Lecture 16:

Real Time Ray Tracing 2 + More on advanced rasterization

Visual Computing Systems

Stanford CS348K, Spring 2022

Real-time ray tracing performance challenges

To simulate advanced effects renderer must trace many rays per pixel to reduce variance (noise) that results from numerical integration (via Monte Carlo sampling)

Ray-scene intersection requires traversal through bounding volume hierarchy acceleration structure

- Unpredictable data access
- Rays are essentially randomly oriented after enough bounces

Incoherent shading

Not discussed last time: building the BVH acceleration structure

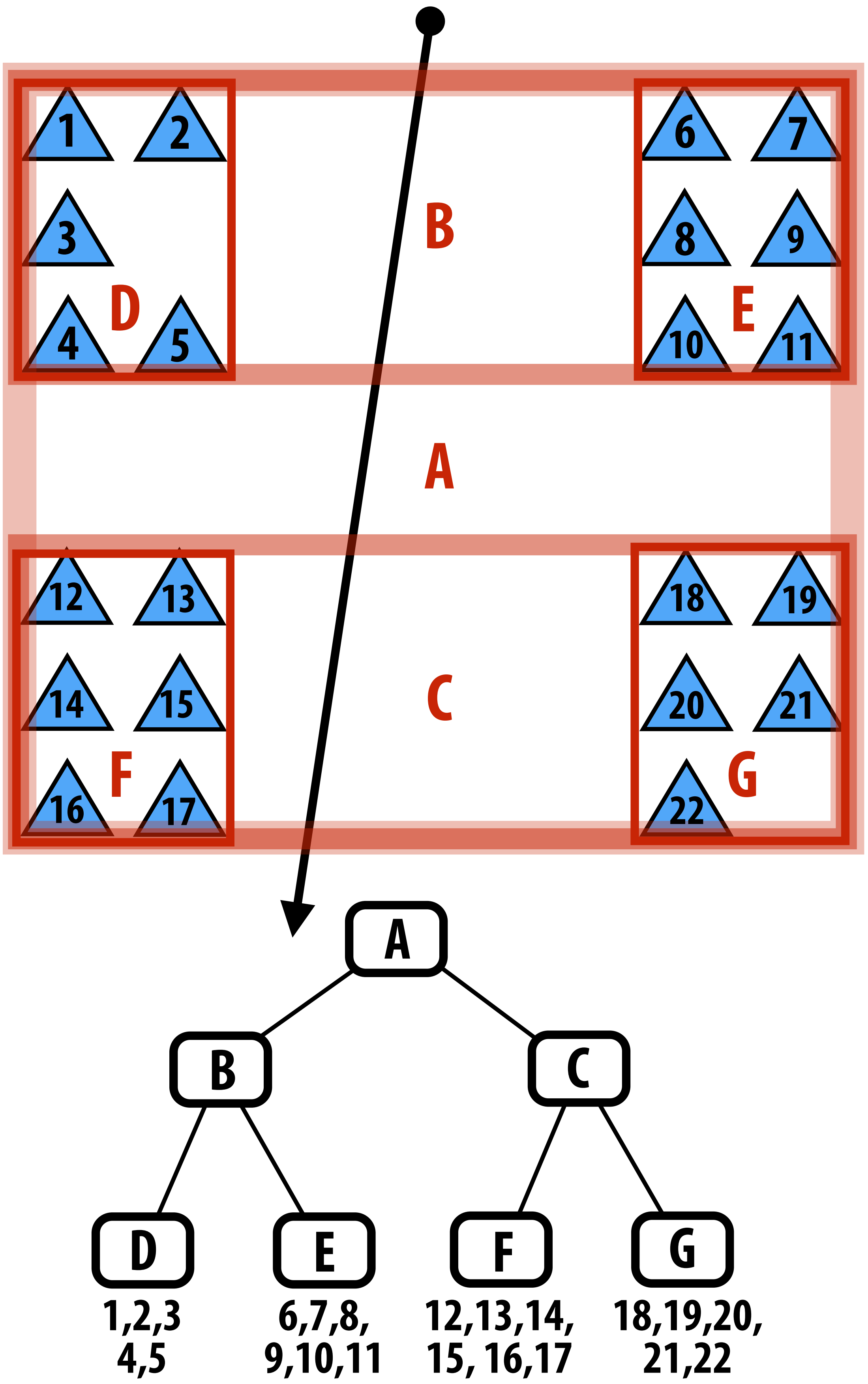
Today

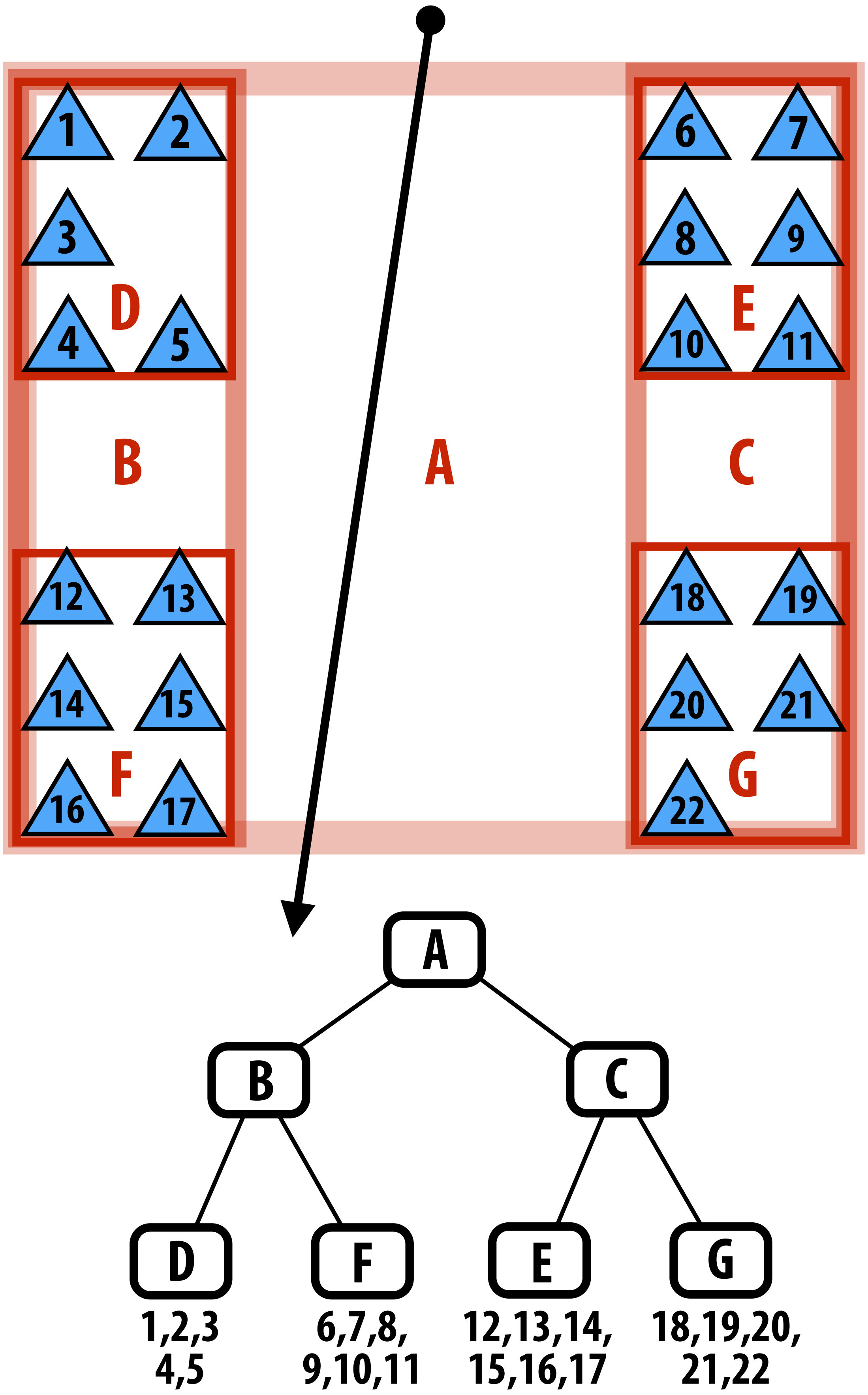
- Finish up real-time ray tracing:
  - Fast BVH construction
  - Real time ray tracing APIs and hardware
  - Role of neural post-processing to improve images

- Small amount of prep for Tuesday's speaker (Brian Karis, Epic)

A quick discussion of how to build BVHs

Bounding volume hierarchy (BVH)





Left: two different BVH organizations of the same scene containing 22 primitives.

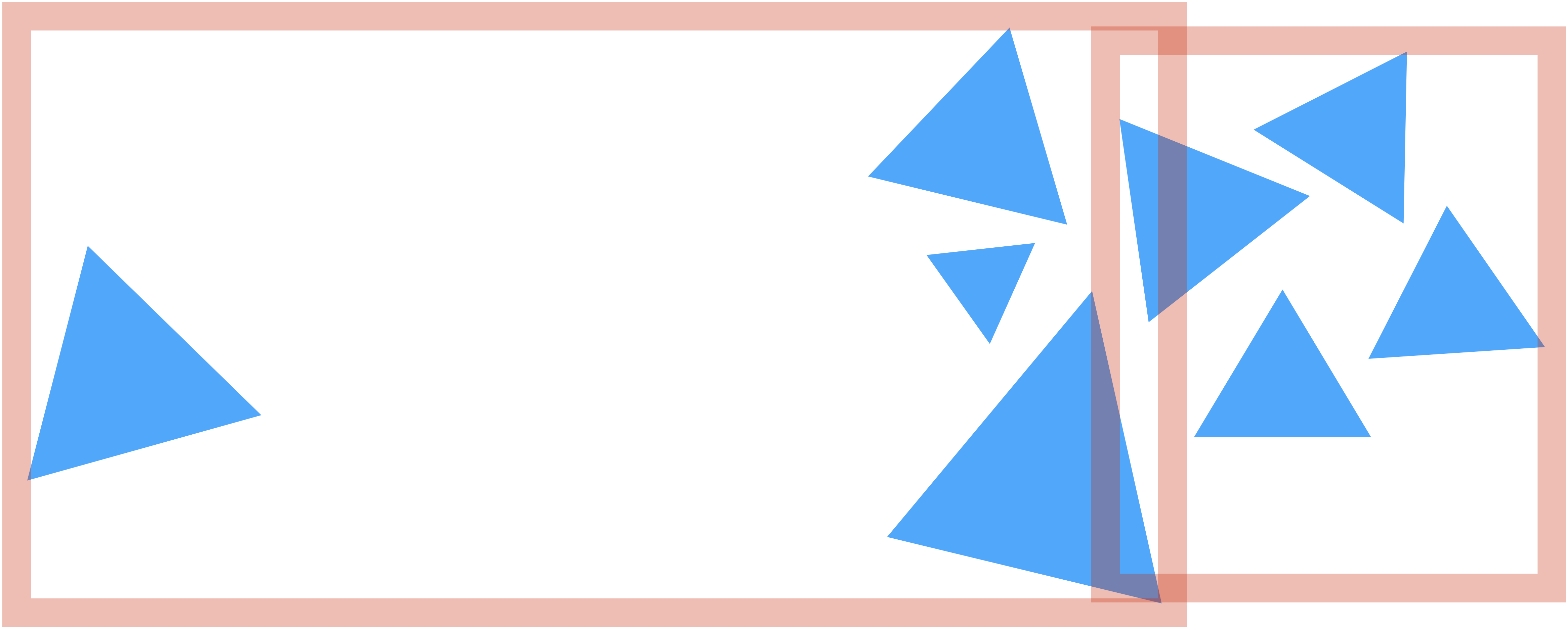
Is one BVH better than the other for THIS PARTICULAR RAY?

For a given set of primitives, there are many possible BVHs

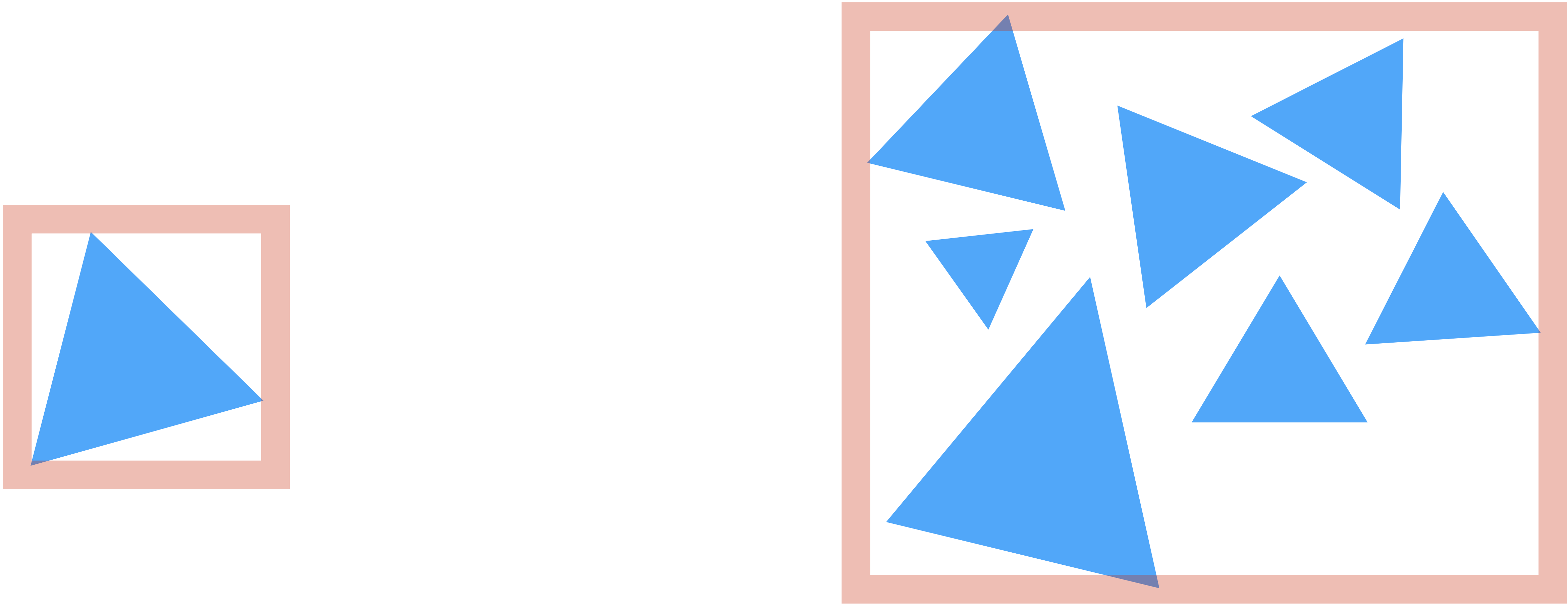
(2^N ways to partition N primitives into two groups)

😱 😱

Intuition about a "good" partition?



