
sgi™

OpenGL Volumizer & Large Data Visualization

Chikai J. Ohazama, Ph.D.
AGD Applied Engineering

Overview

Topics

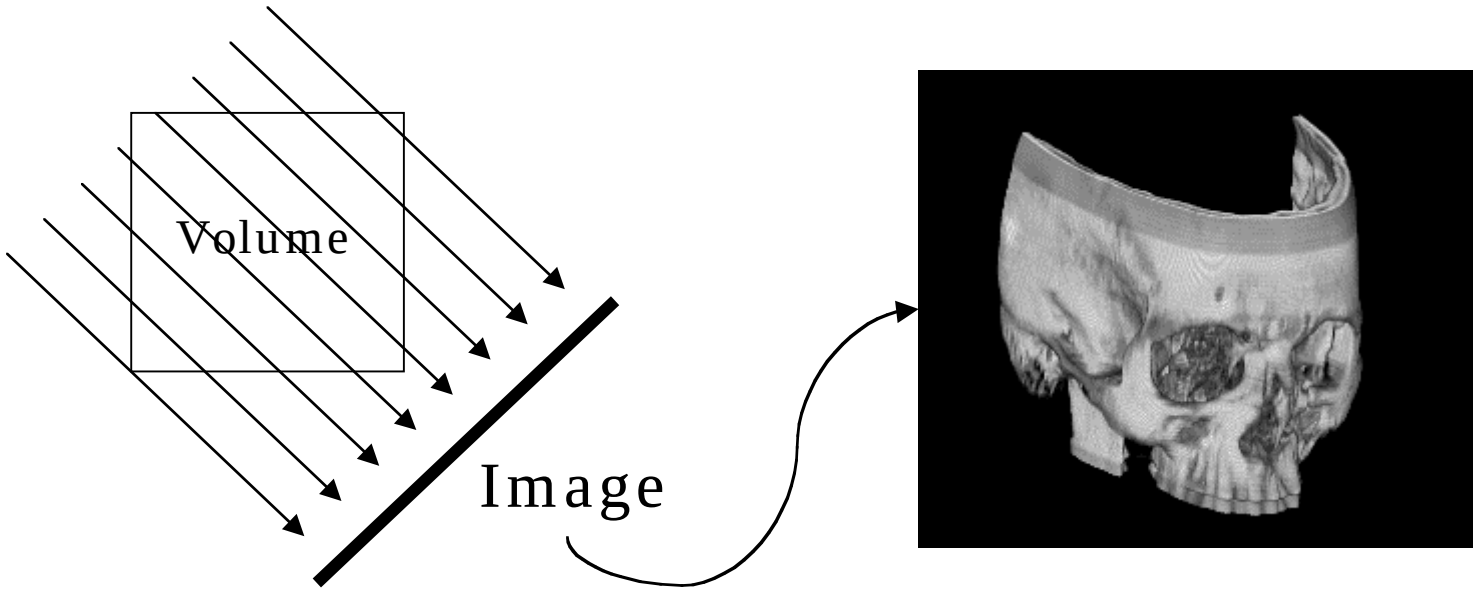
- OpenGL Volumizer
- Parallel Volume Rendering
- Volume Roaming
- Performance Determination

OpenGL Volumizer

Topics

- Volume Rendering
- OpenGL Volumizer
- Code Example
- Applications

Volume Rendering



Volume Rendering

Advantages

- Not necessary to explicitly extract surfaces from volume when rendering
- Can change the Look Up Table (LUT) to bring out different objects within the volume

Volume Rendering

Disadvantages

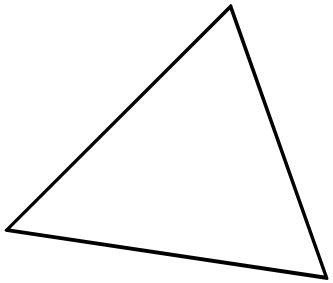
- Do not have explicit surfaces, therefore not straightforward to do computational operations on objects within the volume
- Much more computationally intensive to render volume since not dealing with a set of discrete objects

OpenGL Volumizer

Features

- Tetrahedral Volume Tesselation
- Automatic Brick Management
- Portability across Platforms

OpenGL Volumizer

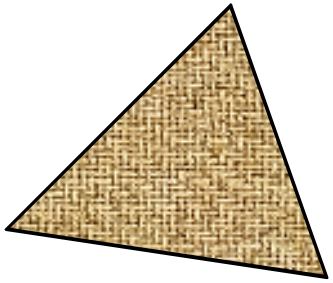


Triangle

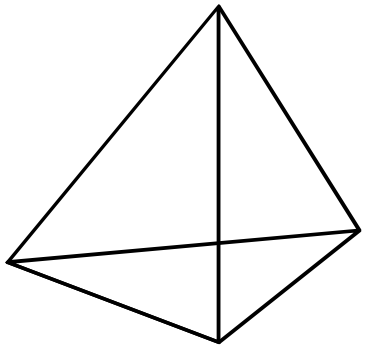
+

2D Texture

=

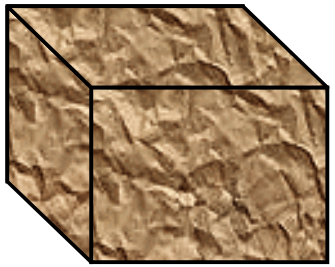


Texture Mapped Triangle



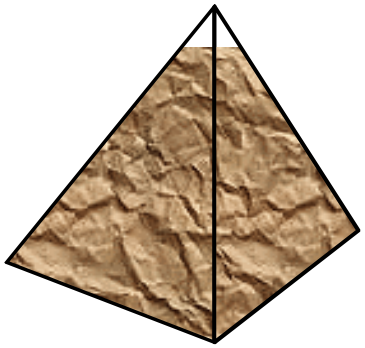
Tetrahedron

+



3D Texture

=



Volume Rendered Tetrahedron

Code Example

Basic Components

- Initialize Appearance
- Initialize Geometry
- Render Volume

Initialize Appearance

```
voAppearanceActions::getBestParameters(
        interpolationType, renderingMode, dataType,
        diskDataFormat,
        internalFormat, externalFormat,
        xBrickSize, yBrickSize, zBrickSize);
```

Automatically calculates brick size and formats necessary for optimal volume rendering.

