Terrain Rendering with Geometry Clipmaps

ALEX GOLDBERG
UNIVERSITY OF CALIFORNIA, SAN DIEGO
Presentation Outline

1. Terrain rendering considerations and common approaches
4. Results and analysis
Real time terrain rendering

- Unsolved problem in computer graphics
  - Area of much active research
- Many challenging issues:
  - Near and far features rendered simultaneously
  - Cohesive structure – no intuitive segmentation
  - Large scale feature variations – steep/high frequency mountains adjacent to rolling plains
  - Small scale detail desired at close proximity
• Terrain can make or break an immersive application...
Terrain LOD

- LOD scheme required for terrain rendering
  - Large data set
  - Aliasing of high frequency detail
- Considerations:
  - View dependence
  - Frustum/occlusion culling
  - Memory and bandwidth limitations
  - Textures vs. geometry
- Ideal goal: Adaptive refinement based on screen-space geometry deviation from source data
  - Not directly conducive to rendering pipeline
Terrain Tiling

- Many traditional techniques partition the terrain into convenient sections
  - LOD meshes are adaptively selected or generated on a tile basis
  - Care must be taken to avoid visual seams on edges between tiles with differing LOD levels
- LOD selection is traditionally viewer position-based
  - Not always ideal – consider viewer standing at the intersection of several tiles
Fixed-Size Tiles

- Irregular tiles are possible, but difficult to work with
  - More tile edge dependencies, more complicated culling, etc
  - Can provide significant speed/memory benefits if handled well

- Regular size (typically square) tiles more convenient
  - Fixed number of edge dependencies per tile
  - Position and bounds can be computed solely from index
  - Simplified occlusion and frustum culling
  - Performance highly dependent on tile size
Tile Tessellation

- Complexity of edge handling scheme, memory requirements, and rendering performance are greatly impacted by tile tessellation.

- General consideration: terrain stitching
  - Many precomputed lighting and physics methods require ‘watertight’ and non-overlapping meshes
  - Large, static objects may need to be ‘stitched’ into the terrain tessellation
  - Complicates fixed-size tiles
Heightfields

• Heightfields have several large advantages:
  ○ Simple, regular structure avoids traversal of complex data structures
  ○ Easy to generate, store, and compress
  ○ Interpolation has fixed complexity
  ○ Greatly simplifies collision detection
  ○ Independent from source data

• …and several disadvantages
  ○ Large tradeoff between precision and storage overhead
  ○ Steep features imprecise or costly – overhangs impossible
  ○ Independent from source data
Geometry Clipmaps Defined

- Uniformly tessellated meshes represent a set of nested, regular grids
- Vertex positions are obtained from a sampled heightfield
- Each level contains $n \times n$ vertices
  - Half the sampling frequency for each successive level
Choosing $n$

- Goal is to have approximately uniform triangle size in screen space.
- Triangle screen size varies proportionally with distance from the viewer.
  - Assume right-angled triangles of size $g_L$ and horizontal view direction
  - If $(0.4)ng_L$ is the average screen space depth over all viewing directions, then screen space triangle size is given by:

$$s = \frac{g_L}{(0.4)ng_L} \frac{W}{2 \tan \frac{\phi}{2}} = (1.25) \frac{W}{n \tan \frac{\phi}{2}}$$
Clipmap Regions

- Each clipmap has three conceptual regions:
  1. Active Region: The extent of the data we’d like to render
  2. Clip Region: The extent of the data actually stored in the mesh
  3. Render Region: The region that will be drawn by the clipmap

- Every frame, the clip region is shifted towards the active region and its vertex data is updated
Rendering Algorithm Overview

• Each frame:
  1. Determine ‘active’ region of each clipmap level
  2. Update clipmap meshes to account for viewer motion
  3. Crop clipmap regions to active regions
  4. Render meshes
Active Region Determination

- The active region of each clipmap level $L$ with grid spacing $g_L$ is defined to be the square $n g_L \times n g_L$ region centered at the viewer’s $xy$ position.
Clipmap Update