Partition into child nodes with equal numbers of primitives



Better partition

Intuition: avoid bboxes with significant empty space

Which partition is fastest?

What is the cost of tracing a ray through a subtree rooted by "node"?

Cost(node)= C_trav

+ Prob(hit L)*Cost(L)

+ Prob(hit R)*Cost(R)

C_trav = cost of traversing a node (e.g., loading node data, computing ray-box intersection)

Cost(L) = cost of traversing left child

Cost(R) = cost of traversing right child

Basic "top-down" greedy BVH build

```
Partition(list of prims) {

  if (termination criteria reached) {
    // make leaf node
  }

  (prim_list_1, prim_list2) = find_cost_minimizing_split_point(list of prims);

  // recursive calls can execute in parallel
  left_child = Partition(prim_list_1)
  right_child = Partition(prim_list_2)
}
```

Modern, fast and high quality BVH construction schemes

- Step 1: build low-quality BVH quickly

- Step 2: Use initial BVH to accelerate construction of high-quality BVH

Building a low-quality BVH quickly

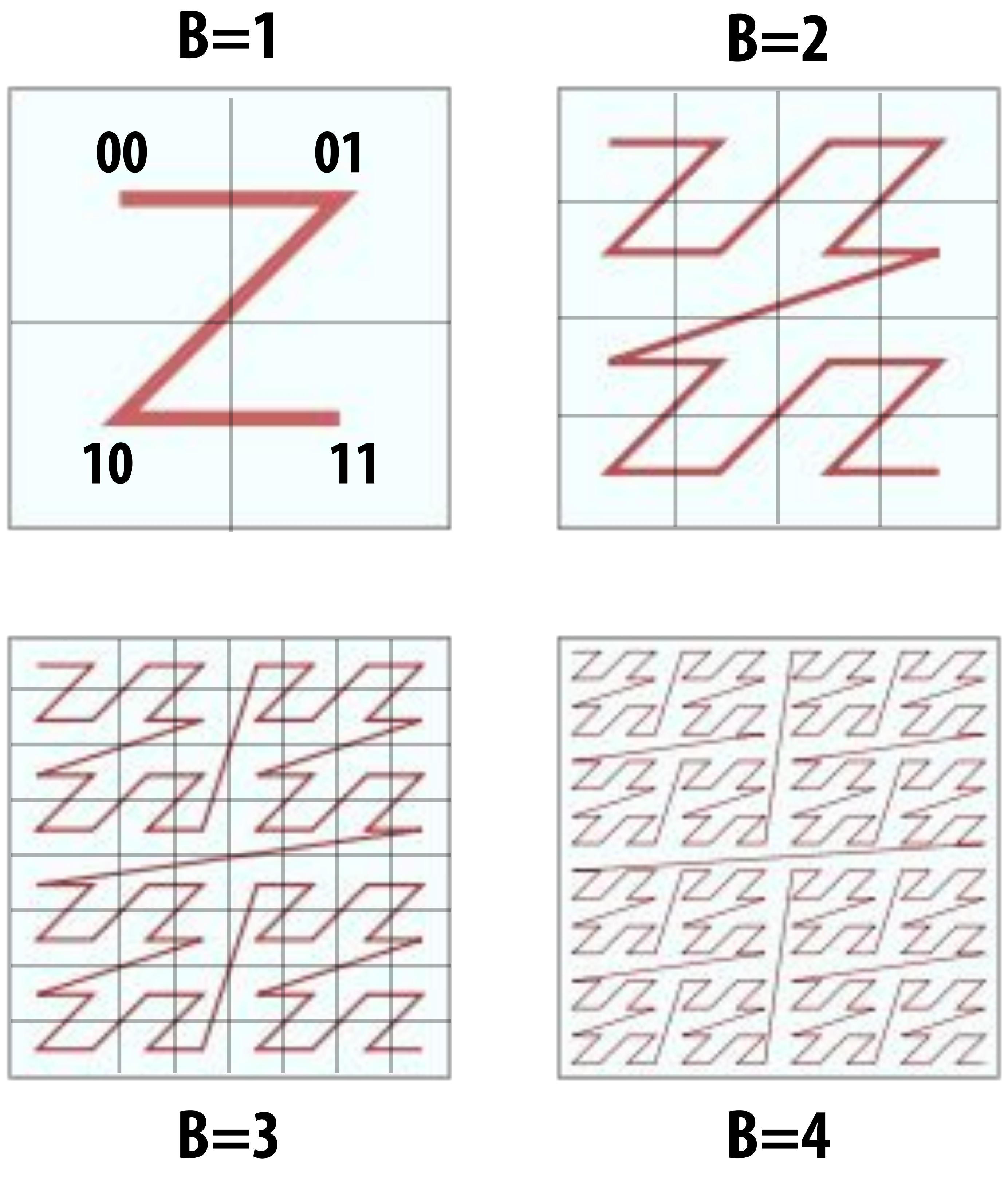
[Lauterbach 09, Pantaleoni 10]

1. Discretize each dimension of scene into 2^B cells
2. [DATA PARALLEL] Compute index of centroid of bounding box of each primitive: (c_i, c_j, c_k)
3. Interleave bits of c_i, c_j, c_k to get 3B bit-Morton code
4. [DATA-PARALLEL] Sort primitives by Morton code (primitives now ordered with high locality in 3D space: in a space-filling curve!)
   - O(N) parallel radix sort

Leads to simple, highly parallelizable BVH build:

```
Partition(int i, primitives):
  node.bbox = bbox(primitives)
  (left, right) = partition prims by bit i
  if there are more bits:
     Partition(left, i+1);
     Partition(right, i+1);
  else:
     make a leaf node
```

2D Morton Order



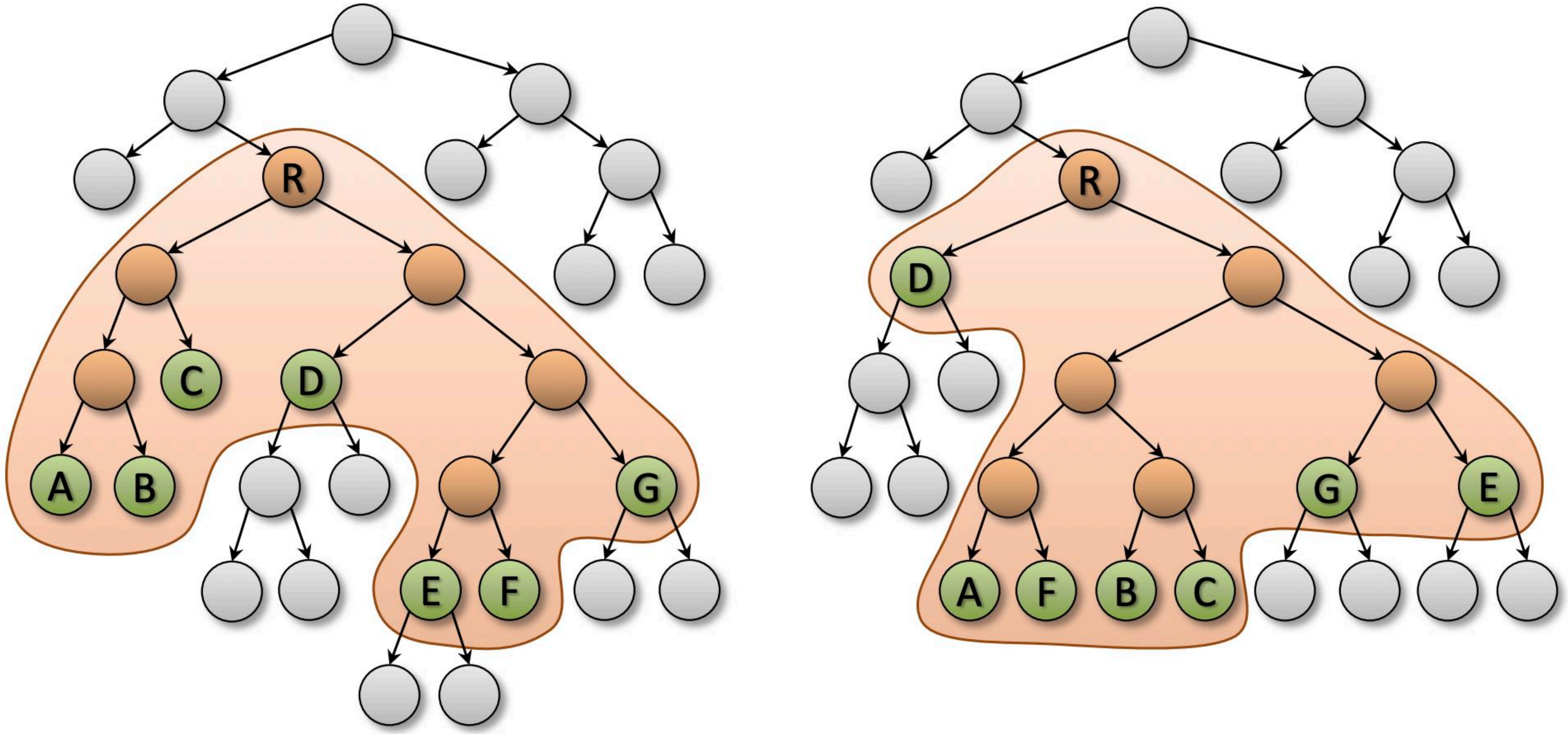
Karras 2013 bottom up treelet-based construction

Step 1: (top down) build low quality BVH quickly using Morton codes

Step 2: (bottom up) walk from leaves toward root forming small treeless

For each treelet, exhaustively try all possible combinations to find optimal (cost minimizing) treelet

- Brute force search implemented using dynamic programming method

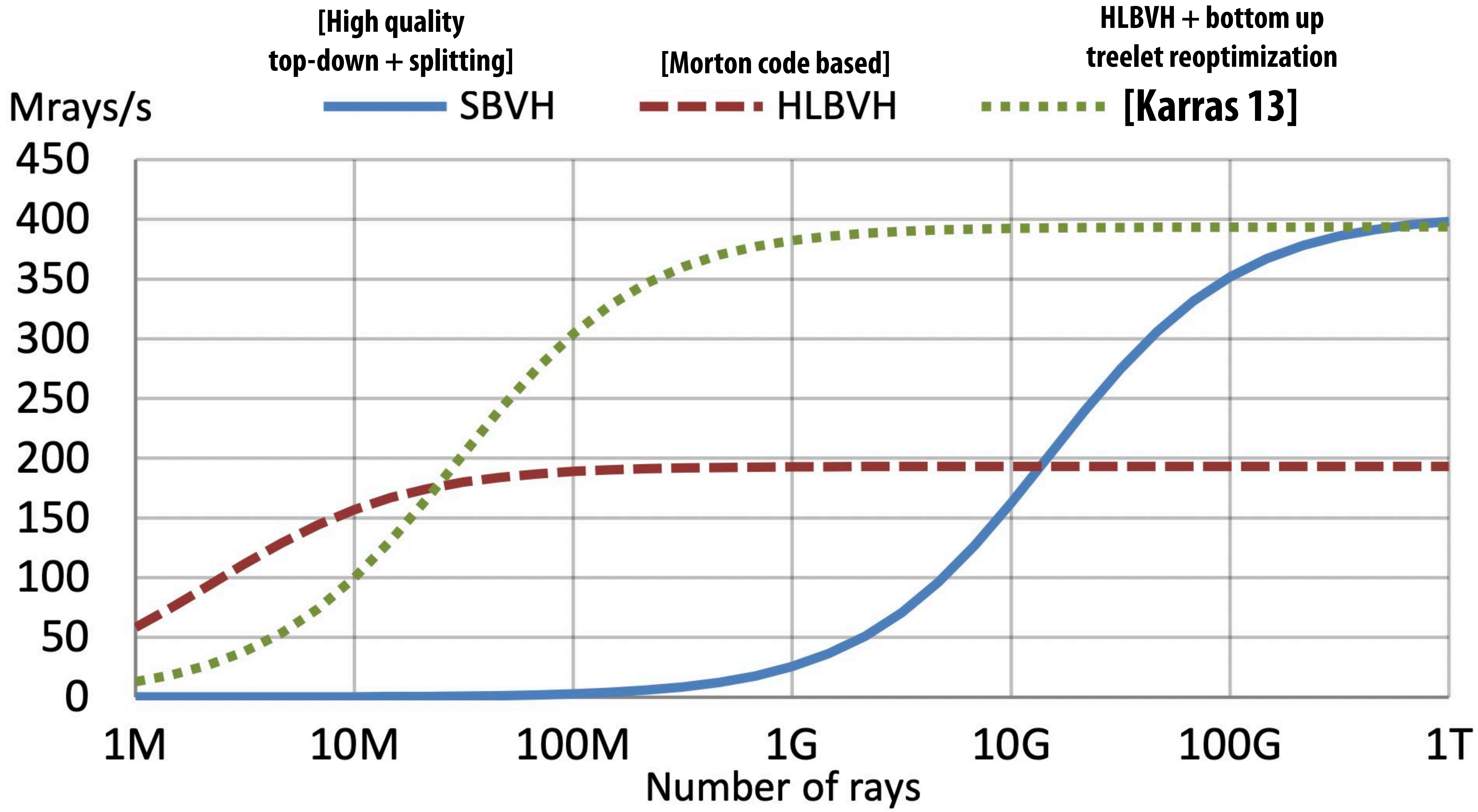


Shaded region:
treelet with 7 leaf nodes

After optimization: this is the optimal treelet for
these nodes (minimal cost)

