Lecture 16:

Optimizing Ray Tracing (Part II)

Visual Computing Systems

Stanford CS348K, Spring 2021

Technologies that are making real-time ray tracing possible

- **Better algorithms: fast parallel BVH construction and traversal algorithms (many SIGGRAPH/HPG papers circa 2010-2017)**

- **GPU hardware evaluation:**
  - **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 number of rays per pixel**
  - **Makes use of existing DNN hardware acceleration**

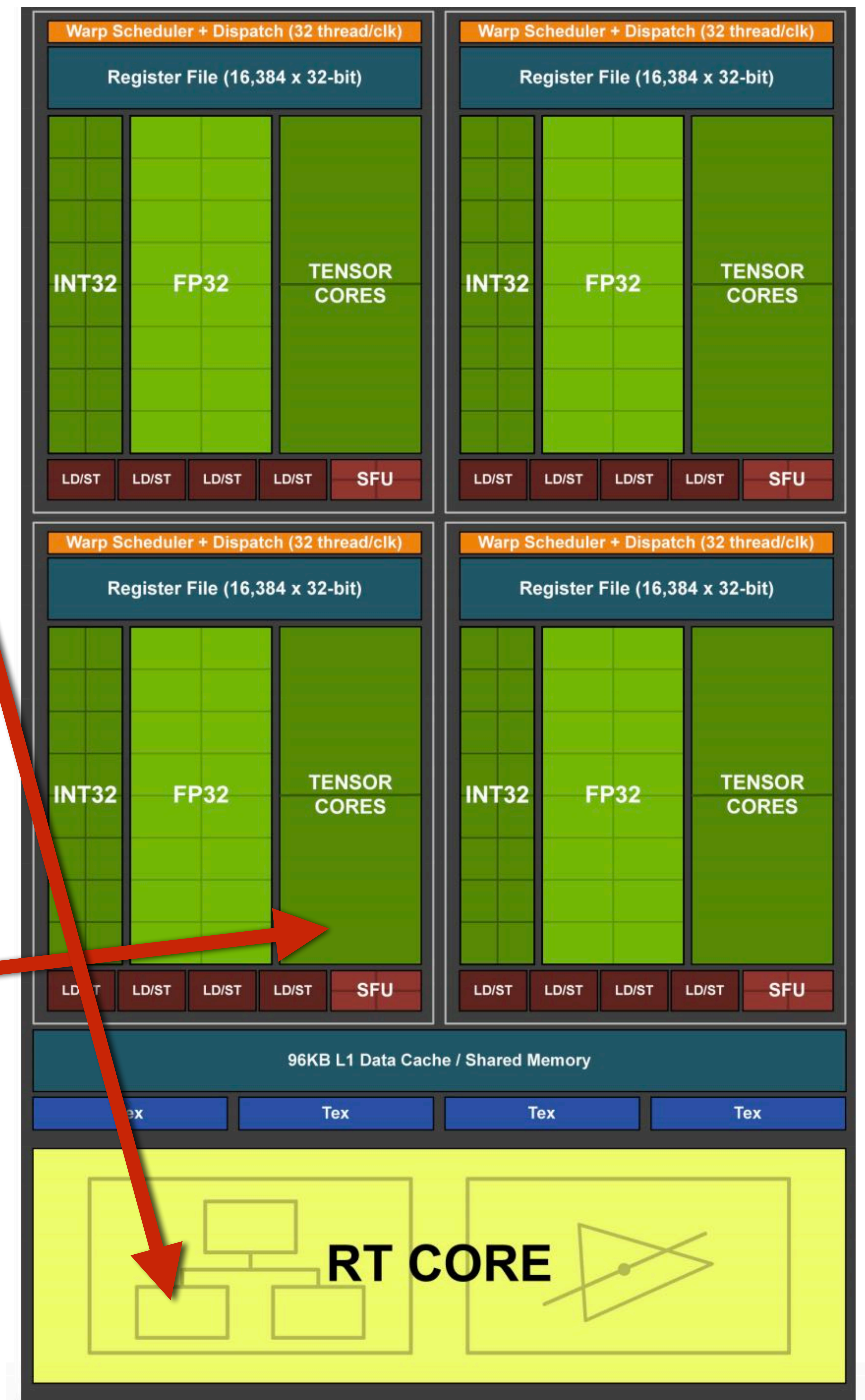
Convenient synergies in HW

- New GPU hardware for ray-tracing operations

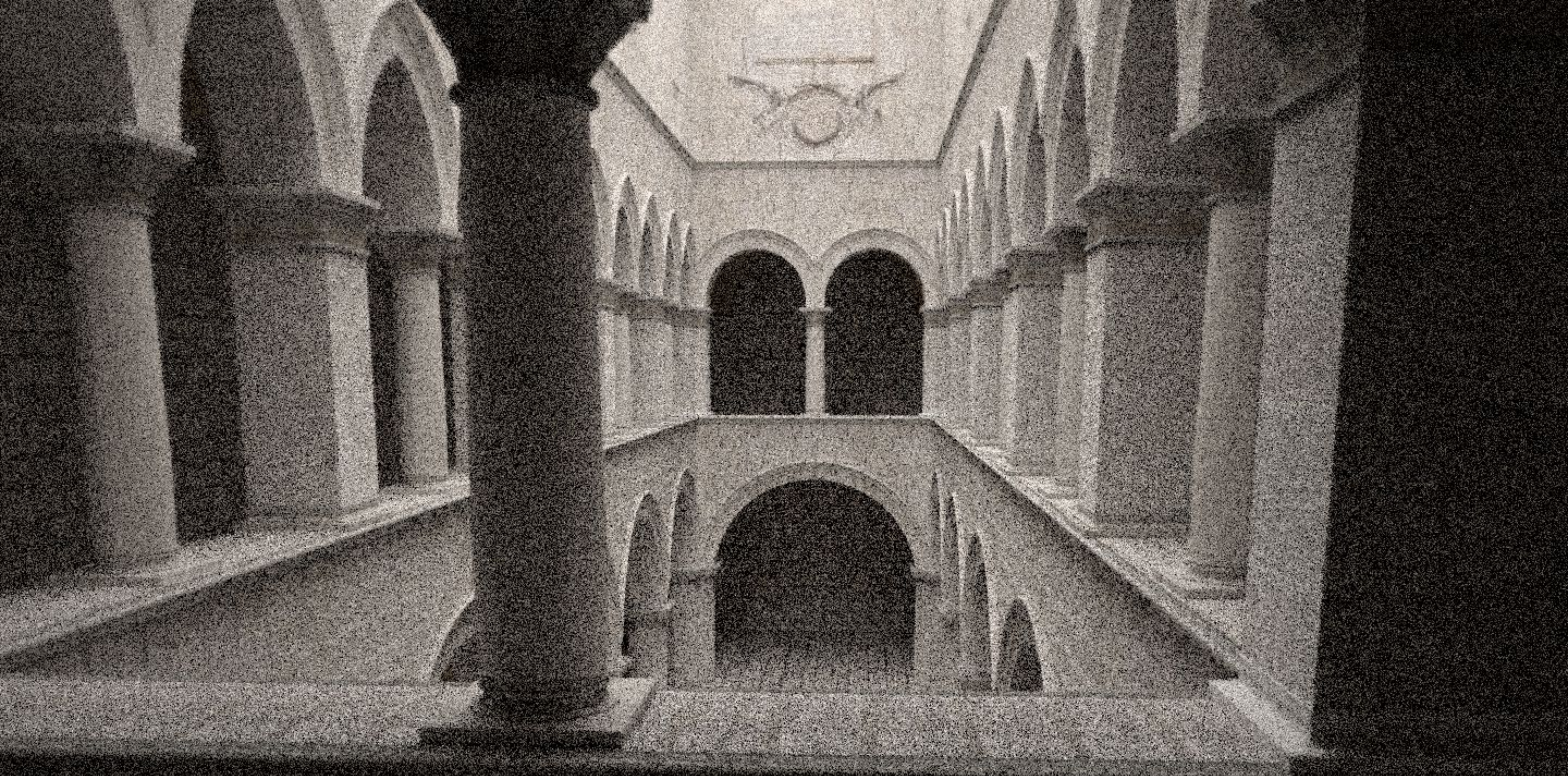
- But ray tracing still too expensive for noise-free images in real-time

- Tensor core: specialized hardware for accelerated DNN computations

  (that can be used to perform sophisticated denoising of ray traced images)



Denoising ray traced images



32 samples per pixel (visible noise)

Deep learning-based denoising

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

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

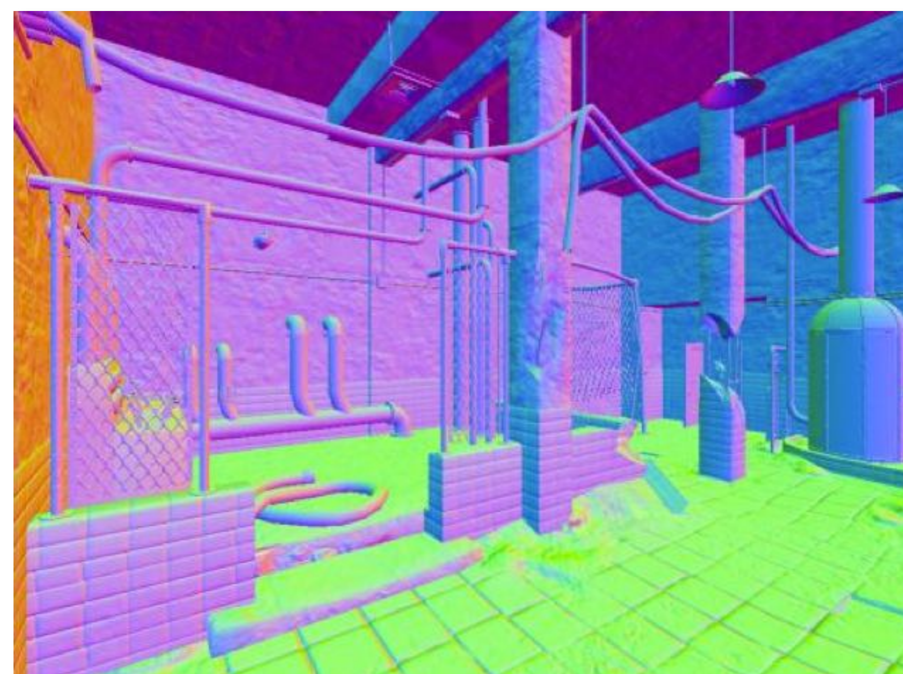
Example: neural denoiser RNN

[Chaitanya 17]

