PantaRay

A Case Study in GPU Ray Tracing for Movies

[Disclaimer: Some slides containing images from Avatar have been removed from this version, in accordance with the licensing terms from Twentieth Century Fox]
Massive Geometry

• 10M – 1B poly per shot

• 10M – 1B scattered shading locations,
  highly non-uniform
Need for good I/O & fast Compute

• Efficient I/O

1.000.000.000.000.000 bytes of live data

• High compute speed

100.000.000.000 points to be shaded
Need for good I/O & fast Compute
Need for good I/O

- $1B$ poly + data = 45GB
- Tesla C1060 = 4GB / GPU
Need for good I/O

• disk bw: <100 MB/s

• CPU → GPU bw: ~6 GB/s

• GPU bw: ~130 GB/s
SH Occlusion

SH Projection

correct transparency
Everything Flows

- streaming RT architecture:
  - streaming spatial index builds
  - paging & caching
  - LOD
MicroGrids

blackbox streaming interface:

read(MicroGrid&)
rewind()
Spatial Index

1. Bucketing
2. Chunking
3. Emission
1. Bucketing

2. Chunking  \textit{Split & Merge}

3. Emission
Spatial Index

1. Bucketing

2. Chunking

3. Emission (1)
1. Bucketing

2. Chunking

3. Emission (2)
• **Shade** one MGStream at a time
Shading

- **Shade** one MGStream at a time

- **Batch** points by spatial proximity, using **K-d splits**
Shading

• **Shade** one MGStream at a time

• **Batch** points by spatial proximity, using **K-d splits**

• **Sort** batches by a **Hilbert curve**
foreach batch:

0. **LOD** determination

1. emit rays

2. **trace** rays

3. process hit points

4. if (...) **goto** step 1 (e.g. transparent hits)

5. write **outputs**
Shading

- each step is a separate kernel
  
  - inputs & outputs live in queues
Tracing broken up in two phases:

1. **working set** determination

2. traversal & intersection
1. **working set** determination:

Brickmap hierarchy approximated
wrt to shading batch, making a cut

Brick covers **large solid angle**  => load treelet

Brick covers **small solid angle**  => treat as **bbox**
1. working set determination:

Keep LRU cache

Stream only **diffs** wrt to previous batch
2. traversal & intersection

```plaintext
trace()
1 while any active ray
2   if any active ray is in leaf node
3     perform wide_leaf_intersection() // Figure 8
4     pop traversal stack
5
6   for i = 0 ... 8 // short while loop
7     if node is a new brick
8         if new brick is not loaded
9             report intersection with new brick’s bbox
10            pop traversal stack
11        else
12            jump to new brick’s root
13      else if node is not leaf
14          traverse to next node
15    else
16      break // node is a leaf
```
2. traversal & intersection

Deep trees → naive packets give very poor utilization, ~20%
LOD Tracing

2. traversal & intersection

SIMD-wide leaf intersection

20% → 60% SIMD utilization
2. traversal & intersection

adaptive precision (doubles are costly!)

*float* intersection +

*double* checks for *near* hits
2. traversal & intersection

- main problems: register pressure divergence

- queues might help [Aila & Karras 2010]
Results (45GB input, 1GB cache)

\(~15\text{M} \text{ rays/s per GPU (Tesla)}\)

98.5 - 99.3\% cache efficiency

I/O: ~2 bytes / ray!
Thank You!