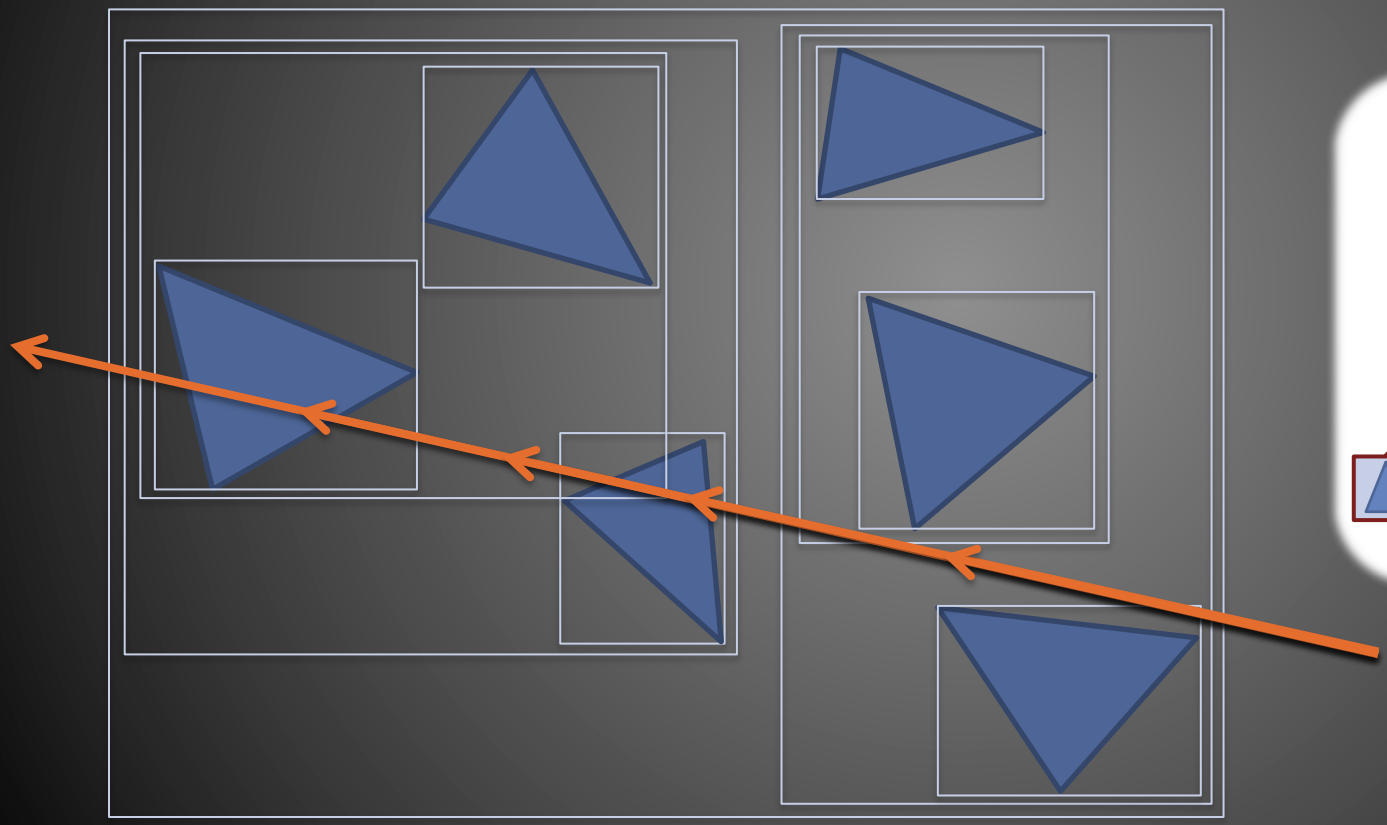
SRDH : Specializing BVH Construction and Traversal Order Using Representative Shadow Ray Sets

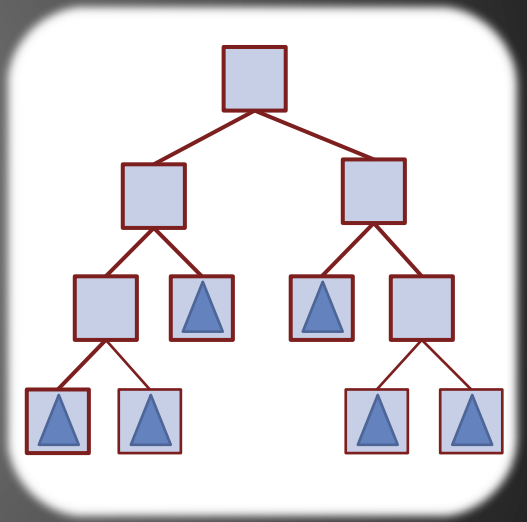
Nicolas Feltman *
Minjae Lee
Kayvon Fatahalian

