PantaRay:
Fast Ray-traced Occlusion Caching of Massive Scenes

J. Pantaleoni, L. Fascione, M. Hill, T. Aila

Marie-Lena Eckert

tum.3D
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

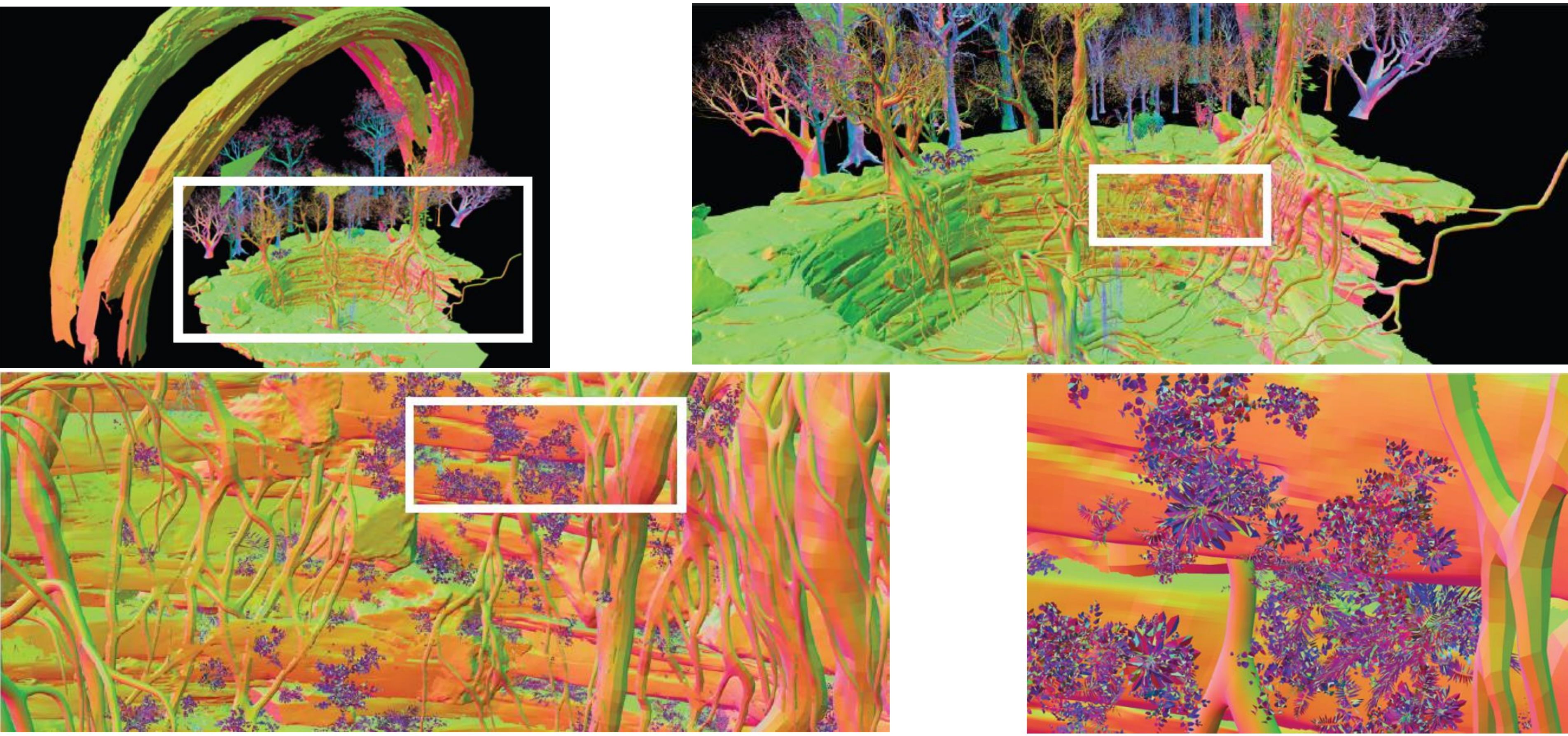
2D image

Introduction: Motivation

- Avatar: unprecedented complexity

Introduction: Motivation

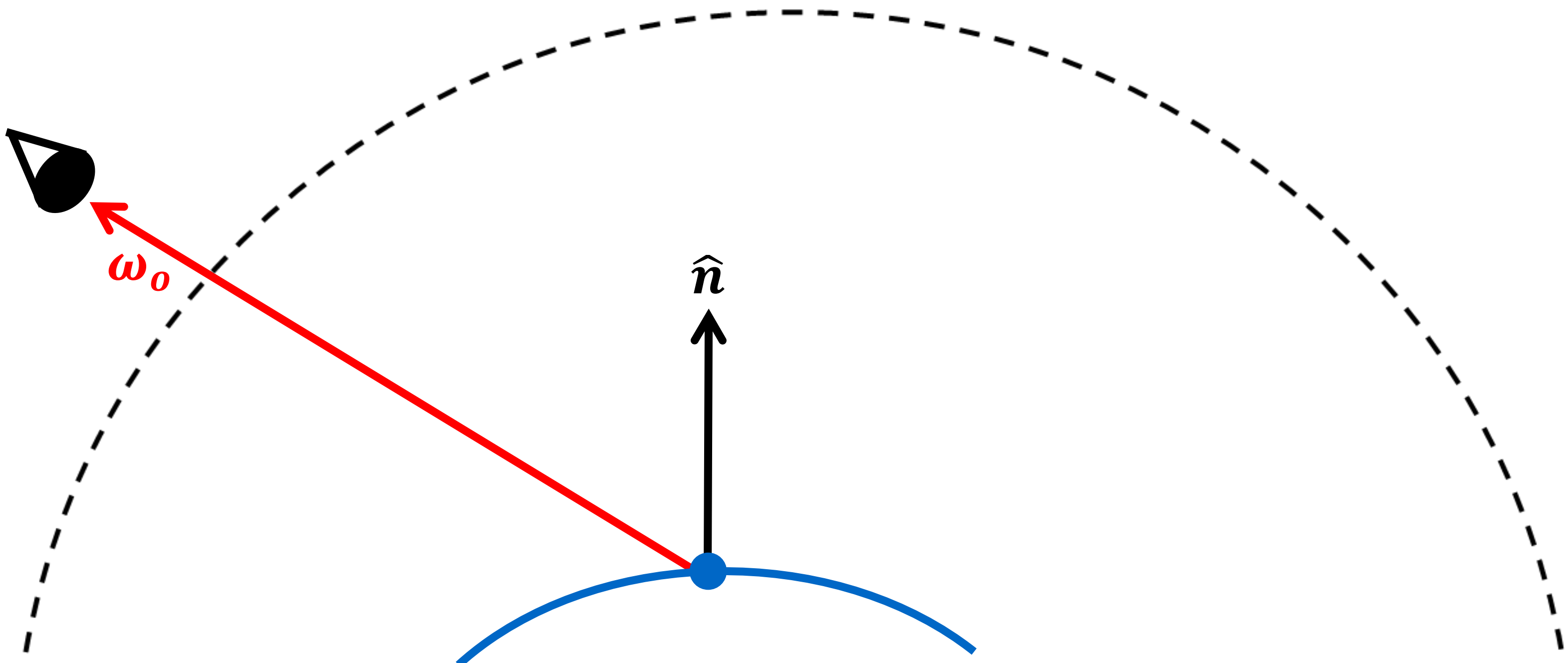
- Avatar: level of detail (LOD)