Initialize Appearance

```
voBrickSetCollection *aVolume =
new voBrickSetCollection(
        xVolumeSize, yVolumeSize, zVolumeSize,
        xBrickSize, yBrickSize, zBrickSize,
        partialFormat,
        1,
        internalFormat,
        externalFormat,
        dataType,
        interpolationType);
```

Creates the bricks, but does not read data or allocate memory.

Initialize Appearance

```
voBrickSetCollectionIterator collectionIter(aVolume);
  for (voBrickSet * brickSet; brickSet = collectionIter();) {
     // iterate over all bricks within the brickCollection
     voBrickSetIterator brickSetIter(brickSet);
     for (voBrick * brick; brick = brickSetIter();)
       voAppearanceActions::dataAlloc(brick);
  }
```

Allocate memory for all copies of the volume.

Initialize Appearance

```
voBrickSetIterator brickSetIter(aVolume->getCurrentBrickSet());
  for (voBrick * brick; brick = brickSetIter();) {

    int xBrickOrigin, yBrickOrigin, zBrickOrigin;
    int xBrickSize, yBrickSize, zBrickSize;

    void *vdata = brick->getDataPtr();

    brick->getBrickSizes(xBrickOrigin, yBrickOrigin,
                zBrickOrigin,
                xBrickSize, yBrickSize,
                zBrickSize);
    getBrick(voldata, vdata,
          xBrickOrigin, yBrickOrigin, zBrickOrigin,
          xBrickSize, yBrickSize, zBrickSize,
          xVolumeSize, yVolumeSize, zVolumeSize);
}
```

Load bricks into memory.

Initialize Geometry

```
static float vtxData[8][3] = {
        0,          0,          0,
    xVolumeSize,    0,          0,
    xVolumeSize,  yVolumeSize,          0,
        0,  yVolumeSize,          0,
        0,          0,   zVolumeSize,
    xVolumeSize,    0,   zVolumeSize,
    xVolumeSize,  yVolumeSize,   zVolumeSize,
      0,        yVolumeSize,   zVolumeSize,
  };
```

Define verticies.

Initialize Geometry

```
static int cubeIndices[20] =
  {
     0, 2, 5, 7,
     3, 2, 0, 7,
     1, 2, 5, 0,
     2, 7, 6, 5,
     5, 4, 0, 7,
  };
```

Define Geometry.

Initialize Geometry

```
voIndexedTetraSet *aTetraSet = new
    voIndexedTetraSet((float *) vtxData, 8, valuesPerVtx,
                      cubeIndices, 20);
```

Construct the tetrahedral set.

Initialize Geometry

```
allVertexData = new voVertexData(100000, valuesPerVtx);

aPolygonSetArray = new voIndexedFaceSet**[maxBrickCount];
for (int j1 = 0; j1 < maxBrickCount; j1++) {
    aPolygonSetArray[j1] =
       new voIndexedFaceSet*[maxSamplesNumber];
    for (int j2 = 0; j2 < maxSamplesNumber; j2++)
       aPolygonSetArray[j1][j2] =
          new voIndexedFaceSet(allVertexData,
                        boundFaceCount(tetraCount));
     }
```

Create storage for transient polygons.

Render Volume

```
voGeometryActions::polygonize(
        aTetraSet,
        aVolume->getCurrentBrickSet(),
        interleavedArrayFormat,
        modelMatrix[pset],
        projMatrix[pset],
        aVolume->getInterpolationType() ==
           voInterpolationTypeScope::_3D ?
           voSamplingModeScope::VIEWPORT_ALIGNED :
           voSamplingModeScope::AXIS_ALIGNED,
        voSamplingSpaceScope::OBJECT,
        samplingPeriod,
        maxSamplesNumber,
        sampletemp,
        aPolygonSetArray[pset]);
```

Polygonize tetraset.

Render Volume

```
voSortAction aSortAction(
   aVolume->getCurrentBrickSet(), modelMatrix, projMatrix);
```

Sort Bricks.