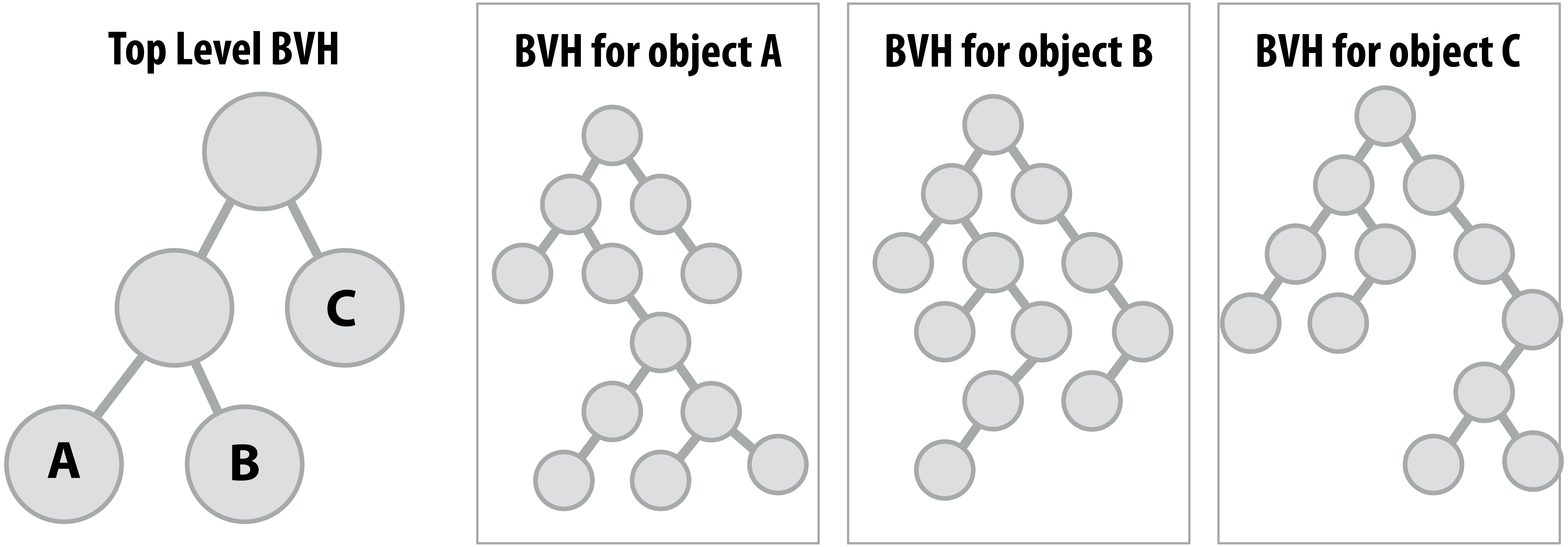
Can afford to build a better BVH if you are shooting many rays (can amortize cost)

- The graph below plots effective ray throughput (Mrays/sec) as a function of the number of rays traced per BVH build
  - More rays = can amortize costs of BVH build across many ray trace operations



Two-level BVHs

- Many scene objects do not move from frame-to-frame, or only move rigidly
- Approach: two-level BVH: build a BVH over per-object BVHs
  - Only rebuild this top level BVH each frame as objects move



Contains hundreds of scene objects

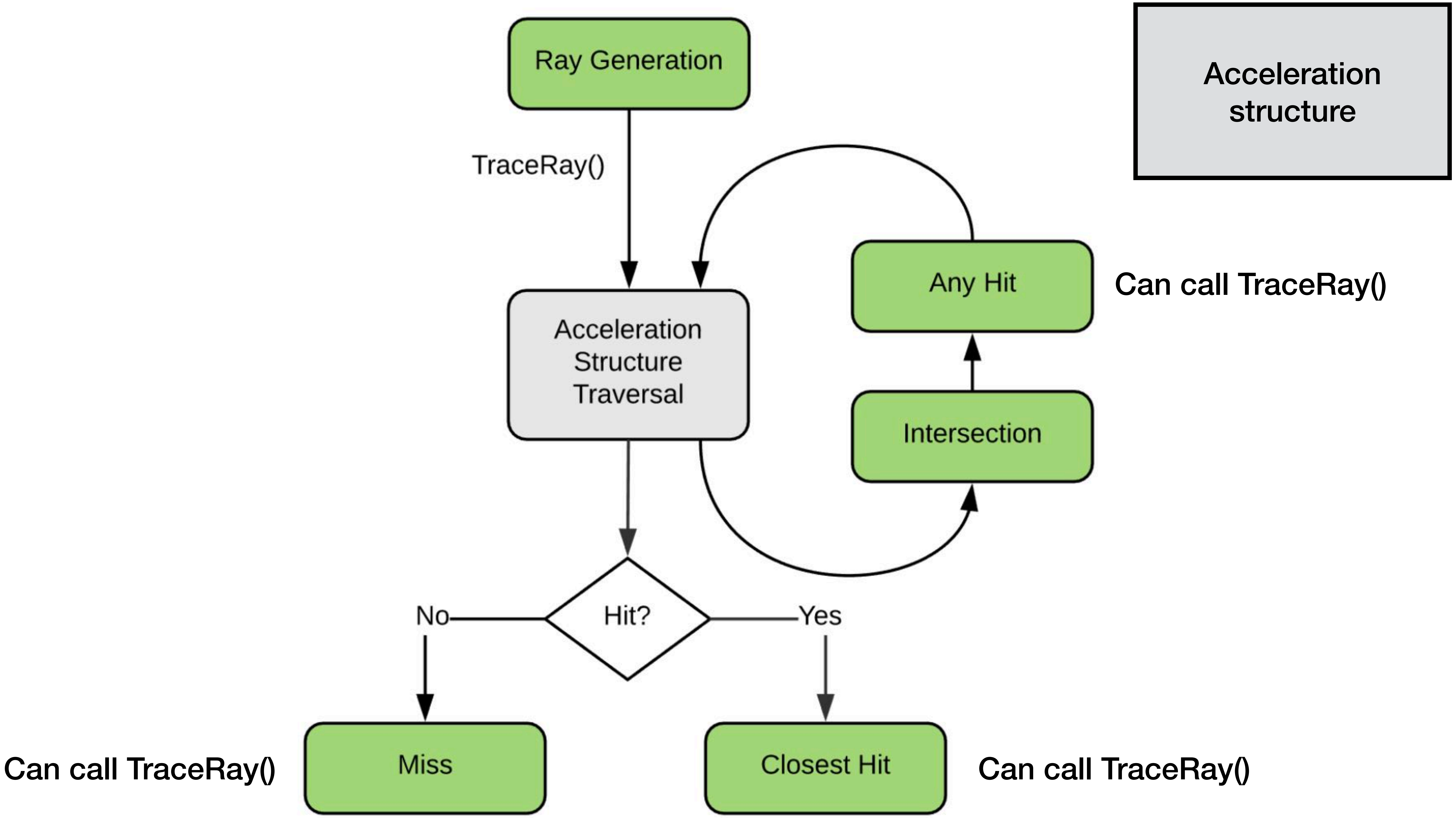
Each per-object BVH might contains tens's of thousands of triangles.
If object's geometry does not undergo relative change
(other than rotation/translation in world)
the BVH can be built once and remain applicable.

Real-time ray tracing APIs

(Recurring theme in this course: increase level of abstraction to enable optimized implementations)

D3D12's DXR ray tracing "stages"

- Ray tracing is abstracted as a graph of programmable "stages"
- TraceRay() is a blocking function in some of those stages



Example: ray generation shader (camera rays)

```
// This represents the geometry of our scene.
RaytracingAccelerationStructure scene : register(t5);

[shader("raygeneration")]
void RayGenMain()
{
    // Get the location within the dispatched 2D grid of work items
    // (often maps to pixels, so this could represent a pixel coordinate).
    uint2 launchIndex = DispatchRaysIndex();

    // Define a ray, consisting of origin, direction, and the t-interval
    // we're interested in.
    RayDesc ray;
    ray.Origin = SceneConstants.cameraPosition.
    ray.Direction = computeRayDirection( launchIndex ); // assume this function exists
    ray.TMin = 0;
    ray.TMax = 100000;

    Payload payload;

    // Trace the ray using the payload type we've defined.
    // Shaders that are triggered by this must operate on the same payload type.
    TraceRay( scene, 0 /*flags*/, 0xFF /*mask*/, 0 /*hit group offset*/,
              1 /*hit group index multiplier*/, 0 /*miss shader index*/, ray, payload );

    outputTexture[launchIndex.xy] = payload.color;
}
```

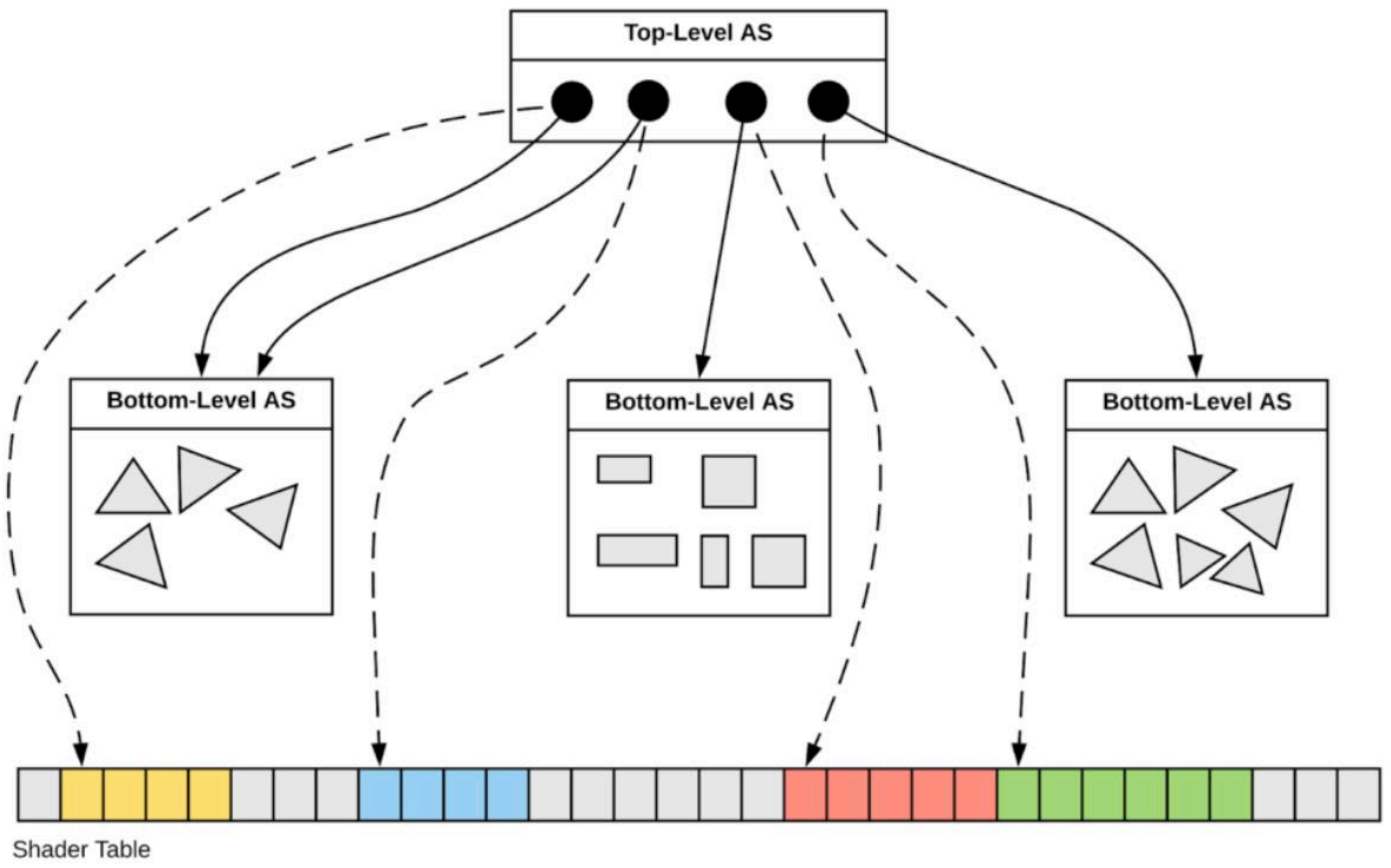
Example "hit shader": Runs on ray hit to fill in payload

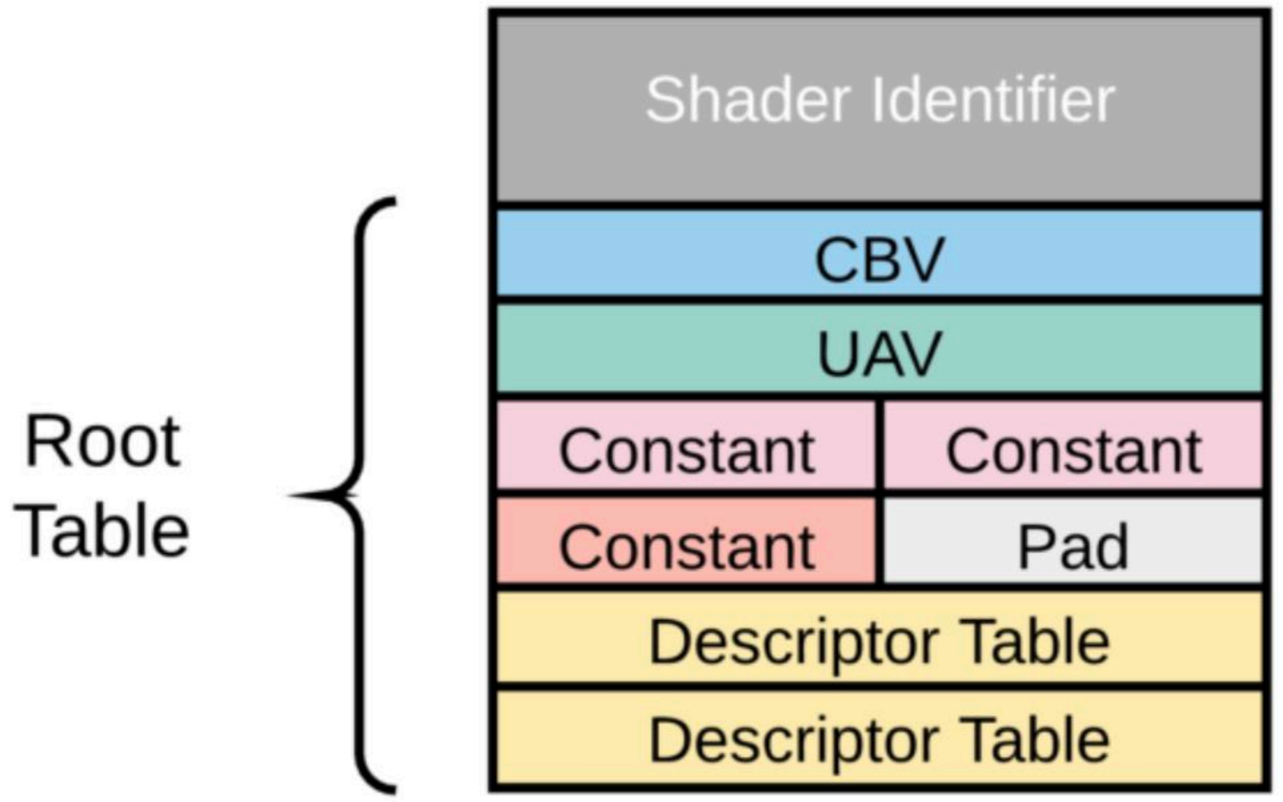
```
// Attributes contain hit information and are filled in by the intersection shader.
// For the built-in triangle intersection shader, the attributes always consist of
// the barycentric coordinates of the hit point.
struct Attributes
{
    float2 barys;
};

[shader("closesthit")]
void ClosestHitMain( inout Payload payload, in Attributes attr )
{
    // Read the intersection attributes and write a result into the payload.
    payload.color = float4( attr.barys.x, attr.barys.y,
                            1 - attr.barys.x - attr.barys.y, 1 );

    // Demonstrate one of the new HLSL intrinsics: query distance along current ray
    payload.hitDistance = RayTCurrent();
}
```

