Efficient Tree Storage and Evaluation for Level of Detail

Gary Jackson
garyj@cs.umd.edu
Today’s Theme

- LoD information can be stored in trees
- Trees are expensive to store and evaluate
- How can we mitigate this cost?
Sequential Point Trees

Carsten Dachsbacher, Christian Vogelgsang, Marc Stamminger

Presented by Gary Jackson
Outline

- Introduction
- Sequential Point Trees
- Implementation and Results
- Conclusion
Introduction
Point Based Rendering

• Idea: instead of rendering sub-pixel triangles, just render points
• No topological information
  • Adjust detail by adding or removing points
• Need to avoid holes
• Focus on point rendering by the GPU
Previous Work

- Q-splats [Rusinkiewicz and Levoy 2000, 2001]
  - Hierarchical, bounding sphere point based rendering
- Hybrid Rendering [Cohen et al. 2001], [Chen and Nguyen 2001]
  - Render mixed points and triangles
Previous Work

- Sorting points back to front [Coconu and Hege 2002]
- Efficient software-based renderer for points [Botsch et al. 2002]
Previous Work

• Random points for fast rendering
  • [Wand et al. 2001]
  • [Stamminger and Drettakis 2001], Dussen et al. 2002]

• Point splatting on the GPU [Ren et al. 2002]

• Point clouds of discrete detail levels [Pauly et al. 2002]
Sequential Point Trees
Point Tree Hierarchy

- Octree
- Single object of uniform color
- Each node: center point $p$, average normal $n$, bounding sphere diameter $d$
- Internal nodes represent union of children
- Leaves should be spaced equally
Perpendicular Error

- $e_p$: Minimum distance between two planes parallel to the enclosing disk

- $\tilde{e}_p$: Image space error

- Dependent on view angle and distance
Tangential Error

• Fit slabs around projected child disks

• $e_t$: diameter of the disk minus the width of the tightest slab

• $\tilde{e}_t$: image space error
Geometric Error

• $e_g$: combination of perpendicular and tangential error

• $\tilde{e}_g$: image space error, not dependent on view angle

  • $\tilde{e}_g = e_g/|v| = e_g/r$

• Maximum is bounding sphere diameter $d$

• When we let $e_g=d$, we get QSplat
Recursive Rendering

• When rendering a node:
  • Let $\tilde{e} := \tilde{e}_g$ or $\tilde{e}_p + \tilde{e}_t$
  • If $\tilde{e}$ exceeds $\epsilon$, recurse
  • Else render the point
  • If $\epsilon$ equals one pixel, all sub-pixel detail is hidden
Benefits of Rendering This Way

- Reflects surface properties
- Large flat regions have small $e_g$ and are rendered as large splats
- Geometrically or visually complex areas get small splats
Sequentialization

- $r$: distance from viewer
- Pre-compute two values for each node
  - $r_{\text{min}}$: error is too great
    - Let $r_{\text{min}} := \frac{e_g}{\varepsilon}$
  - $r_{\text{max}}$: detail is too far to be discerned
- Render point if $r \in [r_{\text{min}}, r_{\text{max}}]$
Picking $r_{\text{max}}$

- Let $r_{\text{max}} := r_{\text{min}}$ of parent?
- Only works when $r$ is constant
- If $r$ varies, we could end up with holes
- Add overlap: distance from parent to child
- Some overdraw, but not a big deal
- So now we can express nodes as a non-hierarchical list
Rearrangement

- Sort list of nodes by $r_{\text{max}}$
- Compute the minimum possible $r$ from the object bounding box
- Throw out everything with $r_{\text{max}} < \min\{r\}$
  - GPU only needs to work on a prefix of the list
View Dependent

• List cull based on $\tilde{e}_g$ as presented
• Render based on more expensive $\tilde{e}_p + \tilde{e}_t$
• Works because $\tilde{e}_p + \tilde{e}_t$ is bound by $\tilde{e}_g$
Rearrangement