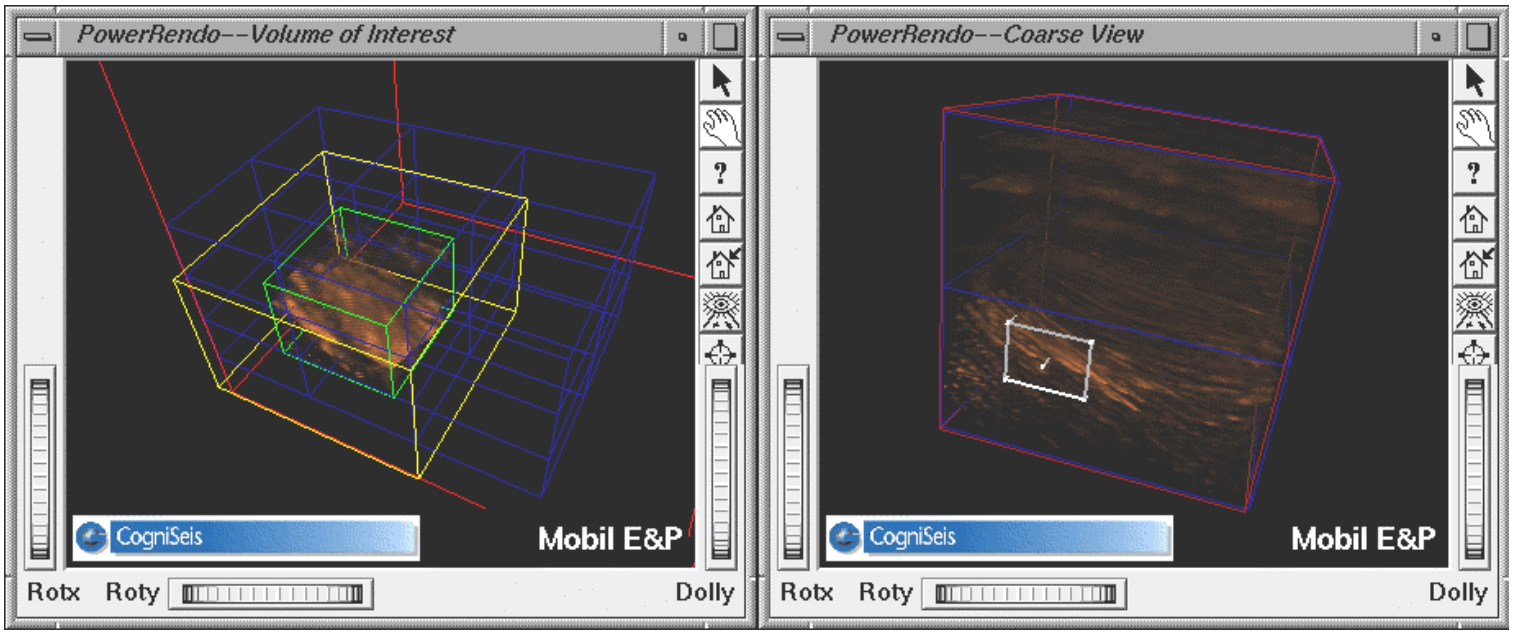
Render Volume – Draw

```
for (brickNo = 0; brickNo < BrickCount; brickNo++) {
  int brickSortedNo = aSortAction[brickNo];

  voBrick *aBrick =
     aVolume->getCurrentBrickSet()->getBrick(brickSortedNo);

  if (!hasTextureComponent(interleavedArrayFormat))
    voAppearanceActions::texgenSetEquation(aBrick);

  voAppearanceActions::textureBind(aBrick);

  for (int binNo = 0; binNo < samplesNumber ; binNo++) {
     voGeometryActions::draw(
        aPolygonSetArray[brickSortedNo][binNo],
        interleavedArrayFormat);
  }
}
```

Render bricks.

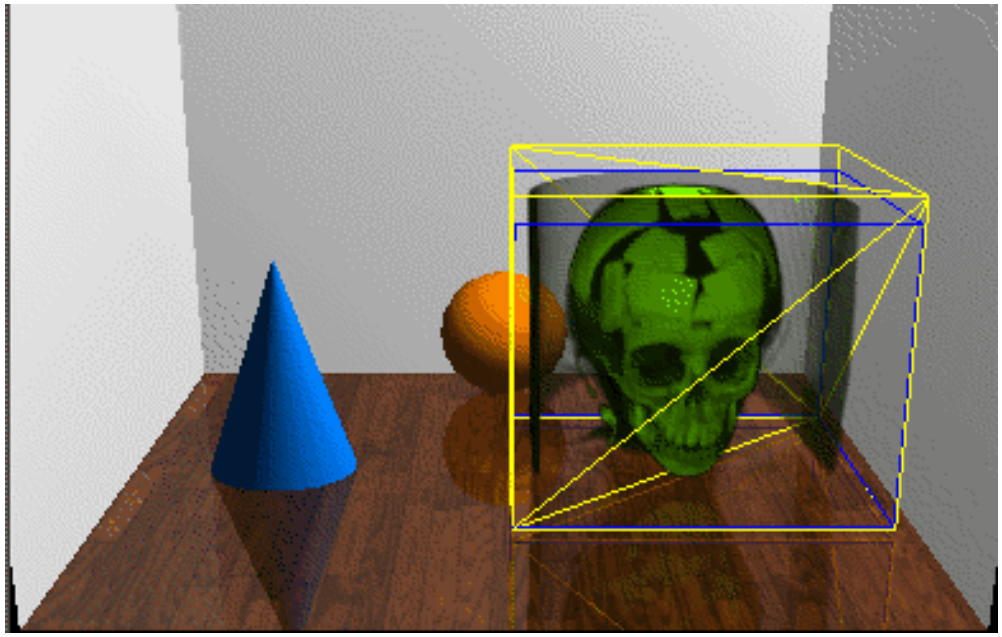
Applications

Volume Roaming



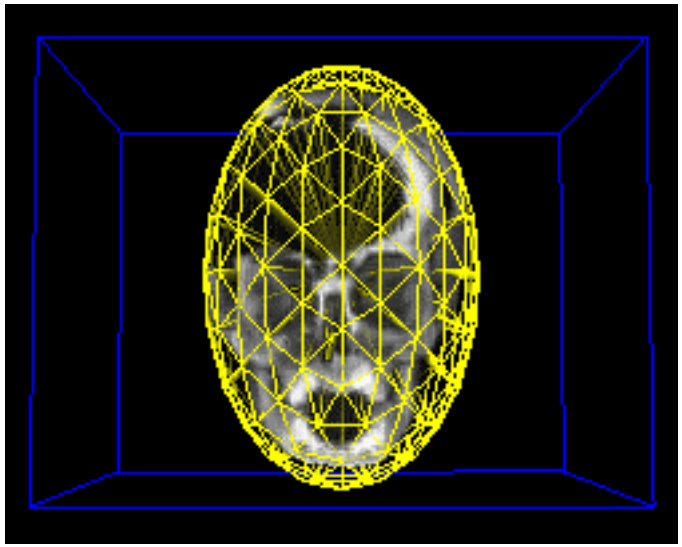
Applications

Heterogenous Rendering (Surface and Volume Objects)