GPU understands format of BVH acceleration structure and "shader table"



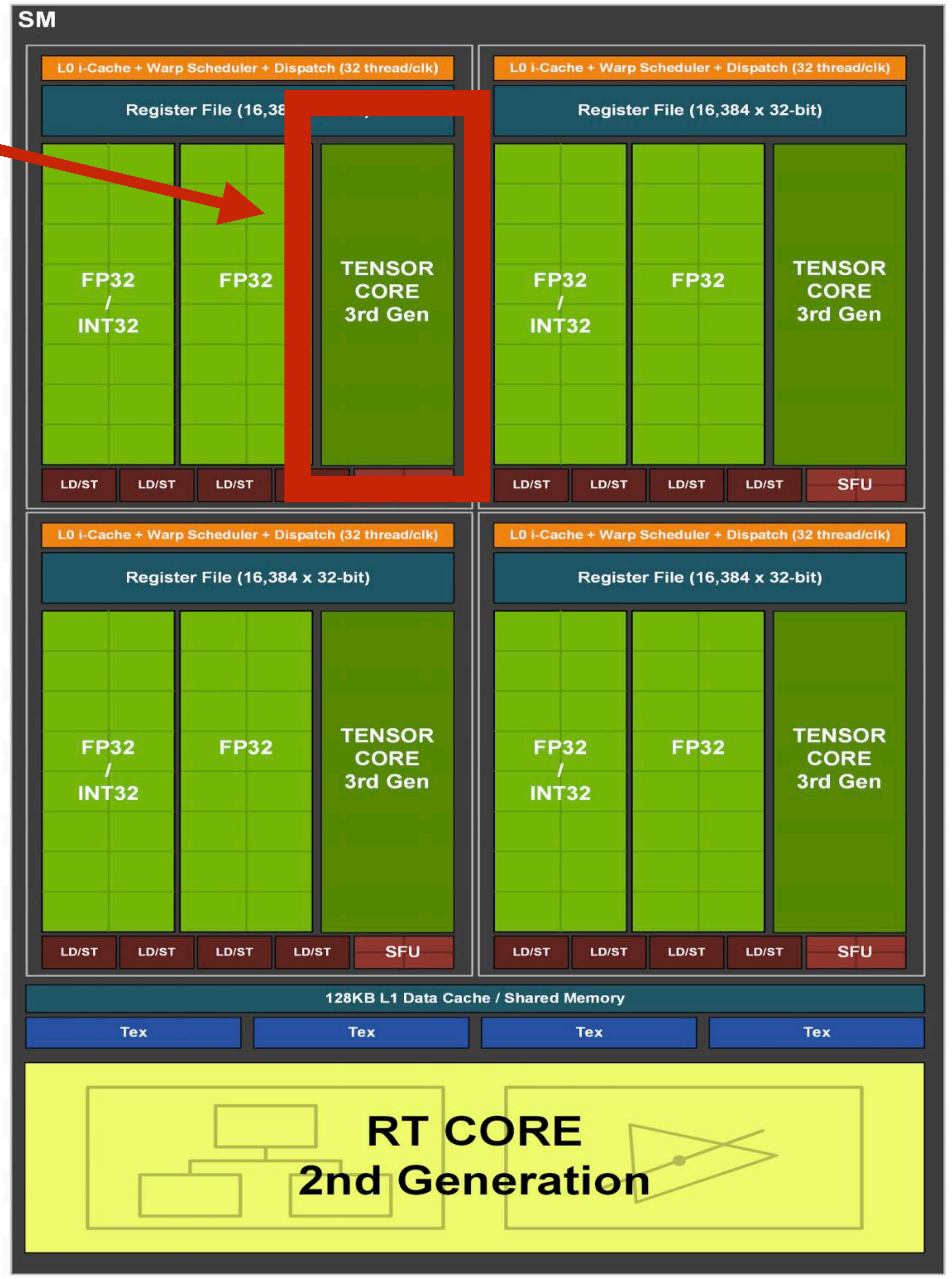


Implementation: NVIDIA Ampere SM (RTX 3xxx series)

- Hardware support for ray-triangle intersection and ray-BVH intersection (“RT core”)

- Very little public documentation of architectural details at this time

- But the RT hardware is not the only fixed-function hardware on a GPU that is important for real-time raytracing…



Denoising ray traced images


VICTROLA

Surface Albedo

Surface normals

Recall: numerical integration of light (via Monte Carlo sampling) suffers from high variance, resulting in images with “noise”

16 paths/pixel


64 paths/pixel


256 paths/pixel


1024 paths/pixel

4096 paths/pixel

Denoised results


16 paths/pixel


16 paths/pixel (denoised)
VICTROLA

64 paths/pixel (denoised)

256 paths/pixel (denoised)

1024 paths/pixel (denoised)

4096 paths/pixel (denoised)

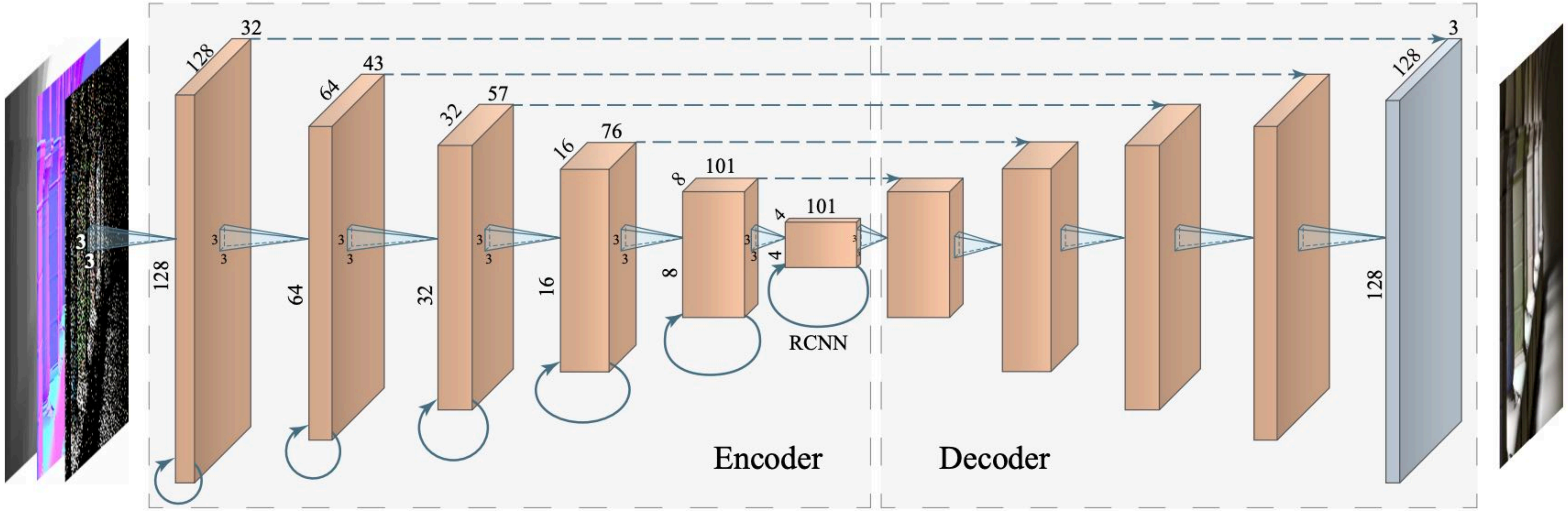
4096 paths/pixel (NOT DENOISED)

Deep learning-based denoising

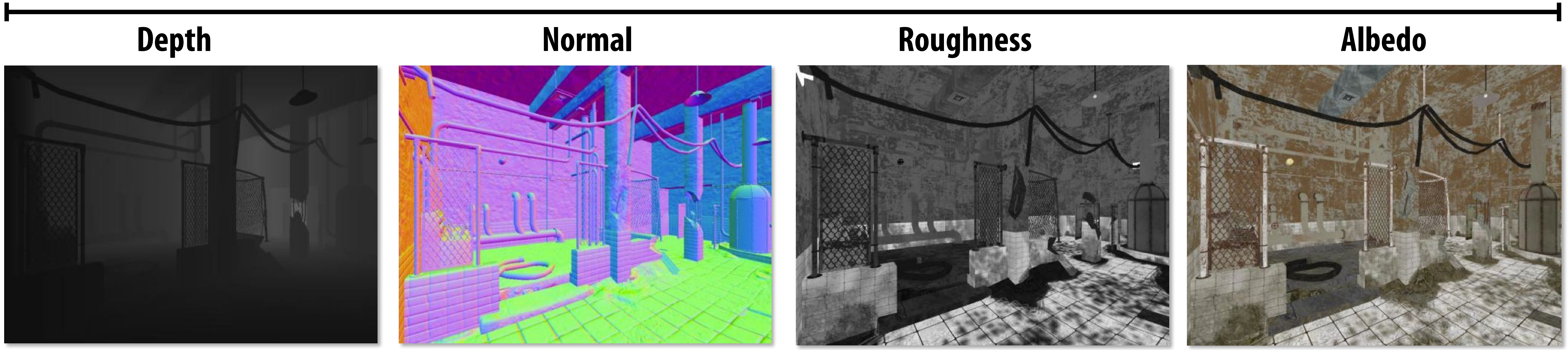
- Can we "learn" to turn noisy images into clean ones?

- Idea: Use neural image-to-image transfer methods to convert cheaper to compute (but noisy) ray traced images into higher quality images that look like they were produced by tracing many rays per pixel

Example: neural denoiser DNN



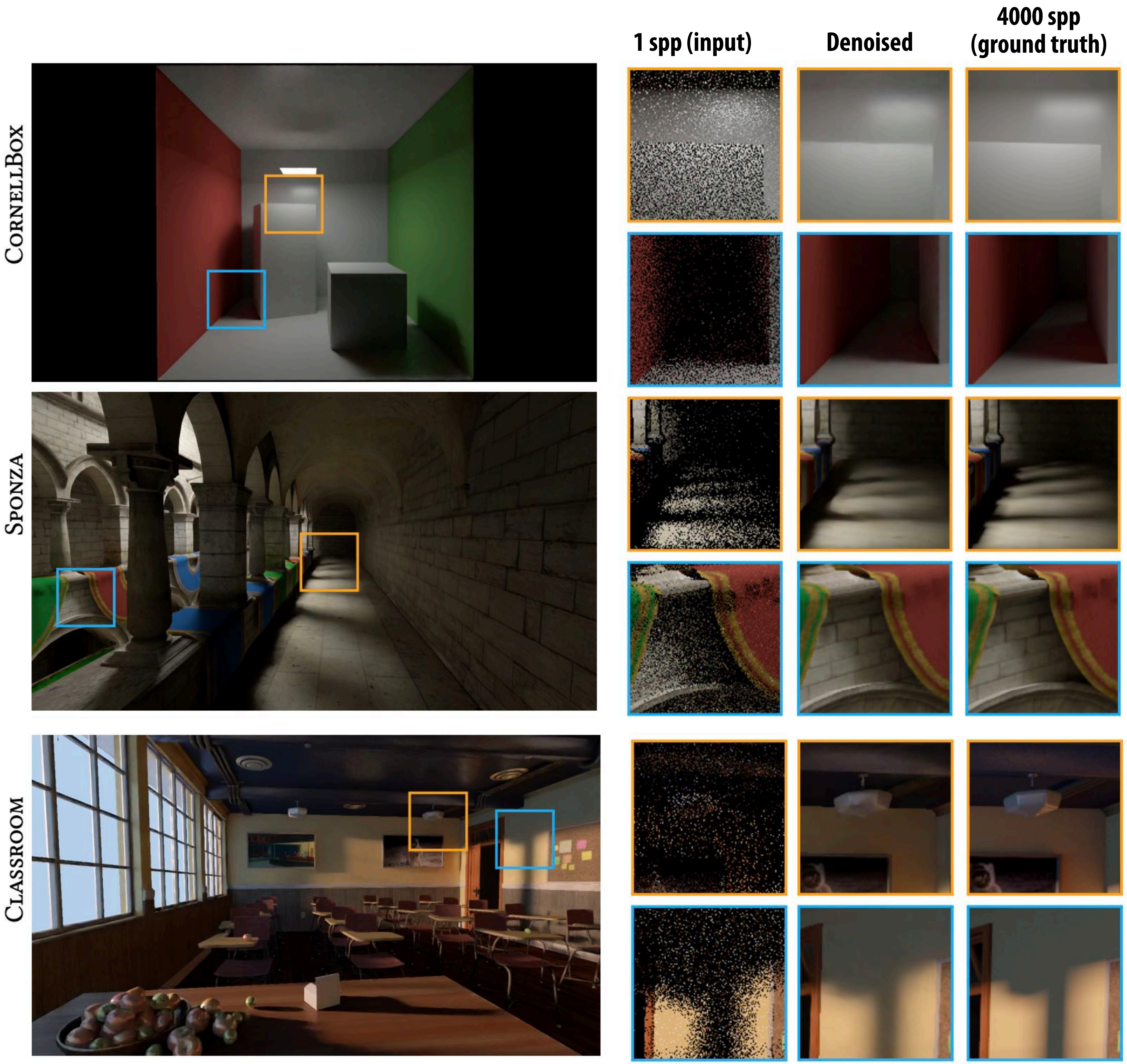
Input to network is noisy RGB image * + additional normal, depth, and roughness channels
(These are cheap to compute inputs help network identify silhouettes, sharp structure)