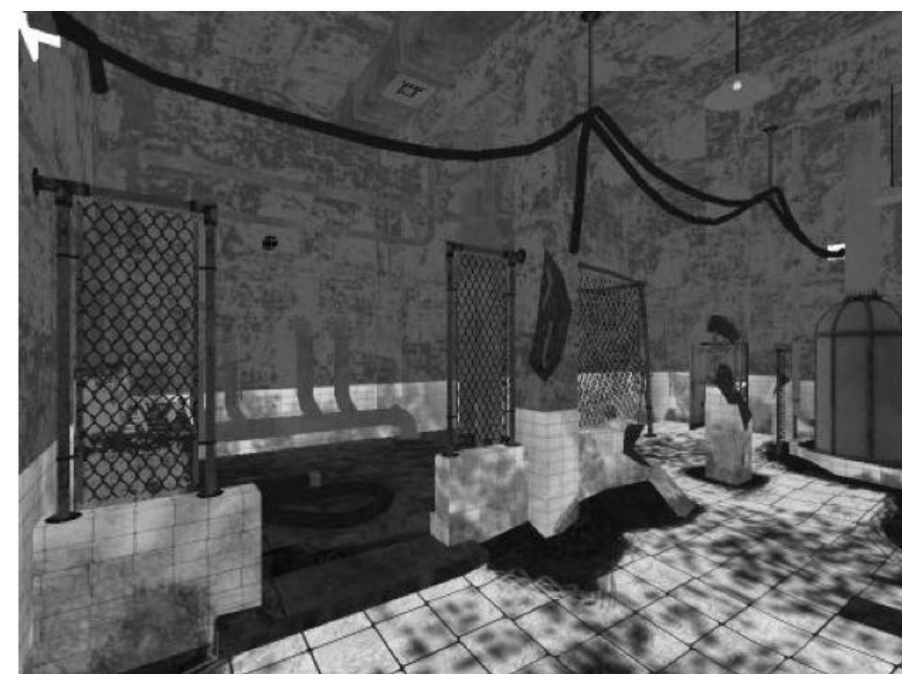
Input to network is noisy RGB image * + additional normal, depth, and roughness channels
(These cheap to compute inputs help network identify silhouettes)

Depth

Normal



Roughness

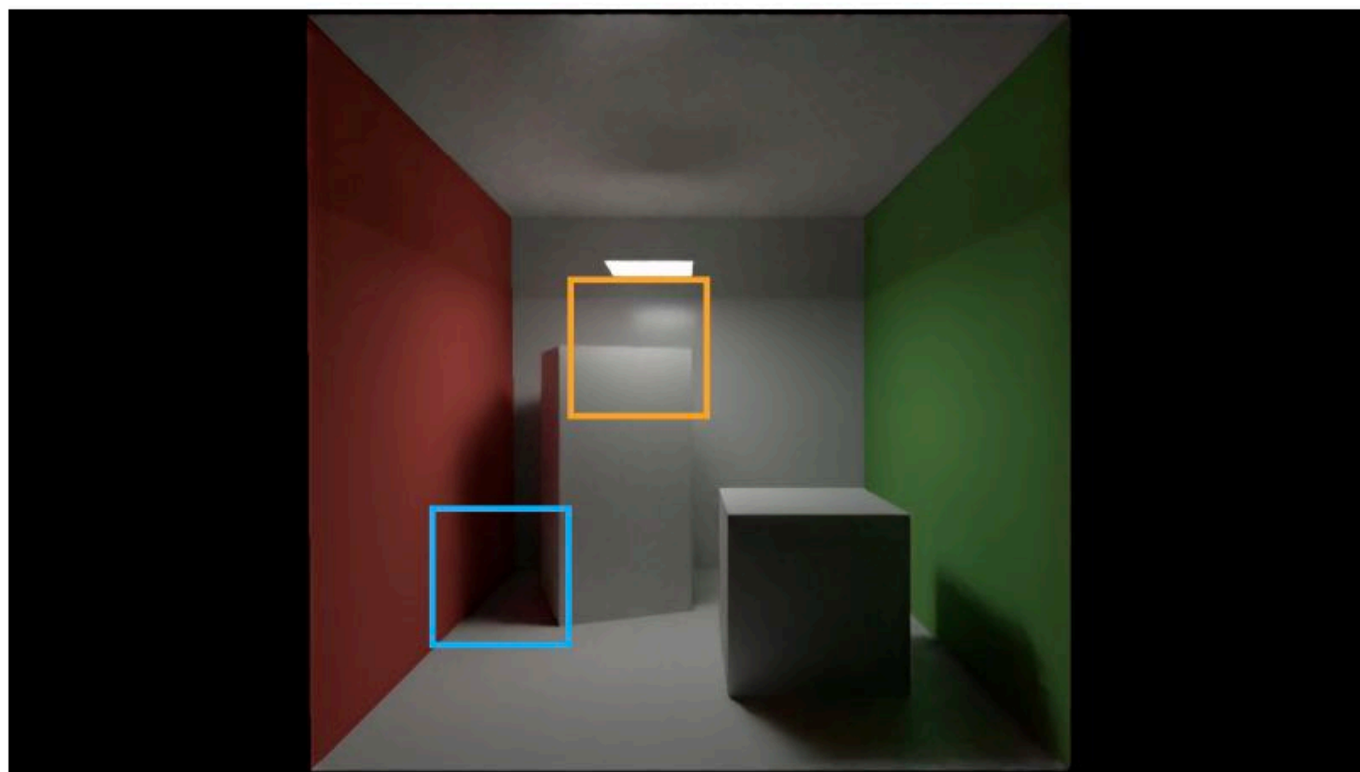
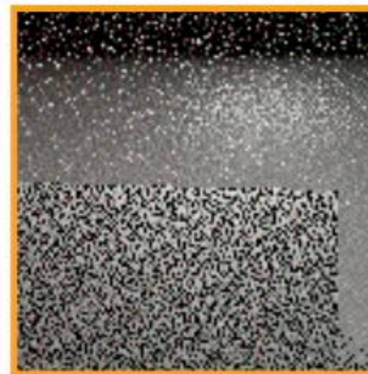
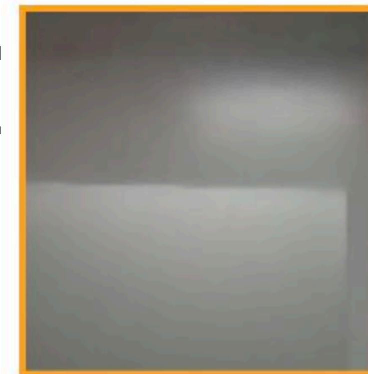
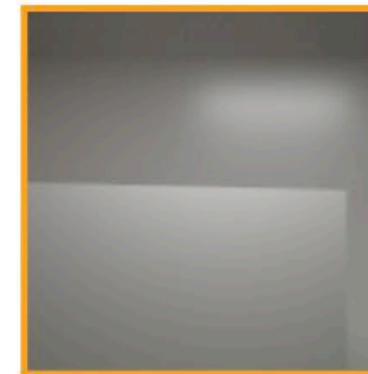
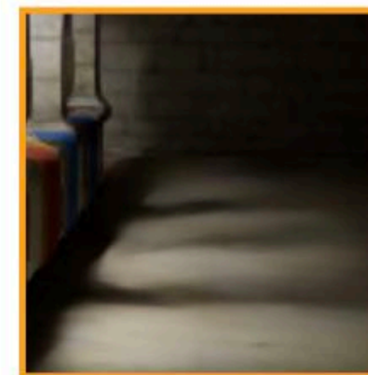
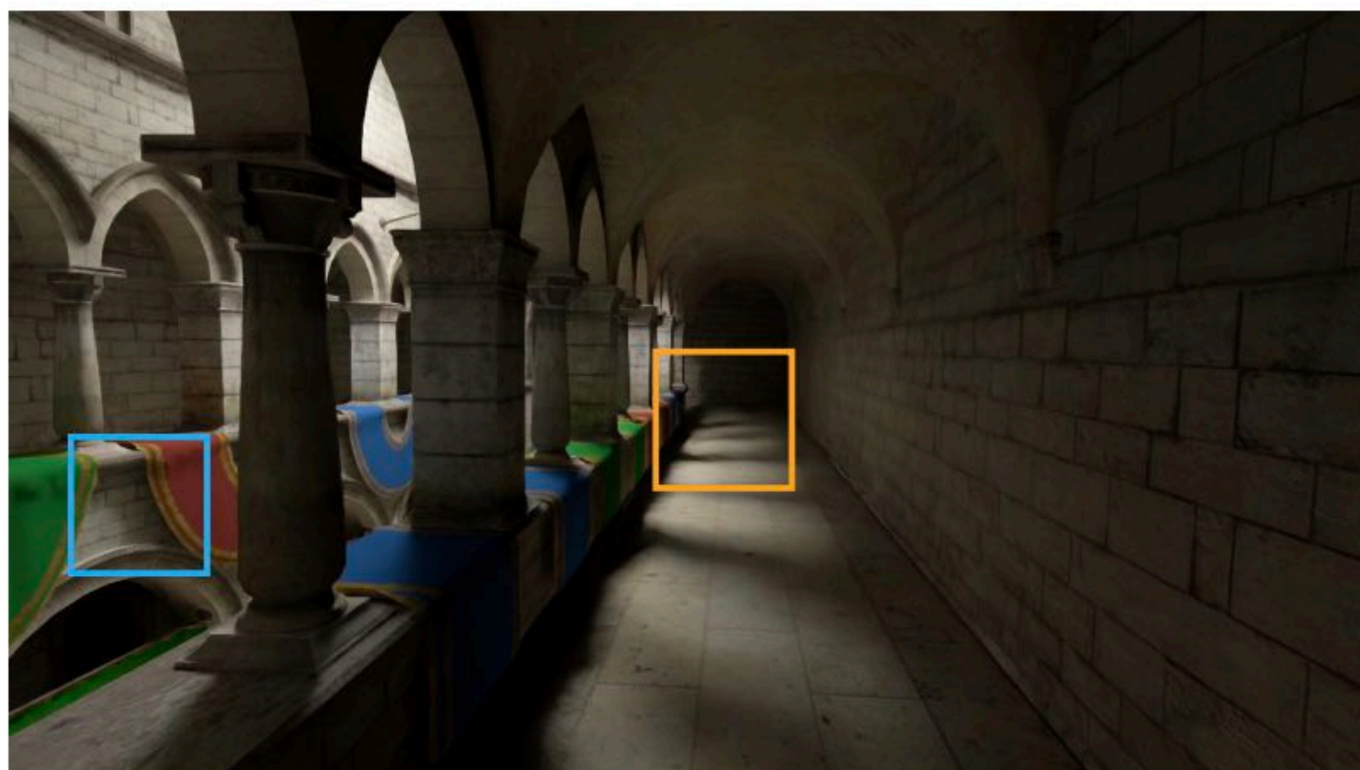
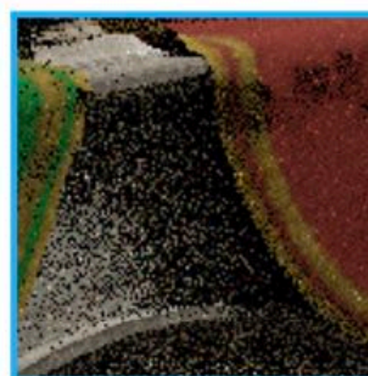
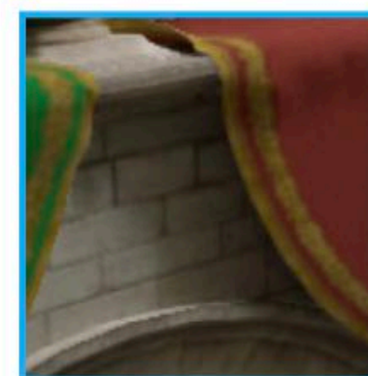
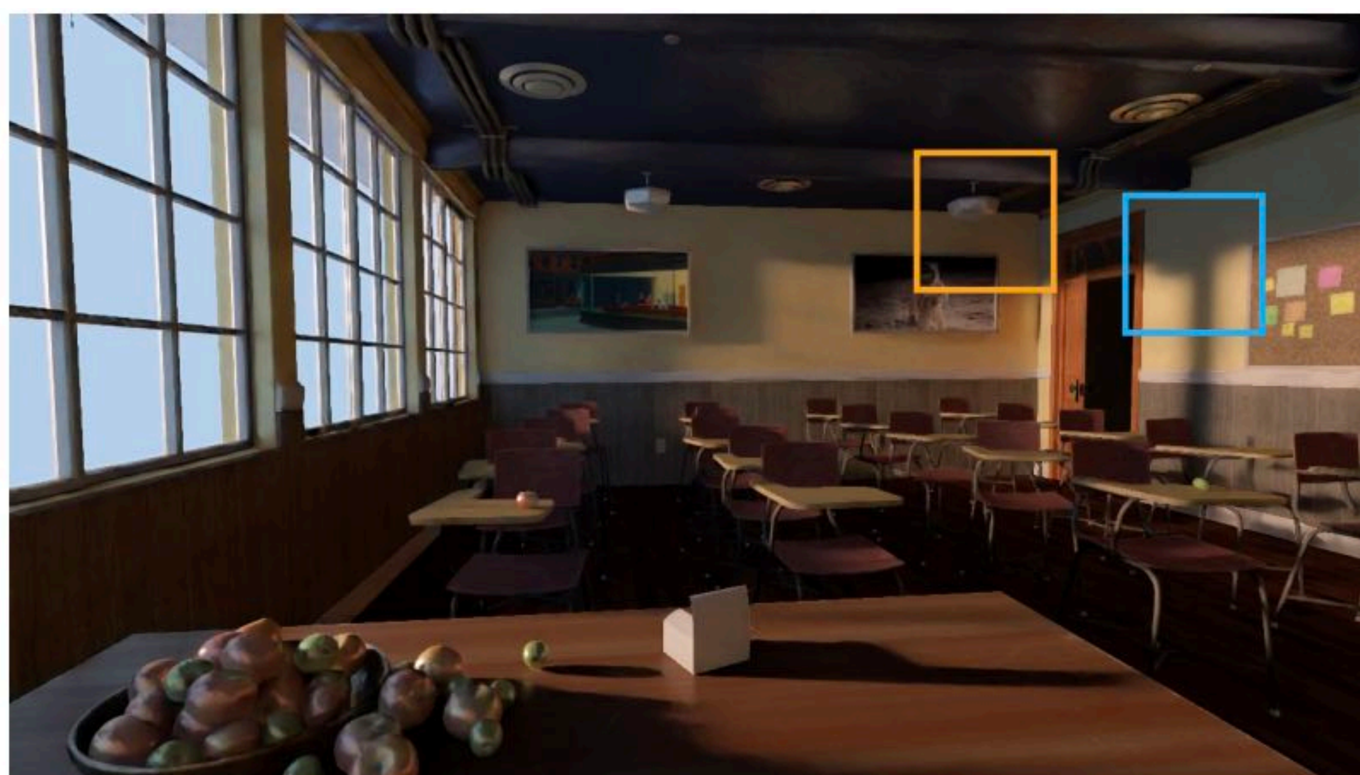
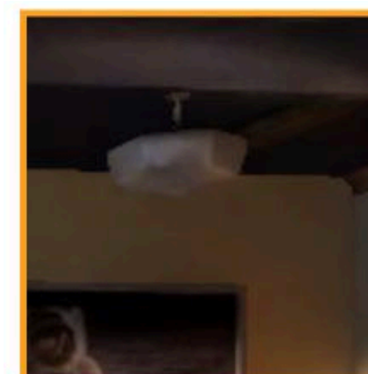
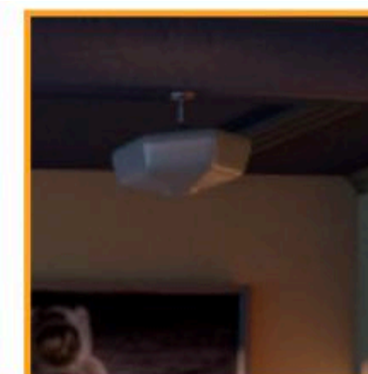
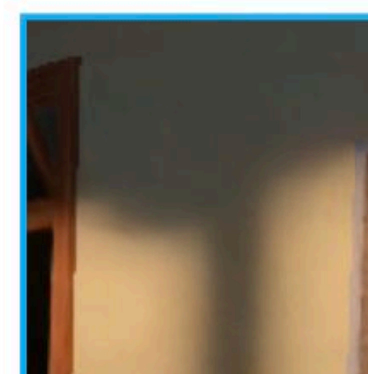
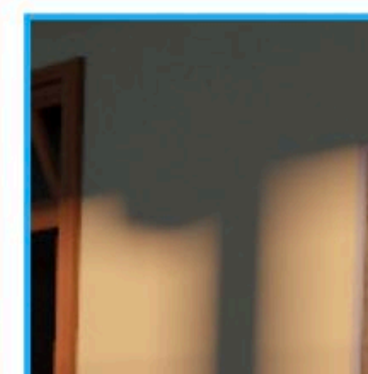