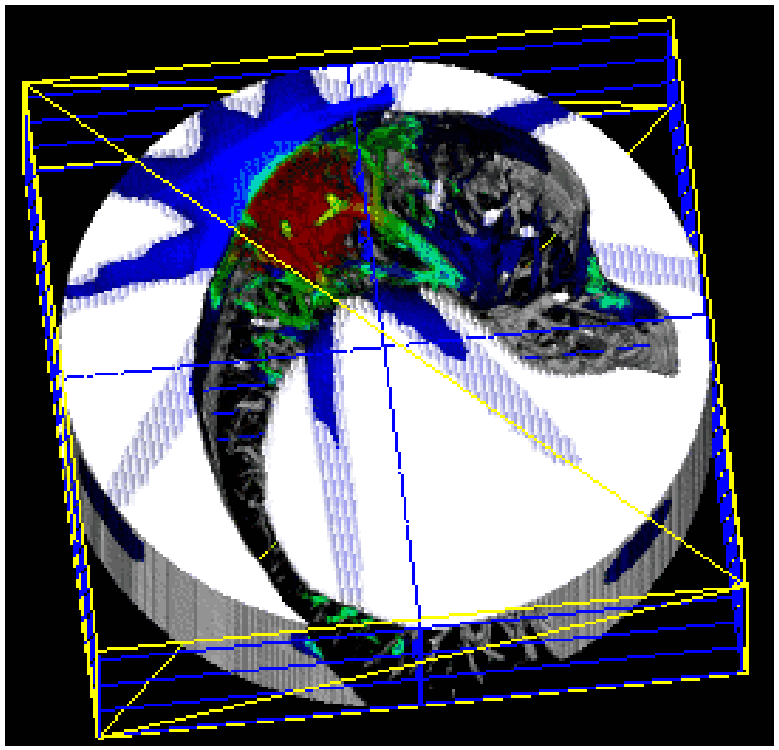
Applications

Region of Interest Volume Rendering



Applications

Multiple Volumes



Applications

- Medical
- Oil&Gas
- Scientific Visualization

Parallel Volume Rendering

Topics

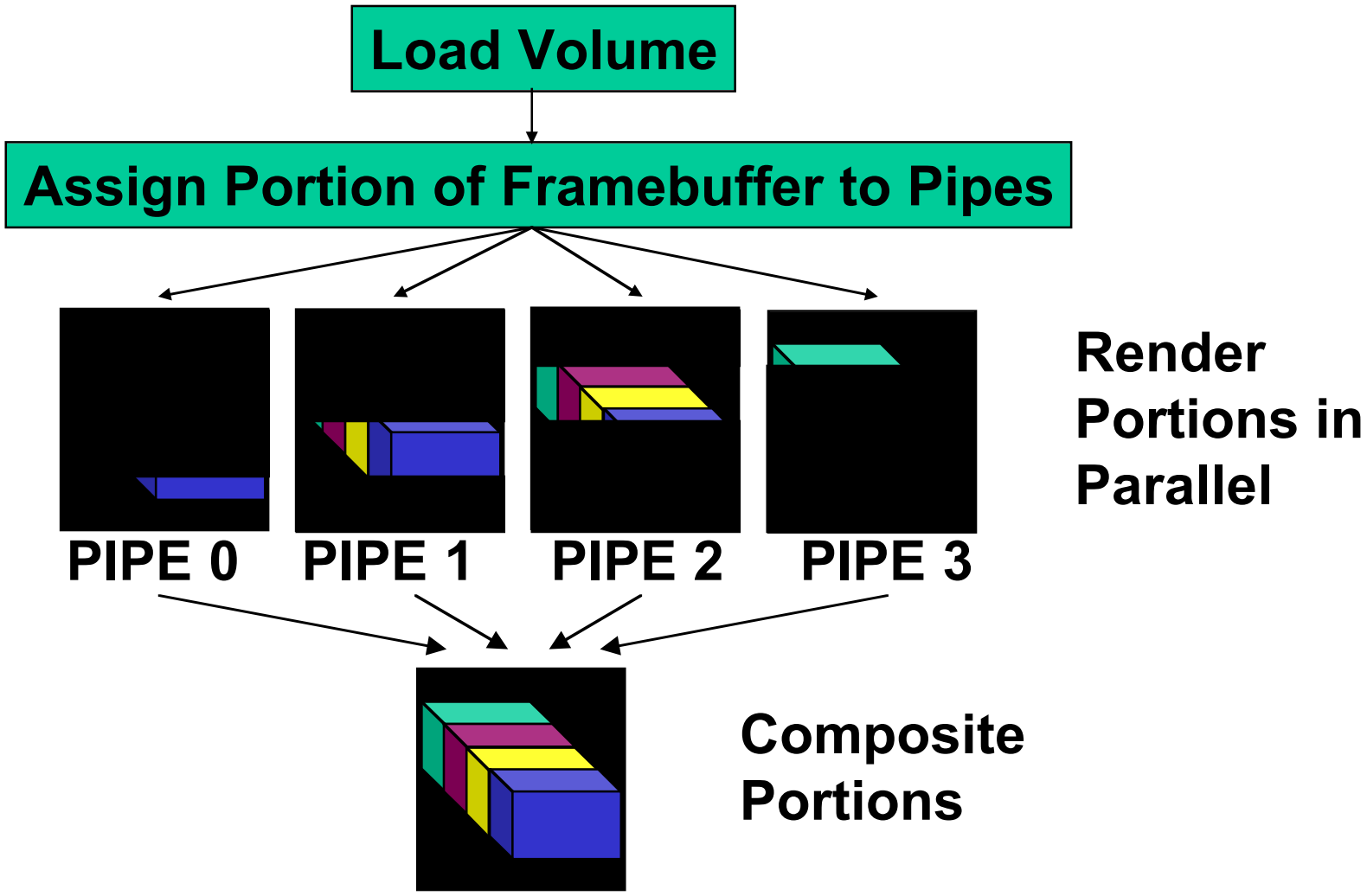
- Basic Architecture
- Scalability
- Code Example