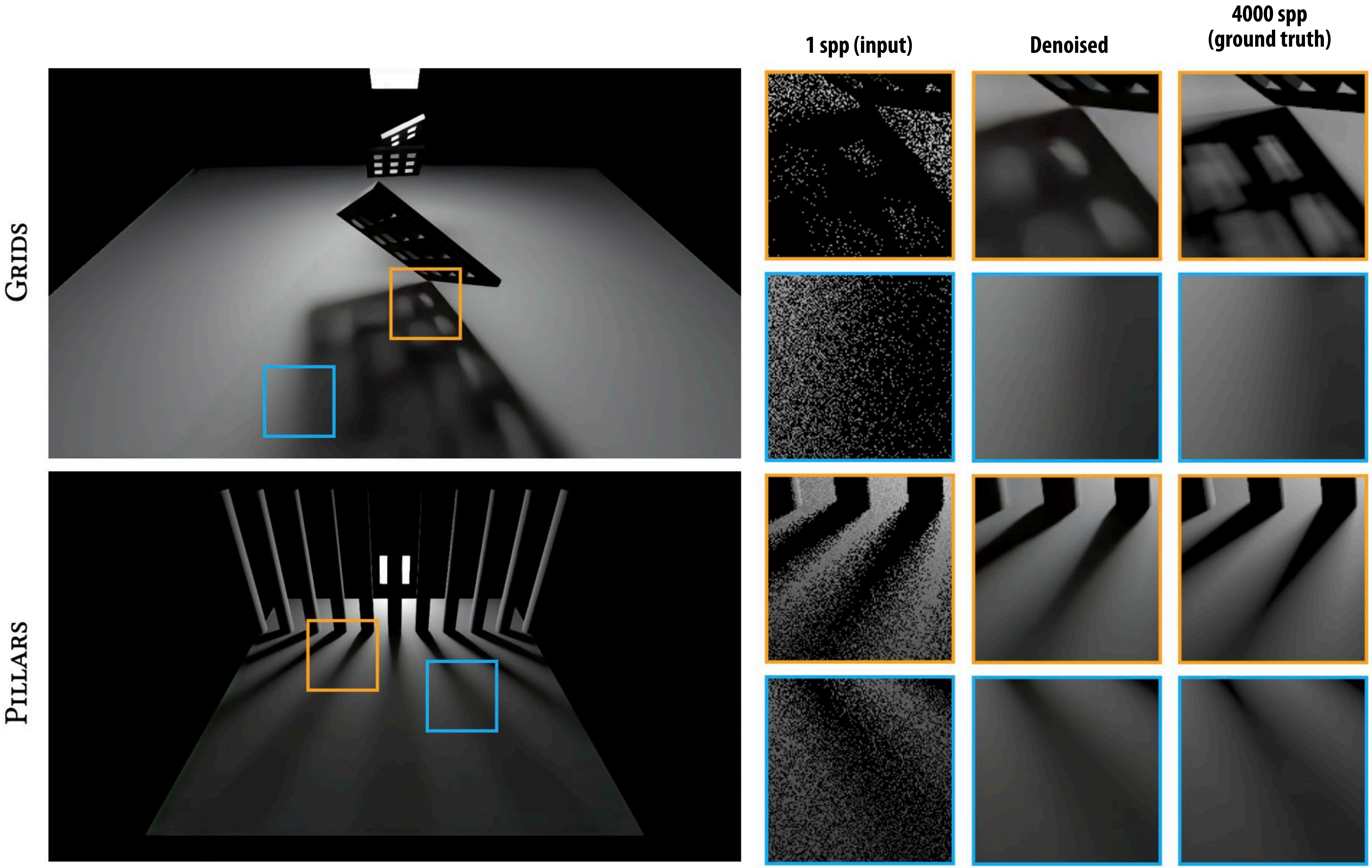
*** Actually the input is RGB demodulated by (divided by) texture albedo (don't force network to learn what texture was)**

[Chaitanya 17]

Denoising results



Denoising results (challenging)



More denoising examples

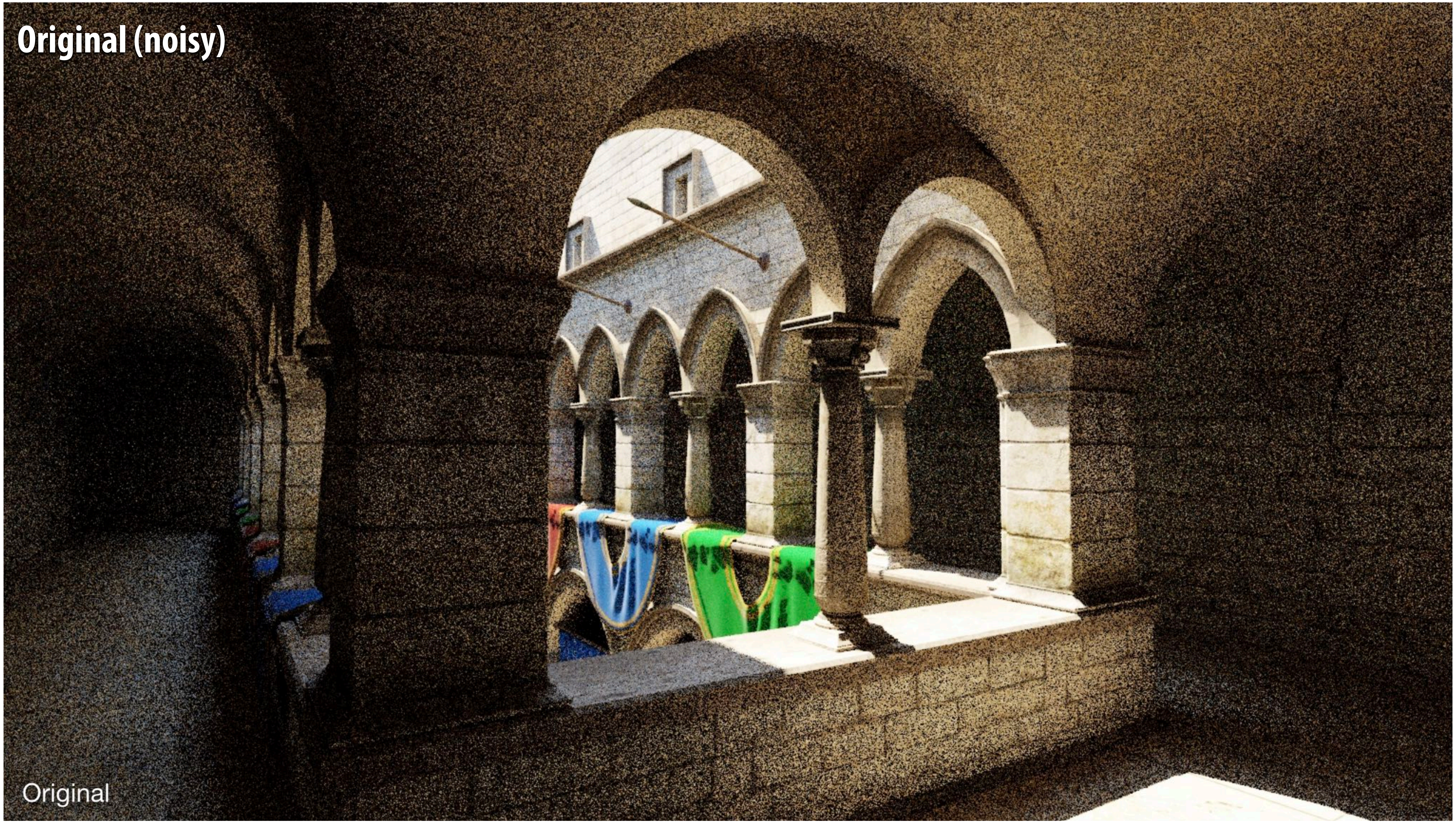


Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/

More denoising examples

Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/

More denoising examples

Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/

More denoising examples

Image credit: Intel Open Image Denoise : https://openimagedenoise.github.io/

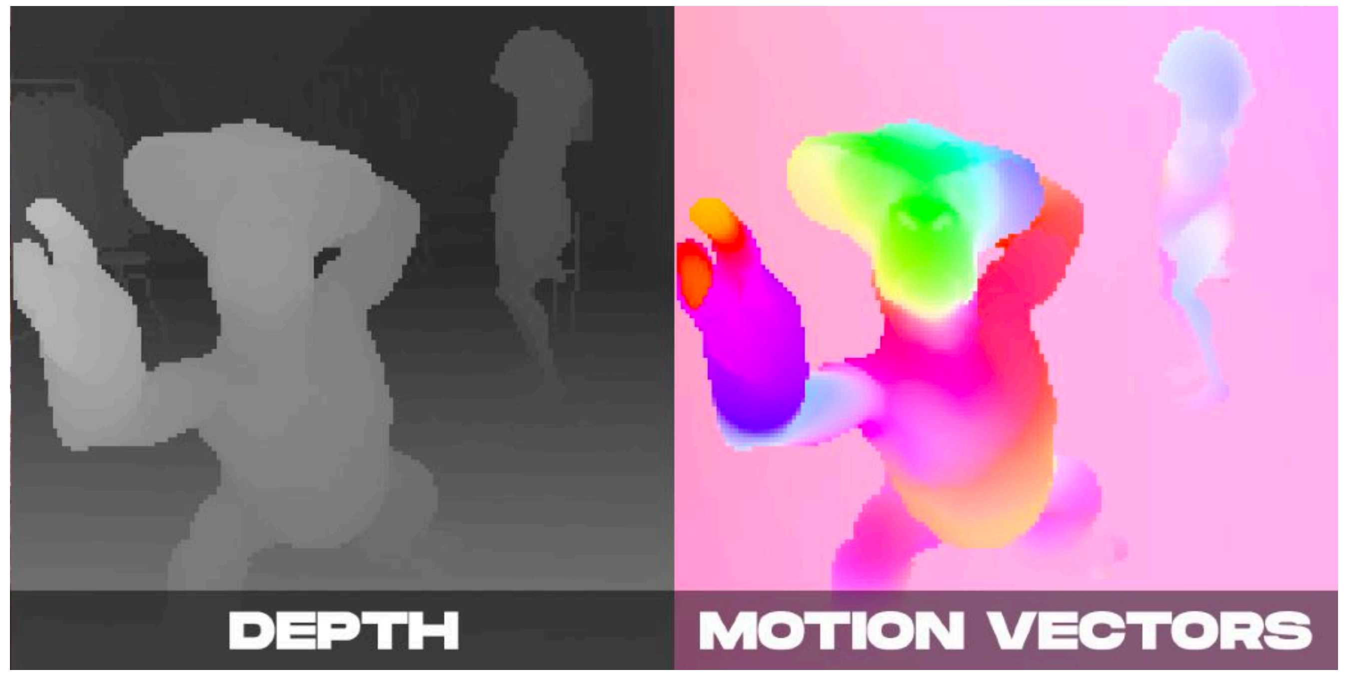
Aside: upsampling low-resolution images to higher resolution images

(This is upsampling, not reducing Monte Carlo noise.)

Examples: NVIDIA's DLSS (performs both anti-aliasing and upsampling)

Neural upsampling (hallucinating detail)

+ auxiliary inputs

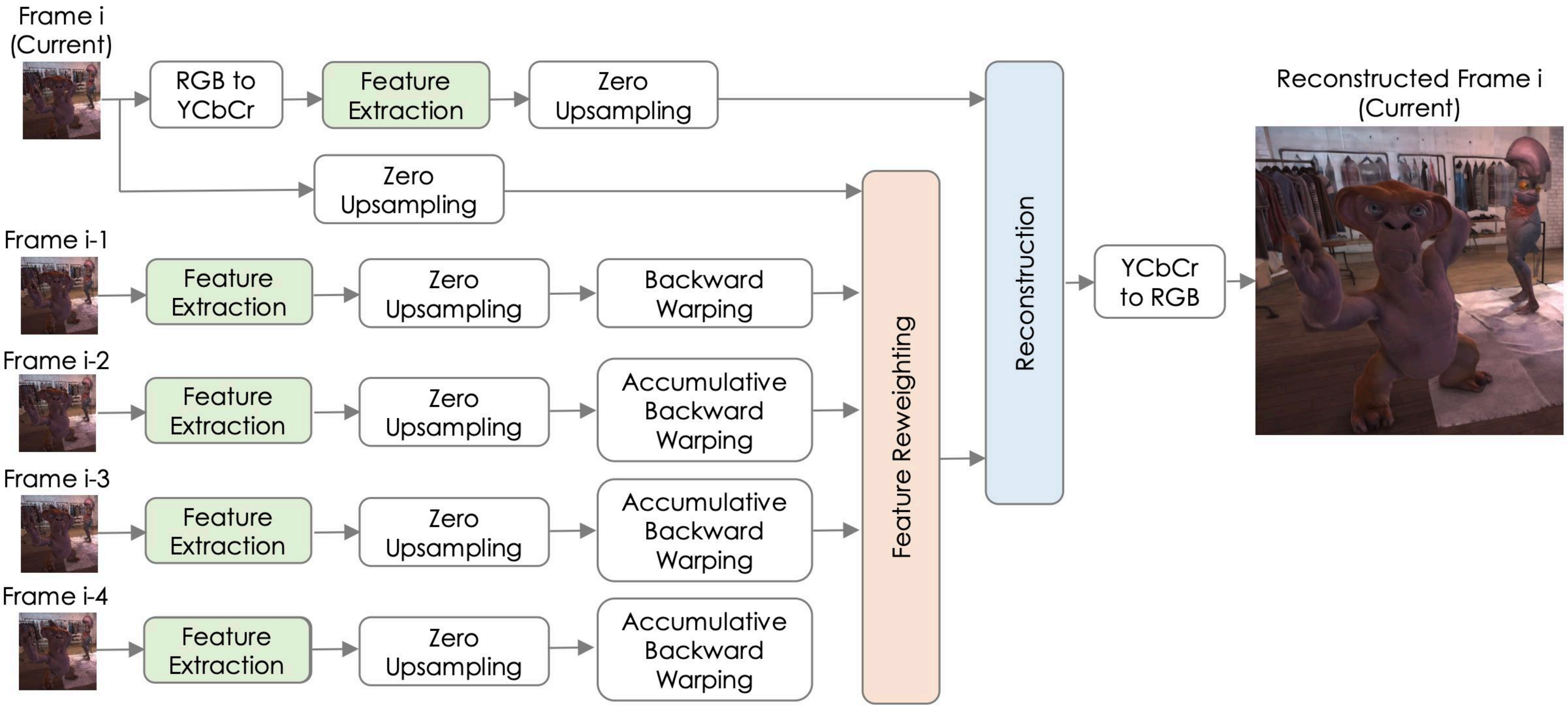


Neural upsampling (hallucinating detail)

4x4 upsampled result (16x more pixels)

Neural upsampling pipeline

[Xiao 20]