Introduction: Basics

- Rendering equation

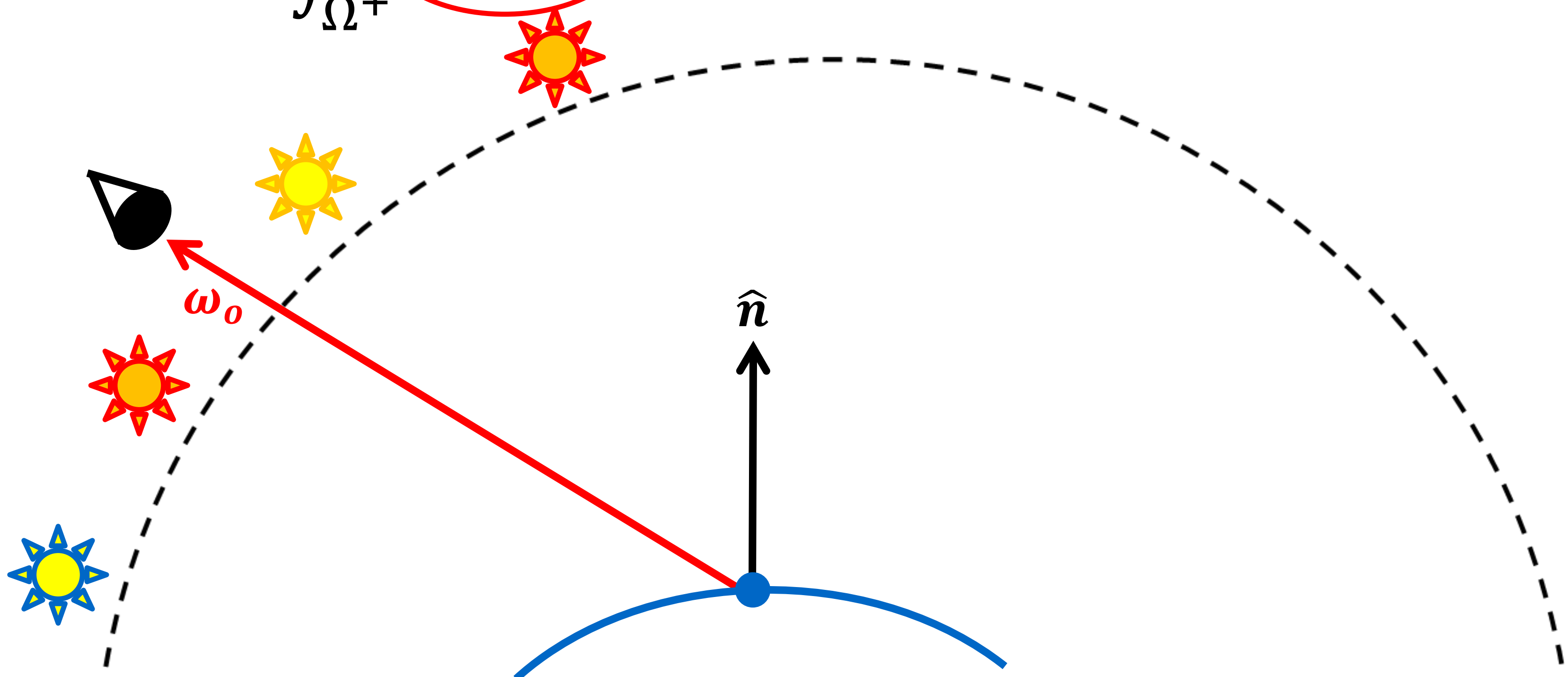
$$L_o(x,\omega_o) = \int_{\Omega^+} L_i(x,\omega)\rho(x,\omega,\omega_o)V(x,\omega)\langle\omega,\hat{n}\rangle\, d\omega$$



Introduction: Basics

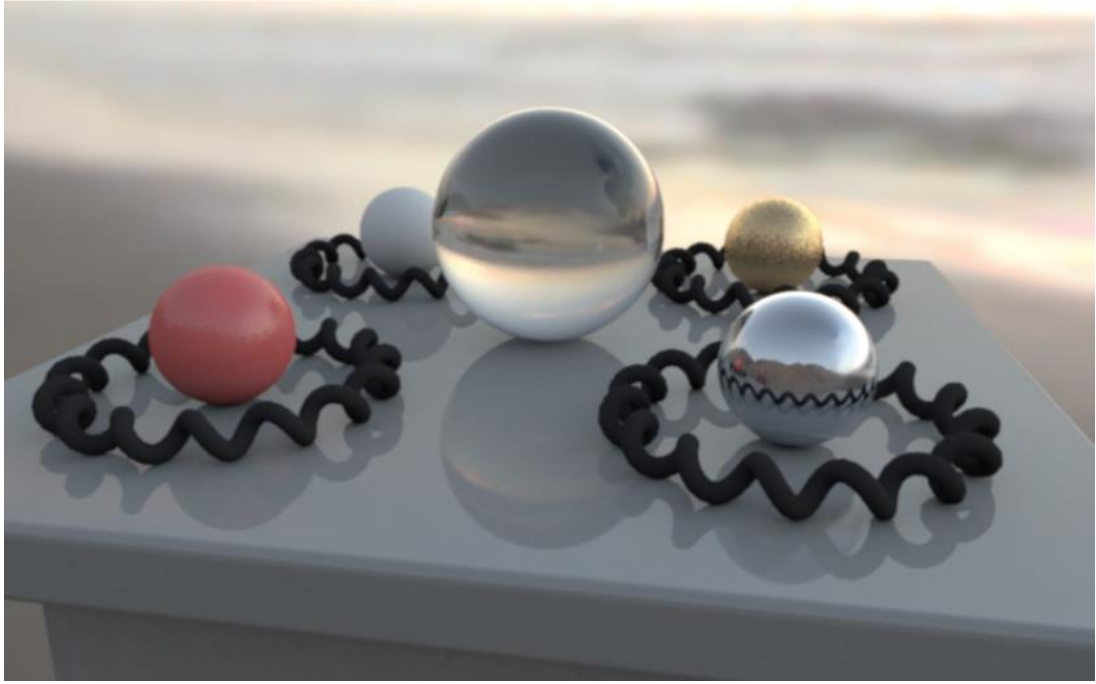
- Rendering equation

$$L_o(x,\omega_o) = \int_{\Omega^+} L_i(x,\omega)\rho(x,\omega,\omega_o)V(x,\omega)\langle\omega,\hat{n}\rangle\, d\omega$$