Albedo

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

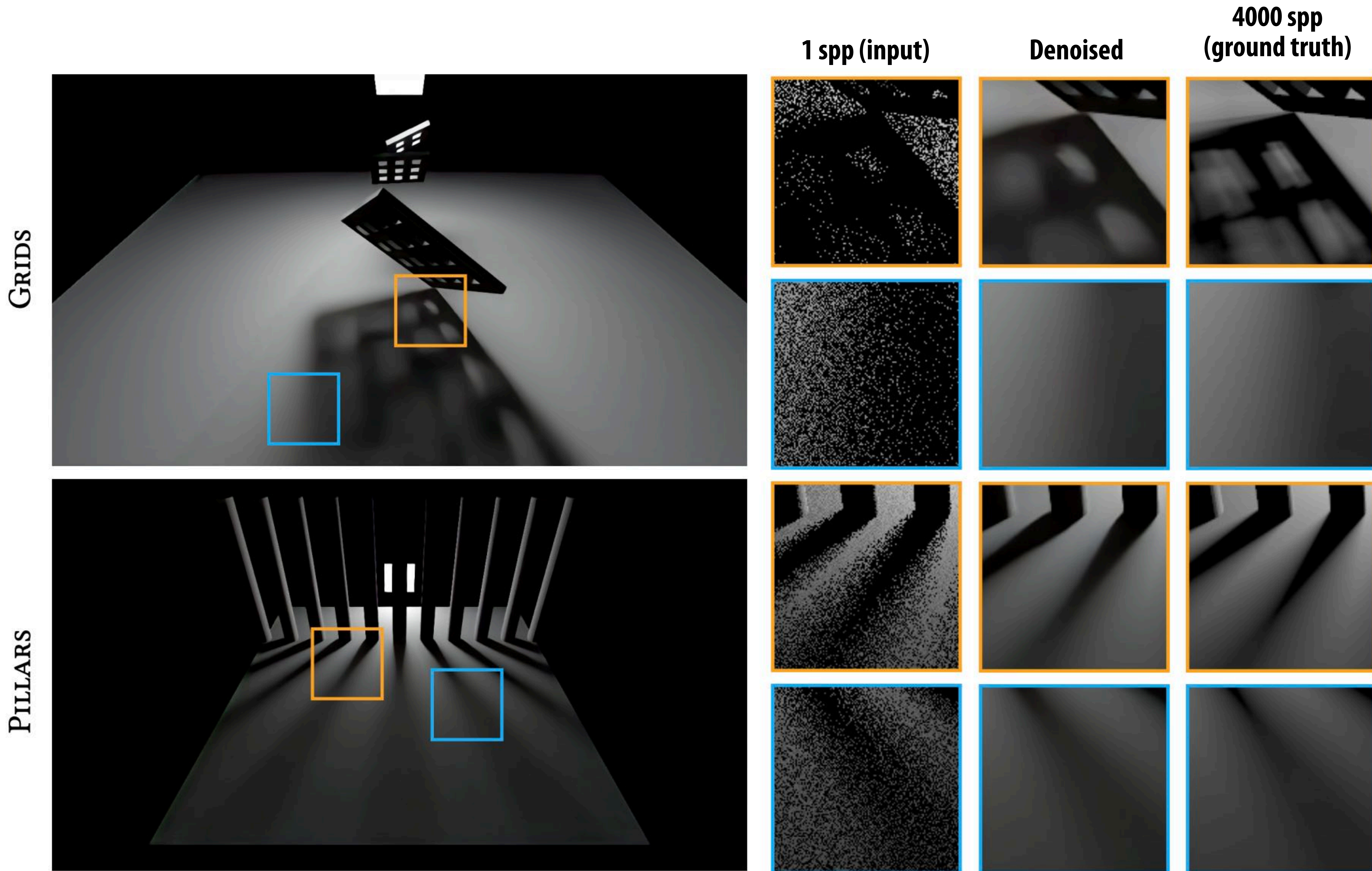
Denoising results

[Chaitanya 17]

	1 spp (input)	Denoised	4000 spp (ground truth)
CornellBox 			
			
Sponza 			
Classroom 			
			

Denoising results (challenging)

[Chaitanya 17]



More denoising examples

Original (noisy)

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

More denoising examples

Denoised

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

More denoising examples

Original (noisy)

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

More denoising examples

Denoised

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.)

