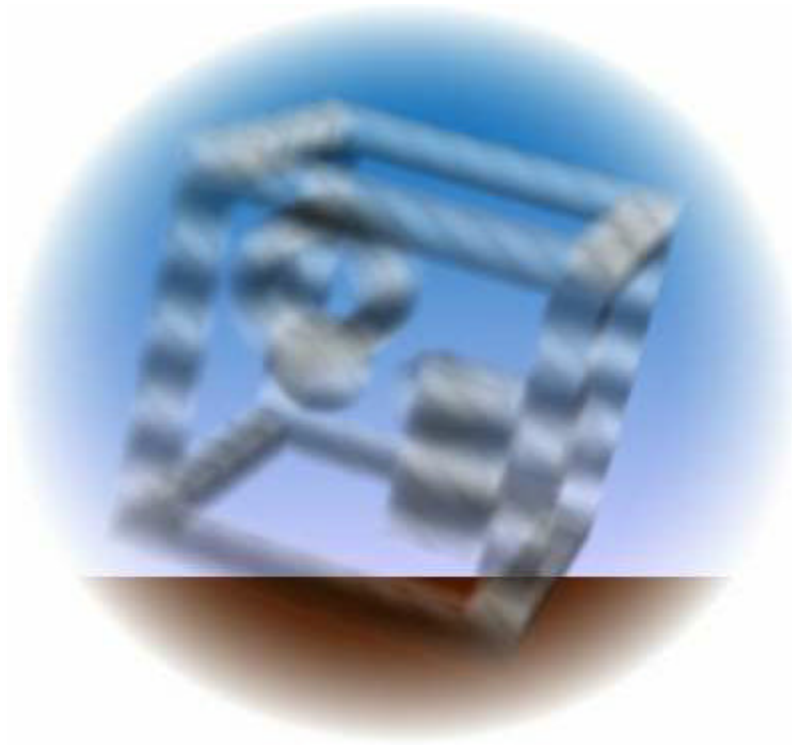




Volume

By implementing some simple volume rendering this example shows the use of 3D texture and glClipPlane.



Volume rendering is an image based problem which is considerably different from conventional geometry based rendering. Many thanks to Rachael Brady for her initial enquiries and sample code which led ultimately to this demo. When volume rendering the task is for each pixel to compute the integral of the ray intersection of the volume data set. This is computationally expensive. SGI hardware supports an OpenGL extension which when used correctly can exploit the power of the texturing hardware to compute the volume rendered image of the dataset. The supported extension is EXT_texture3D and the volume demo uses this extension to present a simplest case example of volume rendering.

There are a few simple steps to rendering a volume dataset using OpenGL:

- the volume data must be loaded into texture memory as a 3D texture
- the volume should be oriented correctly w.r.t. the viewer
- orthogonal slices in eye space must be generated
- these orthogonal slices should be clipped to the bounds of the volume data
- the slices should be drawn in back to front order with 3D texture applied *through* them
- as the slices are drawn they must be blended to accumulate the pixel ray path integrals

Using the demo I'll explain one method of accomplishing this result, it is by no means the only method but

in essence all methods using the 3D texturing approach are similar to the one presented here.

The volume data is loaded to texture memory, using the Performer API this is the same as any other image download but the r texture dimension is >1 and the image data array should be three dimensional. To create the 3D image array the demo calls the MakeImage function but normally this data would be read from disk or some other data source.

```
tex->setImage((uint *) (image), nc, sx, sy, sz);
```

For 3D textures a trilinear filter should be selected as a minimum, remember this is a 3D texture so trilinear in this case is analogous to bilinear for the 2D case:

```
tex->setFilter(PFTEX_MINFILTER, PFTEX_TRILINEAR);  
tex->setFilter(PFTEX_MAGFILTER, PFTEX_TRILINEAR);
```

Our example is volume rendering an image with both intensity and alpha information:

```
tex->setFormat(PFTEX_INTERNAL_FORMAT, PFTEX_IA_8);
```