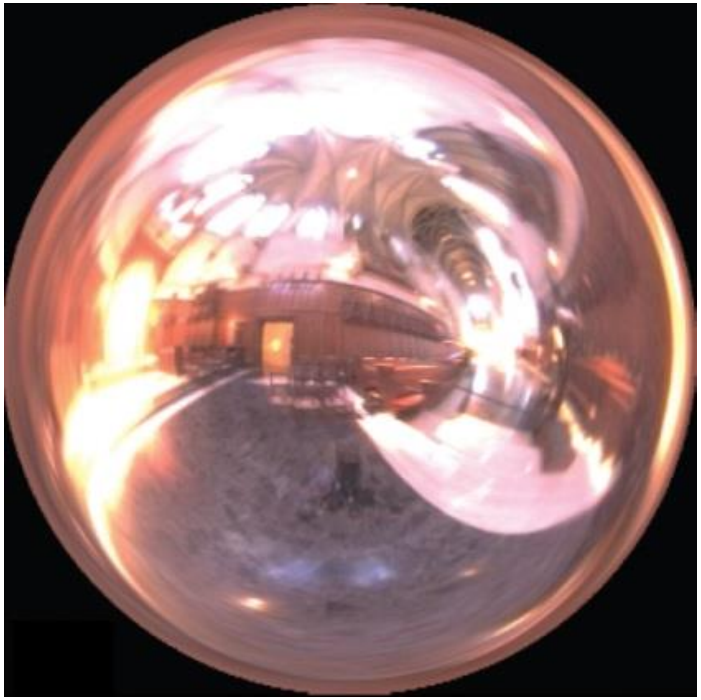
Introduction: Basics

- Image-based lighting (IBL) for global illumination



Introduction: Basics

- Image-based lighting (IBL) for global illumination

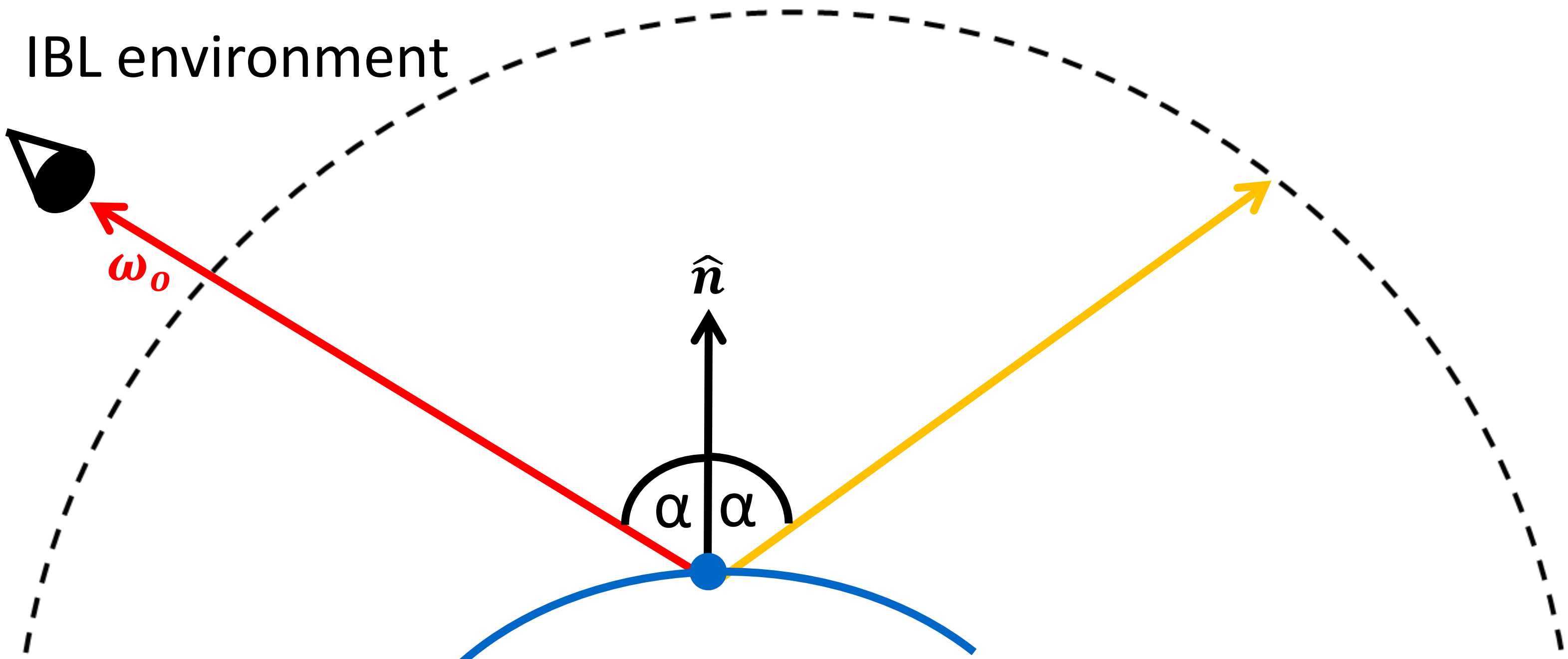


Introduction: Basics

- Rendering equation

BRDF

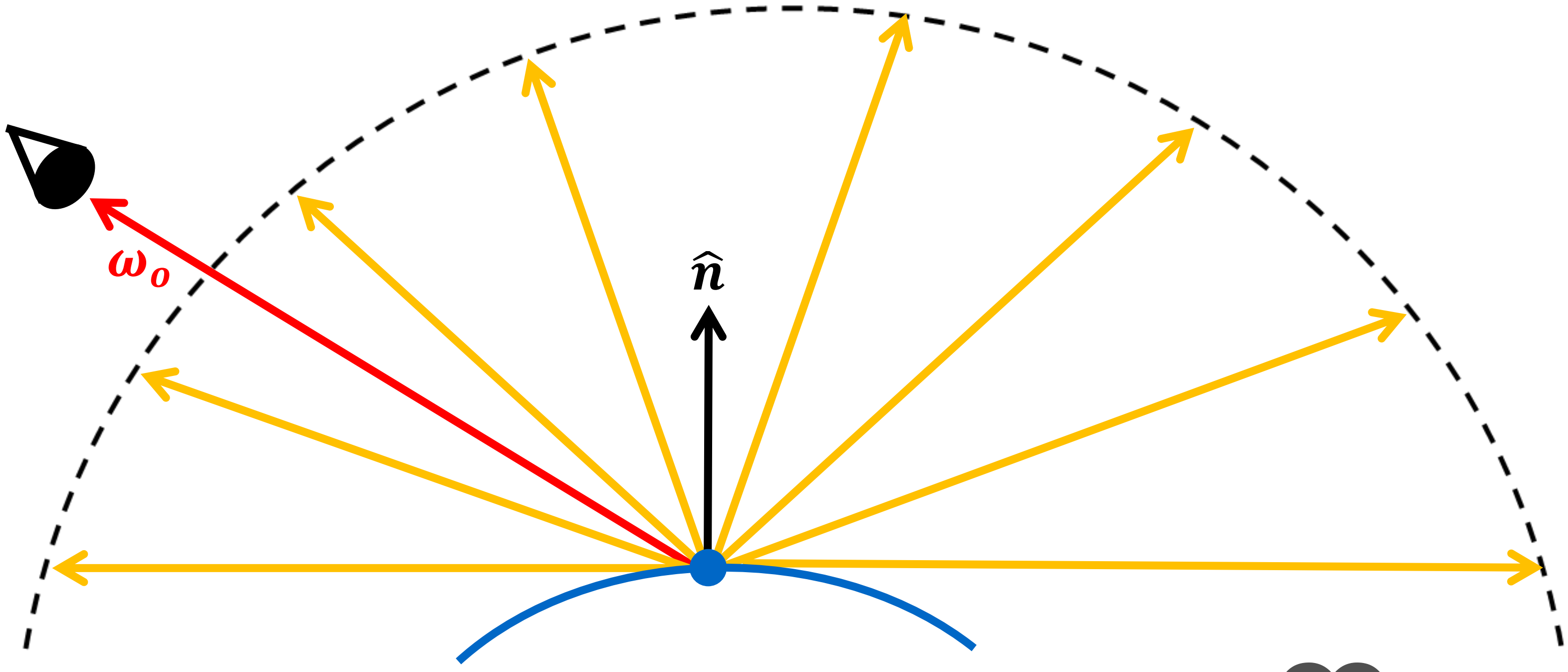
$$L_o(x, \omega_o) = \int_{\Omega^+} L_i(x, \omega)\rho(x, \omega, \omega_o)V(x, \omega)\langle\omega, \hat{n}\rangle\, d\omega$$



Introduction: Basics

- Rendering equation

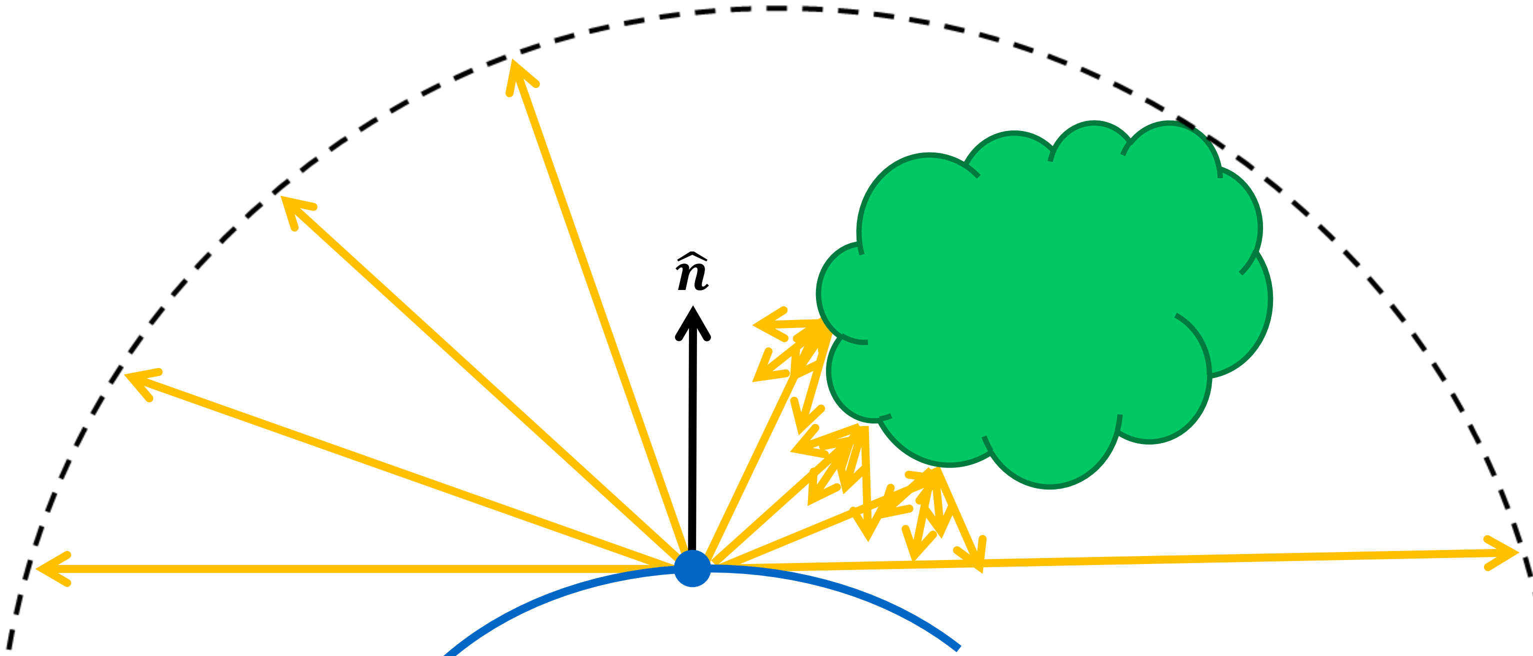
$$L_o(x, \omega_o) = \int_{\Omega^+} L_i(x, \omega) \rho(x, \omega, \omega_o) V(x, \omega) \langle \omega, \hat{n} \rangle \, d\omega$$