Neural upsampling (hallucinating detail)

[Xiao 20]

LOW-RESOLUTION INPUT

+ auxiliary inputs

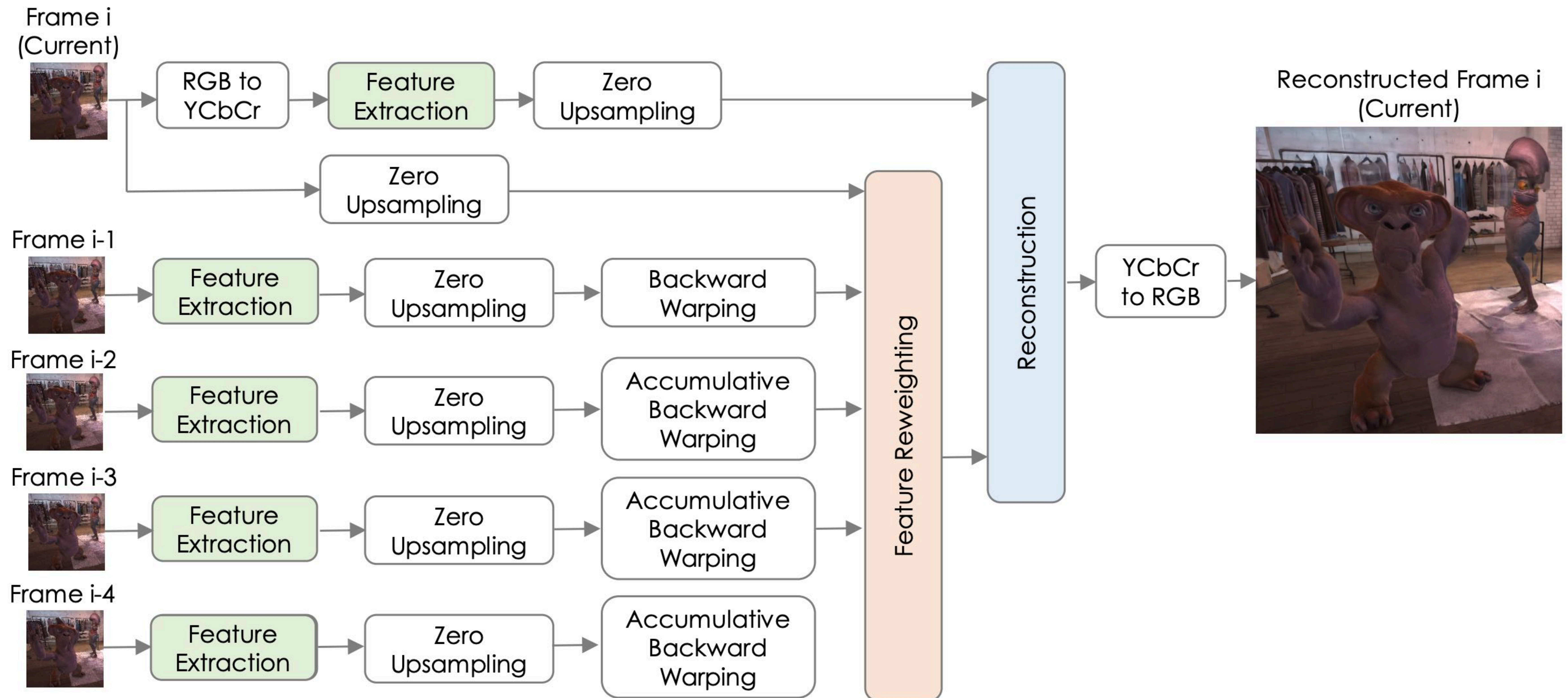
Neural upsampling (hallucinating detail)

[Xiao 20]

4x4 upsampled result (16x more pixels)

Neural upsampling pipeline

[Xiao 20]



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

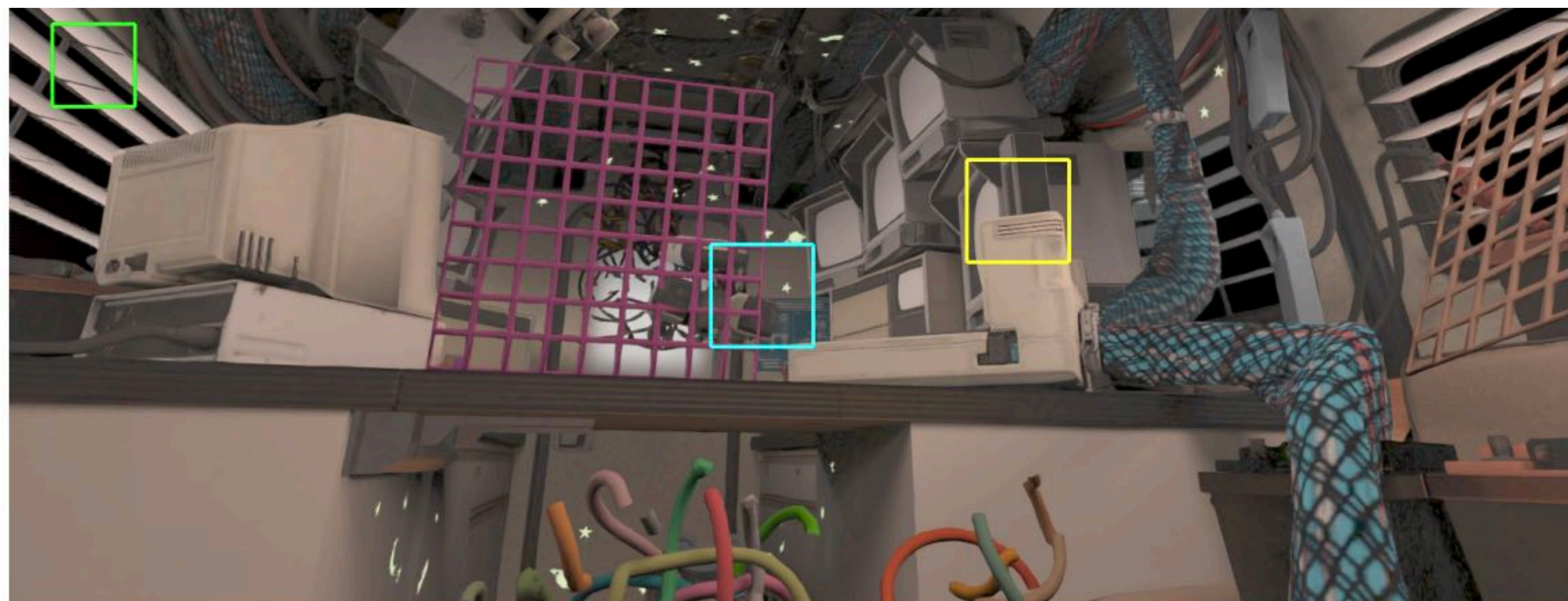
Alignment vectors provided by renderer

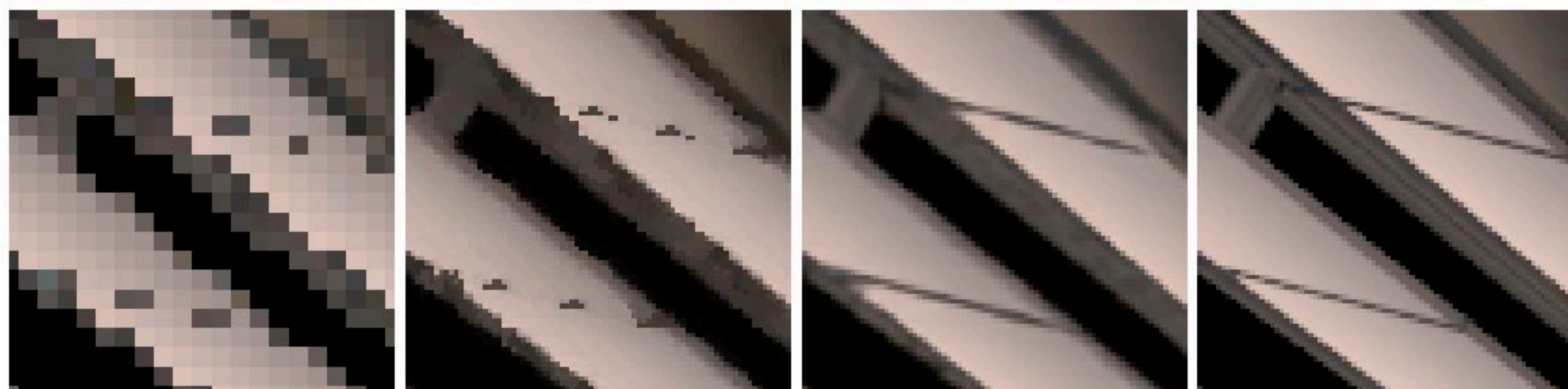
Learn model that determines weights for aligned features ("feature reweighting")

Then decode with neural decoder ("reconstruction")

Closer look

[Xiao 20]