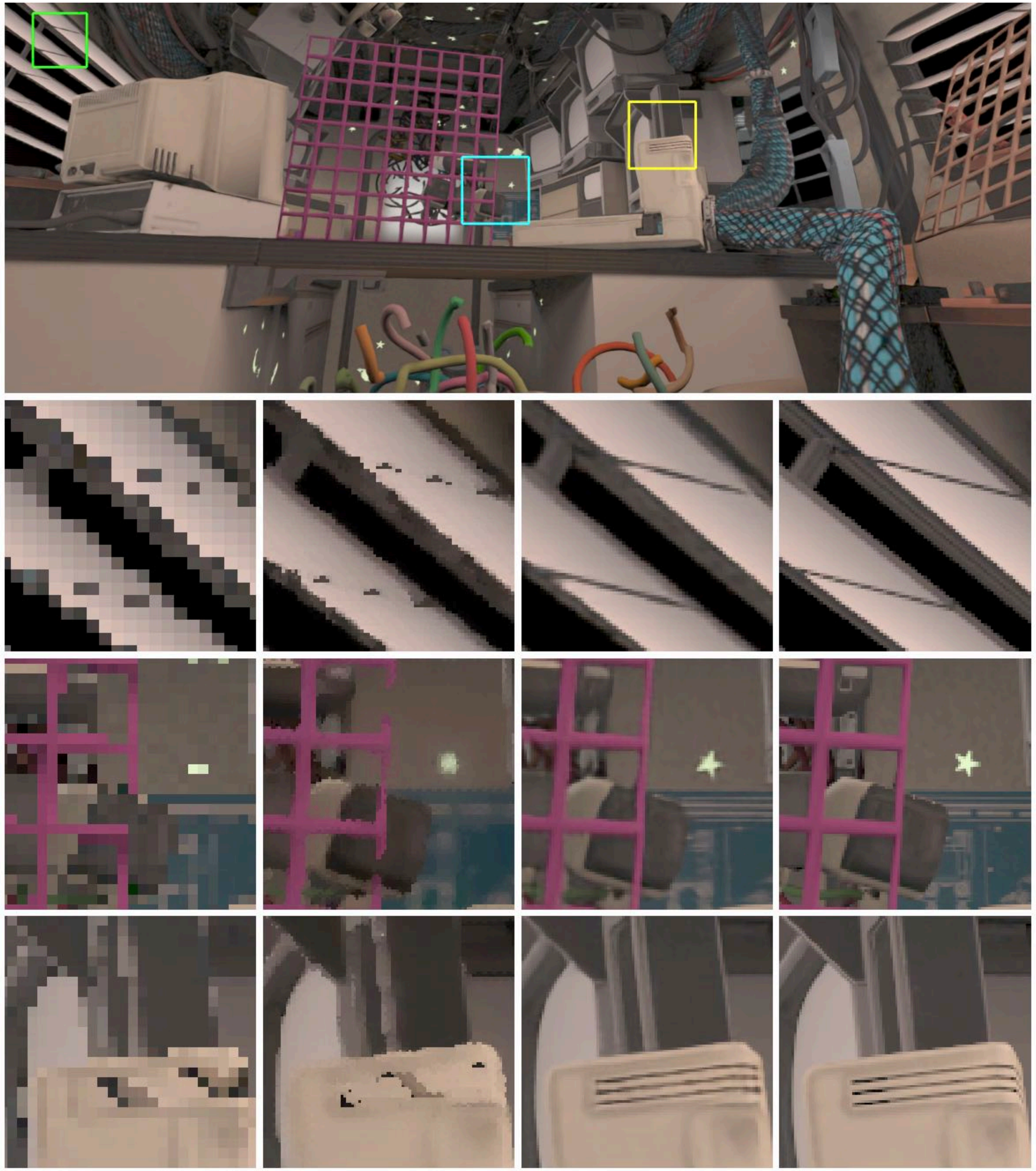
Main idea: gain resolution by aligning and merging multiple recent frames

Alignment vectors provided by renderer

Learn model that determines weights for aligned features ("feature reweighting")

Then decode with neural decoder ("reconstruction")

Closer look



Technologies that are making real-time ray tracing possible

- **Better algorithms: fast parallel BVH construction and ray-BVH traversal algorithms for GPUs and multi-core CPUs (many SIGGRAPH/HPG papers circa 2010-2017)**
  - **Main ideas of traversal: compressed, wider BVHs**
  - **Main ideas of BVH construction: two level BVH (don't rebuild everything), two phase top-down + bottom up build (high performance + high quality)**

- **Emergence of GPU hardware acceleration:**
  - **HW acceleration of ray-triangle intersection, BVH traversal**
  - **Increasingly flexible aspects of traditional GPU pipeline (bindless textures/resources)**

- **DNN-based image post-processing (denoising)**
  - **Can make plausible images using small(er) number of rays per pixel**
  - **Makes use of existing DNN hardware acceleration**

Real time ray tracing: what's next

- Continued development of specialized HW
  - More transistors = more RT cores = more rays/sec
  - Currently no hardware acceleration in game consoles (disincentive to making games completely based on RT)

- Continued application developer work to integrate tech into games
  - Application developers want a smooth adoption path (can't just throw out their current game engines and replace with a ray tracer)

- Substantial algorithmic innovation to reduce required ray counts
  - Key challenge: picking the most important directions for which to sample incoming light
  - Interesting recent results rendering scenes with many lights and with indirect illumination
  - Improvements to neural denoising techniques

Scene with many light sources (Direct lighting only)

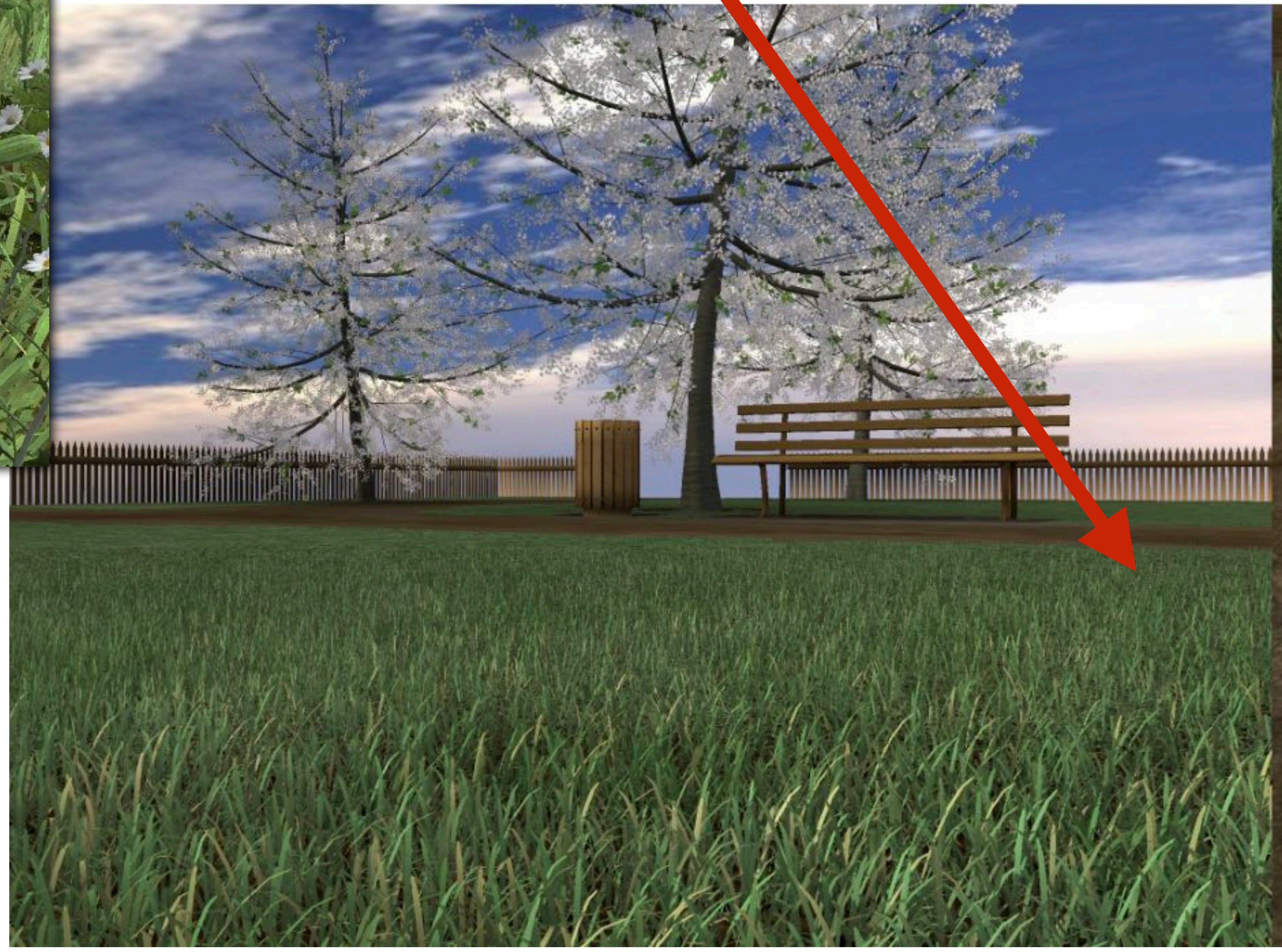
Challenges of high geometric detail scenes

High geometric detail

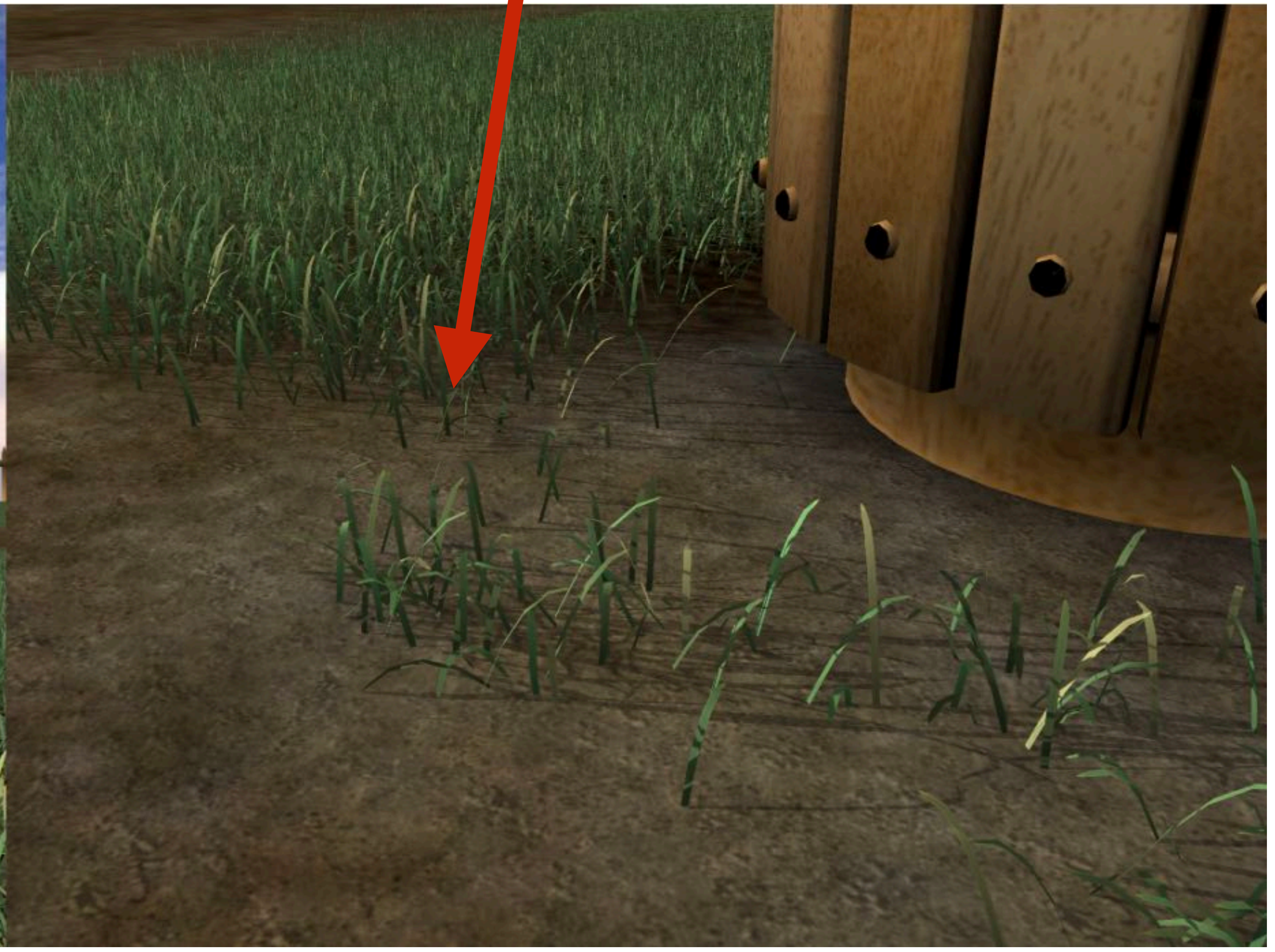
Rendering complex geometry

- How should we represent the geometry?
  - Triangle mesh? Volume (density+rgb), Subdivision surface?

Far?

