


Maintaining Interactivity 2 - Subsampling

aka, the story of my life

Papers

- Frameless Rendering: Double Buffering Considered Harmful [Bishop, et al., SIGGRAPH '94]
 - Interactive Rendering using the Render Cache [Walter, et al. EUROGRAPHICS '99]
 - Qsplat: A Multiresolution Point Rendering System for Large Meshes [Rusinkiewicz and Levoy, SIGGRAPH '01]
-



Frameless Rendering: Double Buffering Considered Harmful

Bishop, Fuchs, McMillan, Scher Zagier
UNC Chapel Hill
SIGGRAPH '94

Frameless Rendering

- Basic idea: improve interactivity by removing latency
 - All samples are rendered with the most up-to-date input
 - Allows smooth motion without needing to finish a whole frame every n milliseconds
-

Frameless Rendering

- Samples updated in a randomized order
 - Crude image when scene is changing, higher quality renderings when the inputs are constant
 - Images appear blurry but have smooth and continuous motion
-

Frameless Rendering

- [videos]
-

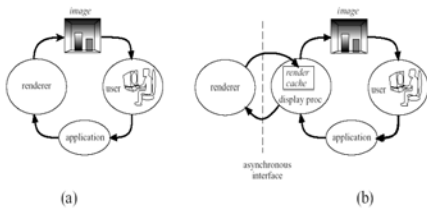
Interactive Rendering using the Render Cache

Walter, Drettakis, Parker
University of Utah
EUROGRAPHICS '99

Render Cache

- Another method of speeding up slow renderers – or at least appearing to do so
- As with frameless rendering, it decouples sample generation from sample display
- Similarly well-suited to interactive raytracing

Render Cache



Render Cache

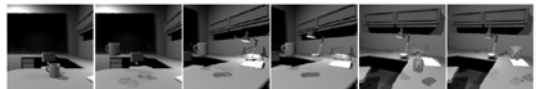
- Uses point sample re-projection and interpolation to respond to updated user input
- Prioritizes updates where samples are sparse
- Uses depth culling to attempt to preserve occlusion even when undersampled

Render Cache

- Basic idea: render to a *point image*, which is just an image that saves the z values at each pixel
- When new input is received, the points making up the previous image are re-projected onto new camera plane

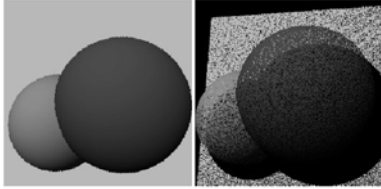
Render Cache

- Example (can you see what's going on here?)



Render Cache

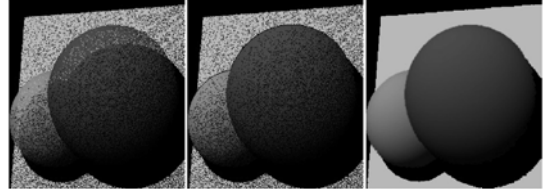
- Example #2 (what about now?)



[Notice the disocclusion artifacts]

Render Cache

- Example #3 – a better reconstruction



Render Cache

- Uses a depth culling stage to remove samples whose depth are inconsistent with their 3x3 neighborhood (e.g., if its depth is more than 10% beyond the average)

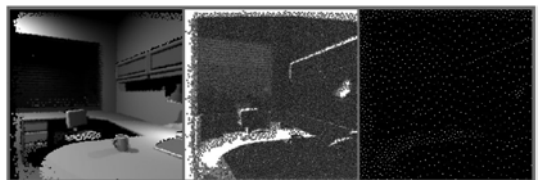
Render Cache

- Does a filtering pass, again on a 3x3 neighborhood
- Performs a weighted average of neighboring samples [weights 4,2,1]

Render Cache

- Must intelligently choose which samples to compute next
- A grayscale *priority image* is constructed and then a diffusion dither is applied to it
- Each pixels's priority is based on the age of the points that map to it
- The dither ensures spatial distribution and concentraion in high priority regions

Render Cache



Render Cache

- Application can supply hints as to which points are likely to need resampling soon
- Eg: moving objects and their shadows, specular highlights

Render Cache

- The cache is fixed-size and just slightly larger than the number of pixels to be displayed
- When resampling, new points overwrite the old ones at the same location
- Ensures stale data will eventually go away and keeps computational cost low


Render Cache

Initialize buffers	0.0046 secs
Point projection	0.0328 secs
Depth cull	0.0085 secs
Interpolation	0.0139 secs
Display image	0.0027 secs
Request new samples	0.0053 secs
Update render cache	0.0027 secs
Total time	0.0705 secs

Table 1. Timings for the display process' generation of a 256x256 image produced on a single 195Mhz R10000 processor. The display process is capable of producing about 14 frames per second in this case, though the actual framerate may be slower if part of the processors time is also devoted to rendering.

