Efficient Sparse Voxel Octrees

Thomas Breekveldt and Gerben van Veenendaal

April 11th 2013
Outline

Voxel representation

Shading attributes

Hierarchy traversal

Optimizations

Results

Conclusions
Voxel representation

Why voxels?
Voxel representation

Straight forward visualisation: cubes.
Voxel representation

Straight forward visualisation: **cubes**.

Undersampling produces clear artifacts.
Voxel representation

Using cubes increases the surface area significantly.

Level 1

Level 2

Level 3
Use a \textit{contour} instead.
Contours

Further increase detail through use of hierarchical structure.

Level 1

Level 2
Contours

Further increase detail through use of hierarchical structure.
Comparison
Voxel storage

How are voxels being stored?
Contour storage

Store contour using 32 bits:

- $3 \times 6$ bits for the **normal**.
- $2 \times 7$ bits for the **position**.
Voxel storage

For a voxel we need:

- Pointer to children.
- Contour information.
Voxel storage

Problem:

At a parent-voxel decisions need to be made using information from children.

Solution:

Basically let a parent store all the information of its children.
Voxel storage

Store voxel in 64 bits using this **child descriptor**.

```
<table>
<thead>
<tr>
<th>child pointer</th>
<th>far</th>
<th>valid mask</th>
<th>leaf mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>contour pointer</th>
<th>contour mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>8</td>
</tr>
</tbody>
</table>
```
Voxel data structure

Store all the voxels in large memory blocks.
Shading attributes

- Phong shading is used.
- So per voxel, we want to store a color and a normal.
- We want to minimize attribute storage.
- Shading attributes are accessed infrequently.
- So the attributes are aggressively compressed.
Color compression

- For color compression, **DXT1** is used.
- Colors are compressed in blocks of 16 colors.
- Two consecutive *child descriptors*: $2 \times 8$ colors.
DXT1 color compression

- **DXT1** stores two color values, $c_0$ and $c_1$ in RGB-565 encoding.
- From these colors, a set of four colors is created by linear interpolation: \( \{ c_0, c_1, \frac{2}{3} c_0 + \frac{1}{3} c_1, \frac{1}{3} c_0 + \frac{2}{3} c_1 \} \)
- The 16 colors use a 2 bit index to reference a color in this set.
- This makes for a total of 
  \[ 2 \times (5 + 6 + 5) + 16 \times 2 = 64 \text{ bits per block}. \]
DXT1 color compression

Figure: DXT1 compression of two blocks of 16 colors.
Normal compression

- Normals are compressed in blocks of 16 normals.
- Normals are defined by a **base normal** $n_b$ and two **grid normals**, $n_u$ and $n_v$.
- The *base normal* lies on the unit-cube.
- $n = n_b + c_u \times n_u + c_v \times n_v$
- Per normal, the coefficients $c_u$ and $c_v$ are chosen from the set $\{-1, -\frac{1}{3}, \frac{1}{3}, 1\}$.

![Diagram](image)

$n_b$, $n_u$, $n_v$
Normal storage

- The *base normal* uses 32 bits of storage, the *grid normal* each 16 bits.
- Per normal, 2 bits are used per grid coefficient.
- This makes a total of $32 + 2 \times 16 + 16 \times 2 \times 2 = 128$ bits.

<table>
<thead>
<tr>
<th>base</th>
<th>face</th>
<th>face x</th>
<th>face y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>u</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Hierarchy traversal

- Most of the time is spent in hierarchy traversal.
- So we want to optimize this as much as possible.
- We can represent a child by a parent node and three axis bits.

```
+---+---+---+
| 01 | 11 |
+---+---+---+
| 00 | 10 |
+---+---+---+
```
Node intersection

- Per node we test for an intersection of the ray with the *contour*.
- We represent the ray as a function of $t$: $p_t = o + t \times d$, with $o$ the origin and $d$ the direction.
- Intersection gives us a range of $t$. 
Traversals operations

We have 3 tree traversal operations:

- **Push**: Pushes the child onto the stack the ray enters first.
- **Advance**: Advances to the sibling the ray enters next.
- **Pop**: Pops the node from the stack, and goes to parent sibling the ray enters next.
Traversals operations

- Push 00
- Advance 01
- Pop 11

Hierarchies traversal optimisations results conclusions
Traversing algorithm
Beam optimization

Accelerate ray casting process.
Beam optimization

Make a very coarse depth map using the cubic voxels.