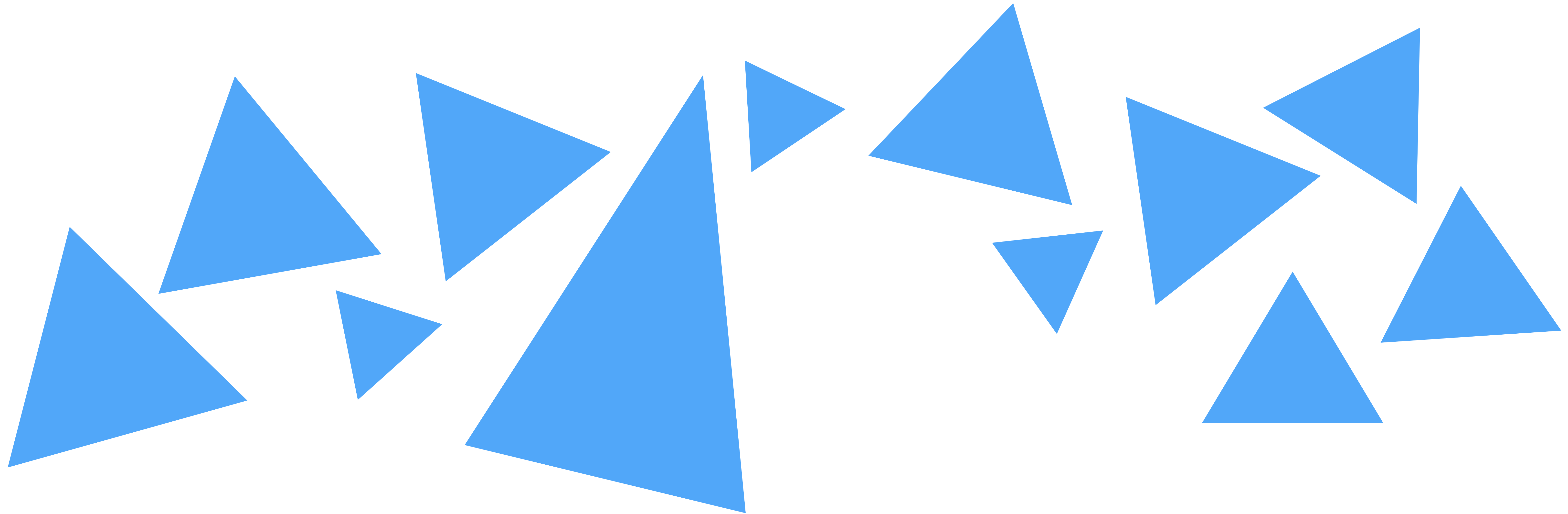




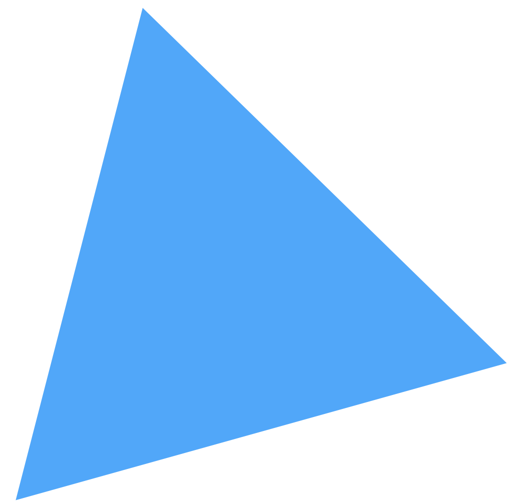
Input Unreal TAAU Ours Reference

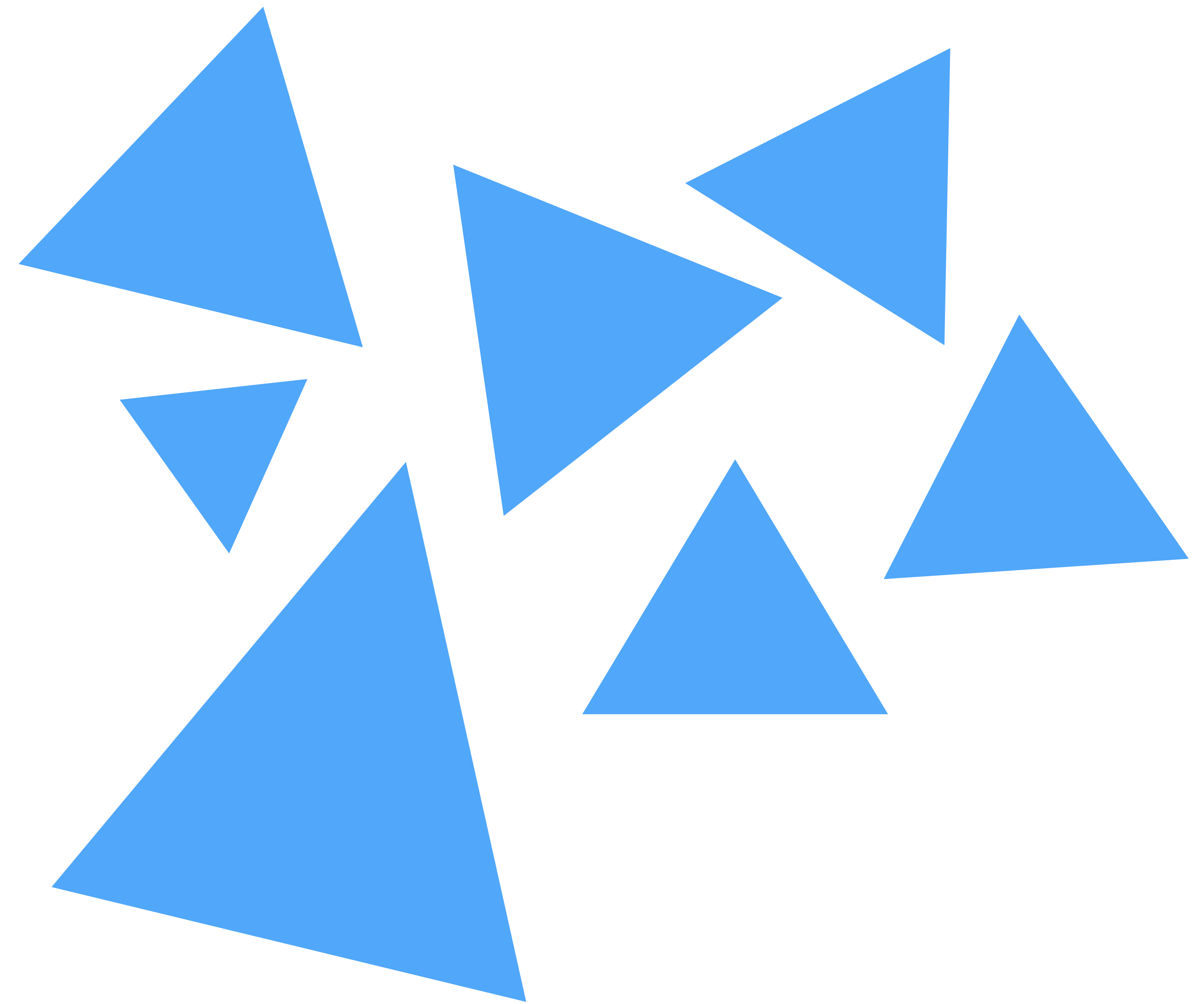
Rapidly Constructing BVH Acceleration Structures

BVH construction review:
How would you partition these triangles into two groups?

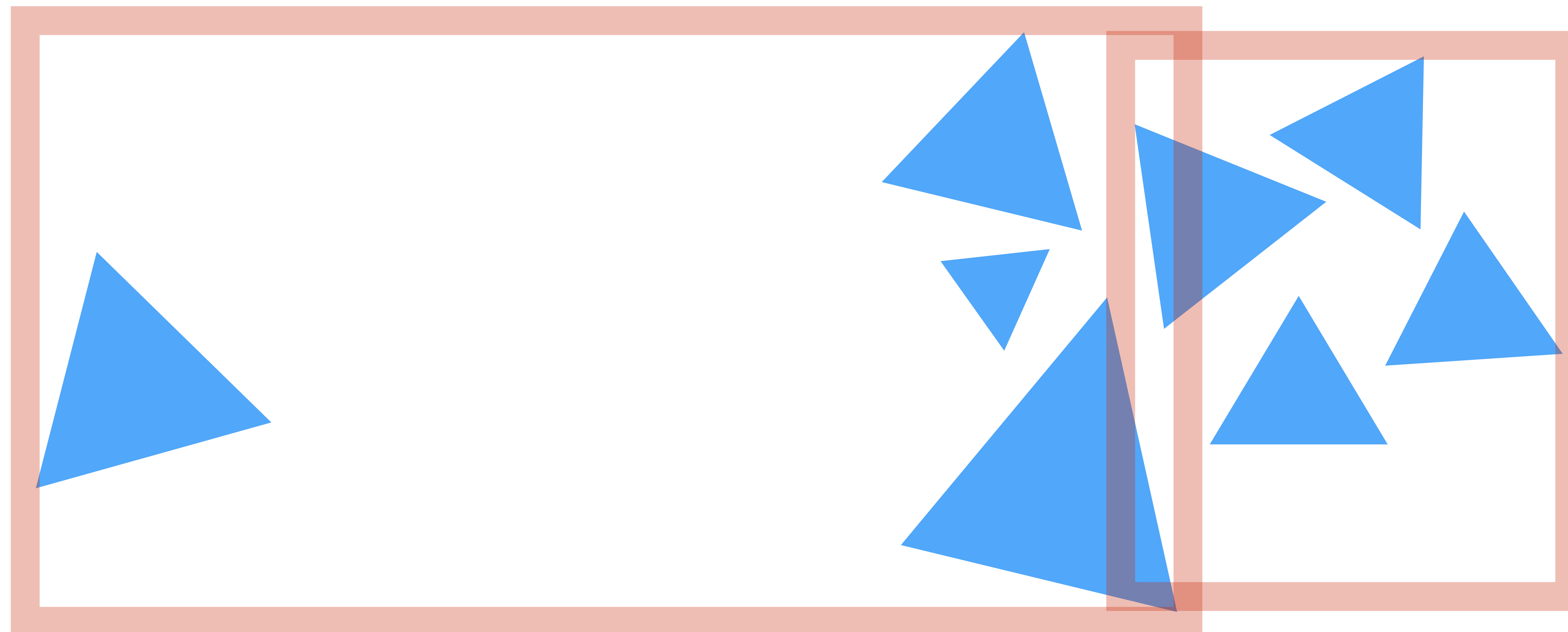


What about these?

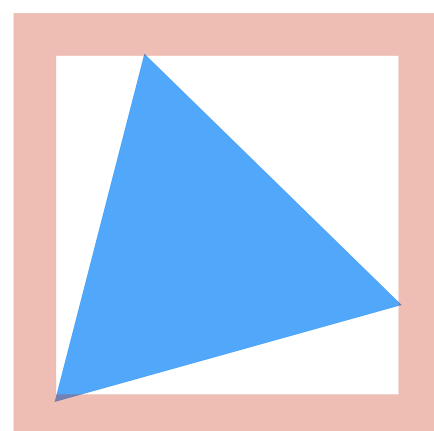


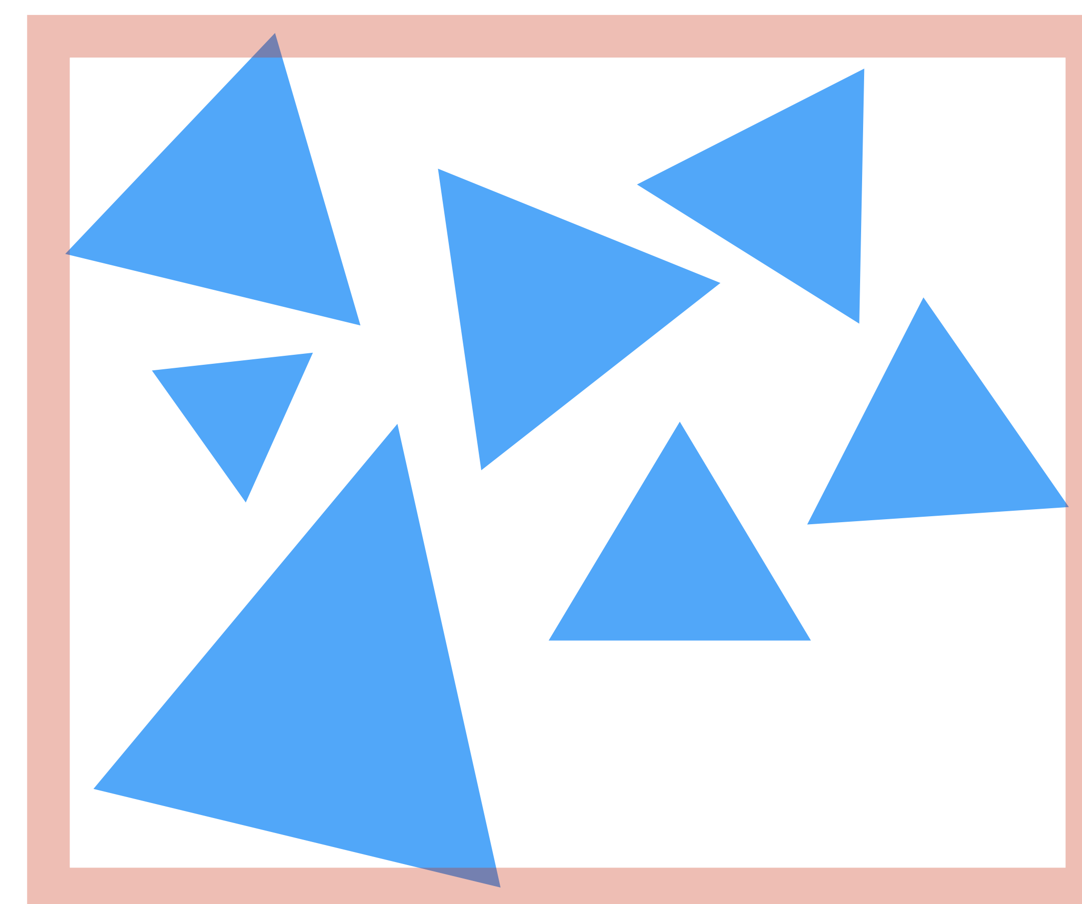


Intuition about a “good” partition?



Partition into child nodes with equal numbers of primitives





Better partition

Intuition: want small bounding boxes (minimize overlap between children, avoid bboxes with significant empty space)

What are we really trying to do?

A good partitioning minimizes the expected <u>cost</u> of finding the closest intersection of a ray with the scene primitives in the node.