Introduction: Basics

- Rendering equation

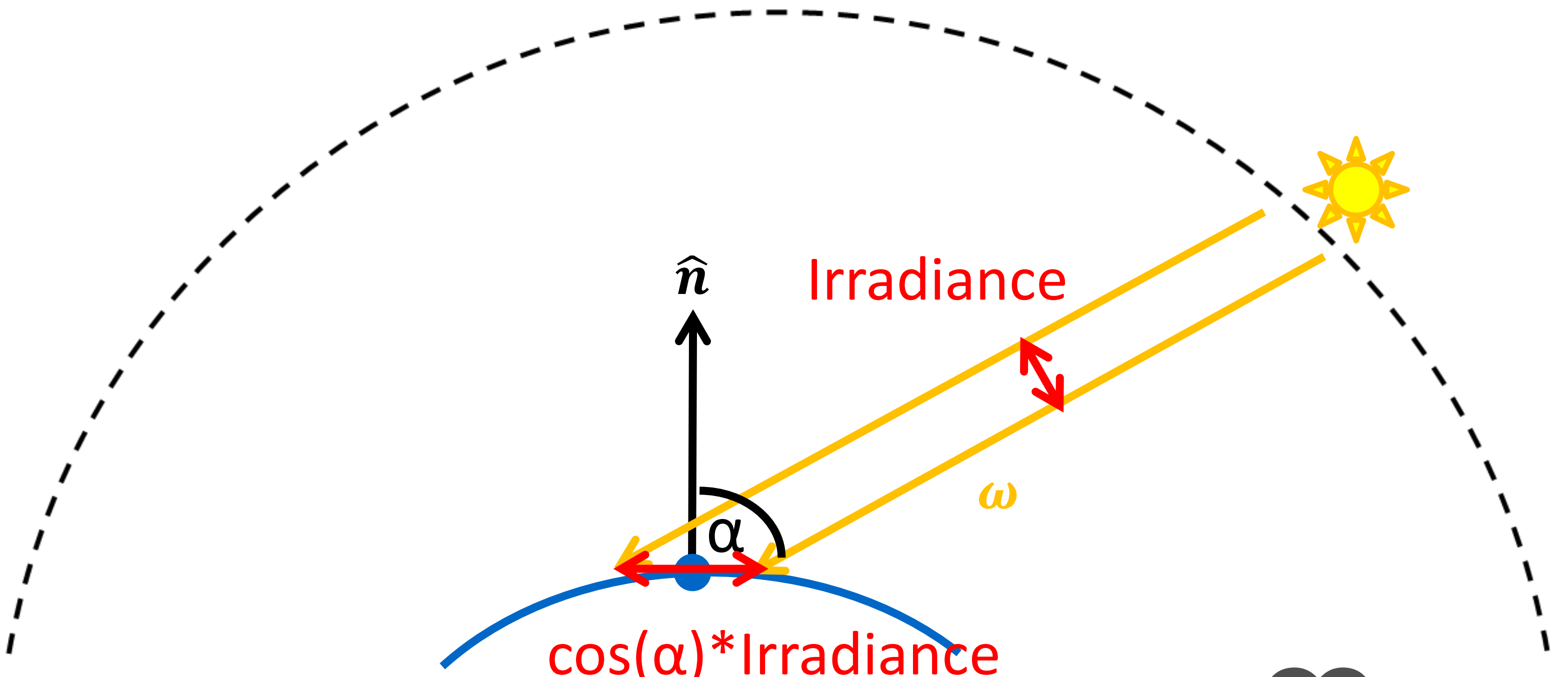
$$L_o(x, \omega_o) = \int_{\Omega^+} L_i(x, \omega)\rho(x, \omega, \omega_o)V(x, \omega)\langle\omega, \hat{n}\rangle\, d\omega$$



Introduction: Basics

- Rendering equation

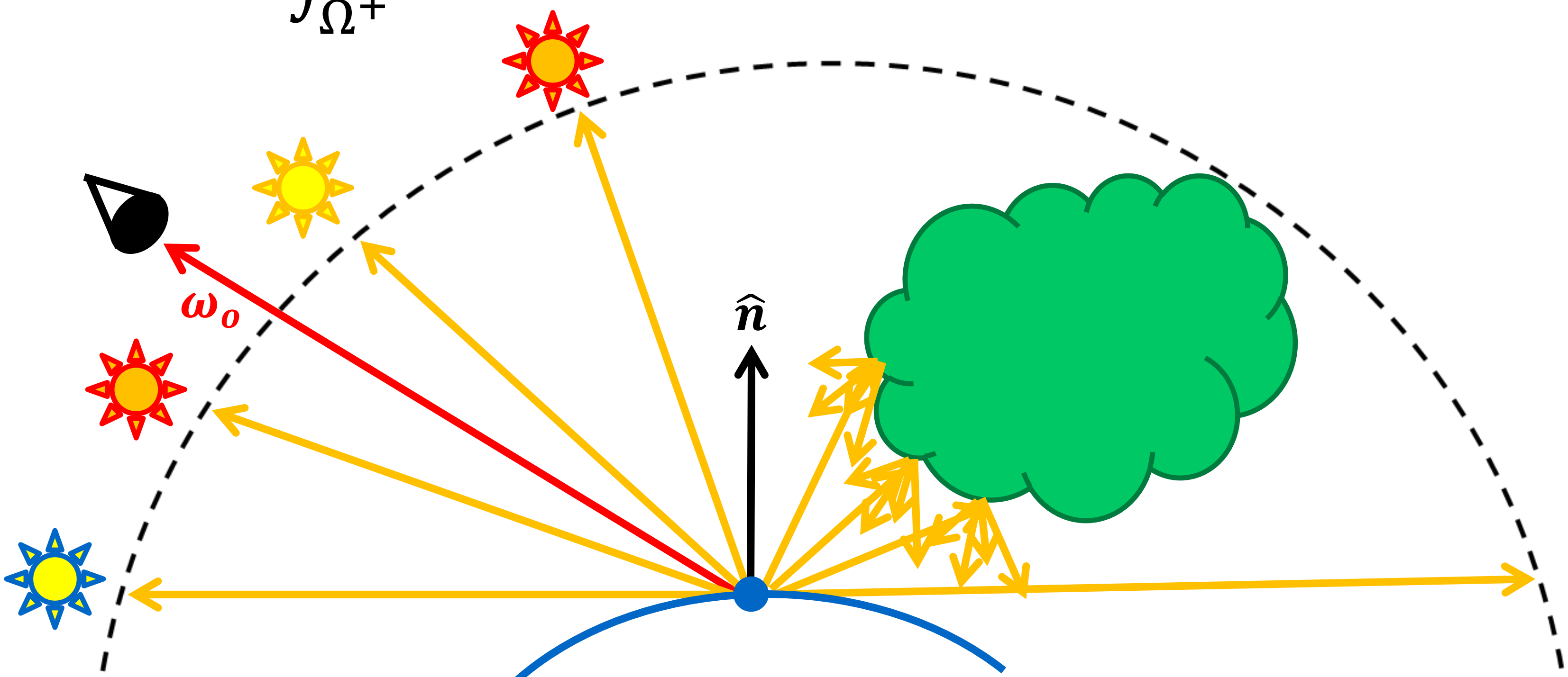
$$L_o(x,\omega_o) = \int_{\Omega^+} L_i(x,\omega)\rho(x,\omega,\omega_o)V(x,\omega)\langle\omega,\hat{n}\rangle d\omega$$



