_QUADS
}
```