Carnegie Mellon University

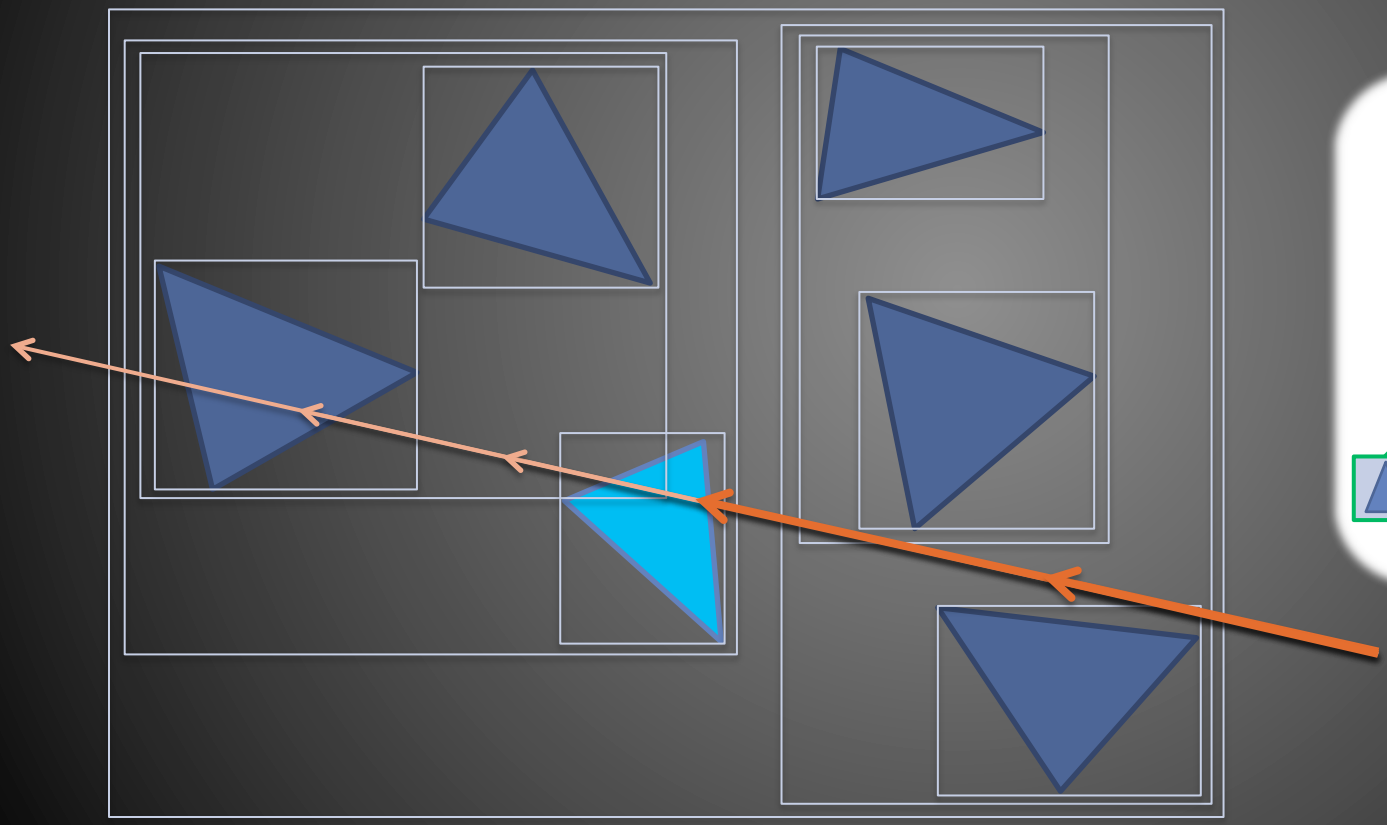
*at Los Alamos CoDesign for the summer

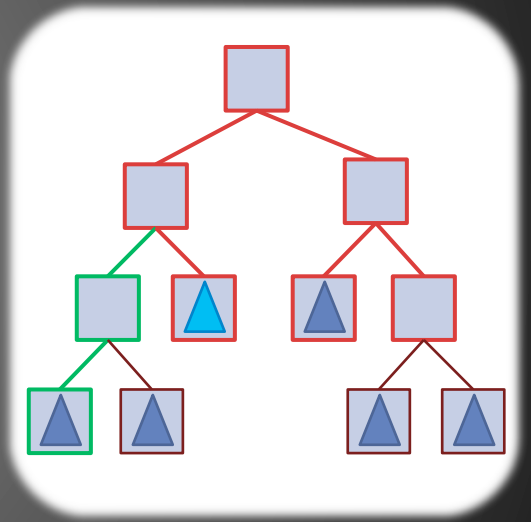
Traversing a BVH (for radiance rays): Find the first hit



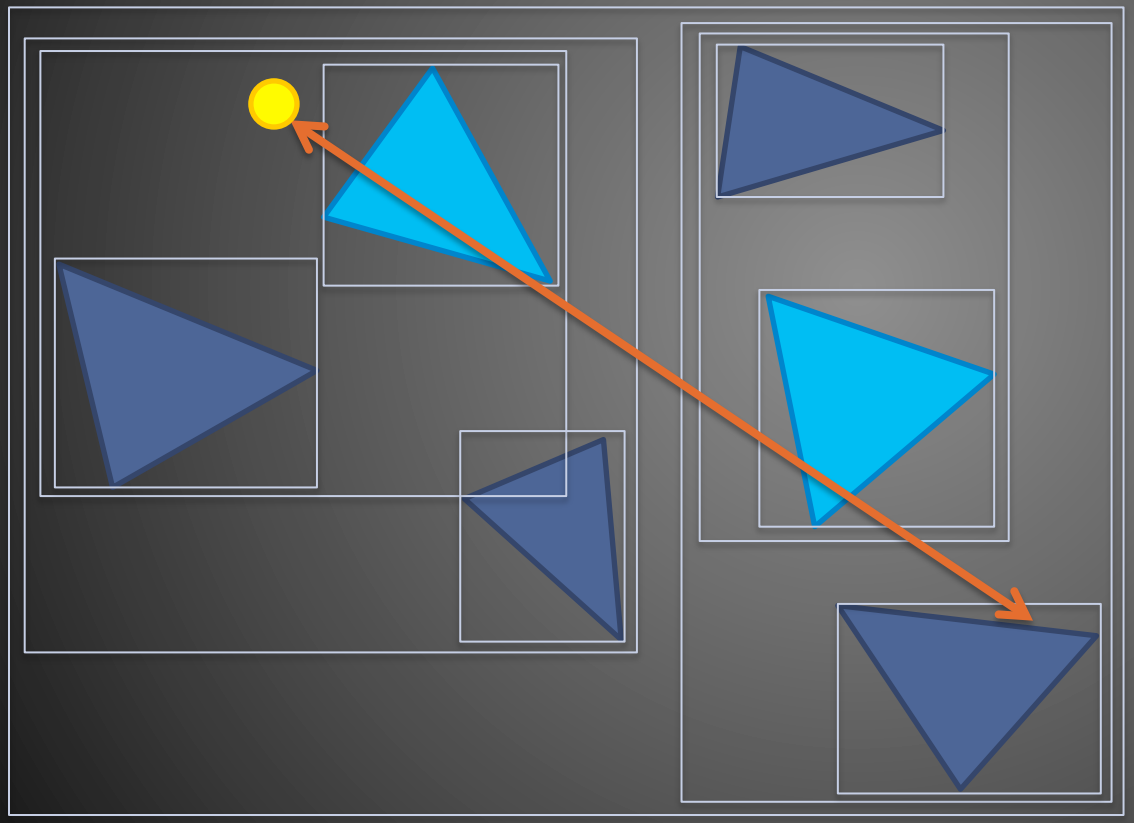


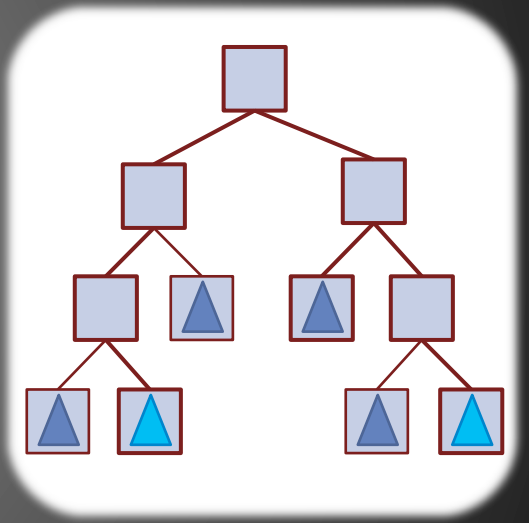
Traversing a BVH (for radiance rays): Find the first hit