If a node is a leaf node (no partitioning):

$$C = \sum_{i=1}^{N} C_{\text{isect}}(i)$$

Where $C_{\text{isect}}(i)$ is the cost of ray-primitive intersection for primitive *i* in the node.

$$= NC_{\text{isect}}$$

(Common to assume all primitives have the same cost)

Cost of making a partition

A good partitioning minimizes the expected cost of finding the closest intersection of a ray with primitives in the node.

$$C = C_{\text{trav}} + p_A C_A + p_B C_B$$

C_{trav} **is the cost of traversing an interior node (e.g., load data + bbox intersection check)**

C_A **and** C_B **are the costs of intersection with the resultant child subtrees**

p_A **and** p_B **are the probability a ray intersects the bbox of the child nodes A and B**

Primitive count is common approximation for child node costs:

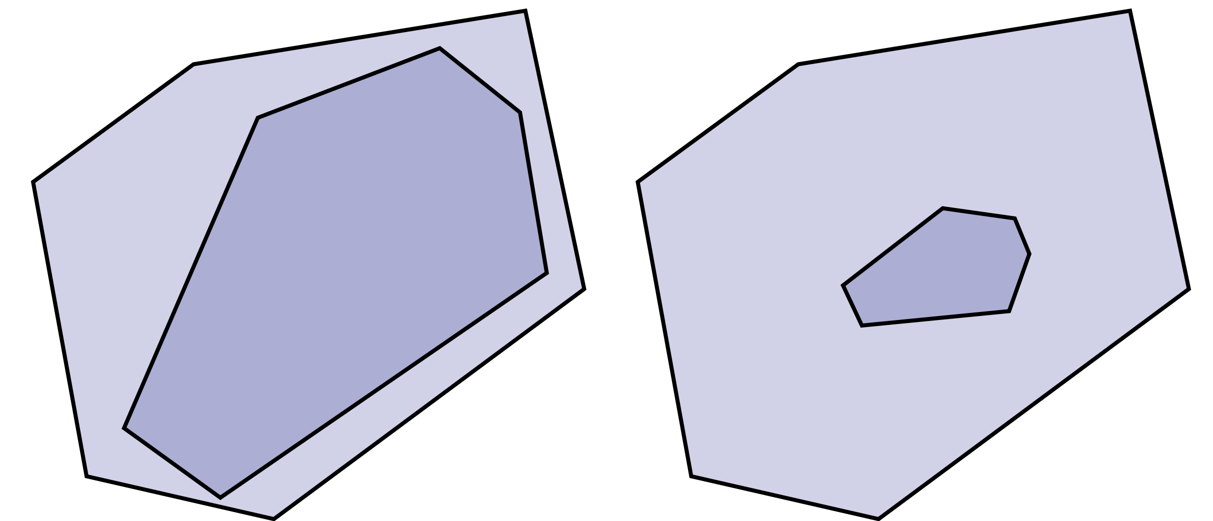
$$C = C_{\text{trav}} + p_A N_A C_{\text{isect}} + p_B N_B C_{\text{isect}}$$

Remaining question: how do we get the probabilities p_A, p_B?

Estimating probabilities

- **For convex object A inside convex object B, the probability that a random ray that hits B also hits A is given by the ratio of the surface areas S_A and S_B of these objects.**

$$P(\mathrm{hit}A|\mathrm{hit}B) = \frac{S_A}{S_B}$$



Leads to surface area heuristic (SAH):

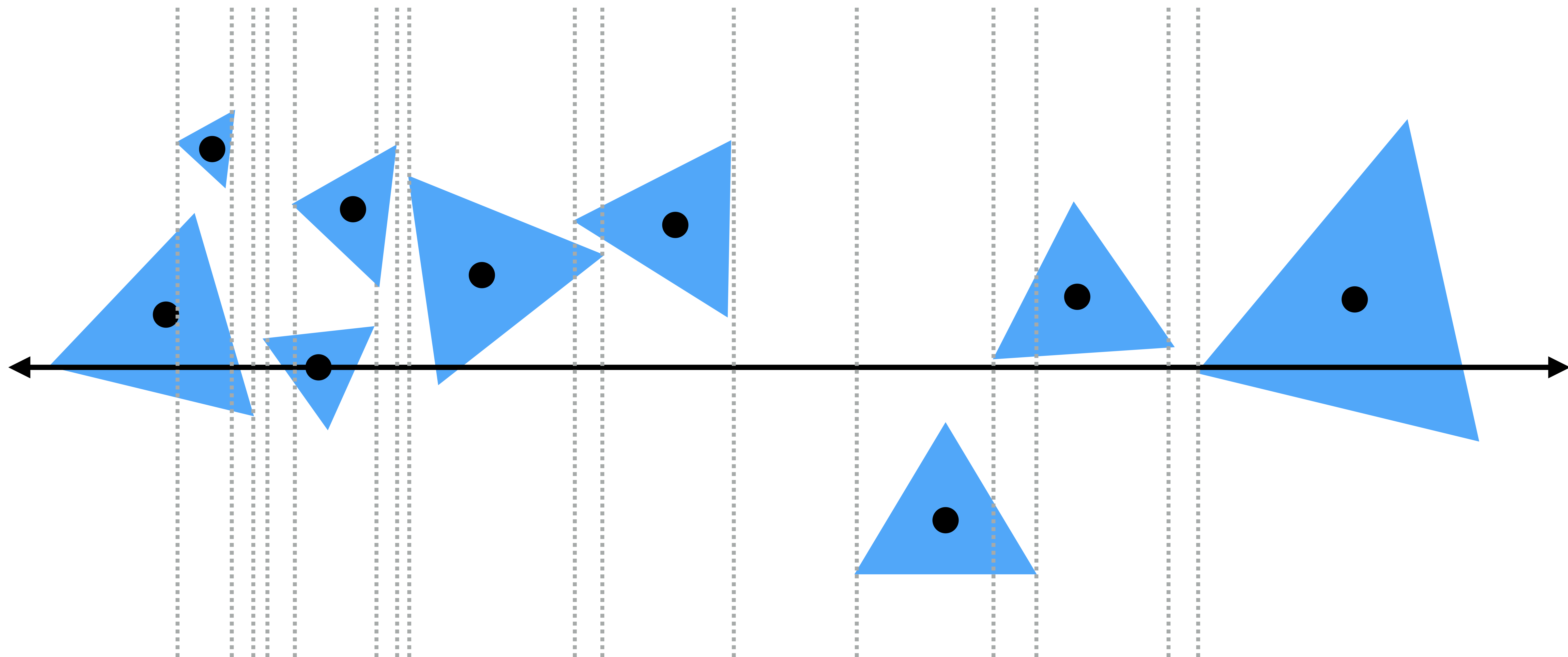
$$C = C_{\mathrm{trav}} + \frac{S_A}{S_N} N_A C_{\mathrm{isect}} + \frac{S_B}{S_N} N_B C_{\mathrm{isect}}$$

Assumptions of the SAH (*which may not hold in practice!*):

- **Rays are randomly distributed**
- **Rays are not occluded**

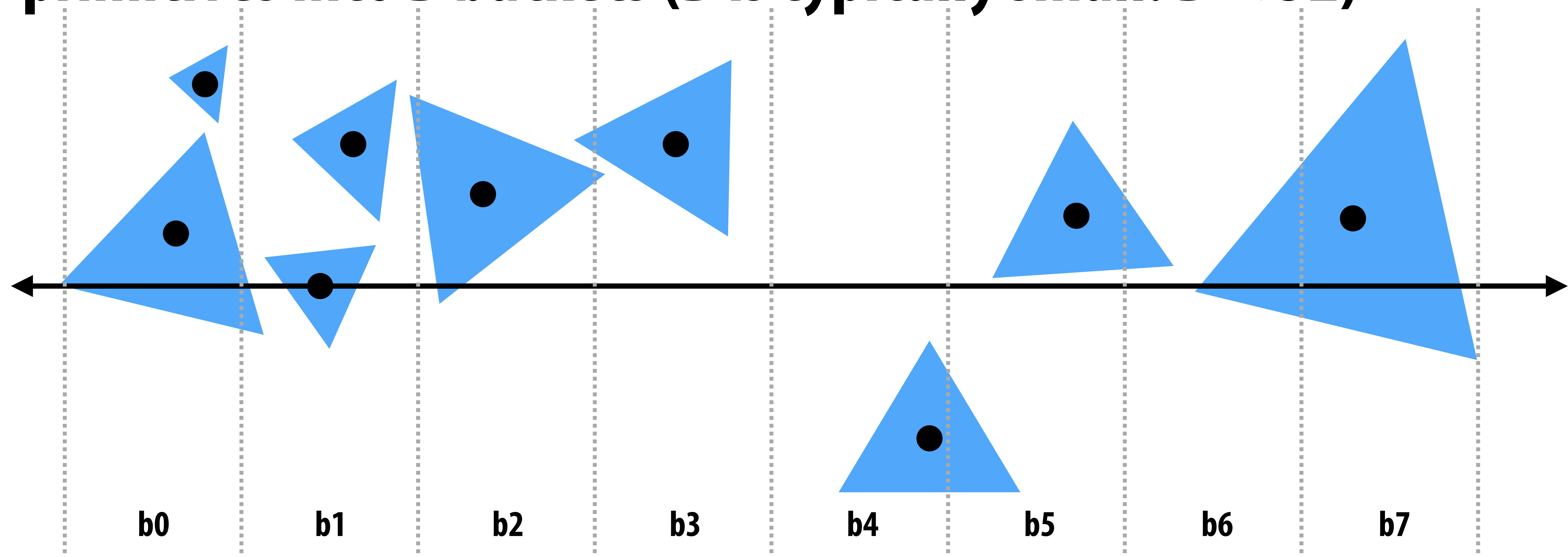
Implementing partitions

- Constrain search for good partitions to axis-aligned spatial partitions
  - Choose an axis; choose a split plane on that axis
  - Partition primitives by the side of splitting plane their centroid lies
  - SAH changes only when split plane moves past triangle boundary
  - Have to consider large number of possible split planes… O(# objects)



Efficiently implementing partitioning

- **Efficient modern approximation: split spatial extent of primitives into B buckets (B is typically small: B < 32)**




```
For each axis: x,y,z:
   initialize bucket counts to 0, per-bucket bboxes to empty
   [POTENTIALLY IN PARALLEL]
   For each primitive p in node:
      b = compute_bucket(p.centroid)
      b.bbox.union(p.bbox);
      b.prim_count++;
   [POTENTIALLY IN PARALLEL]
   For each of the B-1 possible partitioning planes evaluate SAH
      Use lowest cost partition found (or make node a leaf)
```

Top-down BVH construction

```
Partition(list of prims) {

  if (list is small enough, or no cost benefit from SAH split) {
    // make leaf node
  }

  (prim_list_1, prim_list2) = // perform SAH split

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

Building a low-quality BVH quickly

[Lauterbach 09, Pantaleone 10]

1. Discretize each dimension of scene into 2^B cells
2. 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. Sort primitives by Morton code (primitives now ordered with high locality in 3D space: in a space-filling curve!)
   - O(N) parallel radix sort

2D Morton Order

B=1: 00, 01, 10, 11

B=2

B=3

B=4

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
```

Modern, fast BVH construction schemes

- Combine greedy "top-down" divide-and-conquer build with "bottom up" construction techniques