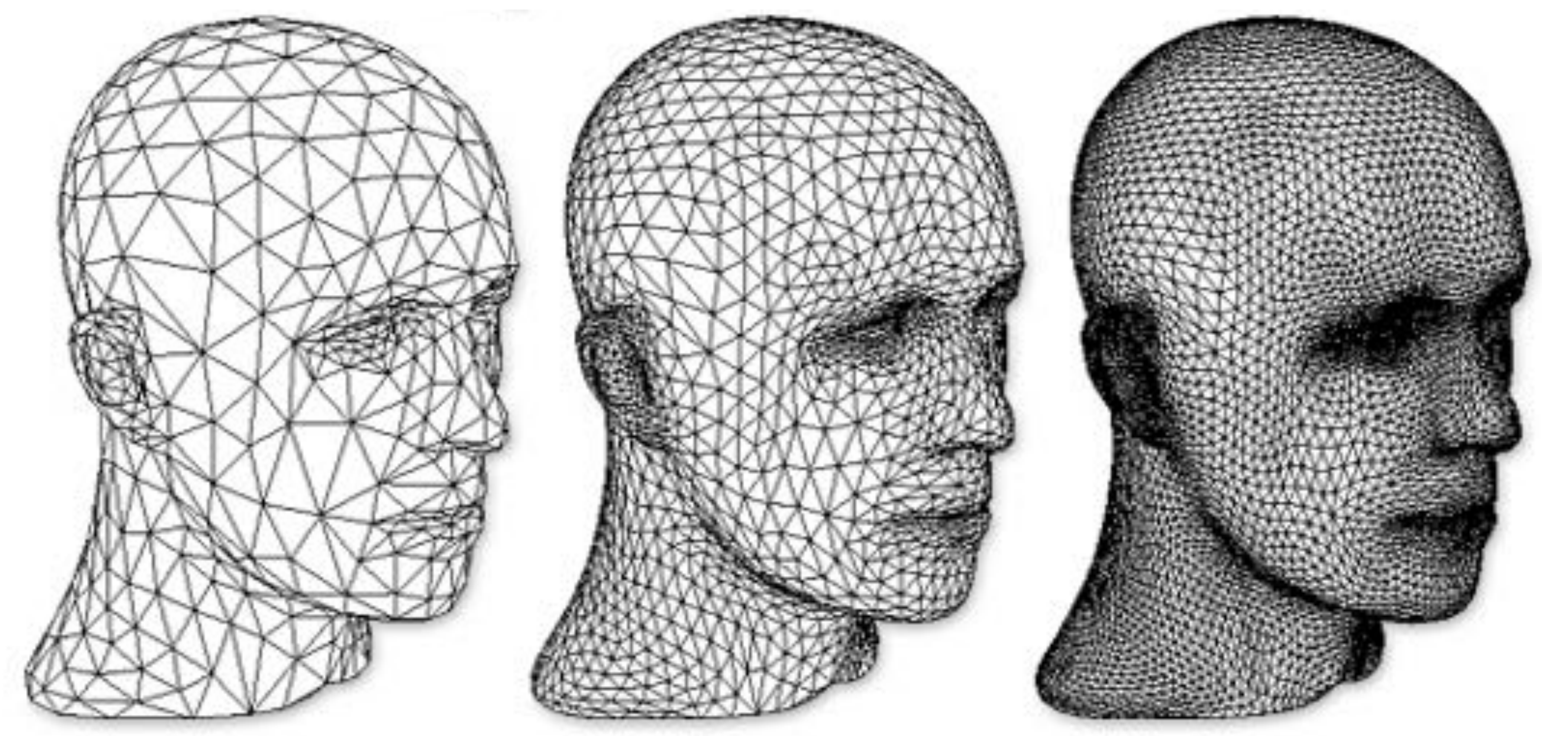


Close: want geometry

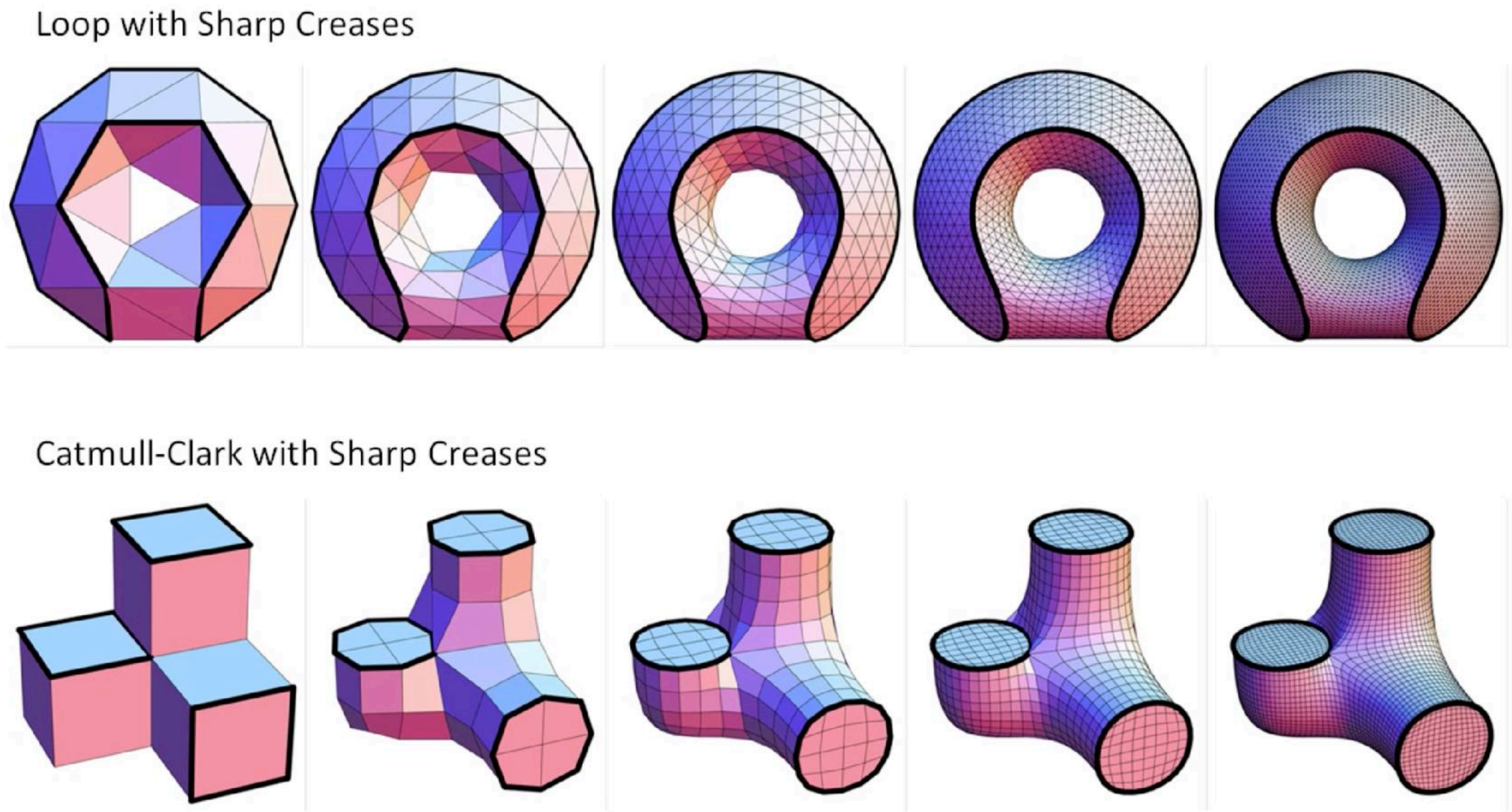


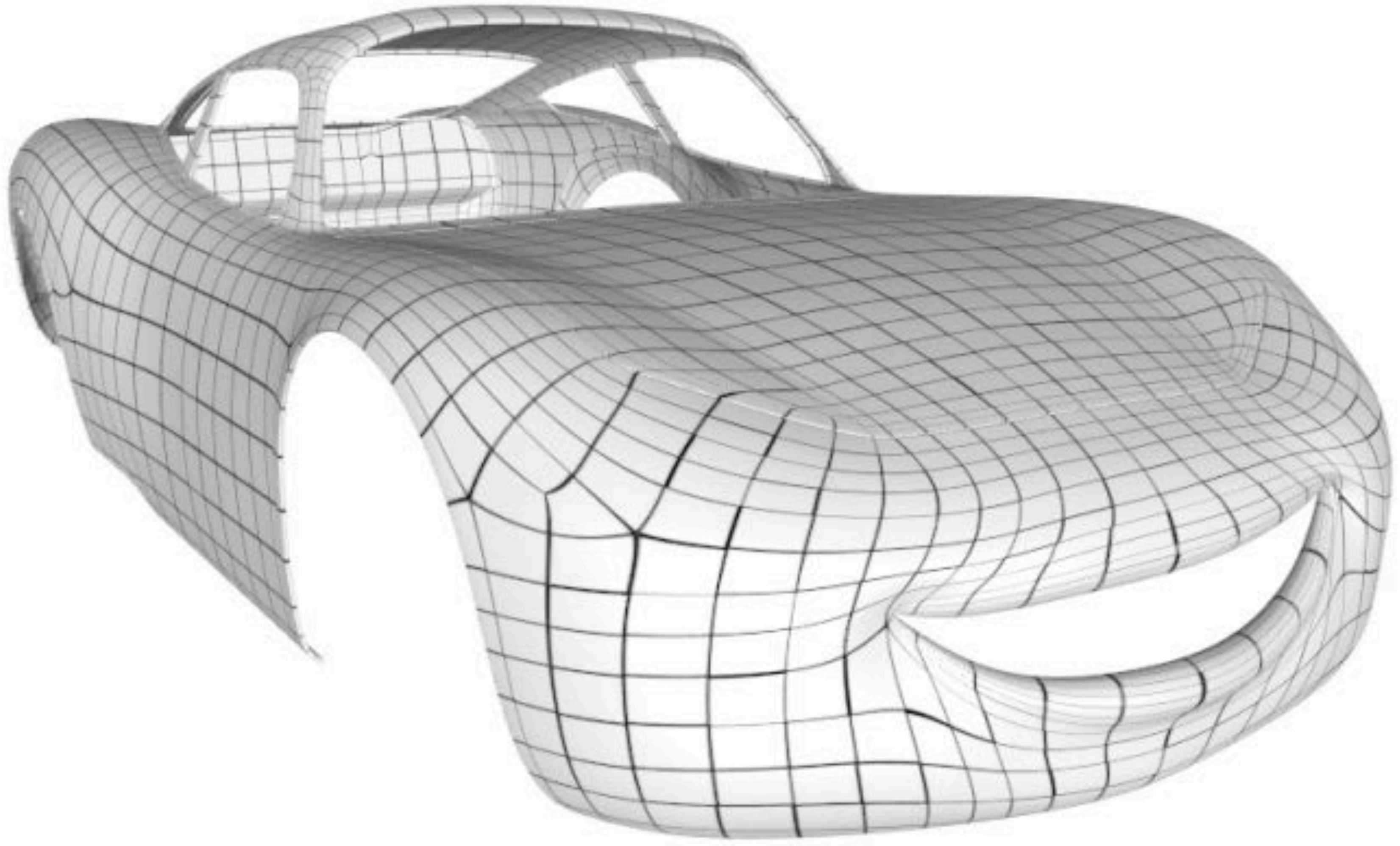
Another example: subdivision surfaces

- Subdivide coarse mesh into finer-scale mesh depending on distance to camera



Loop subdivision





Catmull-Clark control mesh and limit surface

Displaced subdivision surfaces

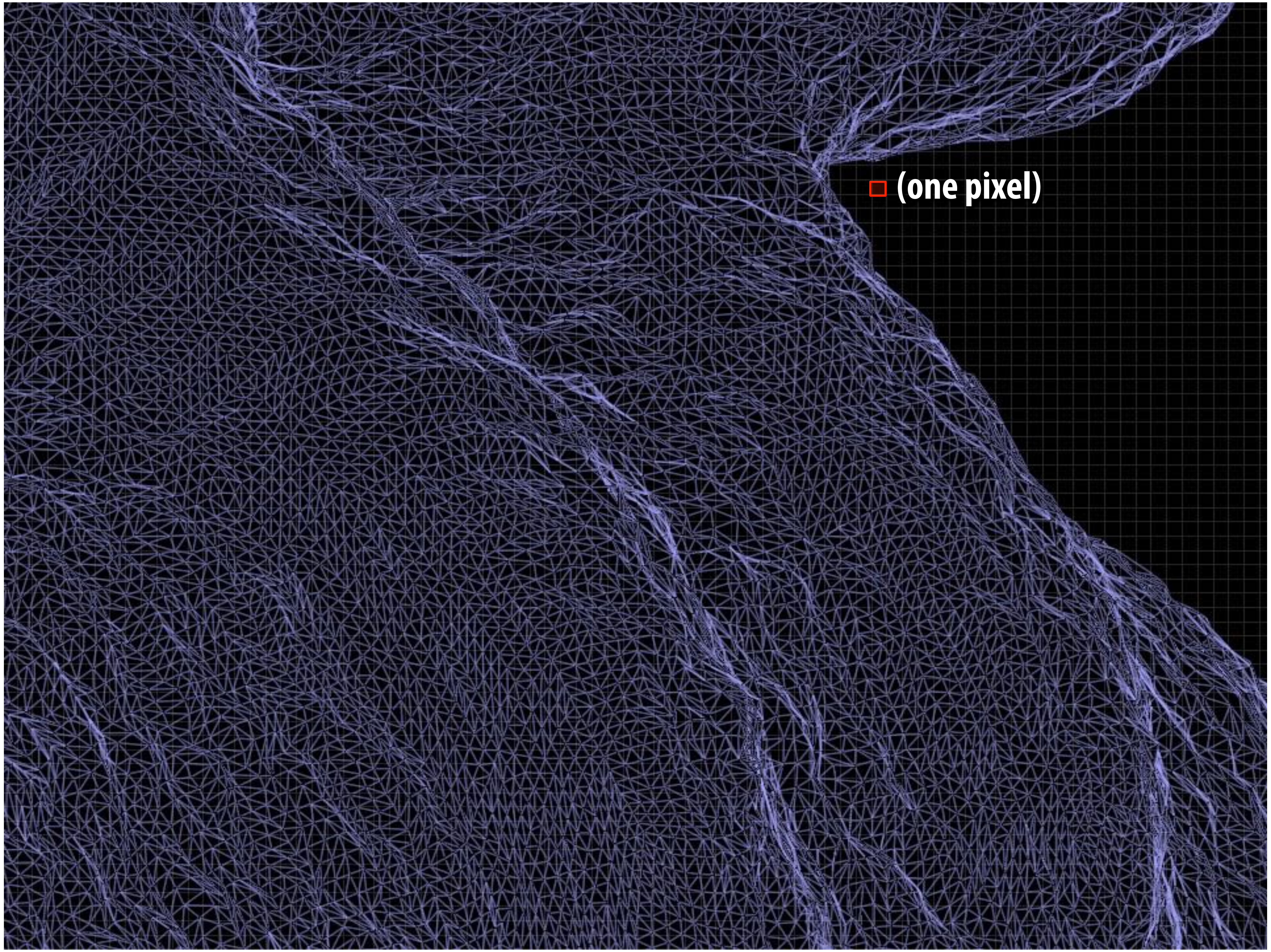
Control cage
(Coarse triangle mesh)

Limit surface
(Renders from fine triangle mesh)

