Explore New Techniques in Volume Rendering and Volume Segmentation with Open Inventor

May 2012

www.vsg3d.com
VSG: Who we are

Proven track record
- 25+ years of expertise
- 25+ developers
- Highly experienced team
- Successfully serving 1000+ first-class customers

Worldwide company
- USA (Boston, Houston, San Diego)
- France (Bordeaux, Paris)
- Germany (Düsseldorf)
- UK (London)

Distributors worldwide
- Asia/Pacific (China, India, Japan, Taiwan, Korea)
- Mexico, Israel, Russia
Integrate high-performance 3D in your applications

Visualize and analyze scientific and industrial data

Join a partner committed to your success

Open Inventor®
3D development toolkit

Avizo®
3D application framework

3D expertise and Professional services

VSG: What we do
Visualization in E&P Workflow

VSG's software is used to develop visualisation applications in:

- Seismic Acquisition
- Seismic Processing
- Seismic Interpretation
- Reservoir Modeling
- Reservoir Characterization
- Reservoir Simulation
- Well Planning
- Production
Open Inventor for Exploration & Production
Visualization Challenges for E&P

1. Manage data
   - Single volume often larger than system or GPU memory
   - And often need multiple volumes simultaneously

2. Rendering
   - Interactive performance and high image quality

3. Computing
   - Integrate computing and rendering (no duplication of data)
   - Able to access volume on GPU for computation

4. Customization and extension
   - Easy to customize one step of rendering, e.g. color blending

5. Clip, edit and extract data
   - Clip against surfaces, shapes, masks, ...
   - Edit/extract by voxel, surface, shape, ...
1.1 Manage data: Divide and conquer

- Divide volume into “tiles”
  - “Random access” to data
  - Load only what you need (by ROI, primitive, clipping, ...)
  - Useful to know things about tiles
    - Constant value?  □  Optimize storage
    - Min/max values  □  Check if completely transparent

Tiles needed to render one slice (line or xline)
1.2 Manage data: 3D Clipmapping (sort of)

- Octree-like hierarchy of resolutions
  - Level $N$ ⊍ Level $N+1$: 1/8 number of tiles
  - Level $N+1$ tiles: same number of voxels, 8x spatial extent

- Choose “best” set of tiles to represent volume in available/allowed memory
  - By assigned priority
  - By distance from camera
  - By ratio of voxels to pixels
  - . . .
1.3 Manage data: “Virtual” texture on GPU

- **Physical**: Actual data tiles packed into data texture
  - Avoids texture bind overhead

- **Virtual**: Shaders always see a complete volume (0..1)
  - Rendering: access neighbor voxels for blending, gradient, . . .
  - Computing: access any voxel in volume, e.g. along trace
  - No need for overlapping tiles

![Diagram showing VvizGetData API, Paging texture, Data texture, and Shader’s view of volume (uniform & continuous).]
2.1 Rendering: GPU volume raycasting
2.2 Rendering: Image enhancement, e.g. edge detection
2.3 Rendering: Edge detection and shadows
2.5 Voxelized ("sugar cube") rendering
2.6 Rendering: Clip by plane, surface, shape, ...
2.7 Rendering: Large horizons also managed
4.1 Customization: Shader API

- Only need to replace specific step in rendering pipe
  - GetData: e.g. apply filter
  - CombineData: e.g. apply mask
  - ComputeColor: e.g. blend colors
  - ...

Other rendering features (lighting, etc) still apply

- Many utility functions in shader API
- Easy to apply custom shaders

```cpp
// Load fragment shader from source file
// and override Compute_Color function
SoVolumeShader* pVolShader = new SoVolumeShader();
pVolShader->setFragmentShader( 
    SoVolumeShader::FRAGMENT_COMPUTE_COLOR,  
    "myBlendFunction.glsl" );
pGroupNode->addChild( pVolShader );
```
### 4.2 Customization: Co-blending example