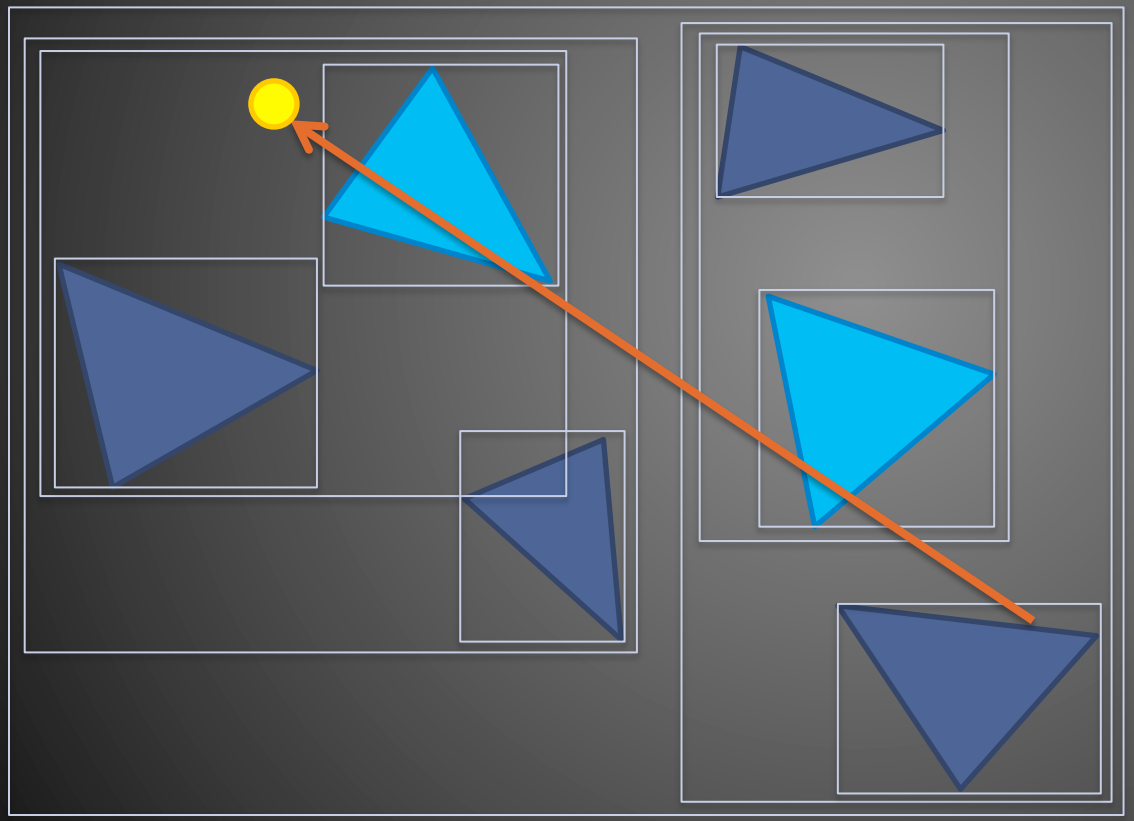


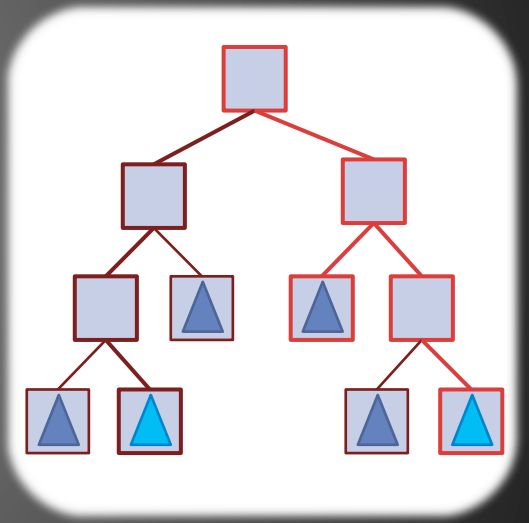
Traversing a BVH (for shadow rays): Find *any* hit



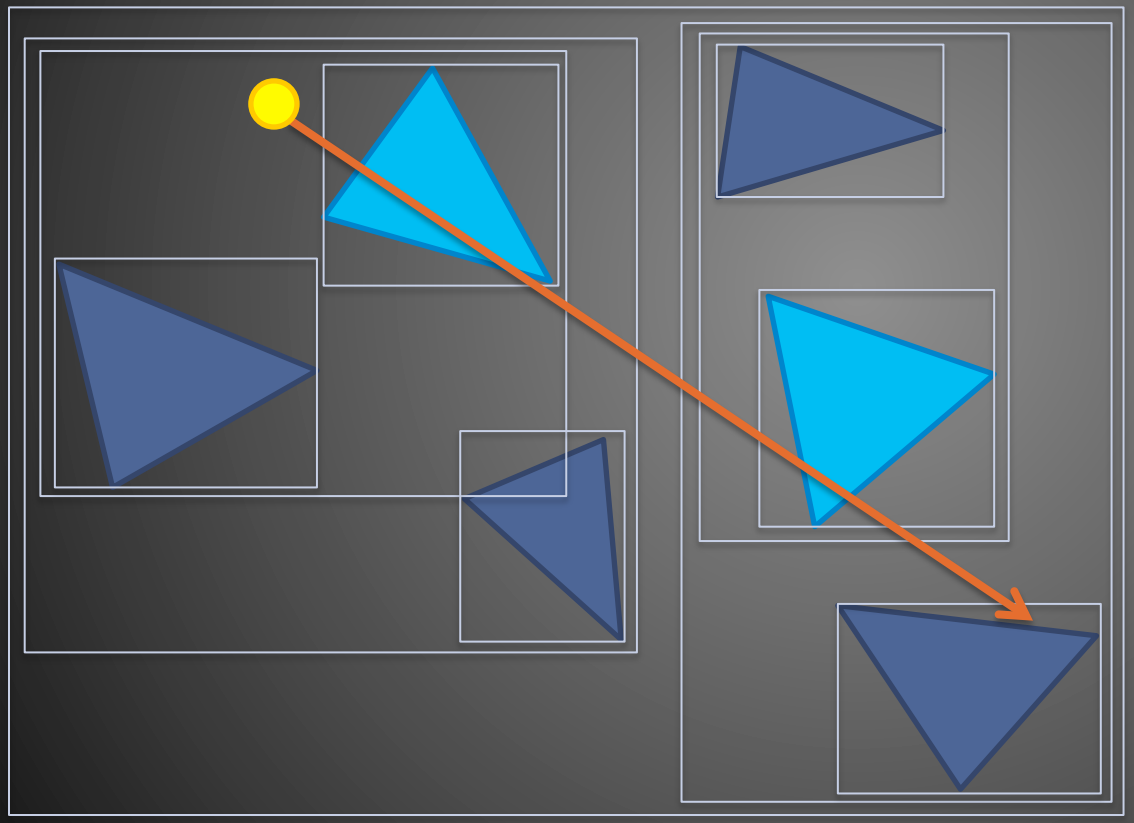


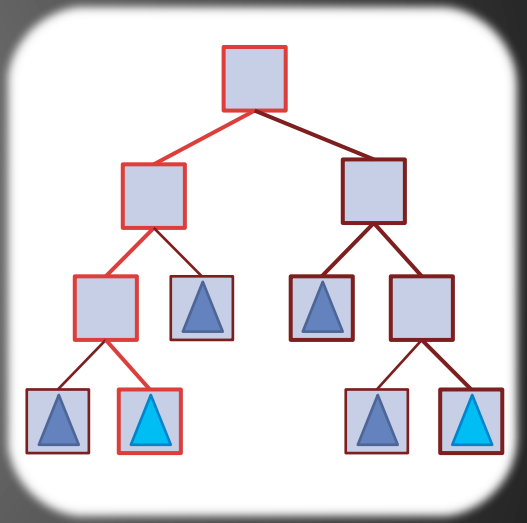
Surface to the light?





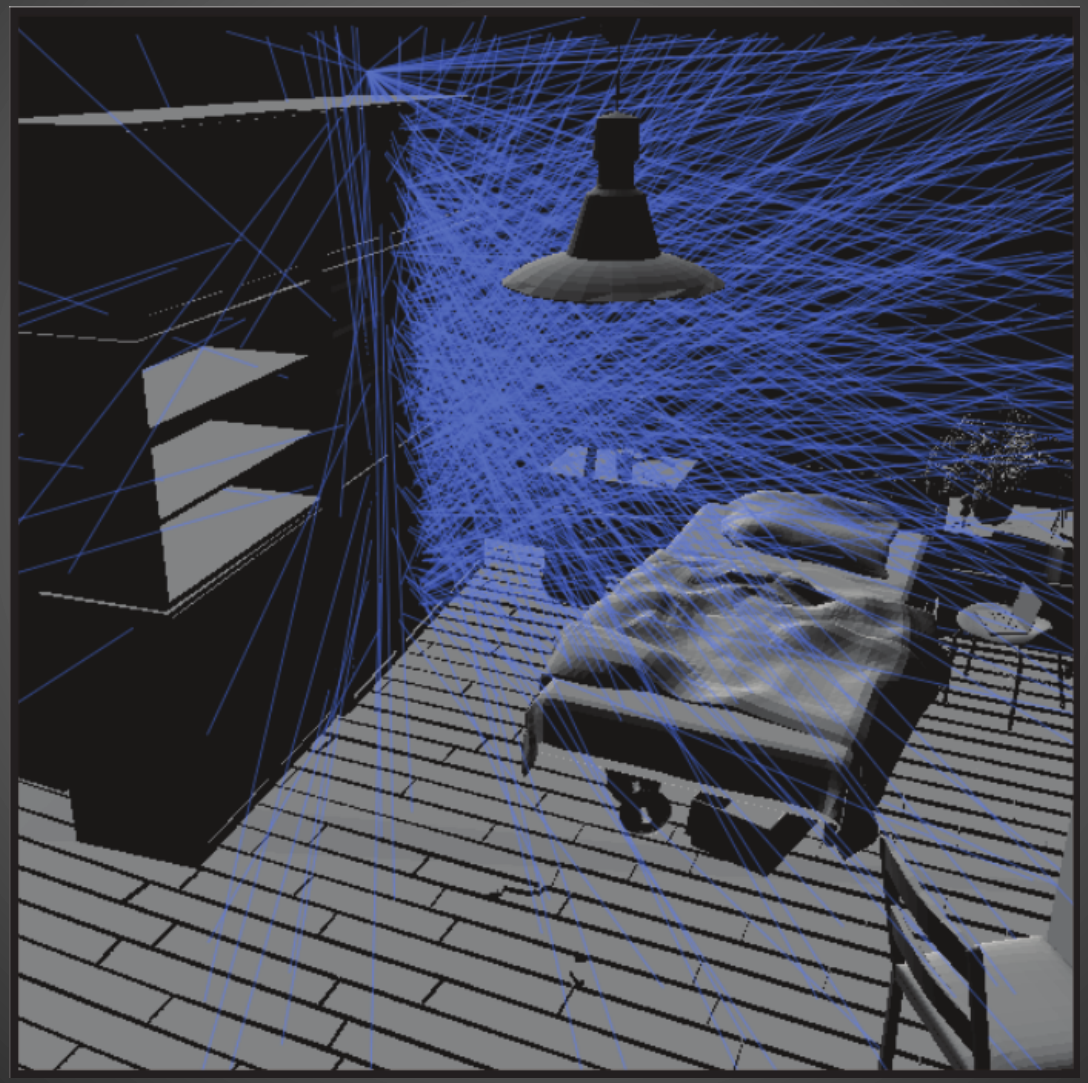
Light to the surface?