- Clipmaps are updated starting with level 0 (coarsest)
- Clipmap data is updated toroidally as the viewer moves
  - Active region wraps around edges of vertex data
  - Data farthest behind viewer is overwritten
  - No translation required for remaining data
- Both vertex and index buffers updated on CPU
Clipmap Cropping

- Ideally, the clip region is shifted to coincide with the active region every update
- If viewer motion is too rapid to fully update, clip region may fall behind
  - Clip region is then clipped against active region
  - Data for the resulting gap still lies in the next lower clipmap (at half resolution)
Clipmap Rendering

- Rendering begins with the highest clipmap containing a non-empty clip region
  - Clipmaps are rendered fine to coarse to exploit hardware occlusion
- Clipmaps are drawn as four quads using batched triangle strips
  - Quads culled against view frustum
Transition Regions

- Without consideration for the clipmap borders, the terrain will exhibit geometry gaps and T-vertices
- To avoid this, vertices near a clipmap’s border have heights linearly interpolated to the edge of the next lower clipmap
Normal Map

- Each clipmap level has an associated normal map
- To avoid storing a separate mipmap pyramid per level, normal map boundaries are blending in a similar manner as for vertex height blending
  - (at the expense of screenspace filtering quality)
Terrain Compression

- Simple pyramid compression scheme allows for terrain compression using interpolatory subdivision
- High frequency residuals either included in compression or synthesized with procedural noise
- (More later...)
## Error Analysis

<table>
<thead>
<tr>
<th>Level</th>
<th>Puget Sound</th>
<th>U.S.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rms($\hat{e}_i$)</td>
<td>$P_{999}(\hat{e}_i)$</td>
</tr>
<tr>
<td>1</td>
<td>0.12</td>
<td>0.58</td>
</tr>
<tr>
<td>2</td>
<td>0.14</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>0.15</td>
<td>0.86</td>
</tr>
<tr>
<td>4</td>
<td>0.15</td>
<td>0.93</td>
</tr>
<tr>
<td>5</td>
<td>0.14</td>
<td>0.96</td>
</tr>
<tr>
<td>6</td>
<td>0.13</td>
<td>0.94</td>
</tr>
<tr>
<td>7</td>
<td>0.11</td>
<td>0.83</td>
</tr>
<tr>
<td>8</td>
<td>0.11</td>
<td>0.75</td>
</tr>
<tr>
<td>9</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Analysis of screen-space geometric error, in pixels. Columns show rms, $99.9^{th}$ percentile, and maximum errors. ($n=255$, $W=640$, $\varphi=90^\circ$, i.e. $s=3$).
GPU-Based Clipmaps

- Previously, incremental vertex and triangle buffer updates performed entirely on CPU
  - CPU will already be loaded with other tasks (physics, audio, compression, AI, etc.)
  - Transferring mesh data between system memory and video memory is a huge bottleneck
- Highly parallel tasks can be solved much faster on the GPU
Clipmap GPU Support

- DirectX 10 introduced a standard method for GPU output of mesh data
  - Relying heavily on DX10 features still unwise at this point…
- However, render-to-texture support is widespread
- Vendor-specific functionality in previous generation of graphics cards
  - nVidia provided texture lookup in the vertex shader since GeForce 6
  - ATI instead supported rendering to vertex buffers
- Geometry clipmaps adapted to GPU in 2005
  - Relies on nVidia’s vertex textures
GPU Clipmap Algorithm

- General approach to (mostly) eliminate CPU from pipeline:
  - Data for each clipmap level stored as a 2D texture heightfield
  - Clipmap texture is sampled in VS to compute vertex heights
  - Normal map, upsampling, and residual synthesis also on GPU