- For constant $r$, algorithm cleanly cuts the tree
- For varying $r$, multiple nodes in a hierarchy may be selected
- Results in a fuzzy boundary of the hierarchy tree
Hybrid Rendering

- Render triangles when the longest side $s$ exceeds the error threshold
  - $s/r \leq \varepsilon$
  - let $r_{\text{max}} := s/\varepsilon$

- Each triangle gets a subtree of points
  - Prune when child $r_{\text{max}} <$ triangle $r_{\text{max}}$
Hybrid Rendering

- Triangles are rendered, some partially
- Partially rendered triangles will also get some child points rendered
- Resorting triangle list breaks up triangle strips and orders optimized for cache hits
Color, Texture, and Material

- Every node in the tree gets a color based on object color and texture
- Inner nodes get averaged color from children

Figure 7: By including color into the error measure, point densities adapt to texture detail. Left: uniform small point size to visualize point densities, right: correct point sizes.
Color, Texture, and Material

• Small geometric error causes large regions to be rendered by large splats

• This washes out color differences

• Fix: increase point error to point diameter where the color differs significantly
Normal Clustering

- Recursively subdivide an octahedron into 128 regions
- Split the sequential node list into the corresponding regions, by normal
- Back face culling
  - Some of those 128 normals will be facing away from the camera
  - Don't process those lists
Normal Clustering

- Increased load on the CPU
- Smaller point lists on the GPU
- Point reduction in 50% more than makes up for the difference
Implementation and Results
Implementation

• Hardware accelerated point primitives is fastest

• Textured splats using programmable pixel shaders failed

• Textured triangles works, but is slow
Results

- Rendered with opaque squares
- 77 million points per second to the GPU, 50 million rendered after culling
- 36 to 90 fps
- CPU load of 5-15%
- ATI Radeon 9700, 2.4GHz Pentium
Results
Proposed Extension

- Gaussian, ellipsoidal splats are expensive
  - z-buffer test with tolerance, or two-pass rendering
- Solution
  - ellipsoidal gauss textures rendered as point sprites
  - blend and renormalize if source fragment is close, depth-wise, to the destination fragment
Conclusion

- Sequential point trees are efficient for GPU rendering
- Fuzzy splats did not work well
- Could be improved by better support for point primitives from drivers and GPUs
Compressed Random Access Trees for Spatially Coherent Data

Sylvain Lefebvre and Hughes Hoppe

Presented by Gary Jackson
Outline

• Introduction
• Subdivision
• Compressed Tree Topology
• Compressed Tree Data
• Applications and Results
• Analysis and Discussion
• Conclusion
Introduction
Coherent Spatial Data

• Distance fields are continuous
• Light maps are smooth except near shadow boundaries
• Alpha mattes are constant except near silhouettes
• HDR images have broad regions of similar luminance
Compressing Coherent Spatial Data

- Very good (JPEG2000, etc.)
- However, it is not random access
- This is not so great when streamed processing needs random access
Block-based Compression

- Random access
- Lack prediction of low-frequency variation
- Uniform bit rate
- High detail is lost
- Waste on low detail
Adaptive Hierarchies

- Multiresolution prediction
- Spatial adaptivity
- Require sequential traversal
Subdivision
Dual Subdivision

- Tree nodes correspond to cells
- Conventional way to divide space
- Hard to interpolate values for pruned levels of the tree
- $3^d$ lookups per level
Primal Subdivision

- Tree nodes correspond to corners
- Nodes at a given level are a superset of nodes at a coarser level
- Easier to interpolate values for pruned levels
- $2^d$ lookups per level
Primal vs. Dual

- Interpolate point D:
- Dual requires lookups to A, B & C
  - and multiquadratic B-spline
- Primal only needs lookups to B & C
Compressed Tree Topology
Conventional Tree

```
struct Node {
    Data data;
    Node * children[2^d];
};
```

- $4 \times 2^d$ bytes per node
Sibling Tree

- For full trees
- 4 bytes per node overhead

```c
struct Node {
    Data data;
    Brood * brood;
};

struct Brood {
    Node nodes[2^d];
};
```
Autumnal Tree