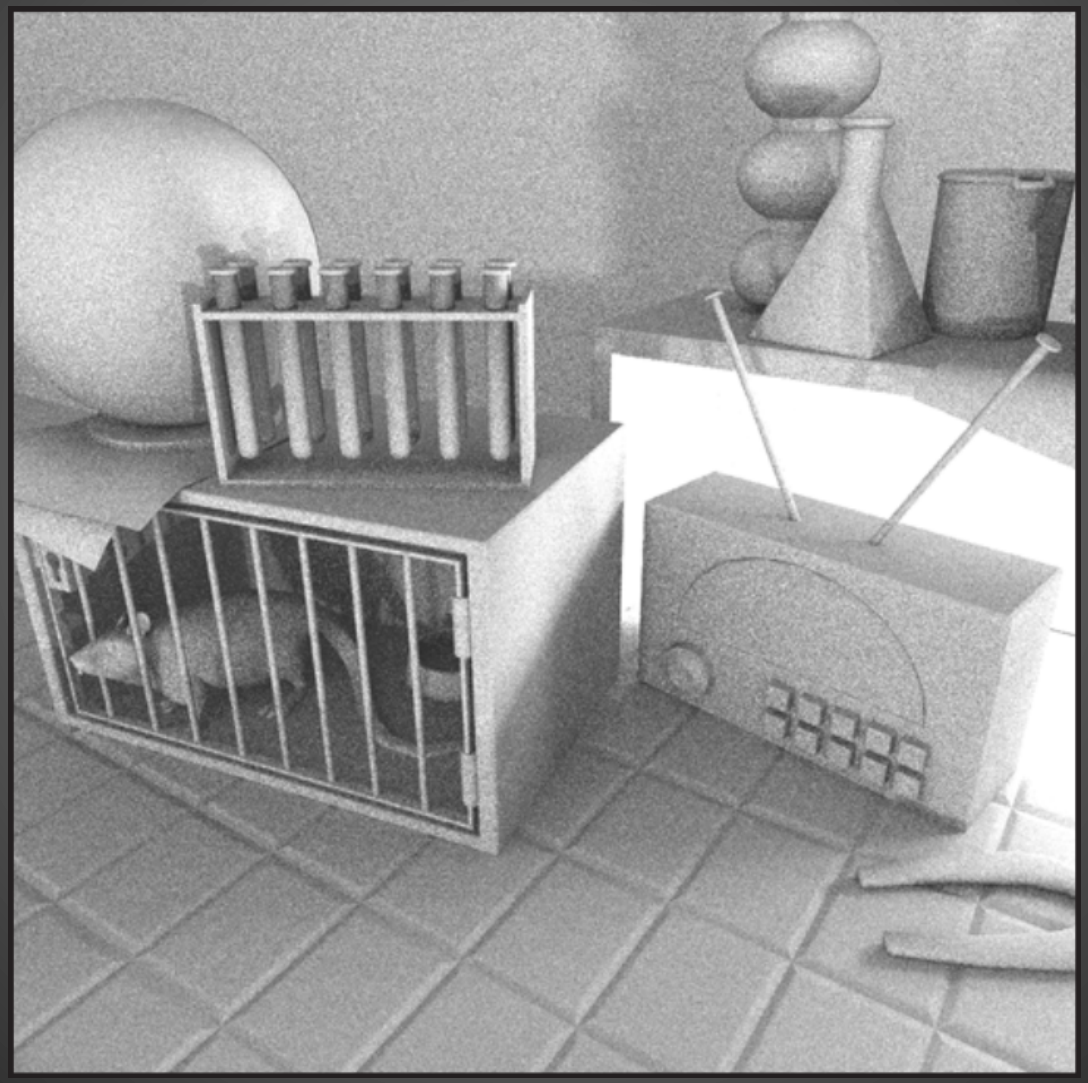


In Bedroom scene, traverse back to front

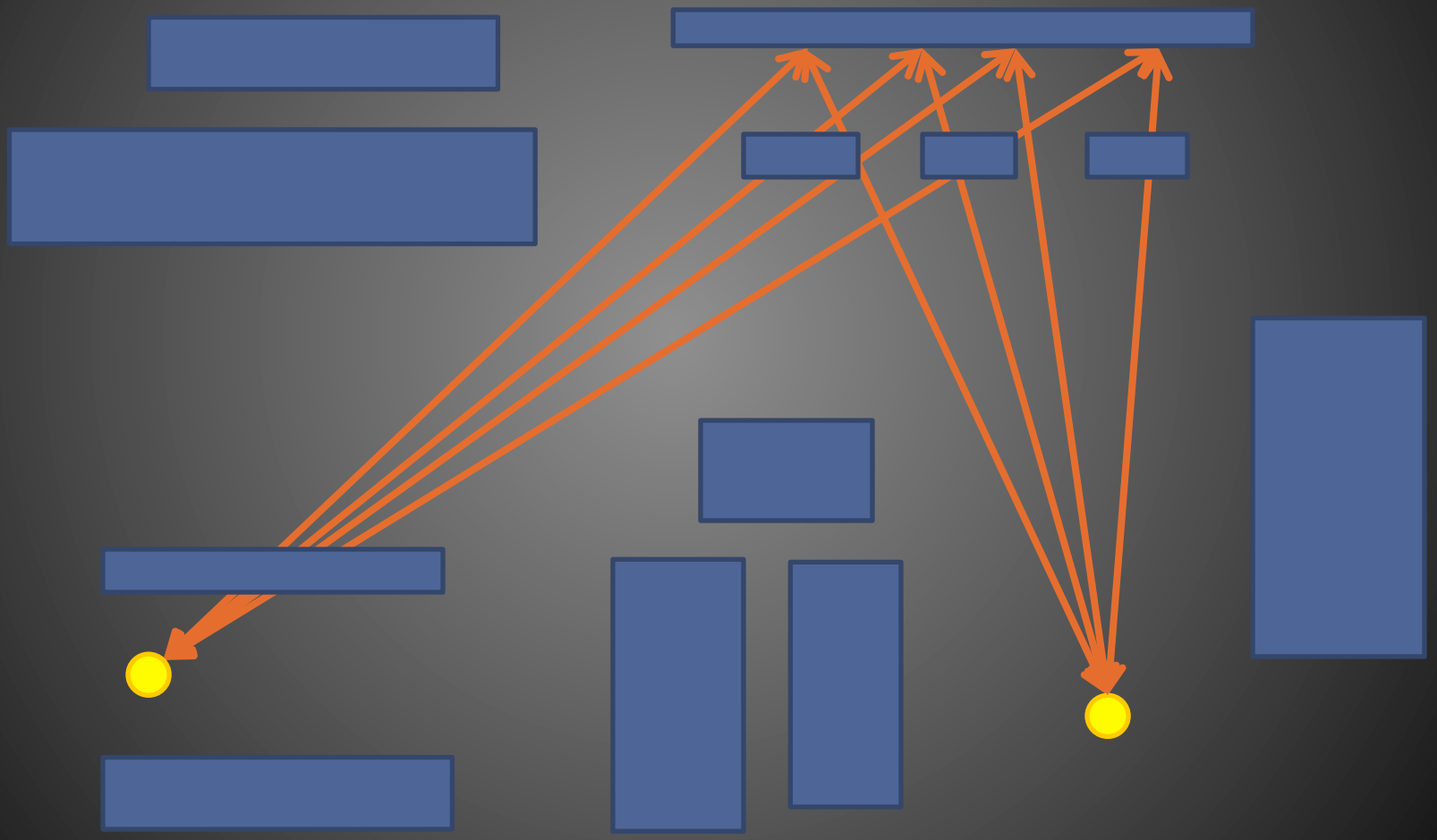
In Bedroom scene, traverse back to front