This only works up to the voxel level where the coarse sampling is oversampling the voxel set.
Beam optimization

With proper sampling we can make assumptions about the depth.
Beam optimization

We can’t if the coarse sampling is mis-aligned.
Comparison

This allows rays to skip the majority of the empty space. Resulting in a lower amount of iterations inside the octree.
Post-process filtering

Smooth out blockiness caused by discrete sampling of shading attributes.
Post-process filtering

A normal blur-filter will also blur the silhouette.

Which is undesired.
Post-process filtering

This is achieved by varying the radius of the blur.

The *most reliable* way to estimate the proper radius: Look at the size of the intersected voxel on the screen.
Scenes
Results

Test hardware:

- NVIDIA Quadro FX 5800 (4GB)
- Q9300 Intel Core2 Quad
- 4 GB RAM
- Windows XP Professional
## Results

<table>
<thead>
<tr>
<th>Scene</th>
<th>resolution</th>
<th>triangle caster ($M$rays/s)</th>
<th>cubical voxels ($M$rays/s)</th>
<th>contours ($M$rays/s)</th>
<th>cont. w/beam ($M$rays/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>City</td>
<td>512×384</td>
<td>46.7</td>
<td>45.1</td>
<td>79.9</td>
<td>88.6</td>
</tr>
<tr>
<td></td>
<td>1024×768</td>
<td>68.5</td>
<td>54.3</td>
<td>89.1</td>
<td>106.0</td>
</tr>
<tr>
<td></td>
<td>2048×1536</td>
<td>77.1</td>
<td>63.9</td>
<td>97.4</td>
<td>123.8</td>
</tr>
<tr>
<td>Sibenik</td>
<td>512×384</td>
<td>64.3</td>
<td>38.7</td>
<td>80.0</td>
<td>82.5</td>
</tr>
<tr>
<td></td>
<td>1024×768</td>
<td>94.1</td>
<td>46.5</td>
<td>94.1</td>
<td>103.6</td>
</tr>
<tr>
<td></td>
<td>2048×1536</td>
<td>107.1</td>
<td>55.1</td>
<td>103.9</td>
<td>122.0</td>
</tr>
<tr>
<td>Sibenik-D</td>
<td>512×384</td>
<td>–</td>
<td>24.8</td>
<td>32.6</td>
<td>37.5</td>
</tr>
<tr>
<td></td>
<td>1024×768</td>
<td>–</td>
<td>30.2</td>
<td>38.7</td>
<td>48.3</td>
</tr>
<tr>
<td></td>
<td>2048×1536</td>
<td>–</td>
<td>37.1</td>
<td>43.6</td>
<td>60.9</td>
</tr>
<tr>
<td>Hairball</td>
<td>512×384</td>
<td>11.6</td>
<td>22.4</td>
<td>24.1</td>
<td>24.1</td>
</tr>
<tr>
<td></td>
<td>1024×768</td>
<td>20.5</td>
<td>22.5</td>
<td>27.9</td>
<td>28.4</td>
</tr>
<tr>
<td></td>
<td>2048×1536</td>
<td>31.2</td>
<td>29.2</td>
<td>36.5</td>
<td>38.2</td>
</tr>
<tr>
<td>Fairy</td>
<td>512×384</td>
<td>63.9</td>
<td>62.1</td>
<td>128.2</td>
<td>132.6</td>
</tr>
<tr>
<td></td>
<td>1024×768</td>
<td>125.1</td>
<td>69.4</td>
<td>145.4</td>
<td>150.9</td>
</tr>
<tr>
<td></td>
<td>2048×1536</td>
<td>155.8</td>
<td>78.6</td>
<td>160.4</td>
<td>169.2</td>
</tr>
<tr>
<td>Conference</td>
<td>512×384</td>
<td>69.1</td>
<td>35.8</td>
<td>97.9</td>
<td>104.4</td>
</tr>
<tr>
<td></td>
<td>1024×768</td>
<td>111.9</td>
<td>43.8</td>
<td>110.3</td>
<td>124.3</td>
</tr>
<tr>
<td></td>
<td>2048×1536</td>
<td>134.0</td>
<td>52.3</td>
<td>120.2</td>
<td>140.8</td>
</tr>
</tbody>
</table>
Advantages and drawbacks

Advantages:

- Real-time rendering of complex voxel based scenes.
- Unique possibilities in per voxel shading attributes.
- Competitive with triangle-based ray casting.
- Unexplored limits of this method.

Drawbacks:

- Static content only.
- Lossy compression.
- Unexplored limits of this method.

Remarks:

- Hard to compare both data structures.
Wrapping it up

We’ve talked about:

▶ Voxel representation.
▶ Compression and storage of shading attributes.
▶ Efficient hierarchy traversal.
▶ Possible optimizations on rendering.
▶ Rendering results.
▶ Advantages and drawbacks.
Questions?