<table>
<thead>
<tr>
<th></th>
<th>Original Implementation</th>
<th>GPU-Based Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elevation Data</td>
<td>In vertex buffer</td>
<td>In 2D vertex texture</td>
</tr>
<tr>
<td>Vertex Buffer</td>
<td>Incrementally updated by CPU</td>
<td>Constant!</td>
</tr>
<tr>
<td>Index Buffer</td>
<td>Generated every frame by CPU</td>
<td>Constant!</td>
</tr>
<tr>
<td>Upsampling</td>
<td>CPU</td>
<td>GPU</td>
</tr>
<tr>
<td>Decompression</td>
<td>CPU</td>
<td>CPU</td>
</tr>
<tr>
<td>Synthesis</td>
<td>CPU</td>
<td>GPU</td>
</tr>
<tr>
<td>Adding Residuals</td>
<td>CPU</td>
<td>GPU</td>
</tr>
<tr>
<td>Normal-Map Update</td>
<td>CPU</td>
<td>GPU</td>
</tr>
<tr>
<td>Transition Blends</td>
<td>GPU</td>
<td>GPU</td>
</tr>
</tbody>
</table>
Clipmap Size

- In order for clipmaps to be properly nested, n must be odd
- Since hardware may be optimized for power-of-two textures, n is chosen to be $2^k - 1$
  - One row & column of texels unused
- Clipmaps will never be exactly centered, allowing finer levels to shift without a coarse update
Vertex and Index Buffers

- For efficiency and improved frustum culling, the bulk of the geometry is drawn with square blocks.
- Since the blocks always have the same number of vertices, a single square mesh can be reused within and across all clipmap levels.
Frustum Culling

- As before, clipmap frustum culling is performed per mesh subset
- Improved granularity due to smaller block size
**Vertex Shader**

- The VS computes vertex xy values using a simple scale/translation, and samples z values with a one-to-one vertex/texel mapping

- Transition regions computed similarly as before
  - Simple equation to compute interpolation parameter $\alpha$
  - Computed height $z' = (1-\alpha)z_f + \alpha z_c$ where $z_f$ is height for current level and $z_c$ is height value for next lower (coarser) level

- Naïve computation of $z'$ requires three vertex texture lookups (expensive!)
  - Instead, $z_f$ and $z_c$ are computed in the update phase.
  - $z_f$ stored in integer portion of float, and $(z_f - z_c)$ stored in fractional part
Pixel Shader

- PS simply samples normal map and performs shading
- Normals blended between fine and coarse samples using similar approach as for height blending
Clipmap Update

- Clipmaps are incrementally updated in a pixel shader using render-to-texture functionality
- 2 – 4 quads are rendered into the heightfield
- No longer need to worry about partial updates and active/clip region clipping
Update Step 1: Upsampling

- Finer level geometry is upsampled in from coarser geometry using interpolatory subdivision scheme
  - C1 smooth, four-point subdivision curve interpolant [Kobbelt, 1996]
- Requires between 1 and 4x4 texture samples.
  - To avoid branching, 16 samples are always taken and a lookup in a 2x2 texture chooses the appropriate mask
Update Step 2: Residuals

- Residuals can be decompressed from a lossy encoded form on the CPU
- GPU synthesized noise can also be used for high frequency details
  - Sampling of a precomputed noise texture with slightly magnified texture coordinates to break periodicity
Update Step 3: Normal Map

- Normals computed using simple cross product of grid samples
- Could potentially eliminate normal map by computing normals in the VS, but vertex textures don’t support bilinear filtering (until DX10) and multiple lookups are expensive
- Can also use noise or precomputed data to yield normal map with much higher sampling frequency than geometry.
Results & Impressions

(images & movies)

<table>
<thead>
<tr>
<th></th>
<th>Previous Implementation*</th>
<th>GPU-Based Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upsampling</td>
<td>3 ms</td>
<td>1.0 ms</td>
</tr>
<tr>
<td>Decompression</td>
<td>8 ms</td>
<td>8 ms**</td>
</tr>
<tr>
<td>Synthesis</td>
<td>3 ms</td>
<td>~0 ms</td>
</tr>
<tr>
<td>Normal-Map Computation</td>
<td>11 ms</td>
<td>0.6 ms</td>
</tr>
</tbody>
</table>

**Still on the CPU.