Introduction: Basics

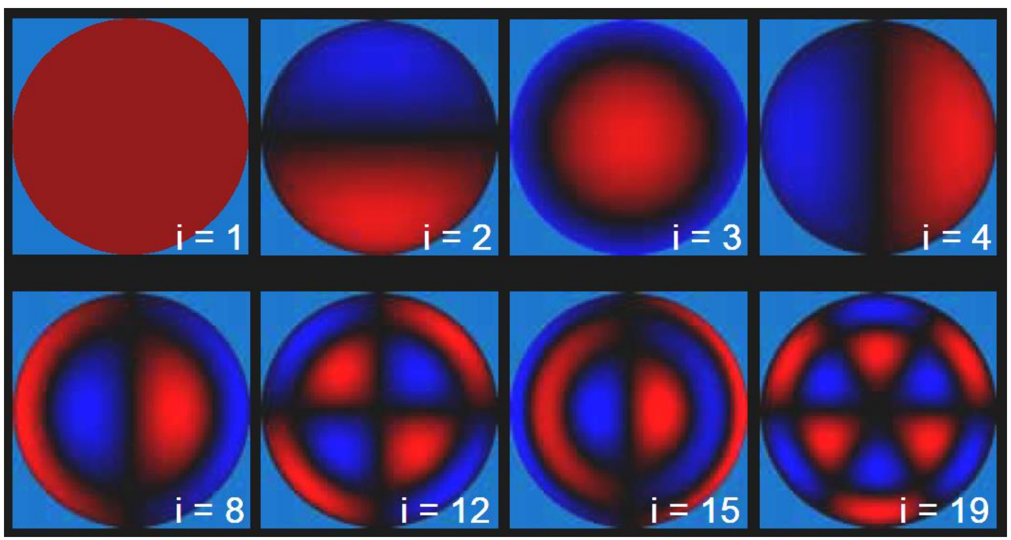
- Rendering equation

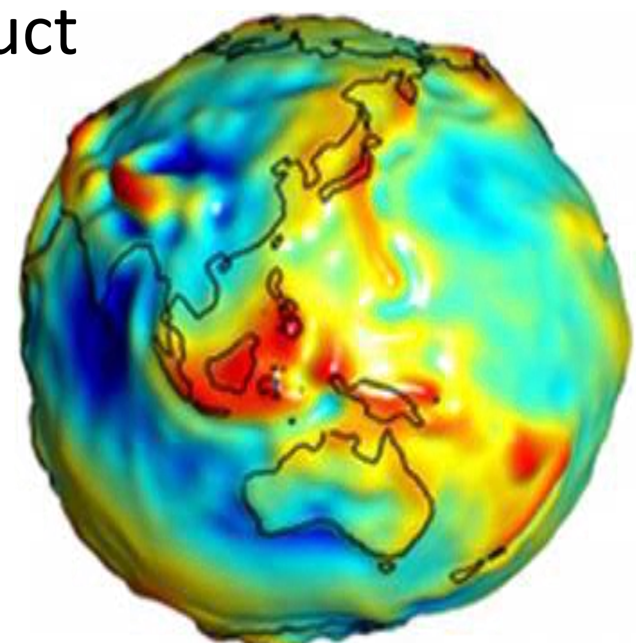
$$L_o(x,\omega_o) = \int_{\Omega^+} L_i(x,\omega)\rho(x,\omega,\omega_o)V(x,\omega)\langle\omega,\hat{n}\rangle\, d\omega$$