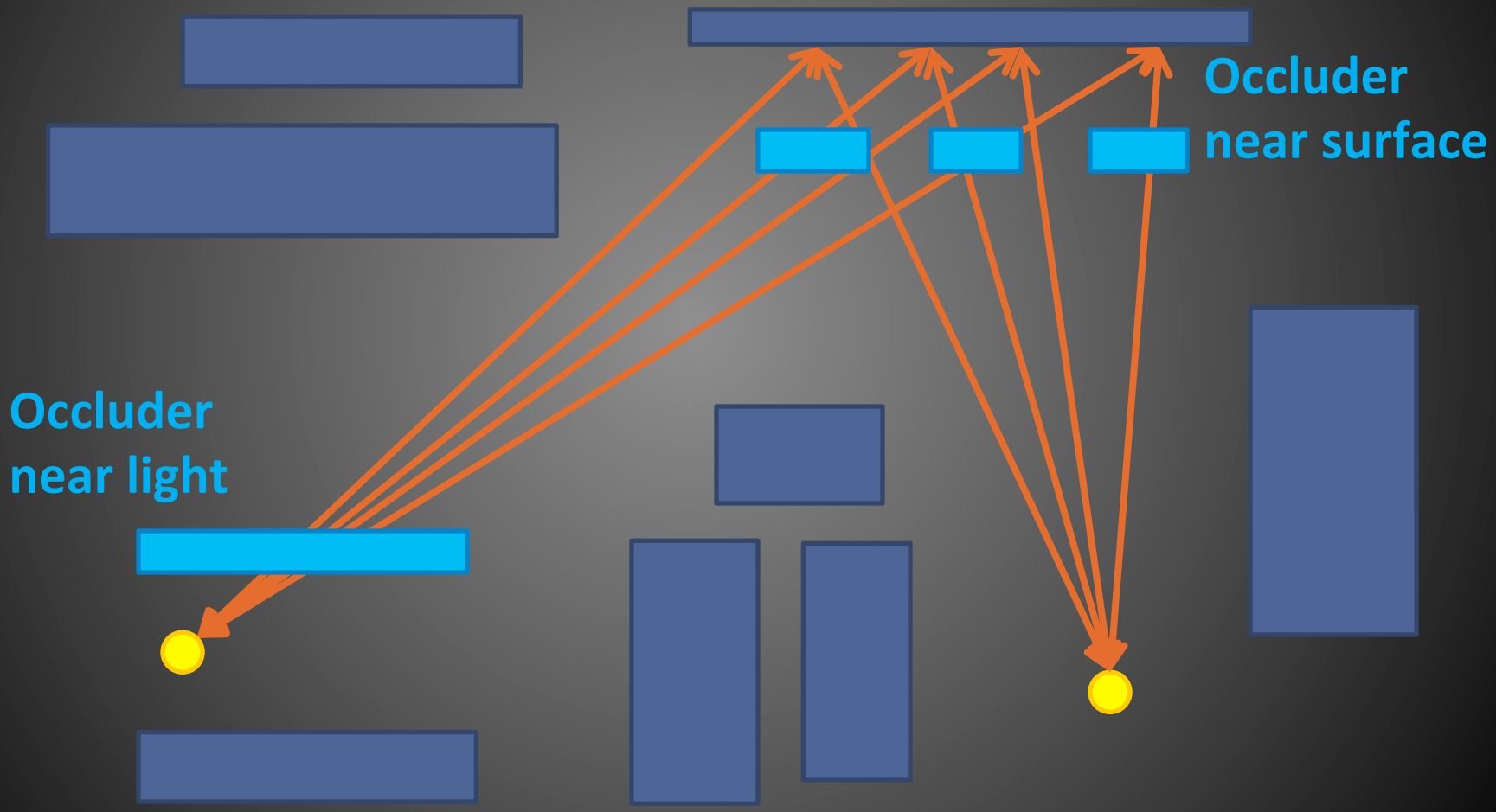
In Mad Science scene, traverse front to back