Each frame the volume texture coordinates are generated using texture coordinate generation. The plane equations are set to produce textures coordinates which form a 3D cube of data which bound the origin from -1 to 1 on all axes, this is associated with the pfGeoState used when drawing the slices.

```
pfTexGen *texgen = new pfTexGen;  
gstate->setMode(PFSTATE_ENTEXGEN, PF_ON);  
gstate->setAttr(PFSTATE_TEXGEN, texgen);  
texgen->setMode(PF_S, PFTG_OBJECT_LINEAR);  
texgen->setMode(PF_T, PFTG_OBJECT_LINEAR);  
texgen->setMode(PF_R, PFTG_OBJECT_LINEAR);  
texgen->setMode(PF_Q, PFTG_OFF);  
texgen->setPlane(PF_S, 0.5f, 0.0f, 0.0f, 0.5f);  
texgen->setPlane(PF_T, 0.0f, 0.5f, 0.0f, 0.5f);  
texgen->setPlane(PF_R, 0.0f, 0.0f, 0.5f, 0.5f);
```

The slice geometry is drawn in world space but never changes w.r.t. the eye since we don't move the eye once we have set up the slices for rendering i.e we have no modelview transformation so object == world == eye space. You will see later that the texture matrix is used to transform the dataset, not the modelview matrix. The depth ordering is built into the dataset:

```
// Set up geosets  
pfVec3 *coords = (pfVec3*) new((4*NUM_QUADS)*sizeof(pfVec3)) pfMemory;  
for(i = 0; i < (4*NUM_QUADS); i+=4)  
{  
static float t = 1.0f;  
coords[i].set( -1.0f, t, -1.0f);  
coords[i+1].set( 1.0f, t, -1.0f );  
coords[i+2].set( 1.0f, t, 1.0f );  
coords[i+3].set(-1.0f, t, 1.0f );  
t -= 2.0f / (NUM_QUADS-1.0f);
```

```

}
pfVec4 *colors = (pfVec4*) new(sizeof(pfVec4)) pfMemory;
(*colors).set(1.0f, 1.0f, 1.0f, SLICE_OPACITY);
pfGeoSet *gset = new pfGeoSet;
gset->setAttr(PFGS_COORD3, PFGS_PER_VERTEX, coords, 0);
gset->setAttr(PFGS_COLOR4, PFGS_OVERALL, colors, 0);
gset->setPrimType(PFGS_QUADS);
gset->setNumPrims(NUM_QUADS);

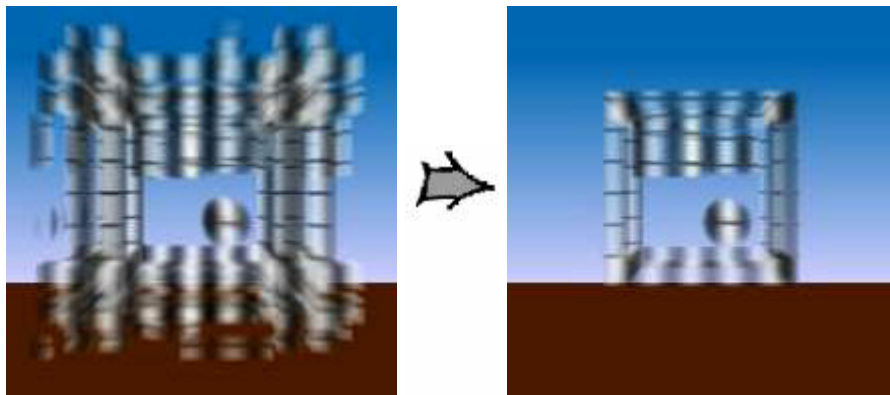
```

When drawing the quads with the texgen applied the cube texture will be applied through the 3D set of cubes and will appear as a ghostly image in 3D, the transparency of the image will depend on both the alpha information in the texture and the polygon alpha determined in our case by the SLICE_OPACITY definition. If the slices are drawn large enough for future cube orientation transformations then we will see the cube image repeating more than once in 3D in much the same way as an unclamped 2D image repeats over a 2D surface, to stop this we can clamp the 3D texture along s, t, and r axes:

```

tex->setRepeat(PFTEX_WRAP_S, PFTEX_CLAMP);
tex->setRepeat(PFTEX_WRAP_T, PFTEX_CLAMP);
tex->setRepeat(PFTEX_WRAP_R, PFTEX_CLAMP);