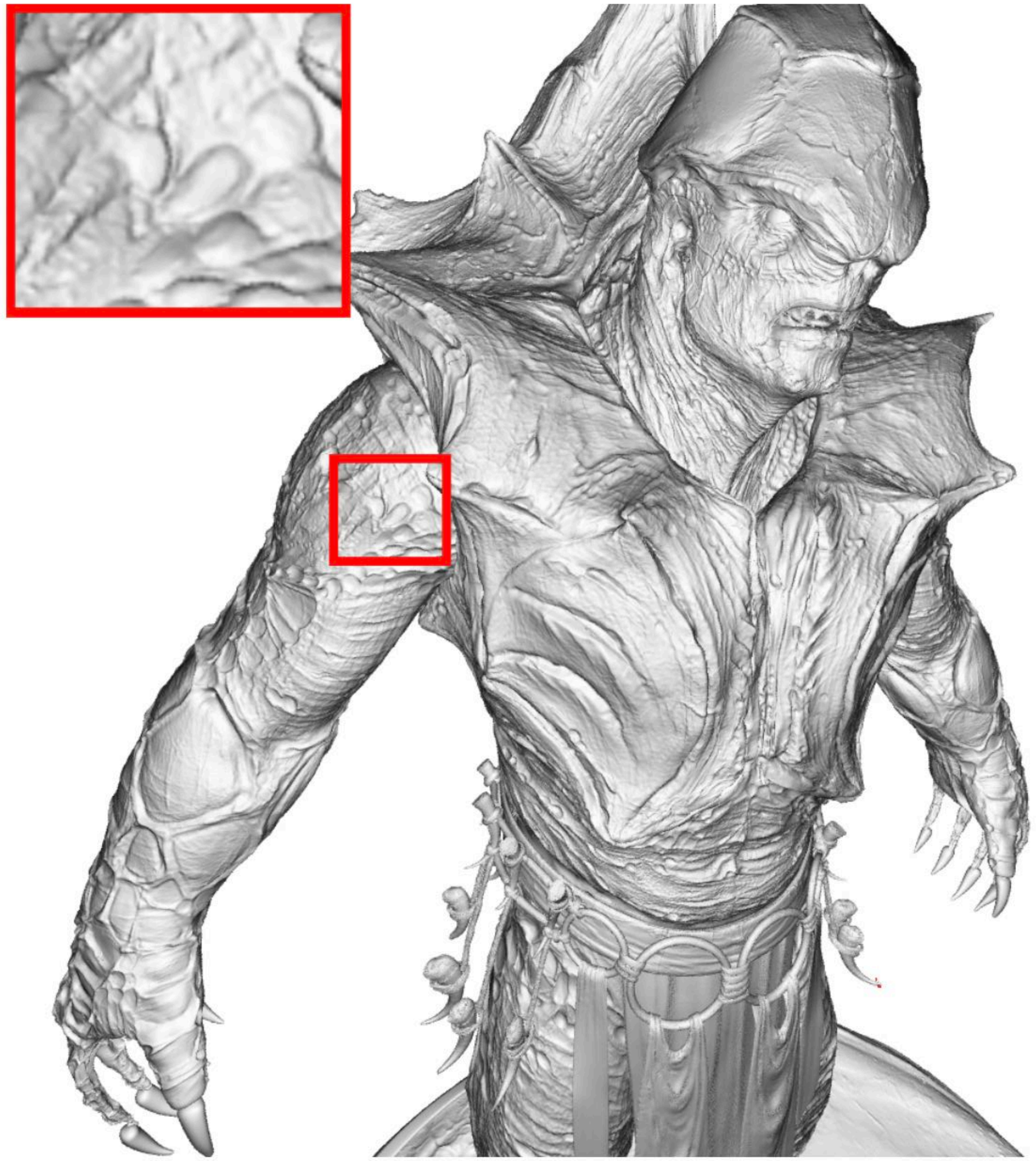
Displaced surface

[Lee 2000]

Result: high-resolution surface detail

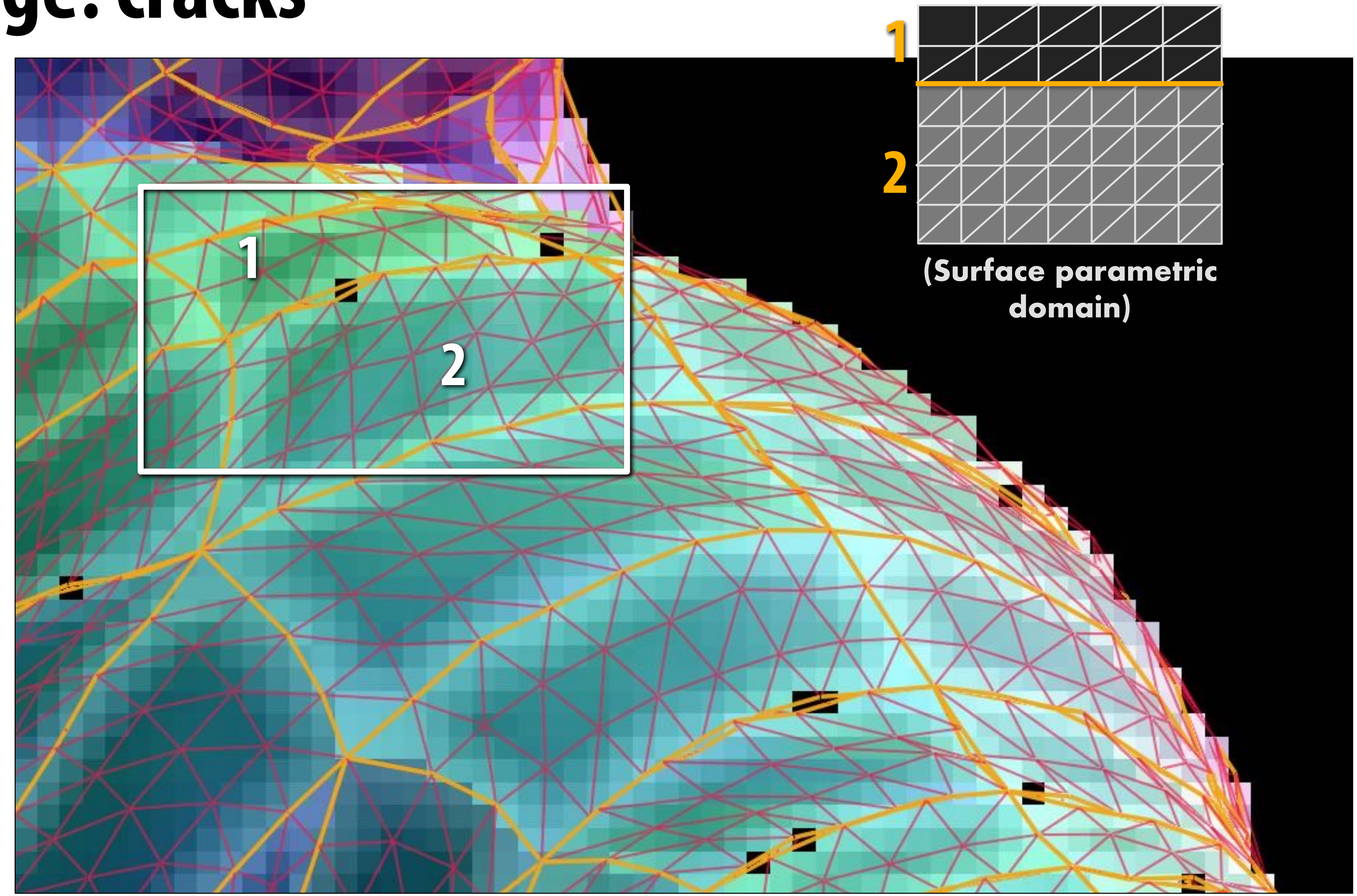


Result: high-resolution surface detail



6M triangles after tessellation and displacement

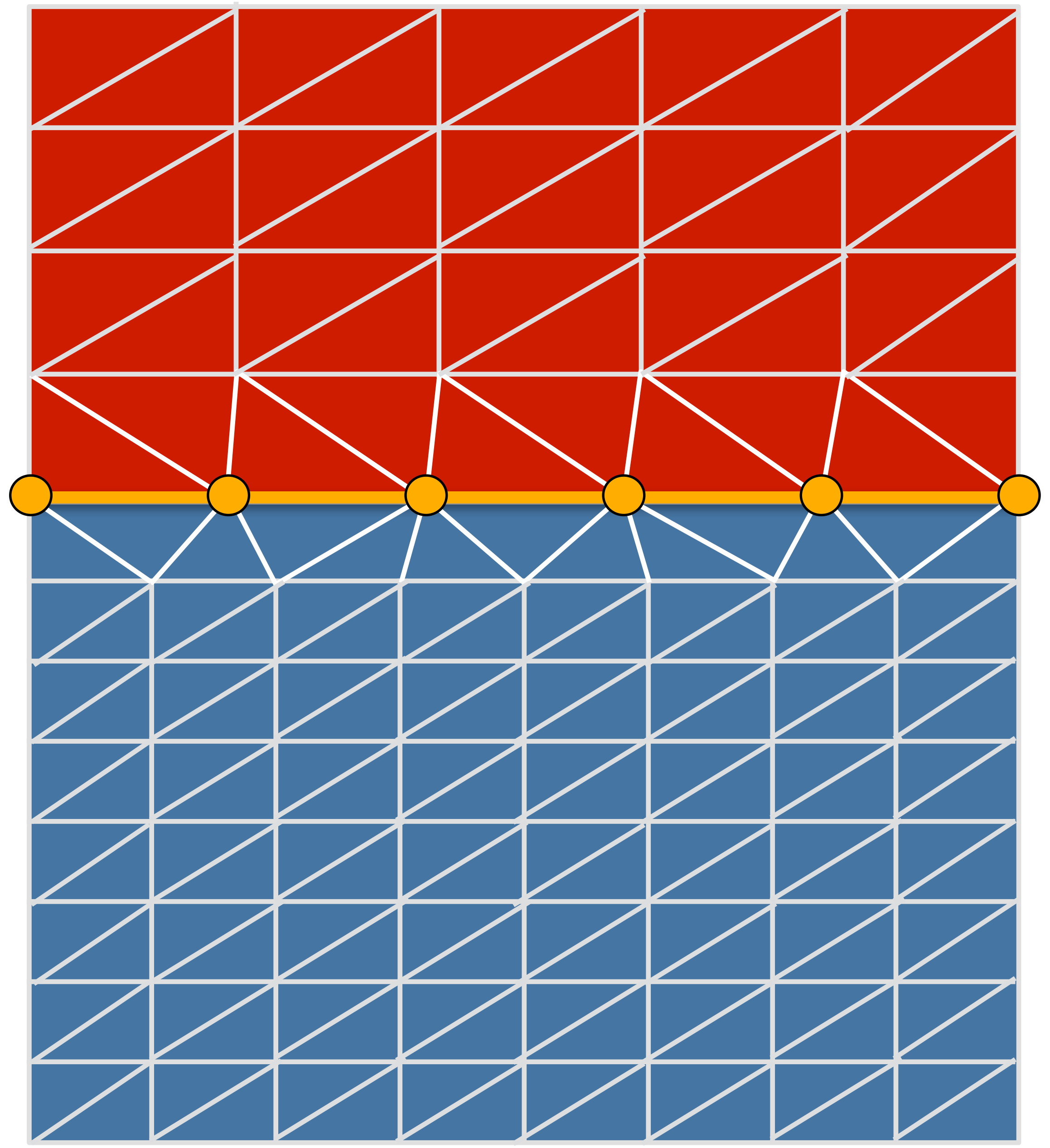
Challenge: cracks



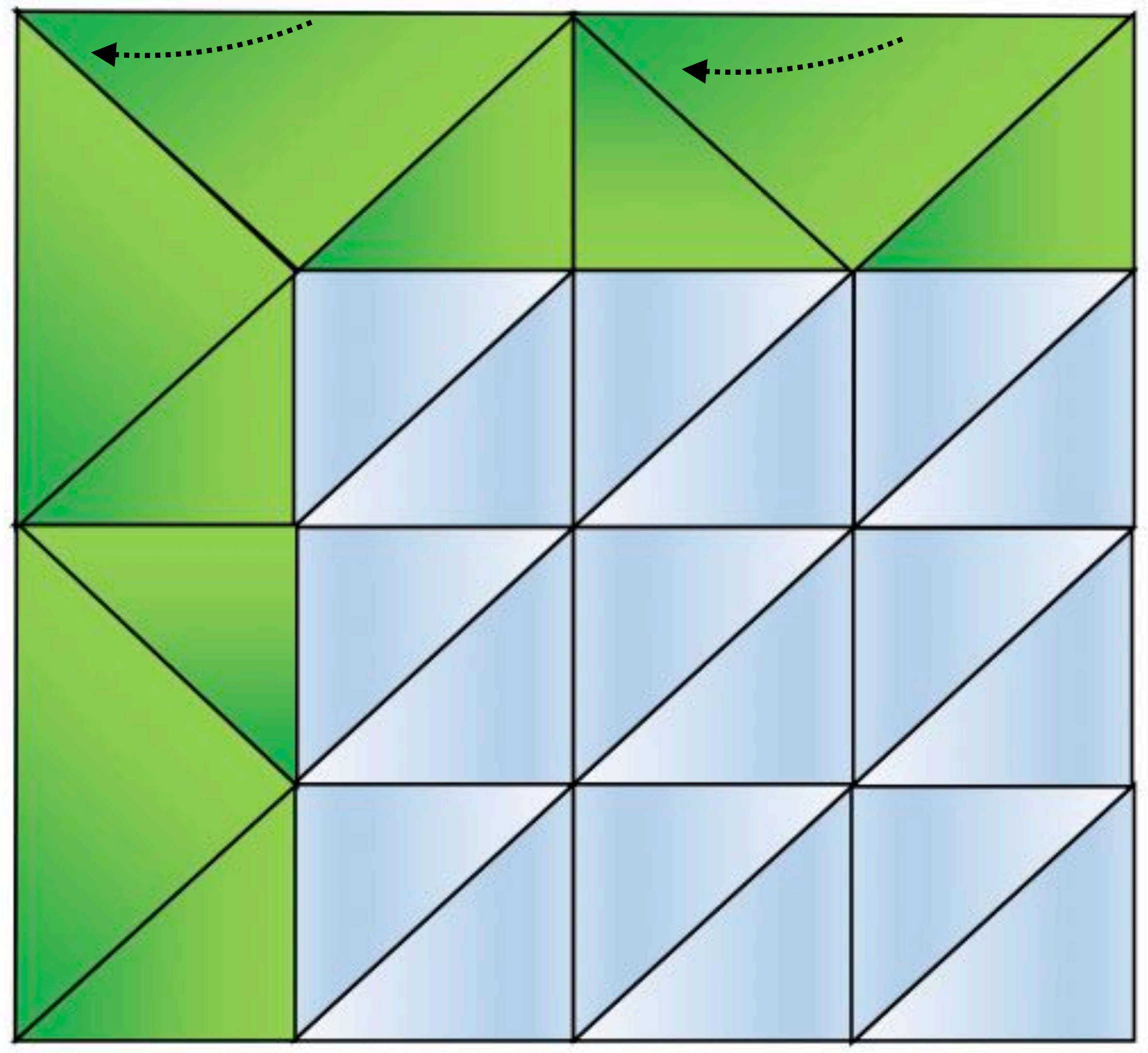
Crack fixing solutions

Key idea: Adjacent regions agree on tessellation along edge

Complicates parallel processing!



Generate irregular topology



5x5 regular vertex grid matching constraints on top & left edge of 3 segments (Vertices moves to create degenerate triangles)

Challenges of high-resolution geometry

- Visibility: have to rasterize large amounts of geometry
  - For each triangle… rasterize, shade triangles
  - Need techniques for building an acceleration structure over scene primitives to quickly discard large numbers of off-screen or occluded primitives
  - Fixed-function rasterization hardware in modern GPUs tends to be optimize for triangles that are at least a few pixels in size.

- Level of detail: how to represent geometry at level of detail needed for current viewing conditions
  - Adaptive level-of-detail introduces challenges of cracks, "popping", etc.

Next time: Unreal Nanite renderer

Modern solutions to rendering high resolution geometry on modern GPUs