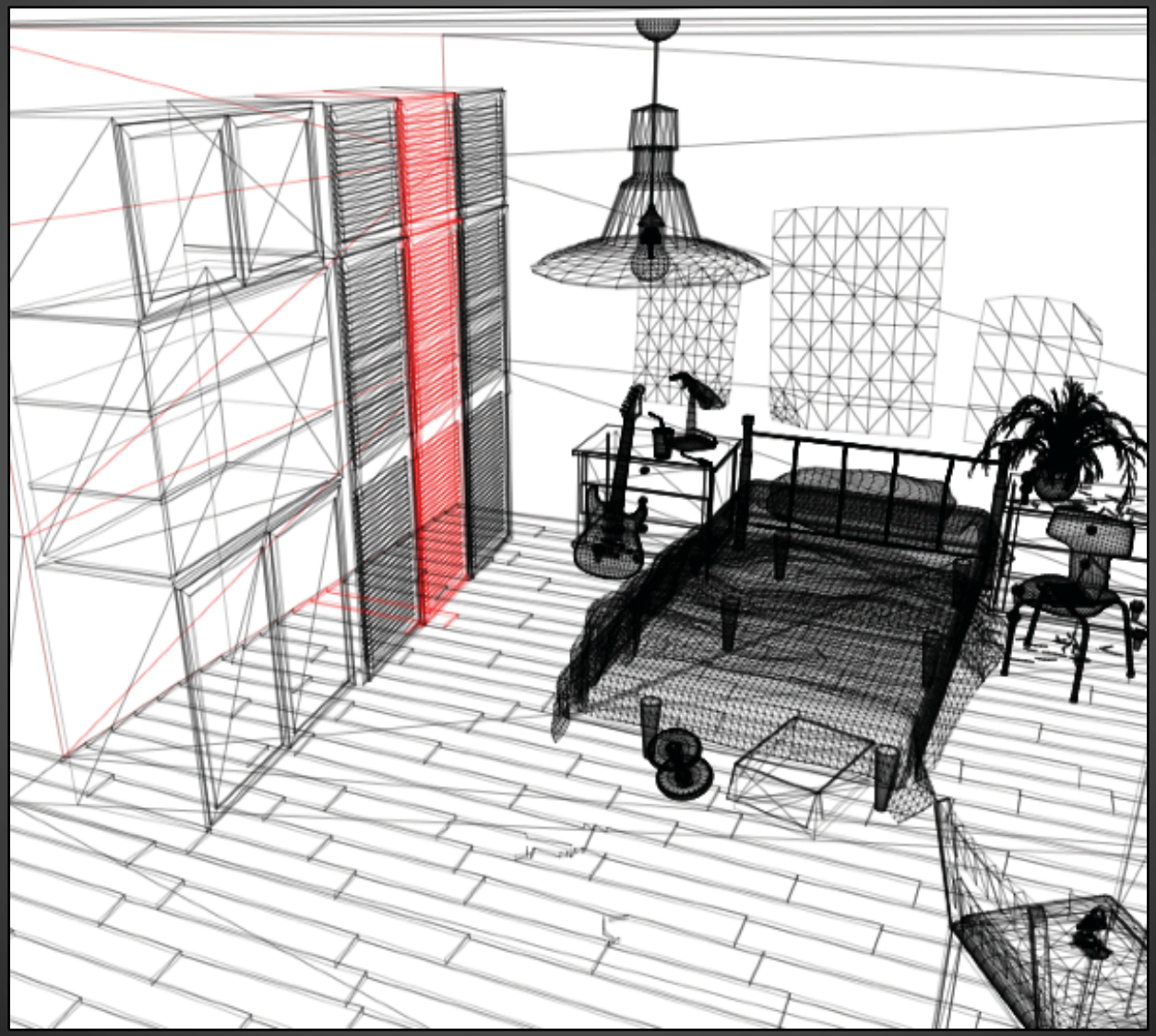
Idea: trace towards *occluders*



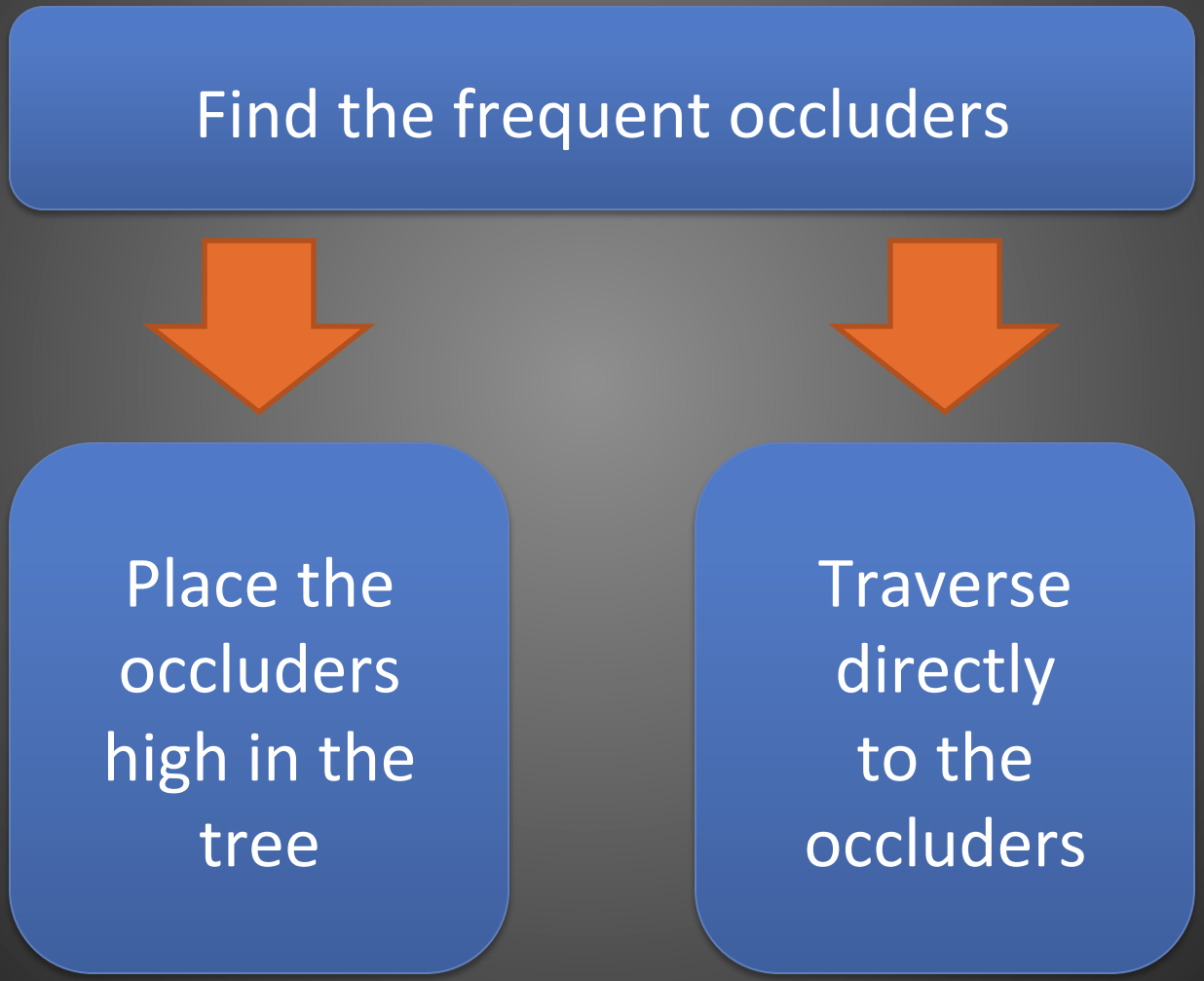
Idea: trace towards *occluders*



Frequent occluders should be closer to root



Optimizing shadow ray traversal: two simultaneous goals



Contribution:
Shadow Ray Distribution Heuristic (SRDH)

Similar Approaches

- RTSAH [Ize & Hansen '11]
  - Starts with standard SAH tree (without ordering).
  - Uses volumetric density as a proxy for occluder likelihood to assign a shadow traversal scheme
- RDH [Bittner & Havran '09]
  - Uses actual ray information to build BVH
  - For use with radiance rays, so no freedom in traversal order

Similar Approaches

- Occluder caches [Haines & Greenberg ‘86]
- Frequent occluder trees

Shadow Ray Distribution Heuristic (SRDH)

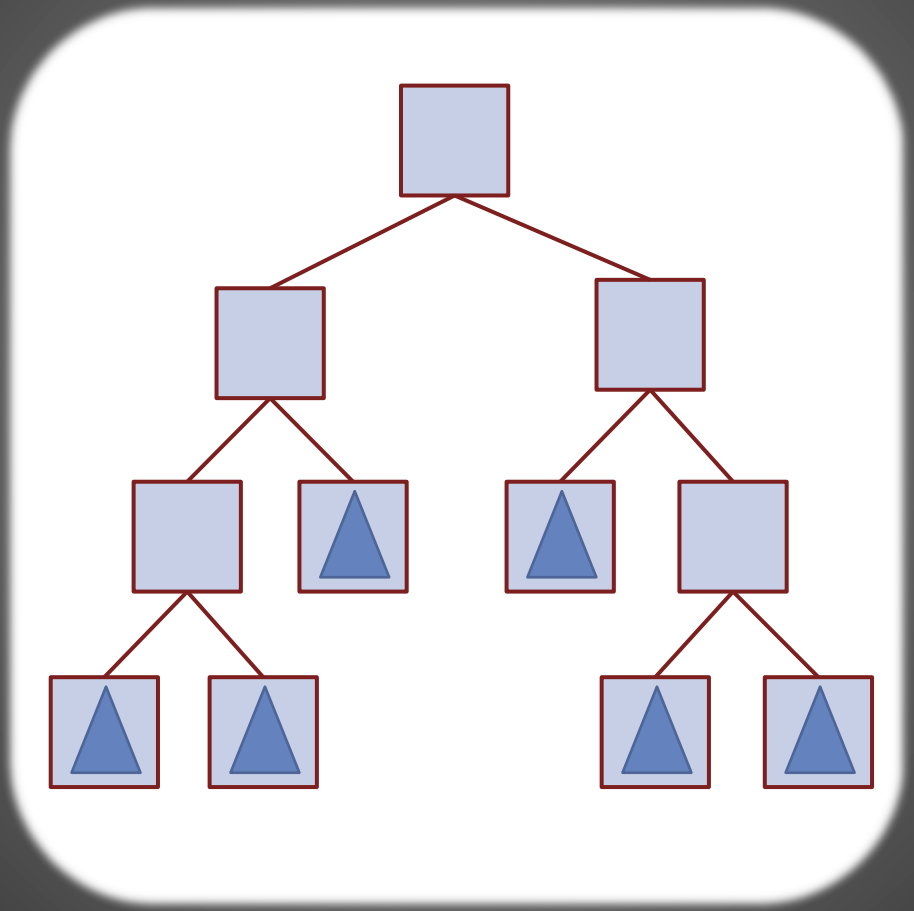
Steps to deriving SRDH:

- Define generalized traversal for shadow rays
- Develop cost metric for our general traversal routine and a *known* ray set
- Use this cost metric to build BVHs