```



So the cube may be drawn but this is hardly an efficient solution, the 2D slices impose a huge pixel fill performance penalty for the extraneous surrounds and unless we have a dataset which is bounded by transparent image data the clamping will not be sufficient to limit the scope of visible pixel writes, even with a transparent border the extra pixel fill load is heavy. We could trim the slices to match the volume extents, but this changes with orientation, so we want to keep our slices but trim them dynamically depending on the dataset orientation w.r.t. the eye. We could do this on the CPU but OpenGL provides arbitrary clipping planes for just such an occasion, we have six cube faces so we need six clipping planes:

```

glClipPlane(GL_CLIP_PLANE0, peqn0);
glEnable(GL_CLIP_PLANE0);
glClipPlane(GL_CLIP_PLANE1, peqn1);
glEnable(GL_CLIP_PLANE1);
glClipPlane(GL_CLIP_PLANE2, peqn2);
glEnable(GL_CLIP_PLANE2);
glClipPlane(GL_CLIP_PLANE3, peqn3);
glEnable(GL_CLIP_PLANE3);
glClipPlane(GL_CLIP_PLANE4, peqn4);

```

```

glEnable(GL_CLIP_PLANE4);
glClipPlane(GL_CLIP_PLANE5, peqn5);
glEnable(GL_CLIP_PLANE5);

```

The next step is to orient the volume w.r.t. the eye, this requires that we apply texture matrix operations to the coordinates generated by the texgen. Remember that the coordinates aren't at the origin, but they bound the origin, so to rotate the cube about a point we need to translate that point to the origin rotate and then translate back:

```

glGetIntegerv(GL_MATRIX_MODE, &mmode);
glMatrixMode(GL_TEXTURE);
glPushMatrix();
// translate to rotate about correct point
glTranslatef(0.5f, 0.5f, 0.5f);
// scale to fit inside geometric cube should be
// a little smaller but I don't want to go to root 3
glScalef(1.55f, 1.55f, 1.55f);
// rotate to animate
glRotatef( angle, -1.0f, -1.0f, 1.0f );
// translate back
glTranslatef(-0.5f, -0.5f, -0.5f);
glMatrixMode(mmode);

```

Note I've also shrunk the cube image mapping a little to fit inside the area of the slice quads we created earlier, this is a bit of legacy from when I was trying to optimize the pixel fill performance prior to adding the glClipPlane slicing. The only remaining step is to orient the glClipPlanes in the same way the texture has been oriented so they clip to the volume bounds, here's an example of the first clip plane:

```

peqn[3] = .65;
matey.makeRot( -angle, -1.0f, -1.0f, 1.0f );
peqn0.xformVec( peqn0, matey );
peqn[0] = peqn0[0];
peqn[1] = peqn0[1];
peqn[2] = peqn0[2];
glClipPlane(GL_CLIP_PLANE0, peqn);
glEnable(GL_CLIP_PLANE0);

```

The .65 fourth plane equation entry is the displacement of the plane along the plane normal vector and this defines the cube wall, the first three terms which describe the plane normal vector are generated by transforming the cube face normal through the matrix 'matey' which holds the 3D data transformation.



Corporate Office

2011 N. Shoreline Boulevard
Mountain View, CA 94043
(650) 960-1980

U.S. 1(800) 800-7441

Europe (44) 118-925.75.00

Asia Pacific (81) 3-54.88.18.11

Latin America 1(650) 933.46.37

Canada 1(905) 625-4747

Australia/New Zealand (61) 2.9879.95.00

SAARC/India (91) 11.621.13.55

Sub-Saharan Africa (27) 11.884.41.47

Copyright © 1998 Silicon Graphics, Inc. All rights reserved.