<table>
<thead>
<tr>
<th>GLSL Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Fragment shader for AmplitudeVelocity example</td>
<td></td>
</tr>
<tr>
<td>// Intensity value from volume1 (seismic amp1)</td>
<td></td>
</tr>
<tr>
<td>// and color value from volume2 (velocity model)</td>
<td></td>
</tr>
</tbody>
</table>
| //!
| oiv_include <VolumeViz/vvizGetData_frag.h>
| //!
| oiv_include <VolumeViz/vvizTransferFunction_frag.h> |
| uniform VVizDataSetId data1;  |
| uniform VVizDataSetId data2;  |
| vec4 VVizComputeFragmentColor( VVIZ_DATATYPE vox, |
| vec3 vecCoord ) |
| {  |
| VVIZ_DATATYPE value1 = VVizGetData(data1, voxCoord);  |
| vec4 color1 = VVizTransferFunction(value1, 0);  |
| VVIZ_DATATYPE value2 = VVizGetData(data2, voxCoord);  |
| vec4 color2 = VVizTransferFunction(value2, 1);  |
| // Color2 modulated by intensity from volume1 |
| color2.rgb *= color1.r;  |
| color2.a *= color1.a;  |
| return color2;  |
| } |

VolumeViz shader framework lets you include libraries of GLSL shaders.

data1 : virtual texture of first volume
data1 : virtual texture of second volume

Method in the VolumeViz pipeline to override for custom color blending

value1 : float value from the first volume
value1 : color from colormap 0
value2 : float value from the second volume
value2 : color from colormap 1

Co-blending

[Diagram showing the process of co-blending with two volumes: Volume One, Volume Two, Example: Seismic Amplitude, Example: Velocity Model, Fragment Shader Program, Combined volume on screen.]
4.3 Co-blending multiple volumes
Geobody extraction workflow

1. Horizon to horizon (slab) clipping

2. Sculpt volume
   A. Draw sculpting outline on screen
   B. Extrude polygonal shape from outline
   C. Clip to shape (and slab)
   D. Repeat B and C
      Combine polygonal shapes using CSG

3. Extract sculpted volume
   A. Create mask volume from original volume
   B. Edit mask using combined shape
   C. Extract voxels in mask
Seismic volume
Horizon to horizon clipping

- Use: `SoUniformGridClipping` class
  or: `SoUniformGridProjectionClipping` class
Sculpt: Draw outline on screen

- Use: SoLassoScreenDrawer class
Sculpt: Extrude shape from outline

- Use: SbExtrusionGenerator class
Sculpt: Clip to polygonal shape

- Use: SoVolumeClippingGroup class
Sculpt: Combine shapes using CSG

- Repeat draw outline, extrude shape cycle
- Add shapes to CSG “tree” using: SoCSGShape node

```cpp
SoCSGShape* extrudedShapeClipped = new SoCSGShape;
extrudedShapeClipped->leftOperand = extrudedShape;
extrudedShapeClipped->rightOperand = sg->mSlabGeometry;
extrudedShapeClipped->csgOperation = SoCSGShape::INTERSECTION;
```
Extract: Create mask volume

- **SoVolumeMask node**
  - Instanciate an `SoVolumeMask` with same properties as the `SoVolumeData` used in the scene (mask is compact using bit field). Set default value to 0.

```java
mVolumeMask = new SoVolumeMask;
mVolumeMask->setProperties(
    mVolumeData->data.getSize(),
    mVolumeData->getTileDimension(),
    mVolumeData->ldmResourceParameters.getValue()->overlapping.getValue(),
    mVolumeMask->extent = mVolumeData->extent.getValue();
    mVolumeMask->dataSetId = mVolumeData->dataSetId.getValue()+2;
    mVolumeMask->setDefaultValue(0);
```

- Create a closed geometry representing the current sculpted area (`SoCSGShape`)
- Edit the volume mask
  Using the `editSolidShape` method with the above geometry, set visible area to 1.

```java
int transactionId;
mVolumeMask->startEditing(transactionId);
int value = mVolumeMask->editSolidShape(mCSGRoot, 1);
mVolumeMask->finishEditing(transactionId);
```

- Access data information using `SoLDMDataAccess` to know whether a voxel is clipped or not
Extracted geobodies (zoom)
Coming soon…

- More rendering performance and image quality
- New rendering effects, e.g. ambient occlusion