Introduction: Basics

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

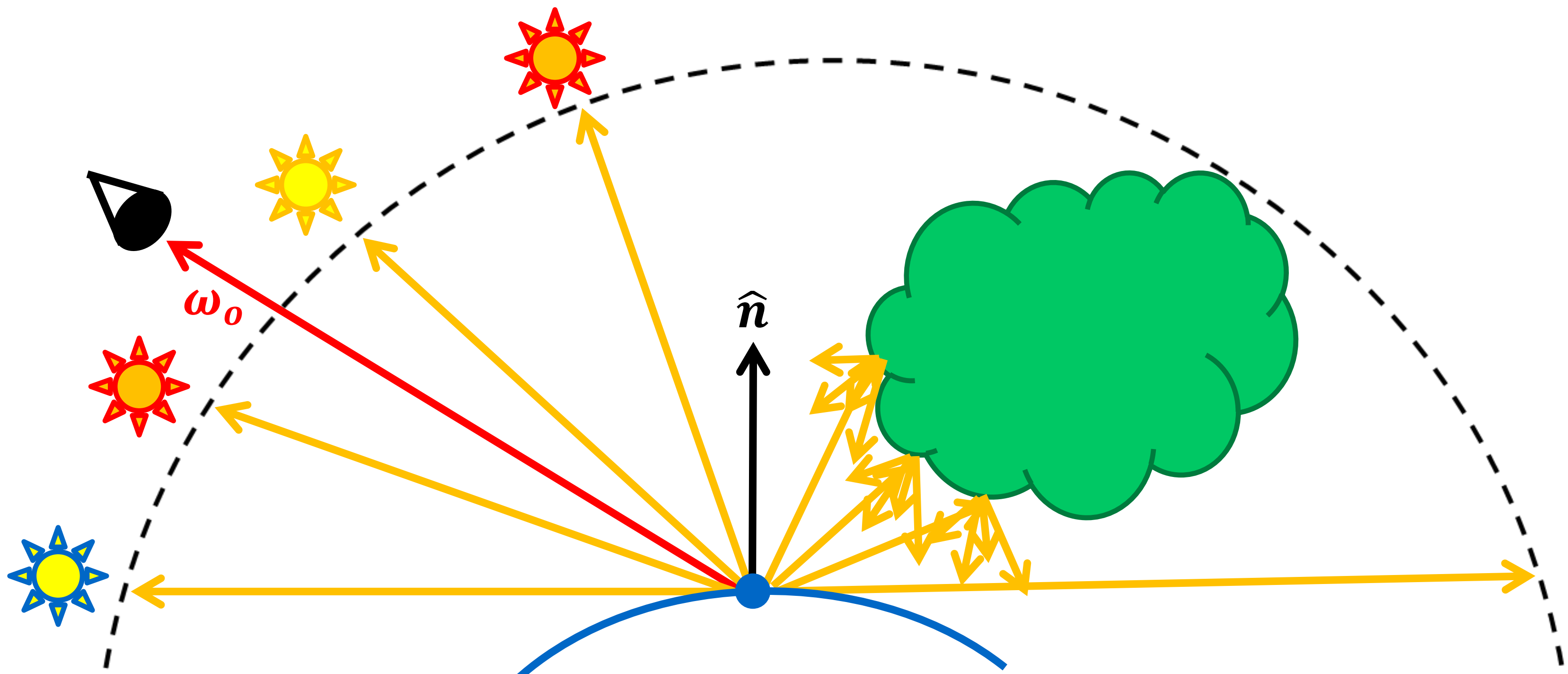




Introduction: Basics

- Rendering equation

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

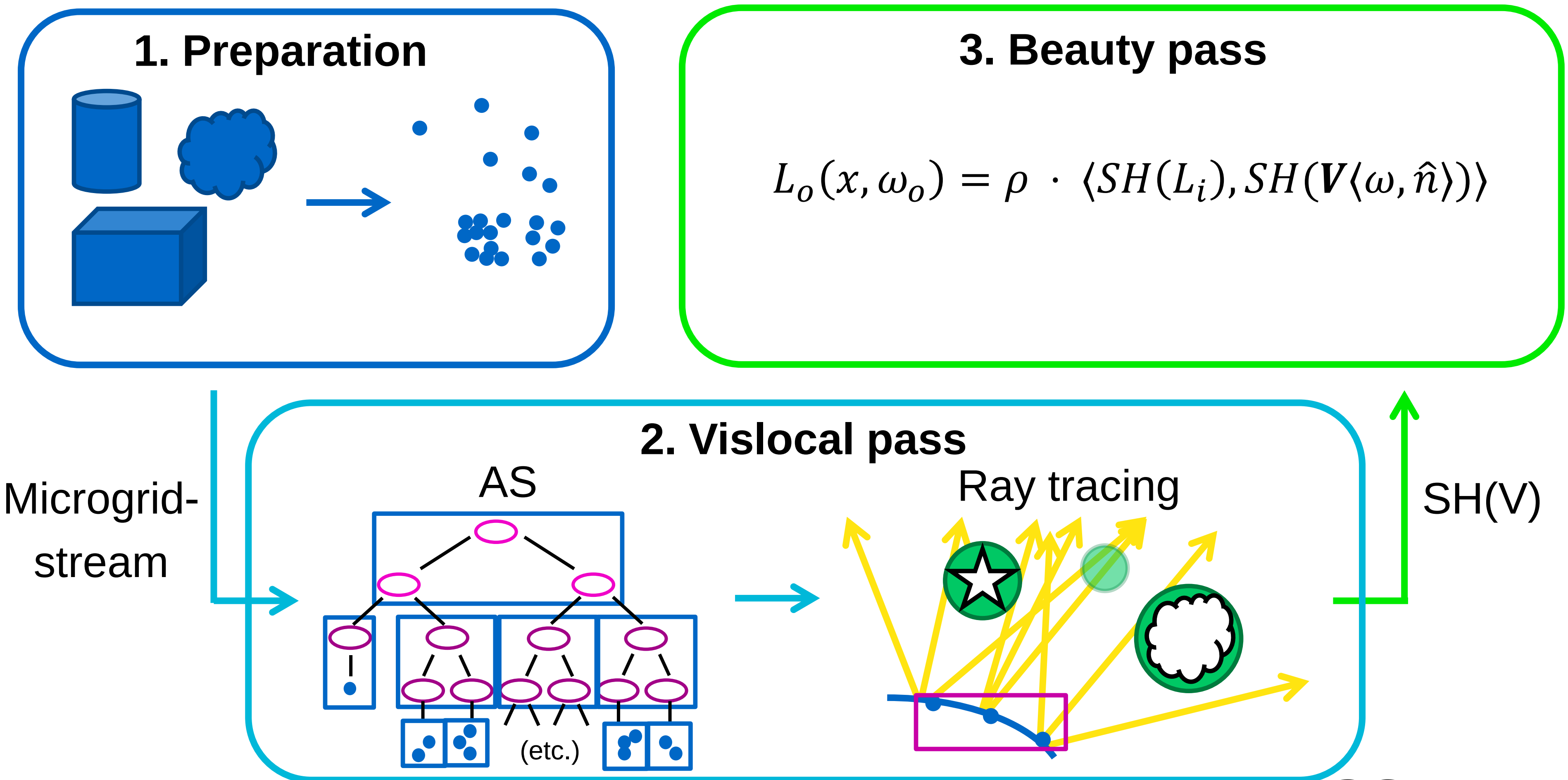


PantaRay

→ 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



PantaRay: Preparation

Microgrid stream:

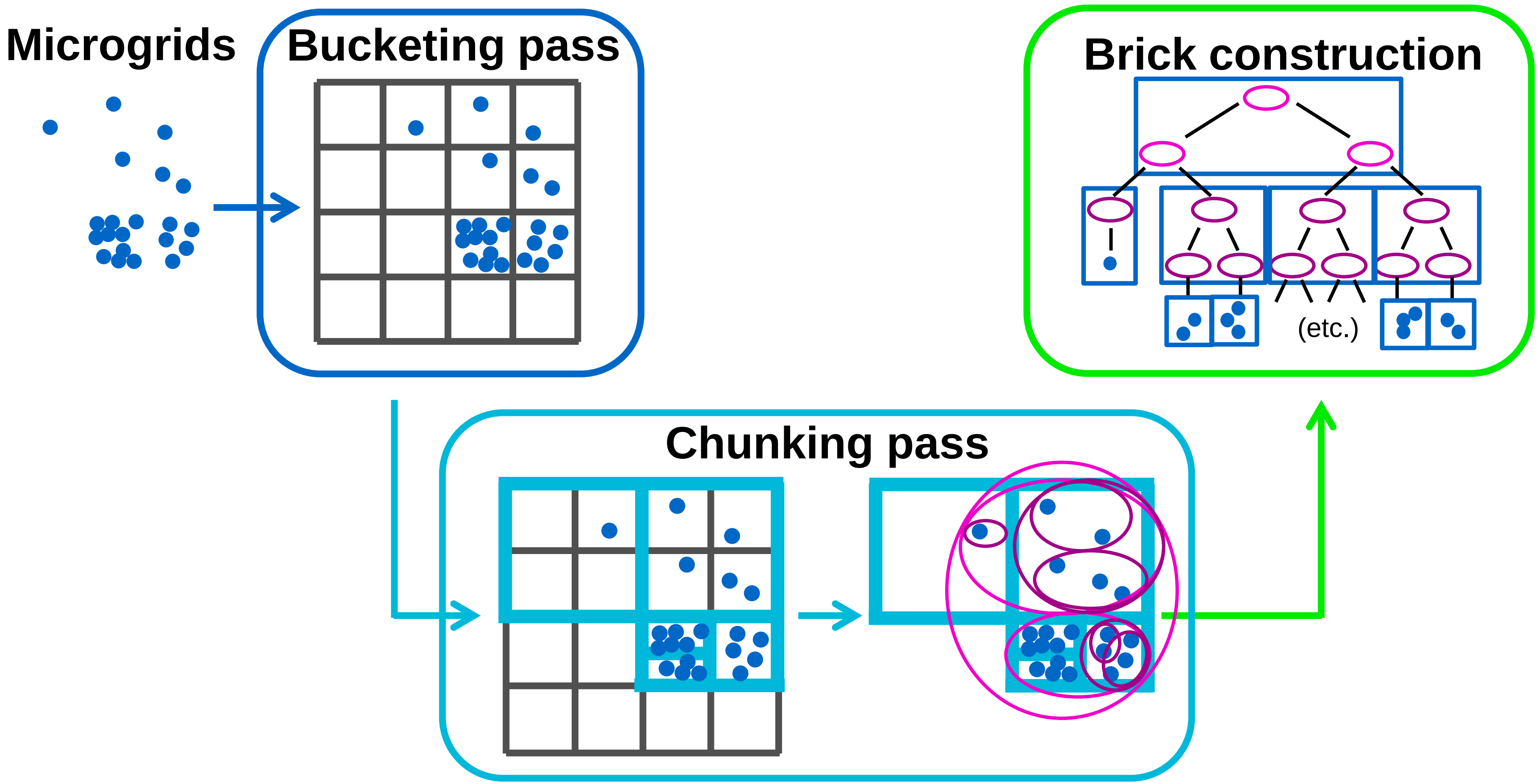
Final render of the scene

Render of bake sets

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

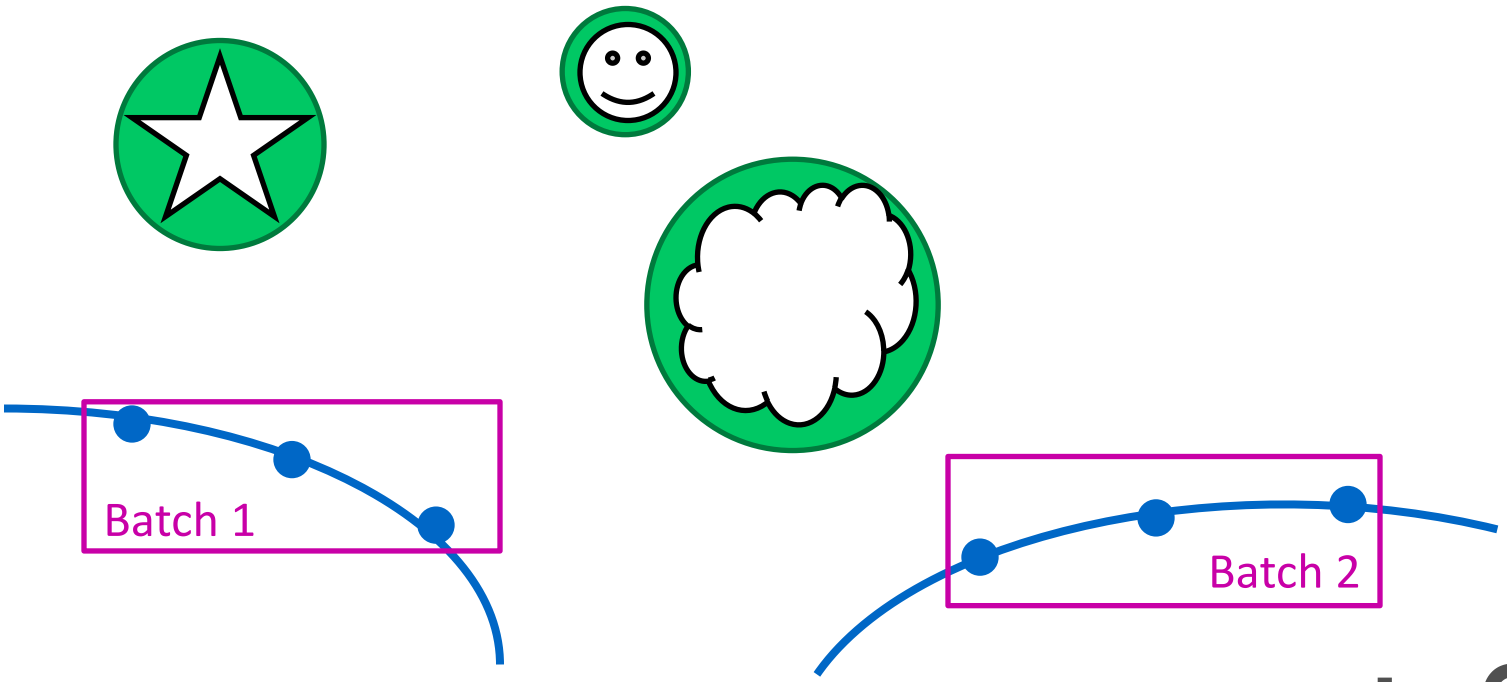


PantaRay: Ray tracing

- 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

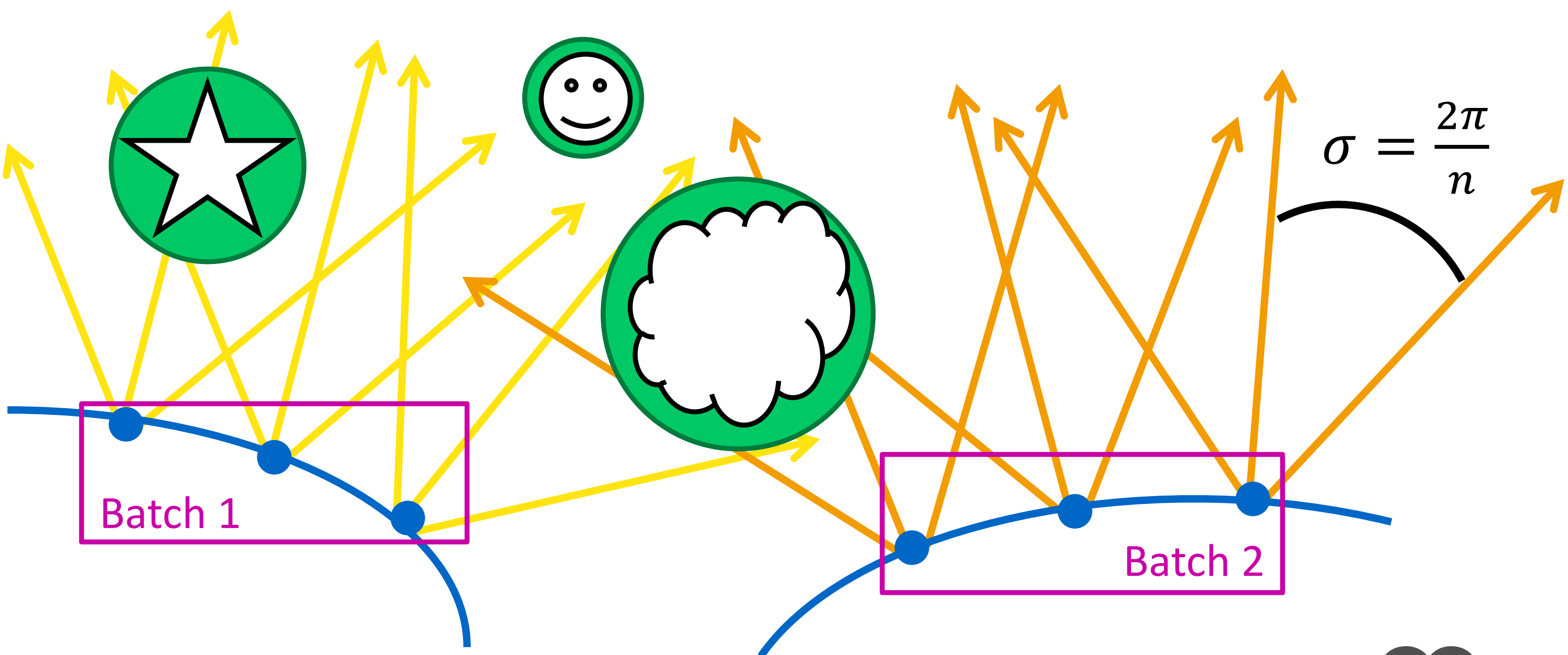
PantaRay: Ray tracing - LOD

- Determine and stream required bricks



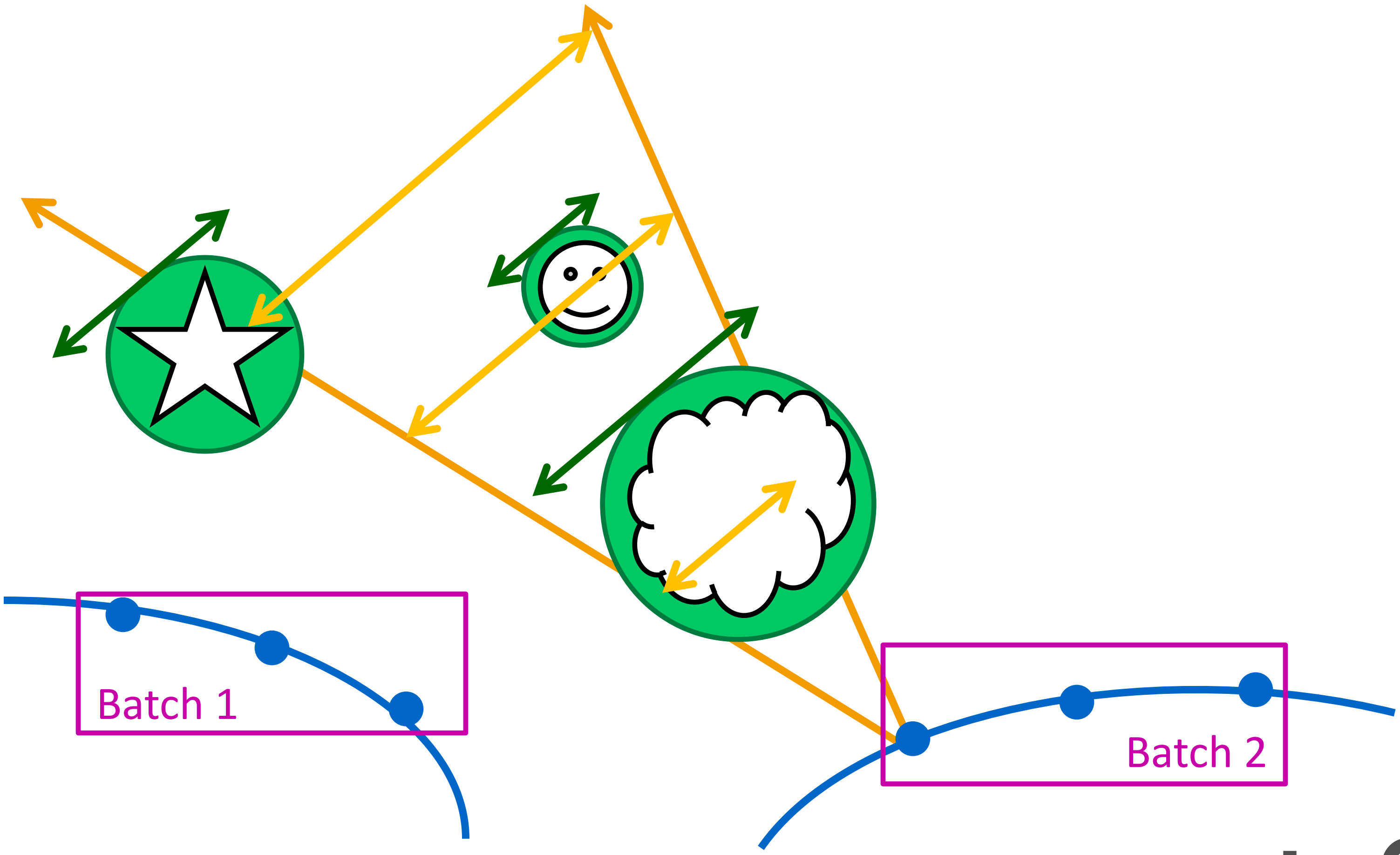
PantaRay: Ray tracing - LOD

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



PantaRay: Ray tracing - LOD

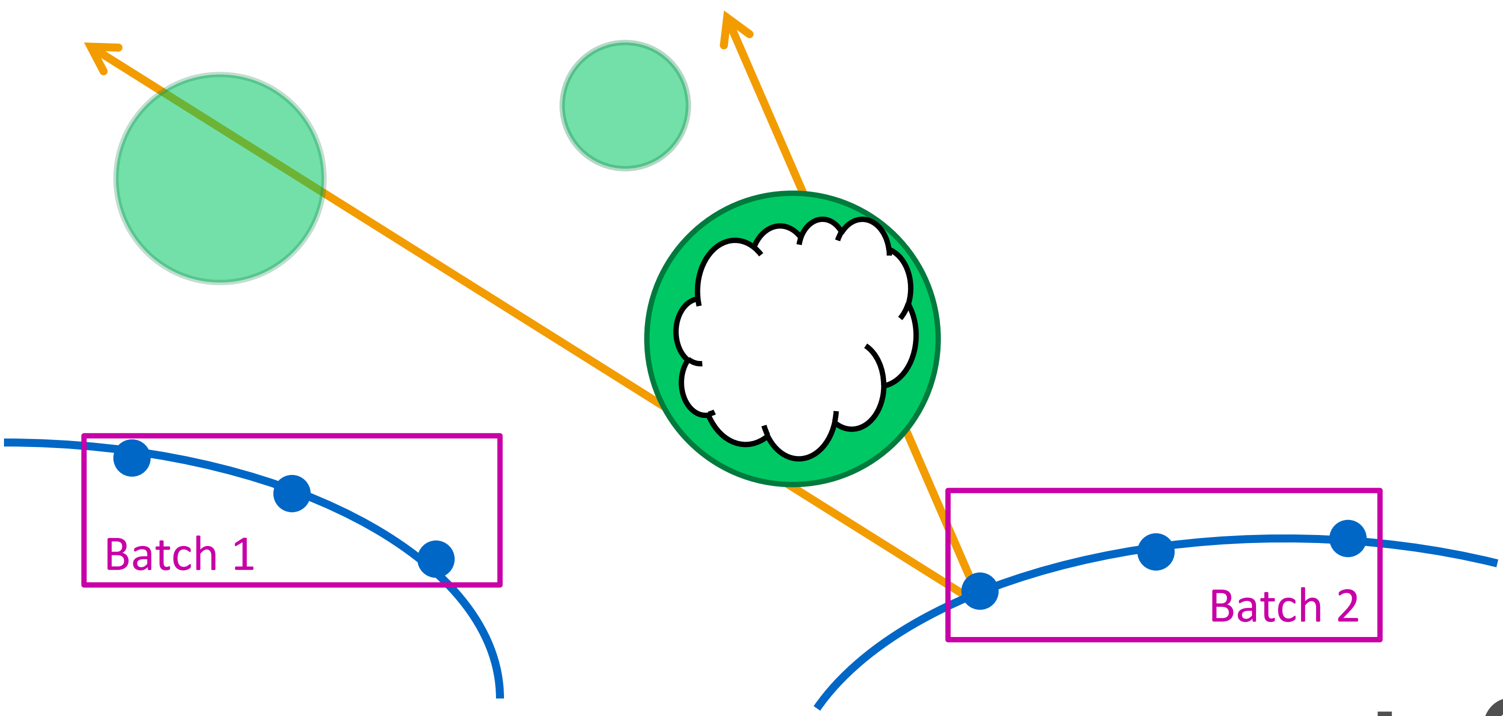
- Bricks loaded if **size(BV(brick))** >= **avg(dist(rays))**



tum.3D computer graphics & visualization

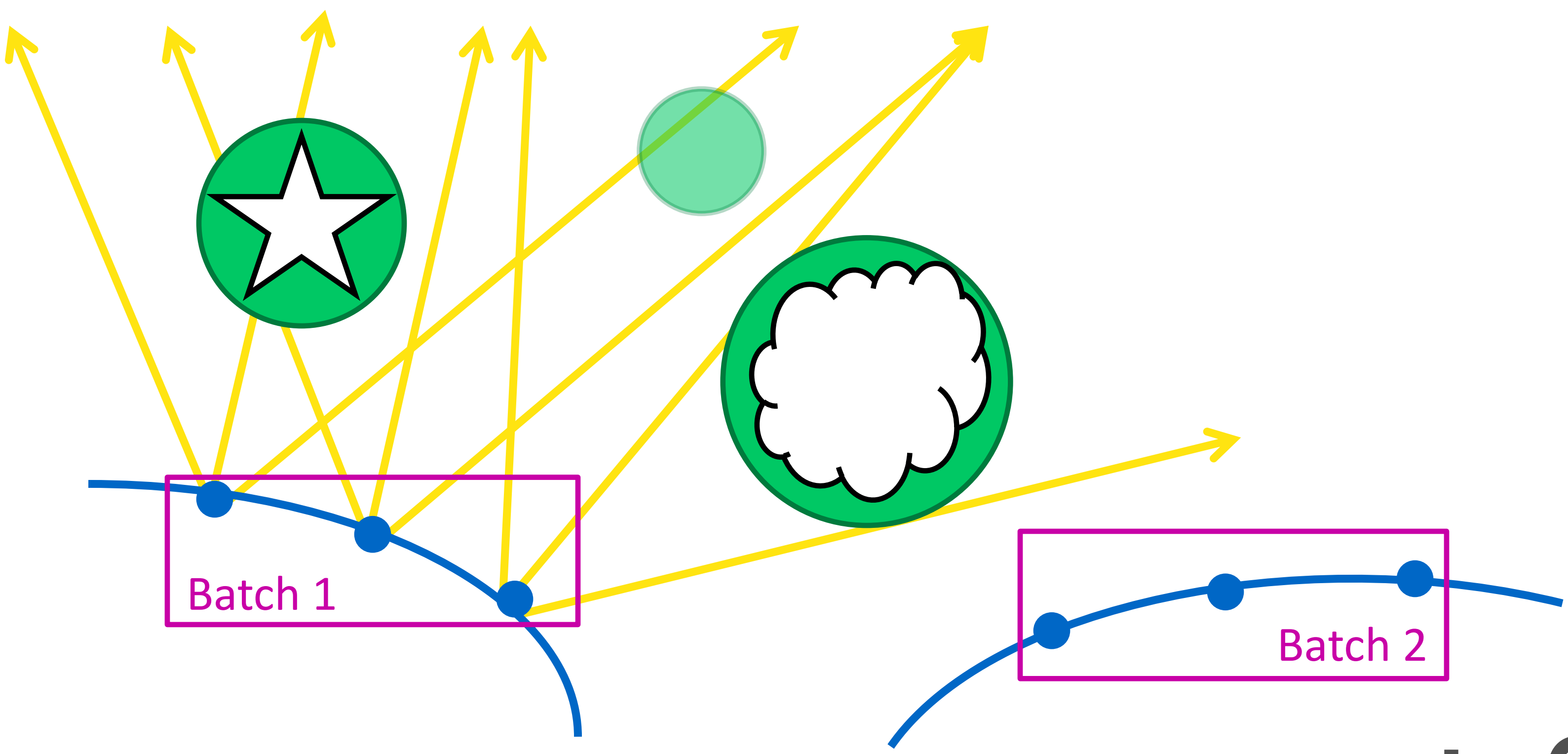
PantaRay: Ray tracing - LOD

- 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 → focus on coherence
- Trace through brick hierarchy
  - [Ail09] while-while traversal kernel
- Intersection computation
  - high scene complexity, high tree depth → 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 → 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]

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In Proc. High-Performance Graphics, 145–149.

[Ram01]

Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques, S IGGRAPH '01, pages 497-500, New York, NY, USA, 2001. ACM.

[Wald05]

Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An interactive out-of-core rendering framework for visualizing massively complex models. In ACM SIGGRAPH 2005 Courses, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

Trace rays algorithm

```