Basic Architecture

Types

- Screen Decomposition
- Data Decomposition

Screen Decomposition



Performance Scalability

Benefits

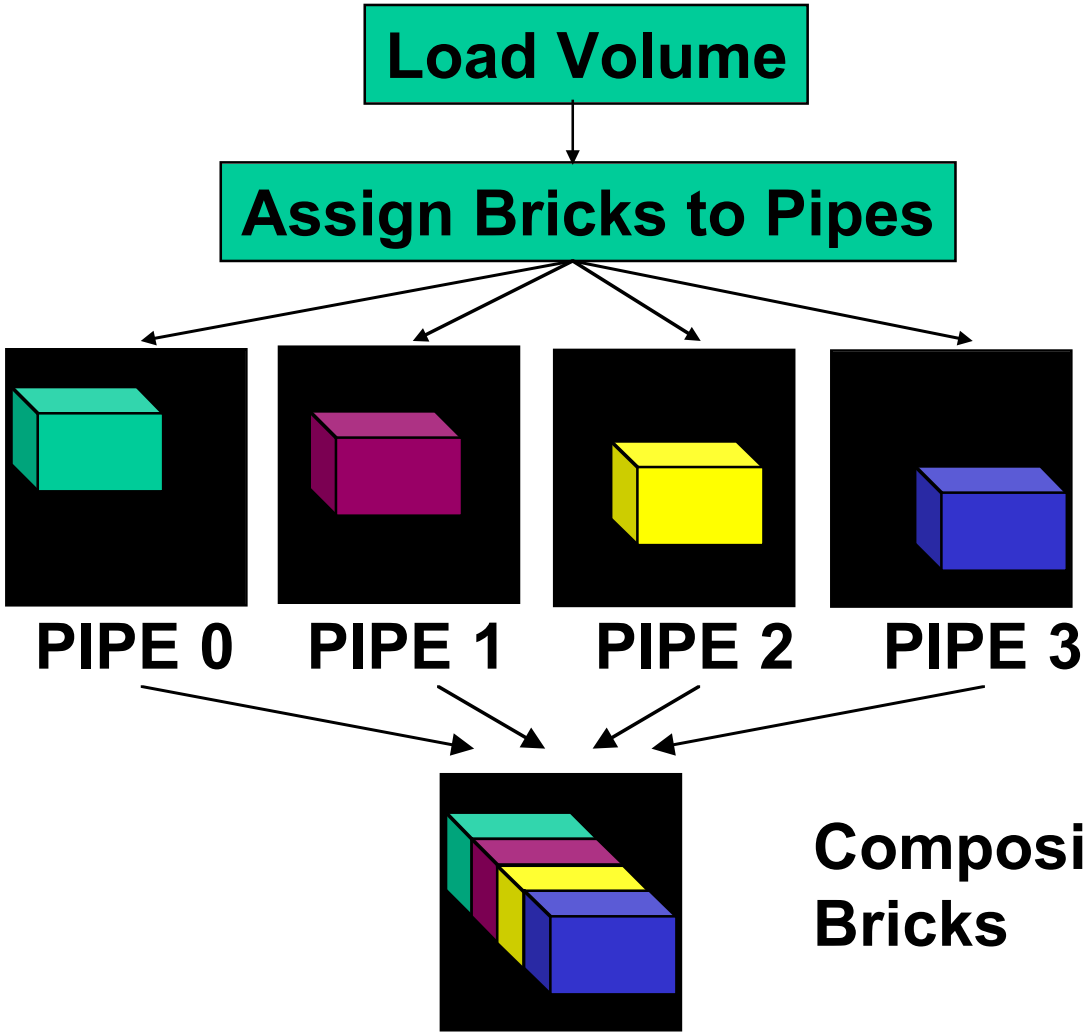
- Linear Increase in Texture Fill Rate
- Framebuffer portion transfer sizes decreases with increased number of pipes, therefore necessary read/write bandwidth is constant

Performance Scalability

Limitations

- Larger the window size, slower the frame rate

Data Decomposition



No Latency Frame Composition

Rendering Setup:

- Pre-Multiplication of Luminance by Alpha (achieved through TLUT)
- Blending Function changed to:

```
glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

Performance Scalability

Benefits

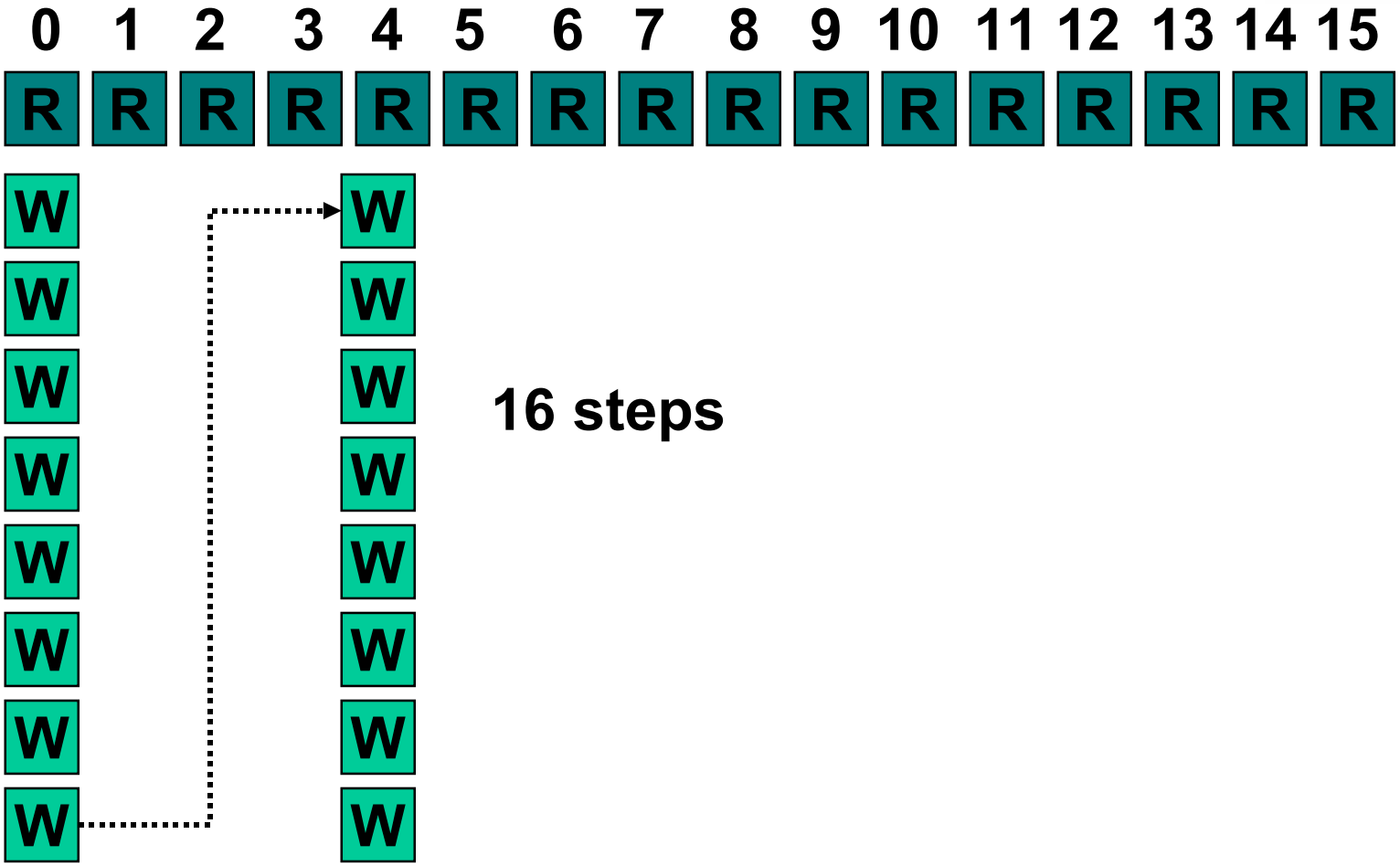
- Linear Increase in Texture Fill Rate
- Linear Increase in Texture Memory
- Linear Increase in Texture Load Rate

Performance Scalability

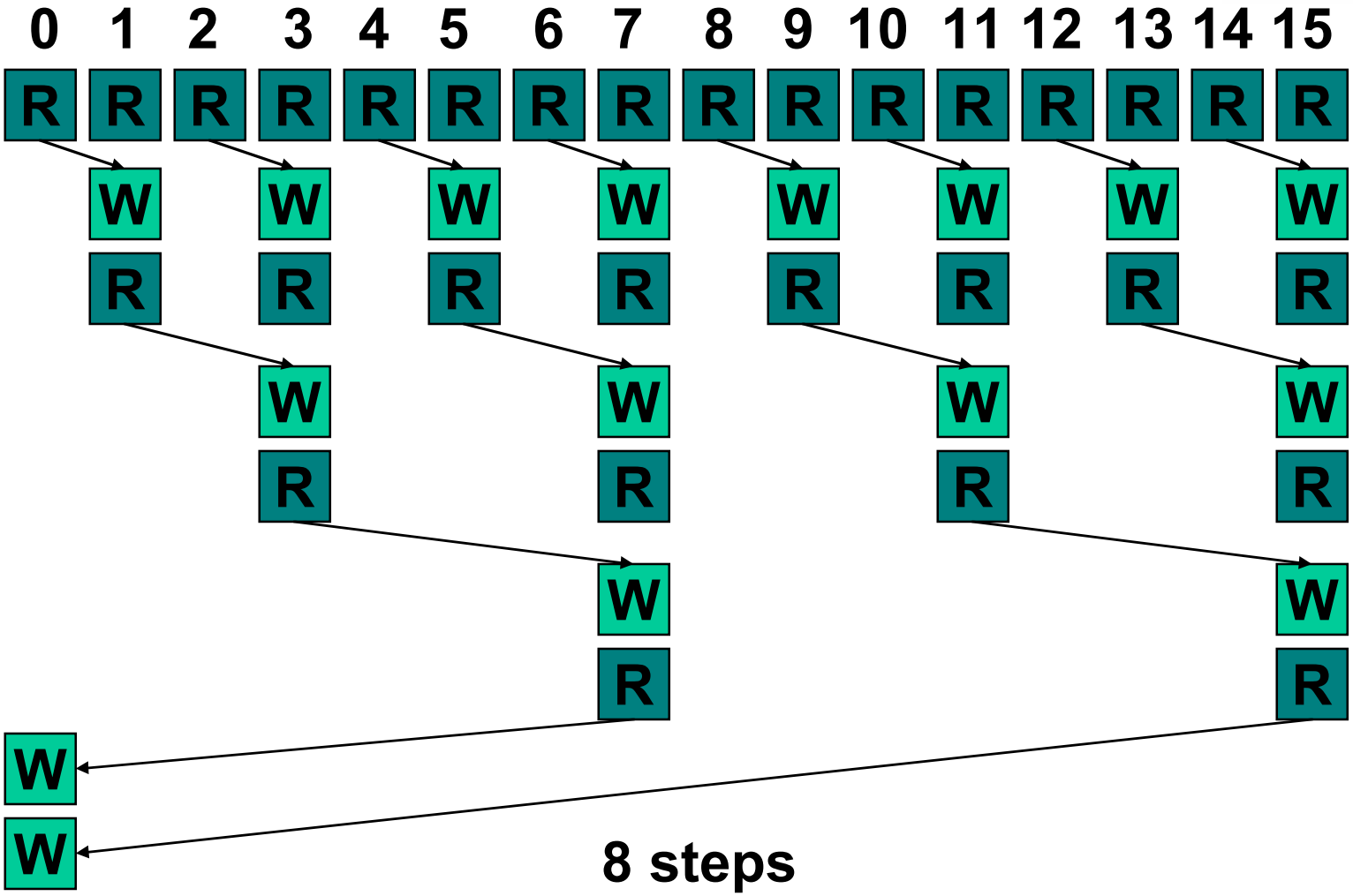
Limitations

- Read/Write Pixel Rate has significant effect on frame rate
- Larger the window size, slower the frame rate

Composition – Linear



Composition – Subdivision



Sample Code

- Available on OpenGL Volumizer 1.1 release

  /usr/share/Volumizer/src/apps/Multipipe

Sample Code Example