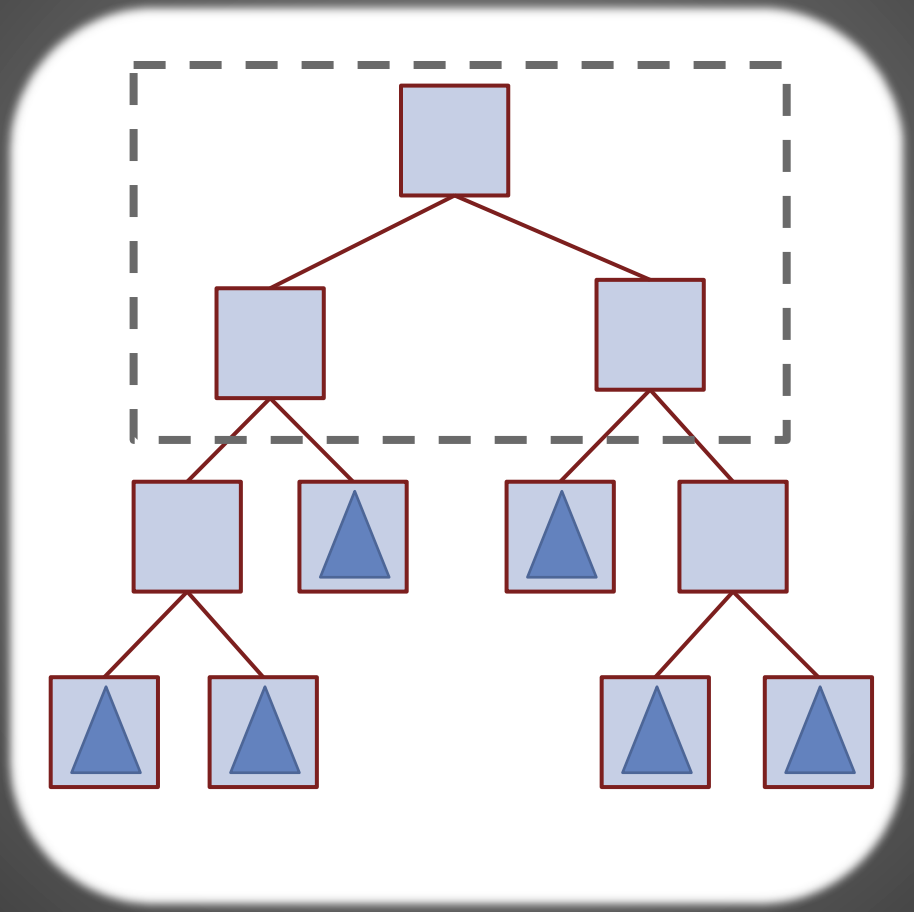
Steps to deriving SRDH:

- Define generalized traversal for shadow rays
- Develop cost metric for our general traversal routine and a *known* ray set
- Use this cost metric to build BVHs