trace()

while any active ray
    if any active ray is in leaf node
        perform wide_leaf_intersection() // Figure 8
        pop traversal stack

    for i = 0 ... 8 // short while loop
        if node is a new brick
            if new brick is not loaded
                report intersection with new brick's bbox
                pop traversal stack
            else
                jump to new brick's root
        else if node is not leaf
            traverse to next node
        else
            break // node is a leaf
```

Compute intersections algorithm

```
wide_leaf_intersection()
  tidx = threadIdx.x; // [0,32)
  // count the number of ray-primitive tasks.
  // haveLeaf indicates if the current node is a leaf
  [lo,hi) = scan(haveLeaf ? numPrims : 0);
  numIsect = hi from thread 31 // how many left?

  while (numIsect > 0)
  {
    // select up to 32 ray-primitive tasks
    foreach primitive p, with lo+p ∈ [0,32)
      write (tidx,p) pair to shared[lo+p]

    // get a ray-primitive task to execute
    (srcThread,p2) = shared[tidx];

    // copy needed variables from "srcThread" thread
    foreach 32bit variable var needed in intersection
      shared[tidx] = var; // write my own variable
      var2 = shared[srcThread]; // read from "srcThread"

    // intersection using vars copied from "srcThread"
    if (tidx < numIsect)
      shared[tidx] = intersectRayPrim(p2);

    // collect intersections of my ray
    foreach prim p, with lo+p ∈ [0,32)
      modify ray according to shared[lo+p]

    lo = lo-32; hi = hi-32; numIsect = numIsect-32;
  }
```