```
mvInitMonster(numPipes, pipeList, winX, winY);
mvLoadVolume(volume, x, y, z, MV_TRUNCATE);
mvStartMonster();
mvRenderVolume();

mvClippingPlaneOff();
mvClippingPlaneOn();
mvSetLut(table);
```

Volume Roaming

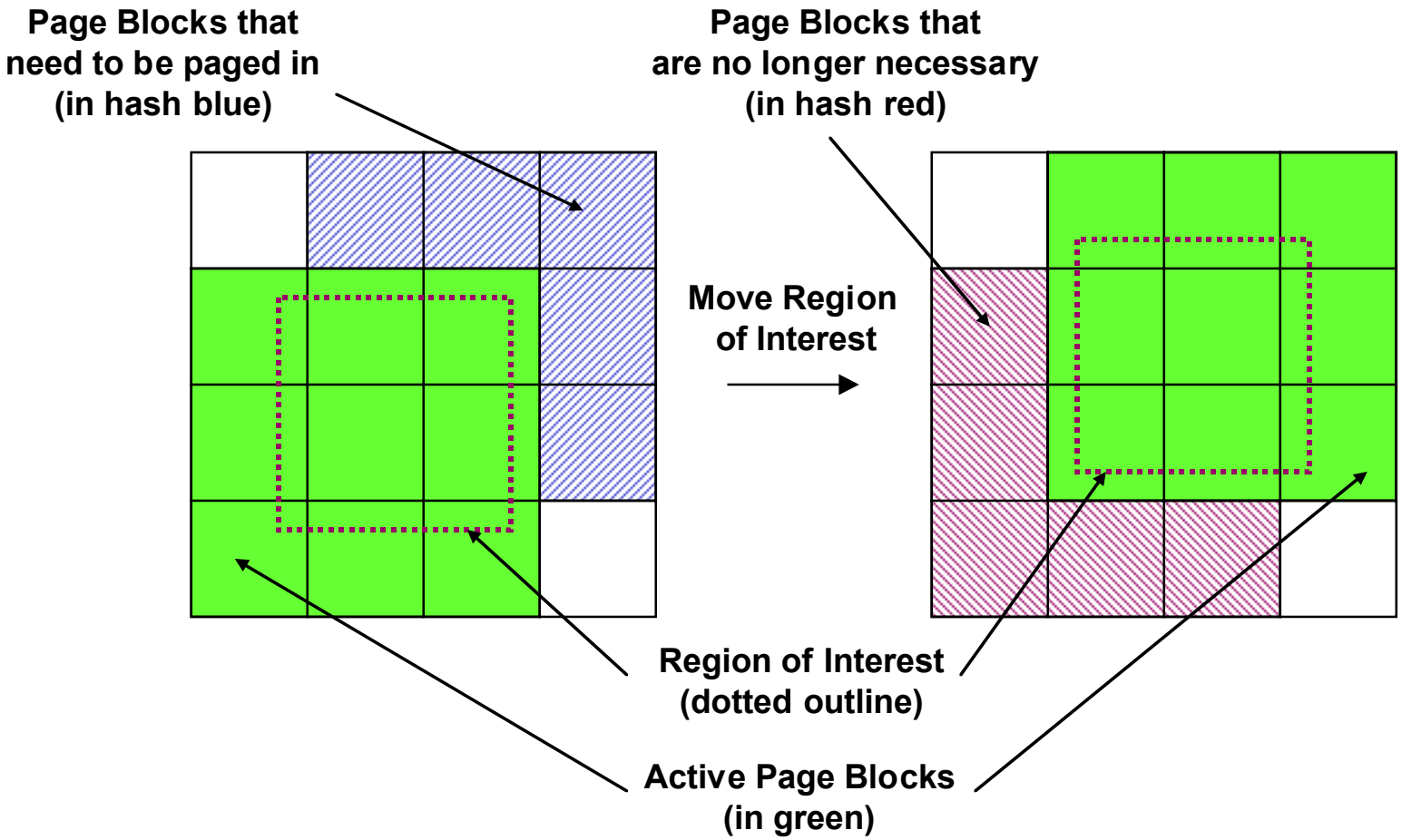
Topics

- Page based Volume Roaming
- SubLoad based Volume Roaming

Note: All explanations will be done in 2D, but the implementation is done in 3D.

Volume Roaming – Page Based

Page based Volume Roaming



Volume Roaming – Page Based

Pros

- Smooth traversal through volume.
- No directional bias when traversing through the volume.

Volume Roaming – Page Based

Cons

- Visible volume is only a fraction of volume loaded in texture memory.
- Diagonal movements faults a significant portion of the cached volume.
- Data must be reformatted into “bricks”.
- Small bricks must be used to keep bandwidth requirements low.

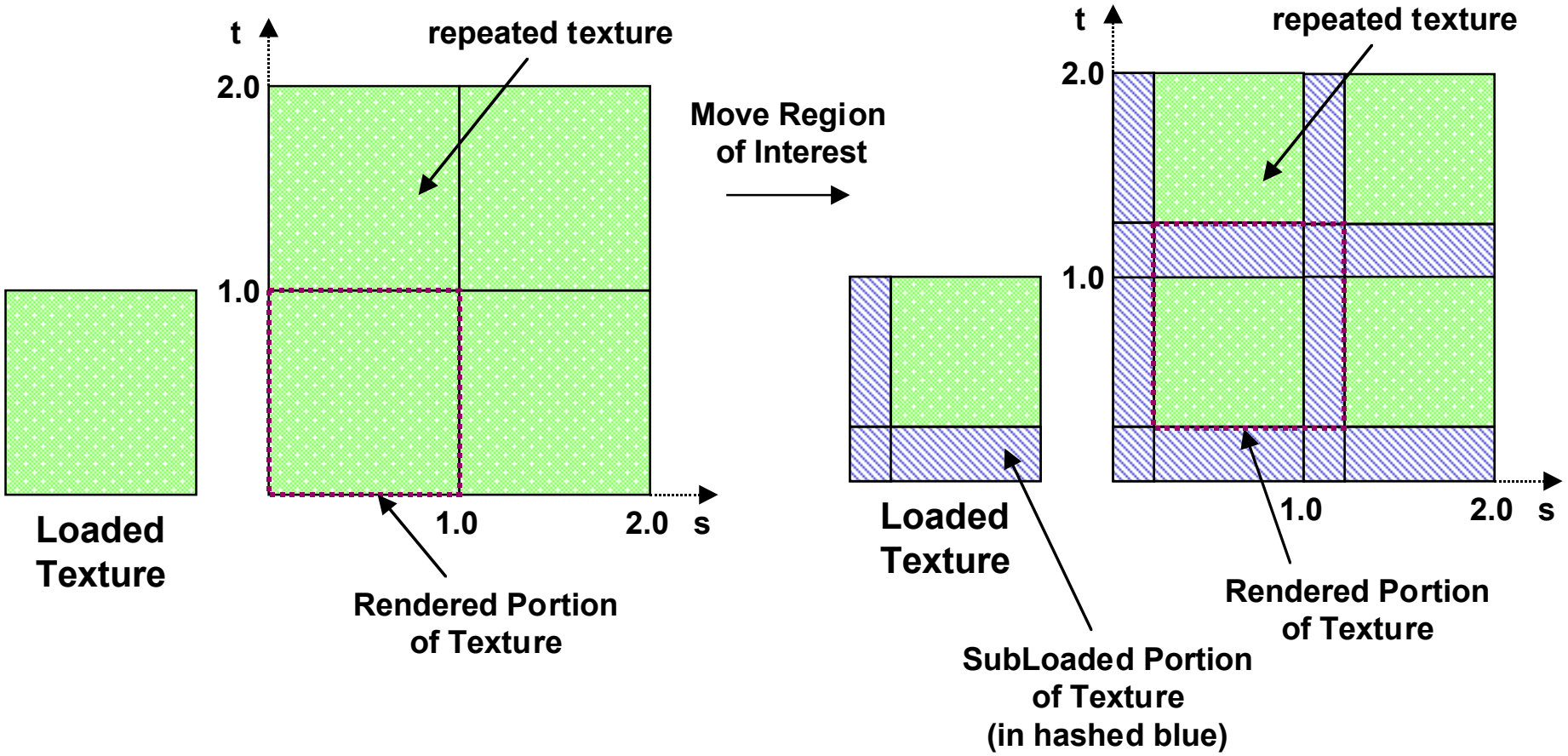
Volume Roaming – Page Based

Cons (cont.)

- Small bricks requires more tetrahedra, therefore increases polygonization time when using OpenGL Volumizer.
- Small bricks decrease download bandwidth on Onyx2 Infinite Reality.

Volume Roaming – SubLoad Based

SubLoad based Volume Roaming



Volume Roaming – SubLoad Based

Pros

- Smooth traversal through volume.
- Entire volume in texture memory is visible.
- Only new part of the volume is uploaded into texture memory.
- No reformatting of volume necessary (but maybe beneficial for disk-based volume roaming).
- Minimal number of tetrahedras needed when using OpenGL Volumizer.

Volume Roaming – SubLoad Based

Pros (cont.)

- Minimal number of tetrahedras needed when using OpenGL Volumizer.

Volume Roaming – SubLoad Based

Cons

- Directional bias when traversing through volume.
- Small subloads decrease download bandwidth on Onyx2 Infinite Reality (but on the otherhand, less bandwidth is needed for smooth traversal).

Performance Determination

Topics

- Performance Criteria
- Example: Volume fitting in Texture Memory
- Example: Volume larger than Texture Memory