Render Cache

- [video]



Qsplat: A Multiresolution Point Rendering System for Large Meshes

Qsplat

- Addresses the opposite problem – too *many* samples (e.g., 100M – 1B)
- Organizes samples into a bounding sphere hierarchy that represents the data re-sampled at multiple resolutions

QSplat

■ Rendering Algorithm:

```
TraverseHierarchy(node)
{
  if (node not visible)
    skip this branch of the tree
  else if (node is a leaf node)
    draw a splat
  else if (benefit of recursing further is too low)
    draw a splat
  else
    for each child in children(node)
      TraverseHierarchy(child)
}
```

QSplat

- Visibility culling: frustum and backface culling
- If a leaf node is reached or recursion is not beneficial, the current sphere is splatted onto the screen
- Recursion depth based on projected size of splat on screen

QSplat

■ Preprocessing Algorithm:

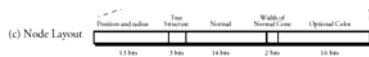
```
BuildTree(vertices(begin..end))
{
  if (begin == end)
    return Sphere(vertices(begin))
  else
    midpoint = PartitionAlongLongestAxis(vertices(begin..end))
    leftsubtree = BuildTree(vertices(begin..midpoint))
    rightsubtree = BuildTree(vertices(midpoint+1..end))
    return BoundingSphere(leftsubtree, rightsubtree)
}
```

QSplat

- Preprocessing begins with a triangle mesh
- Triangles only used for computing sphere sizes and normals; connectivity is thrown away
- Does recursive spatial subdivision
- Nodes combined to give average branching factor of about 4
- Overall process is very fast

QSplat

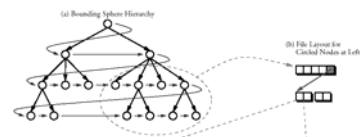
■ How to be miserly with bits:



■ Quantization, quantization, quantization

QSplat

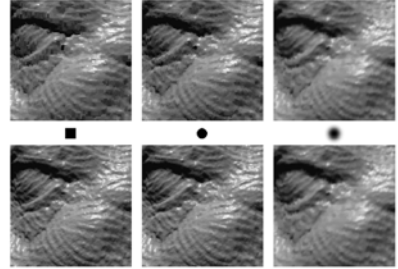
- Tree is arranged on disk and in memory in breadth-first order for fast loading and traversal



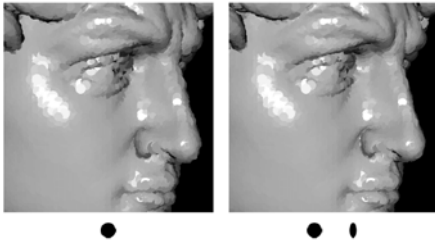
QSplat

- Splat shape can be determined by user
- ...this is largely a time/quality tradeoff
- Examples:
 - OpenGL points (squares)
 - Textured polygons
 - Fuzzy alpha-blended spot
 - Circles or ellipses

QSplat



QSplat



QSplat



Figure 5: Comparison of renderings using point and polygon primitives.

QSplat



Typical performance	David's head, 1mm, color		David, 2mm		St. Matthew, 0.25mm	
	Interactive	Static	Interactive	Static	Interactive	Static
Traverse ms	22 ms	448 ms	30 ms	392 ms	27 ms	951 ms
Compute position and size	19 ms	126 ms	30 ms	307 ms	31 ms	879 ms
Frustum culling	1 ms	4 ms	1 ms	3 ms	1 ms	3 ms
Backface culling	1 ms	22 ms	2 ms	25 ms	1 ms	35 ms
Draw splats	77 ms	664 ms	46 ms	324 ms	50 ms	1281 ms
Total rendering time	129 ms	858 ms	109 ms	1051 ms	110 ms	3149 ms
Points rendered	125,183	931,093	267,542	2,026,496	263,915	8,110,665
Preprocessing statistics						
Input points (= leaf nodes)	2,000,651		4,251,890		137,072,827	
Interior nodes	974,114		2,608,752		50,285,122	
Bytes per node	6		4		4	
Space taken by pointers	1.3 MB		2.7 MB		84 MB	
Total file size	18 MB		2.7 MB		761 MB	
Preprocessing time	0.7 min		1.4 min		59 min	

QSplat

- Future work:
 - Even more space savings, e.g. with Huffman Coding (Oy Vey!!)
 - When rendering speed is more important than size, incremental encoding could be removed
 - Analysis of temporal coherence and caching behavior needed

QSplat

- [demo]
-