- Can be used when pointer and data size are the same
- \(4 \times 2^{-d} + \frac{1}{8}\) bytes per node for tree topology overhead
- 1.125 bytes/node for a quadtree

```c
struct PointerOrData {
  bit leafchild;
  union {
    Node * pointer;
    Data data;
  }
};
```

```c
struct Node {
  Data data;
  PointerOrData children[2^d];
};
```
Encoded Local Offsets

• Store relative offsets instead of pointers
• Encode offsets as $y = s_l x$
  • $s_l$ is a per-level scaling parameter
  • $x$ is a 7-bit value
• Tree is built from fine to coarse
• Data is a 7 bits
Waste

• Coarse levels are dense, no need for adaptivity
• Traversal adds runtime cost
• Padding space waste
• Vector quantization is ineffective due to small number of data nodes

<table>
<thead>
<tr>
<th>Level $l$</th>
<th>Num. nodes</th>
<th>Scaling $s_l$</th>
<th>Padding (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>500</td>
<td>58</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>316</td>
<td>354</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>135</td>
<td>897</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>61</td>
<td>960</td>
</tr>
<tr>
<td>4</td>
<td>76</td>
<td>26</td>
<td>1016</td>
</tr>
<tr>
<td>5</td>
<td>202</td>
<td>10</td>
<td>555</td>
</tr>
<tr>
<td>6</td>
<td>486</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1228</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>3218</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>8322</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
• First 5 levels (0 .. 4) are stored as a mipmap

• Finest level also stores pointers to the resulting subtrees

• For example from table, storing tree topology only needs 0.36 bytes/node
Compressed Tree Data
Construction of adaptive tree

- Create a complete residual mipmap tree $T$
  - Level zero has actual values
  - Lower levels have residual values relative to higher levels
- Apply brood-based vector quantization to make tree $T'$
- Adaptively prune $T'$ to satisfy tolerance $\tau$
Vector Quantization

• Use k-means (or something like it) to come up with 128 (or 256) centroid vectors

• Per level code books

• Store index in to code book at each level
Adaptive Pruning

• Brood-based
• Interpolate intermediate values
• Does the error from the interpolation create too much error?
• More details in the paper
Tree Evaluation

- Descend down the tree
- Not just one branch, but the four bounding points
- Adding up interpolated residuals
- Interpolate when we're at the desired level or we hit a pruned subtree
Applications and Results
Light Maps/Alpha Mattes

- Light Map: 2.2bpp vs 4bpp for BC4U (at same numerical accuracy)
- Alpha Matte: 0.7bpp vs 4bpp for BC4U (at same numerical accuracy)
Distance Fields

- Modification to pruning:
  - Error can be larger
  - Sign must be preserved
- Error vs. vector not visible at under 10k x 10k pixels

Figure 8: Representation of a vector shape (3.2KB) as a signed-distance field at 1025² resolution using a randomly accessible compressed tree (7.8KB), and its benefits for resolution-independent antialiasing and magnification. A binary image would require 131KB and would not magnify as a smooth shape outline as shown in (g).
High Dynamic Range Images

- Tree-compress log(RGB)
- Conventional compression on actual log(RGB) - compressed log(RGB)
- Result: overall compression gets 5bpp
Texture Atlases

- Basic idea
- Incomplete mipmap
- $k$-means handles undefined data
- Result: 2.05bpp
Color Images

- Good on images with large smooth areas
- Tree is too dense to benefit over traditional block-based approach
Analysis and Discussion
Memory Bandwidth

• 8 accesses per level during evaluation
• Strategies
  • Cache end results
  • Buffer intermediate results of tree evaluation
    • Morton order traversal helps this
Large Datasets

- Must construct entire mipmap first, so construction doesn't scale
- Future work: should be possible to more concisely accumulate error for pruning
- In the mean time, tiling works
Conclusion

• Good for certain types of images
• Bad for most common color images
• Random access?
  • Must traverse tree from root to child to evaluate