- Example: Monster Mode Volume Rendering
- Key Features

Performance Determination

Performance Criteria

- Texture Fill-Rate
- Texture Download Rate
- Texture Memory Size

Performance Determination – Example

Volume fitting in Texture Memory

- Limiting factor is Texture Fill-Rate

$$\text{Frame Rate} = \frac{\text{Fill-Rate}}{\text{Volume Size} \times \text{Scale}^2}$$

Performance Determination – Example

Volume fitting in Texture Memory

- Onyx2 Infinite Reality – 1 pipe (4 RMs)
- 64 MV volume (512x512x256)
- Scale = 1.0

$$\text{Frame Rate} = \frac{450 \text{ Mpixels/s}}{64 \text{ Mvoxels} \times 1.0^2} = 7 \text{ fps}$$

Performance Determination – Example

Volume larger than Texture Memory

- Limiting factor is Texture Download Rate

$$\text{Frame Rate} = \frac{\text{Texture Download Rate}}{\text{Volume Size} \times \text{Scale}^2}$$

Performance Determination – Example

Volume larger than Texture Memory

- Onyx2 Infinite Reality – 1 pipe (4 RMs)
- 128 MV volume (512x512x512)
- Scale = 1.0

$$\text{Frame Rate} = \frac{240\ \text{MB/s}}{128\ \text{MV} \times 1.0^2} = 2\ \text{fps}$$

Performance Determination – Example

Monster Mode Volume Rendering

- Screen Decomposition
- Scales Fill–Rate
- Texture Memory does not scale.

$$\text{Frame Rate} = \frac{\text{Fill-Rates x No. of Pipes}}{\text{Volume Size x Scale}^2}$$

Performance Determination – Example

Monster Mode Volume Rendering

- Data Decomposition
- Scales Fill-Rate
- Scales Texture Memory.

$$\text{Frame Rate} = \frac{\text{Fill-Rates x No. of Pipes}}{\text{Volume Size x Scale}^2}$$

Texture Memory = No. of Pipes x Texture Memory Size

Performance Determination – Example

Monster Mode Volume Rendering

- Onyx2 Infinite Reality – 16 Pipes (4 RMs)
- 1 GV volume
- Data Decomposition

$$\text{Frame Rate} = \frac{450 \times 16}{1\ \text{GV} \times 1.0^2} = 7\ \text{fps}$$

Texture Memory = 16 x 64 MB = 1024 MB = 1 GB

Performance Determination

Key Features

- 3D Texture Mapping
- Texture Look Up Table (TLUT)


sgi™
The solution is in sight.