PantaRay:
Fast Ray-traced Occlusion Caching of Massive Scenes
J. Pantaleoni, L. Fascione, M. Hill, T. Aila

Marie-Lena Eckert

computer graphics & visualization
Agenda

• Introduction
  – Motivation
  – Basics

• PantaRay
  – Accelerating structure generation
  – Massively parallel ray tracer

• Conclusion
  – Summary
  – Critics
Introduction: Motivation

- Computer-generated imagery (CGI), rendering:

3D model

2D image
Introduction: Motivation

• Avatar: unprecedented complexity
Introduction: Motivation

• Avatar: level of detail (LOD)
Introduction: Basics

• Rendering equation

\[ L_o(x, \omega_o) = \int_{\Omega^+} L_i(x, \omega) \rho(x, \omega, \omega_o) V(x, \omega) \langle \omega, \hat{n} \rangle \, d\omega \]
Introduction: Basics

- Rendering equation

\[ L_o(x, \omega_o) = \int_{\Omega^+} L_i(x, \omega) \rho(x, \omega, \omega_o) V(x, \omega) \langle \omega, \hat{n} \rangle \, d\omega \]
Introduction: Basics

- Image-based lighting (IBL) for global illumination
Introduction: Basics

- Image-based lighting (IBL) for global illumination
Introduction: Basics

- Rendering equation

\[ L_o(x, \omega_o) = \int_{\Omega^+} L_i(x, \omega) \rho(x, \omega, \omega_o) V(x, \omega) \langle \omega, \hat{n} \rangle d\omega \]
Introduction: Basics

- Rendering equation

\[ L_o(x, \omega_o) = \int_{\Omega^+} L_i(x, \omega) \rho(x, \omega, \omega_o) V(x, \omega) \langle \omega, \hat{n} \rangle d\omega \]
Introduction: Basics

- Rendering equation

\[ L_o(x, \omega_o) = \int_{\Omega^+} L_i(x, \omega) \rho(x, \omega, \omega_o) V(x, \omega) \langle \omega, \hat{n} \rangle \, d\omega \]
Introduction: Basics

• Rendering equation

\[ L_o(x, \omega_o) = \int_{\Omega^+} L_i(x, \omega) \rho(x, \omega, \omega_o) V(x, \omega) \langle \omega, \hat{n} \rangle d\omega \]
Introduction: Basics

- Rendering equation

\[ L_o(x, \omega_o) = \int_{\Omega^+} L_i(x, \omega) \rho(x, \omega, \omega_o) V(x, \omega) \langle \omega, \hat{n} \rangle \, d\omega \]
Introduction: Basics

• Spherical Harmonics (SH)
  – [Ram01]: 9 coefficients, 1% error
  – Store visibility term and BRDF
  – Rendering time: only scalar product
Introduction: Basics

- Rendering equation

\[ L_o(x, \omega_o) = \rho \cdot \langle SH(L_i), SH(V(\omega, \hat{n})) \rangle \]
PantaRay

→ Precompute and cache visibility term

• Scenes:
  – Unprecedented complexity
  – Bigger than memory
  – Varying density

• Production pipeline of Weta Digital

• Panta rei: everything flows
PantaRay: Pipeline computation passes

1. Preparation

2. Vislocal pass

3. Beauty pass

$$L_0(x, \omega_0) = \rho \cdot \langle SH(L_i), SH(V(\omega, \hat{n})) \rangle$$

Microgrid-stream

AS

Ray tracing

SH(V)
PantaRay: Preparation

Microgrid stream:

Final render of the scene

Render of bake sets

PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes
Marie-Lena Eckert
PantaRay: Acceleration Structure

• Vislocal pass
  – Build Bounding Volume Hierarchy (BVH)
  – Traverse for intersection computation
    • LOD
    • Ray tracing

• Main bottleneck: I/O
  – Touch objects multiple times
  – Ten of thousands concurrent processes
PantaRay: Acceleration Structure

Microgrids → Bucketing pass → Brick construction

Brick construction

Chunking pass

PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes
Marie-Lena Eckert
PantaRay: Ray tracing

- Shading of bake sets
  - Point stream into batches
  - Simple ray tracer: CPU
    - Simple reflection occlusion
    - Area light shadow shaders
  - Massively parallel ray tracer: GPU
    - Complex spherical sampling queries
      - SH occlusion
      - Indirect lighting shaders
  - Billions of points per scene → compute-intensive, LOD
PantaRay: Ray tracing - LOD

• Determine and stream required bricks
PantaRay: Ray tracing - LOD

• 3 rays emanating from each surface point
• Each ray represents solid angle $\sigma = \frac{2\pi}{n}$, $n = 3$
PantaRay: Ray tracing - LOD

- Bricks loaded if $\text{size}(\text{BV(brick)}) \geq \text{avg}(\text{dist(rays)})$
PantaRay: Ray tracing - LOD

- Partially transparent Bounding Volumes (BV)
PantaRay: Ray tracing - LOD

- Custom far-field for batch 1
PantaRay: Generate and trace rays

• $m$ GPUs
  – $\frac{1}{m}$ points, subset $\rightarrow$ focus on coherence

• Trace through brick hierarchy
  – [Ail09] while-while traversal kernel

• Intersection computation
  – high scene complexity, high tree depth $\rightarrow$ low
    Single Instruction Multiple Thread (SIMT) utilization
  – increase from 20% to 50-60%
Conclusion: Summary

• Raised 2 orders of magnitude in terms of both speed and scene size
• Comparison AS generation
  – [Wald05]: 350M triangles: 1d
  – PantaRay: 575M micropolygons: 54min
• Custom far-field \(\rightarrow\) low I/O (few MBs per batch)
• Ray tracing speed: up to 22M shaded rays per second
Conclusion: Summary

I/O: 2MB per batch, trace time: 16 h, 41 m, 8.7M shaded rays/s
Conclusion: Critics

• Loss of details in computing visibility term (SH, LOD)
• Float precision: missing some intersections
• Parallel construction of AS
Thank you!
PantaRay: References

[Ail09]

[Ram01]

[Wald05]
Trace rays algorithm

```c
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
```
Compute intersections algorithm

```c
wide_leaf_intersection()
1:     tidx = threadIdx.x; // [0, 32)
2:     // count the number of ray-primitive tasks.
3:     // haveLeaf indicates if the current primitive is a leaf
4:     (lo, hi) = scan(haveLeaf ? numPrims : 0);
5:     numIsect = hi from thread 31 // how many left?
6:     while (numIsect > 0)
7:     {
8:         // select up to 32 ray-primitive tasks
9:         foreach primitive p, with lo+p ∈ [0, 32)
10:            write (tidx,p) pair to shared[lo+p]
11:     }
12:     // get a ray-primitive task to execute
13:     (srcThread,p2) = shared[tidx];
14:     // copy needed variables from ”srcThread” thread
15:     foreach 32bit variable var needed in intersection
16:        shared[tidx] = var; // write my own variable
17:        var2 = shared[srcThread]; // read from ”srcThread”
18:     // intersection using vars copied from ”srcThread”
19:     if (tidx < numIsect)
20:        shared[tidx] = intersectRayPrim(p2);
21:     // collect intersections of my ray
22:     foreach prim p, with lo+p ∈ [0, 32)
23:         modify ray according to shared[lo+p]
24:     lo = lo-32; hi = hi-32; numIsect = numIsect-32;
25:     }
```