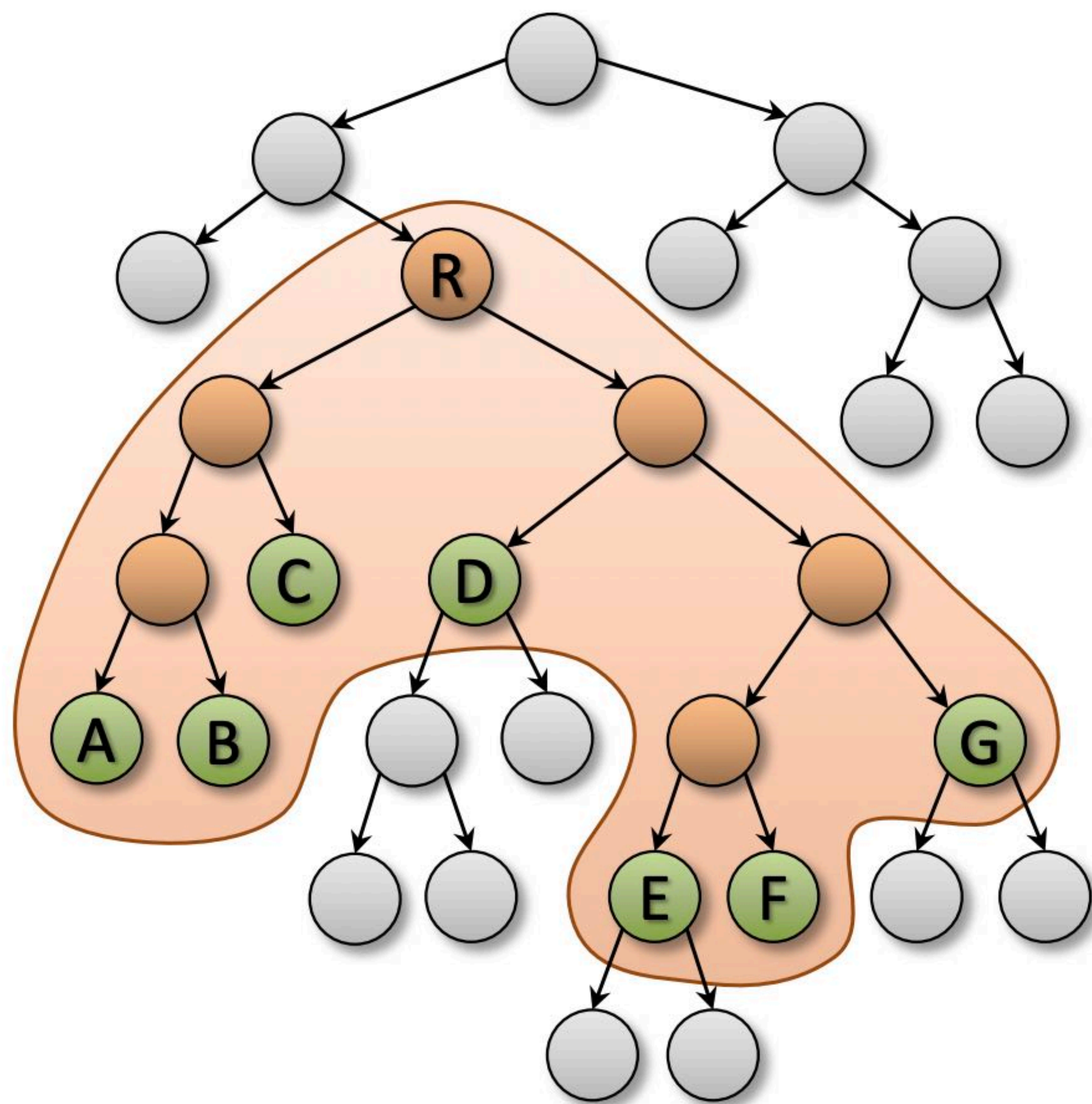
- Step 1: build low-quality BVH quickly (e.g, using Morton codes)

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

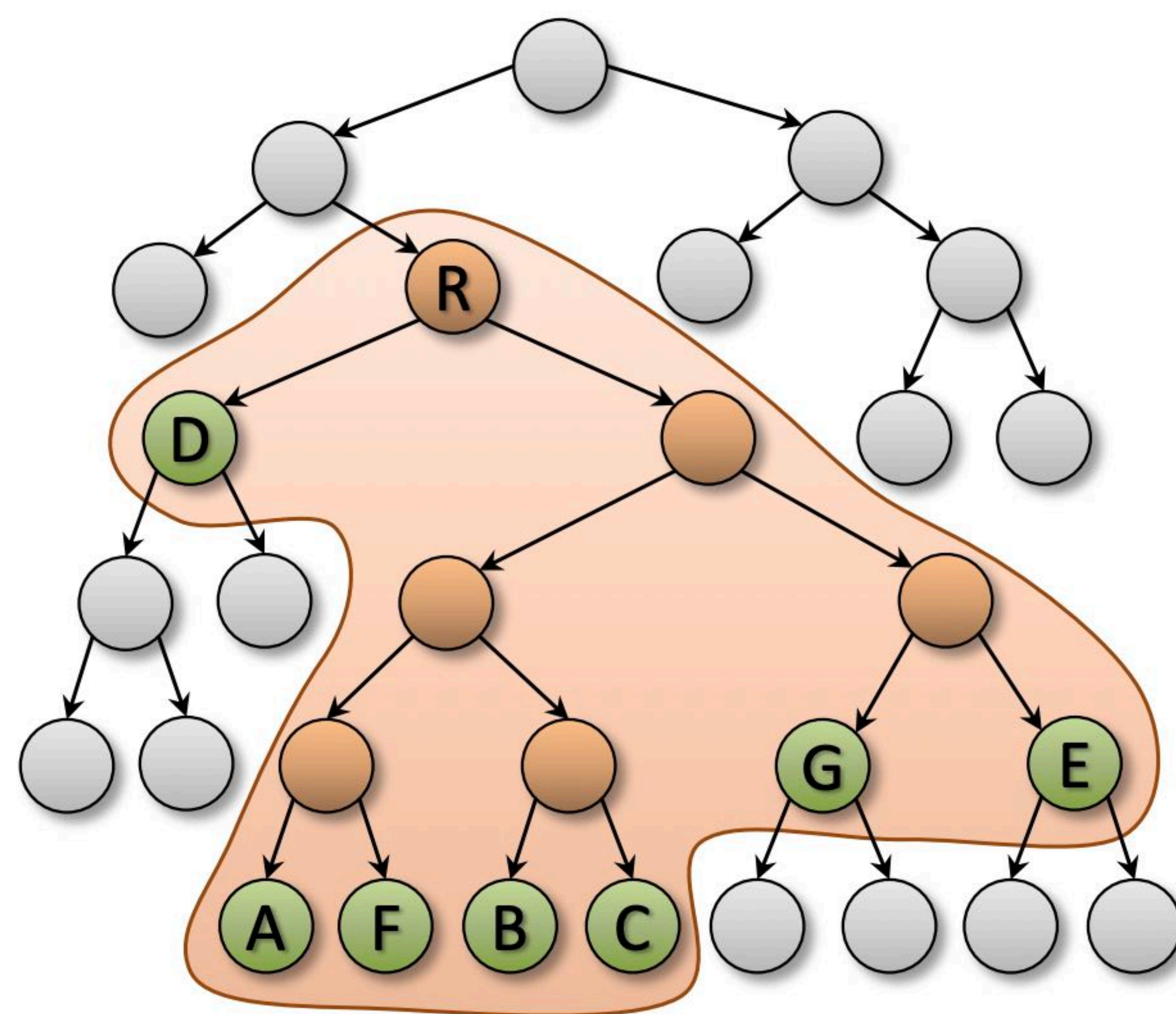
Kerras 2013 bottom up treelet-based construction

[Kerras 13]

- 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 (SAH) treelet
  - Brute force search implemented using dynamic programming method



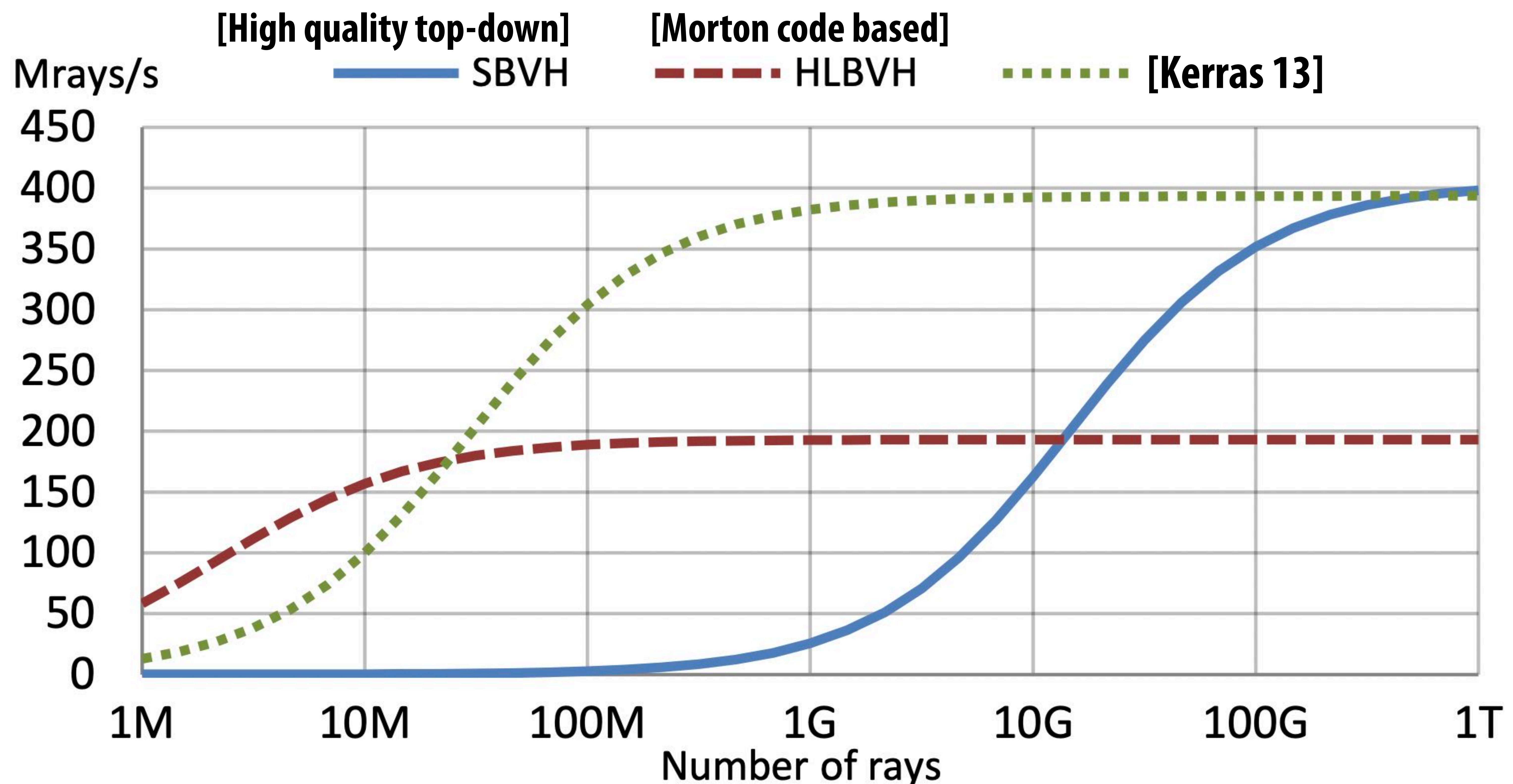
Shaded region: treelet with 7 leaf nodes



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

Can afford to build a better BVH if you are shooting many rays

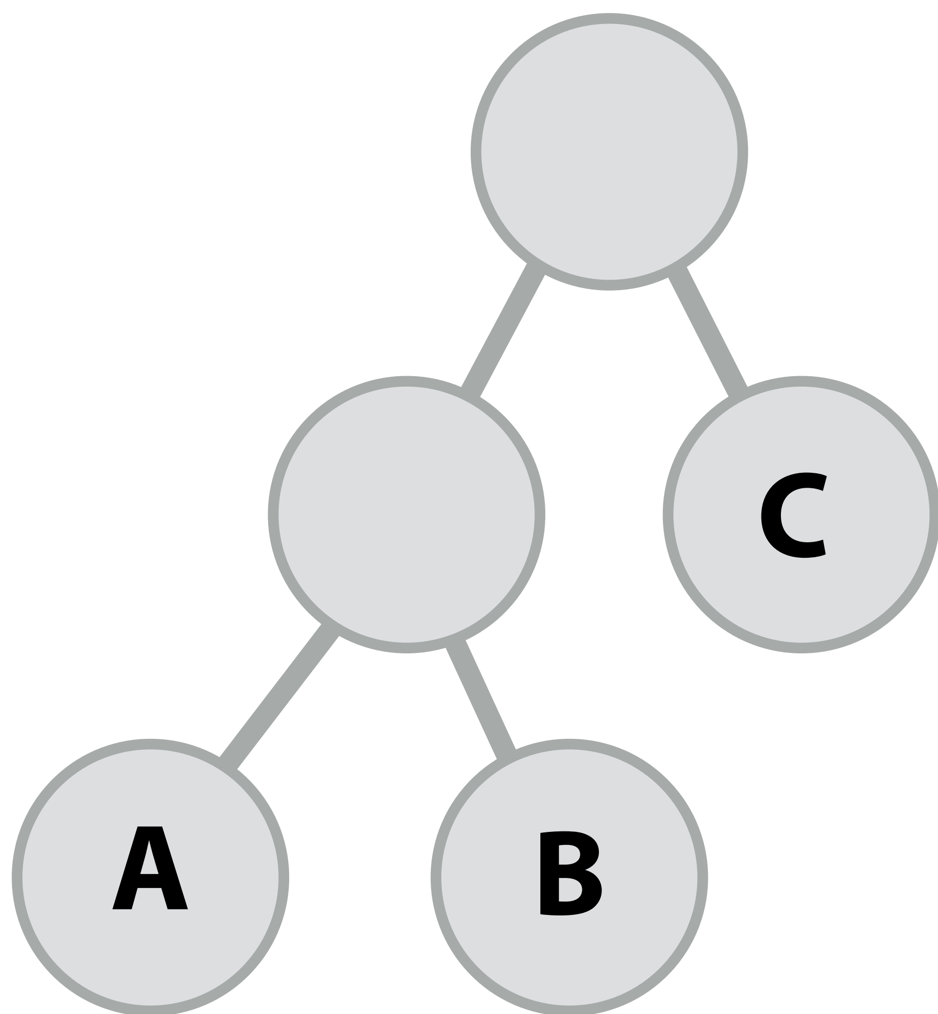
- The graph below plots effective ray throughput (Mrays/sec) as a function of the number of rays traced per BVH build
  - More rays = can amortize costs of BVH build across many ray trace operations



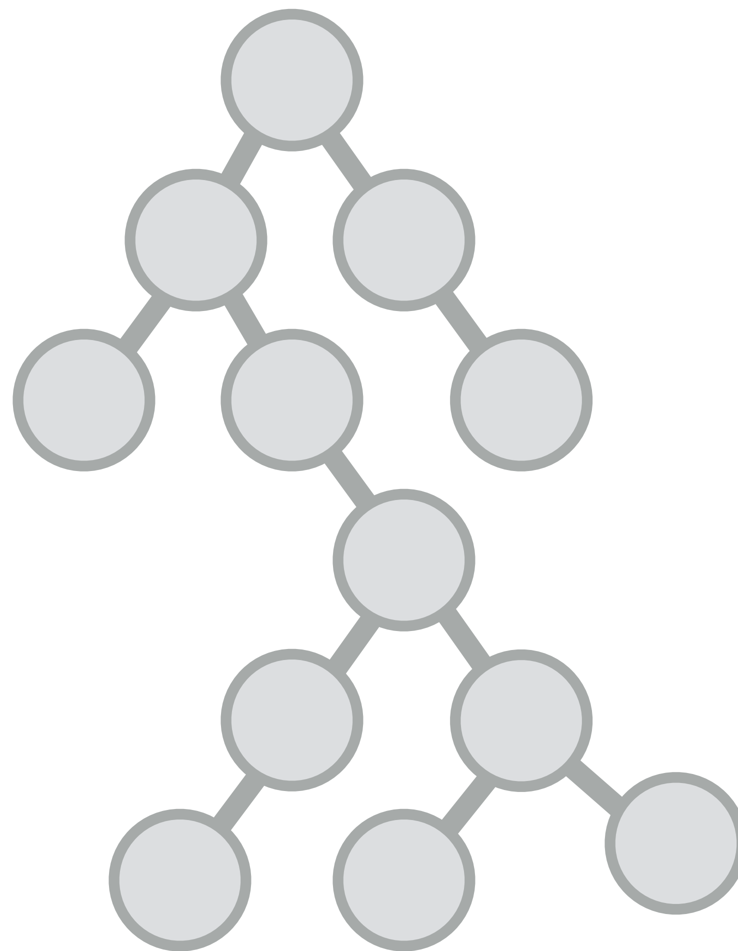
Two-level BVHs

- 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

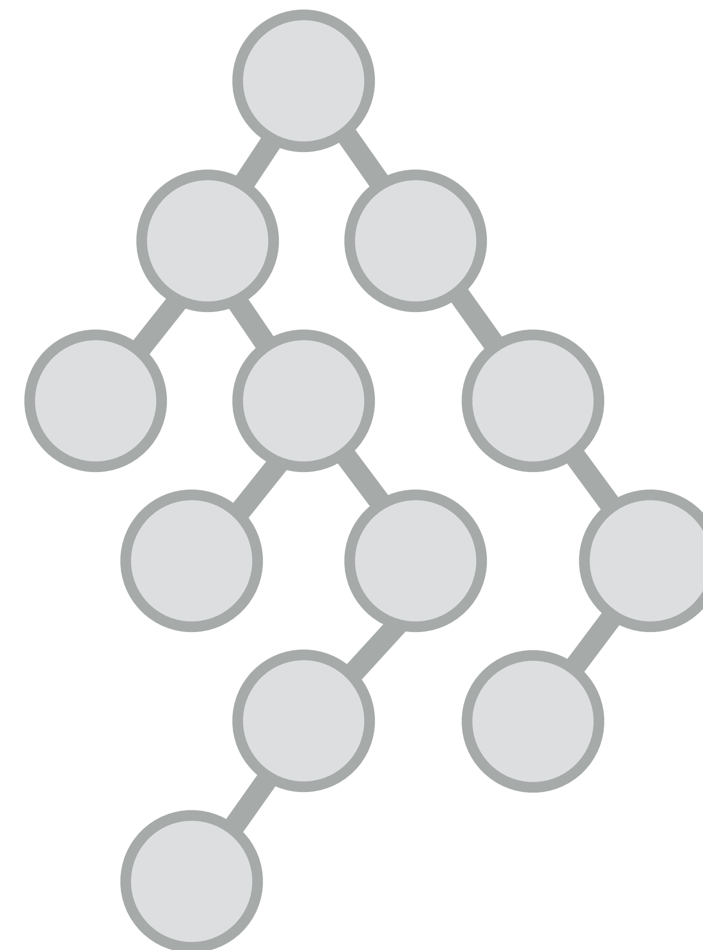
Top Level BVH



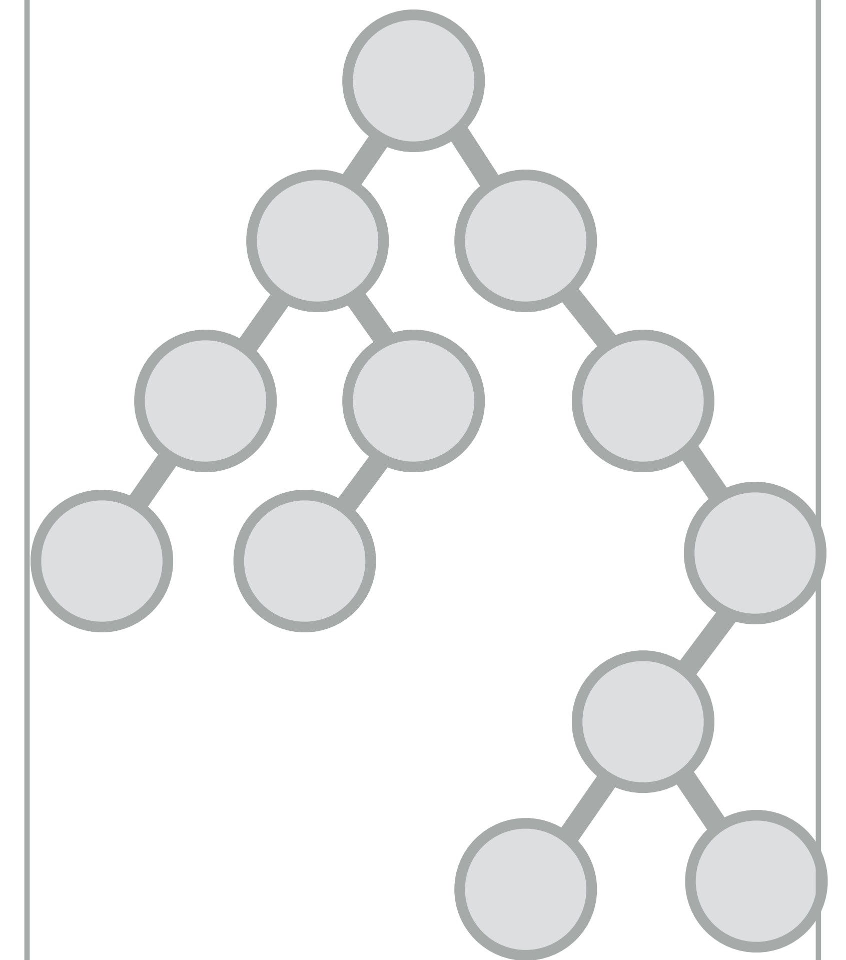
BVH for object A



BVH for object B



BVH for object C



Contains hundreds of scene objects

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

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

- 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
  - Interesting recent results rendering scenes with many lights (we'll have a guest speaker on June 1)
  - Improvements to neural denoising techniques