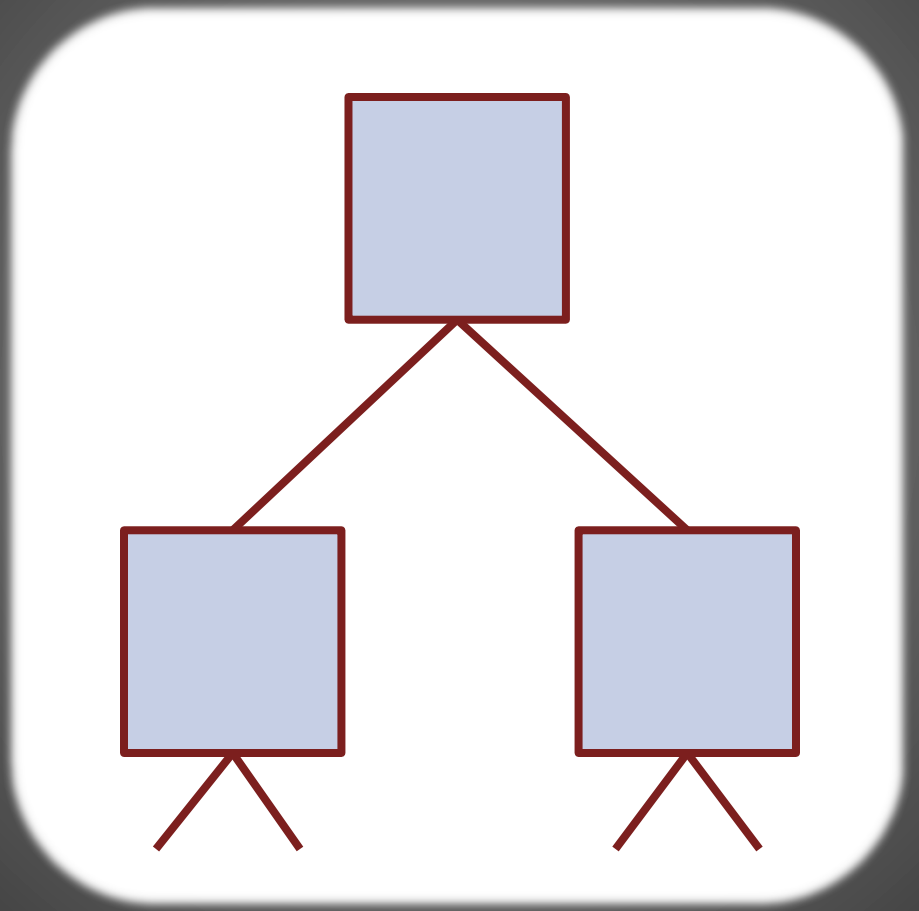
Traversal per node



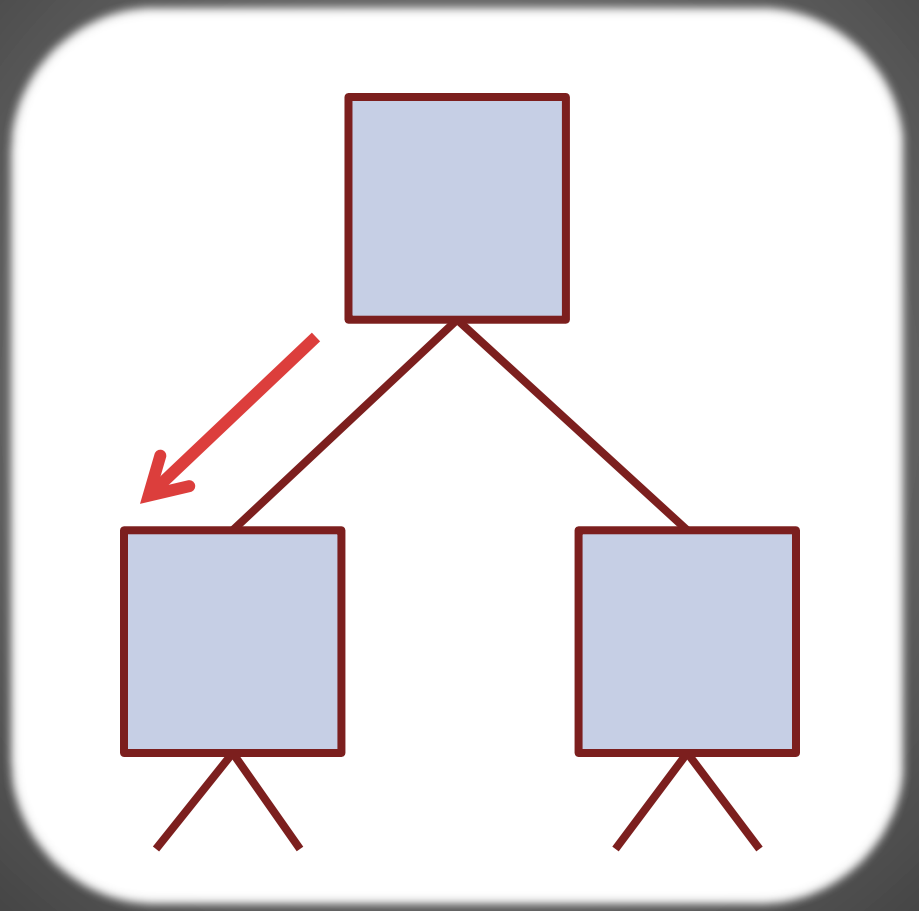
Traversal per node



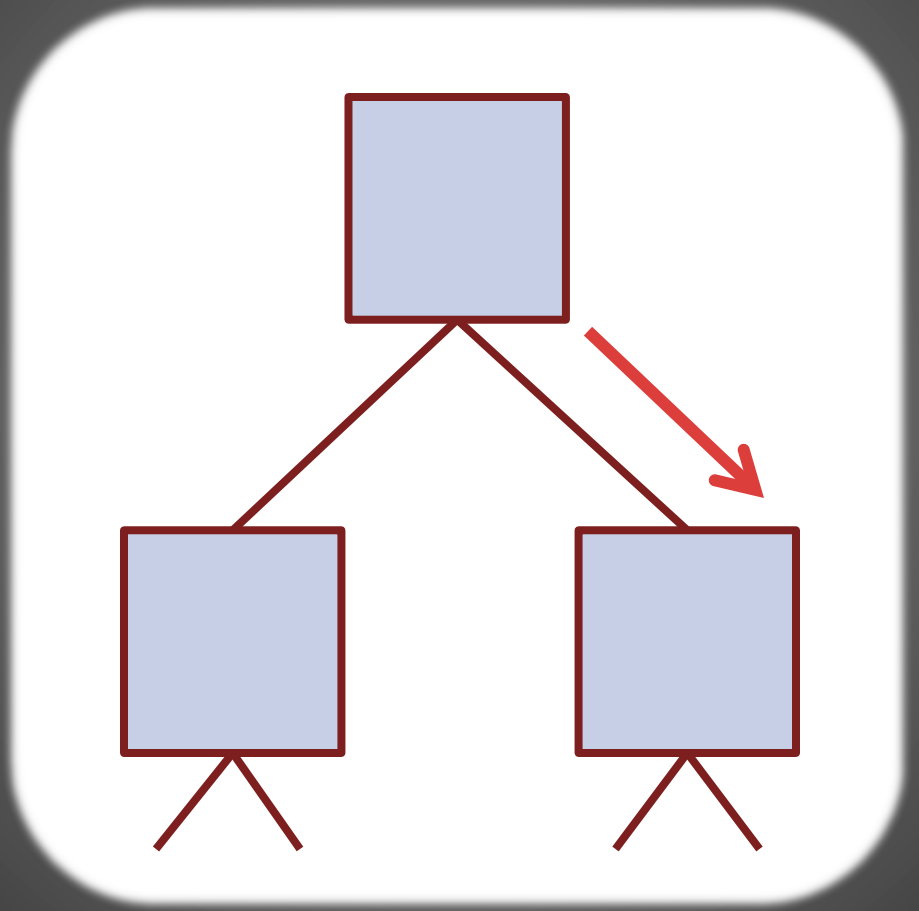
Traversal at a branch



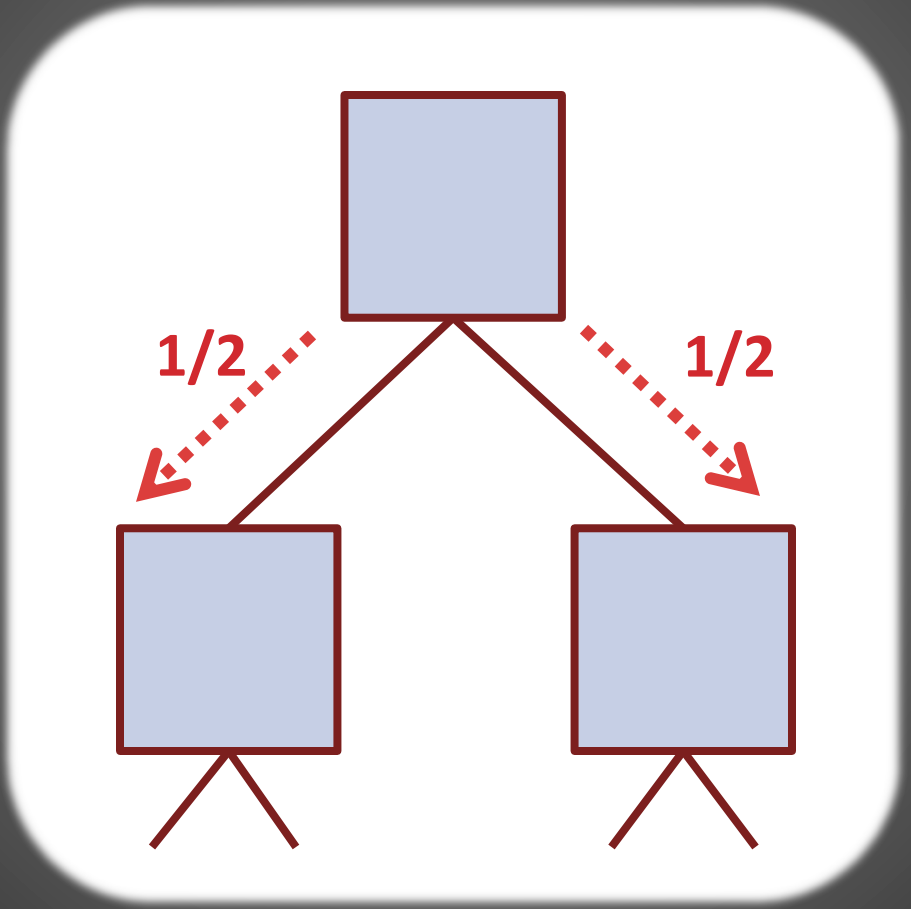
Traversal at a branch



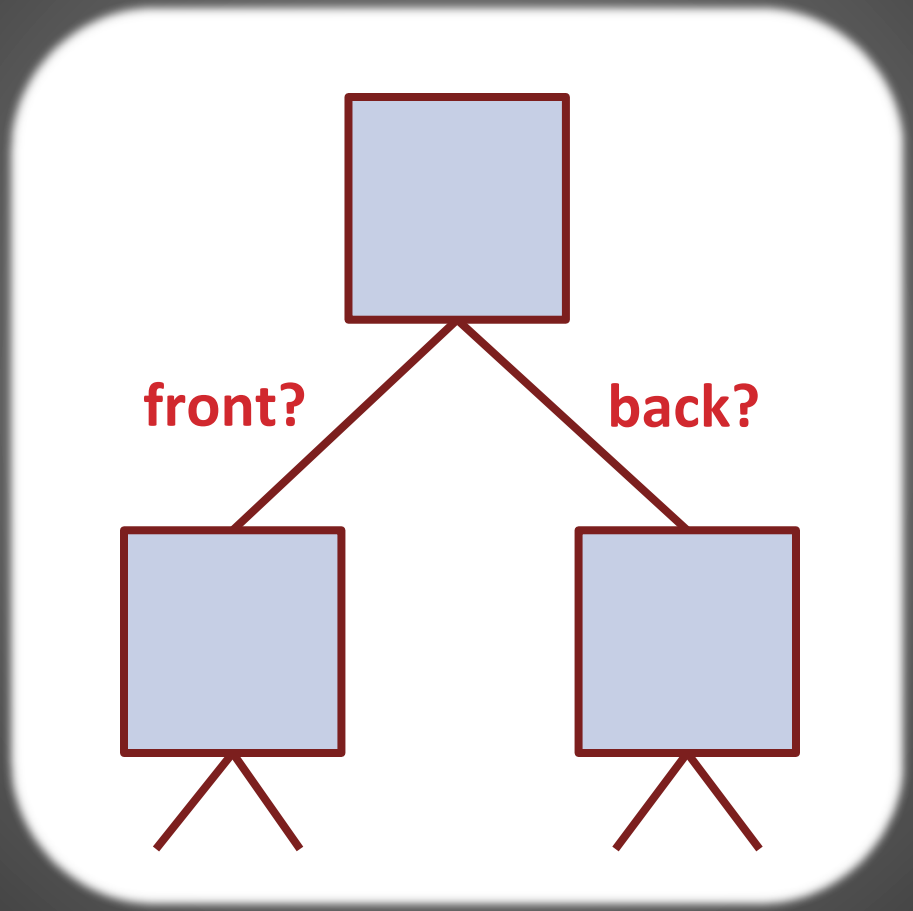
Traversal at a branch



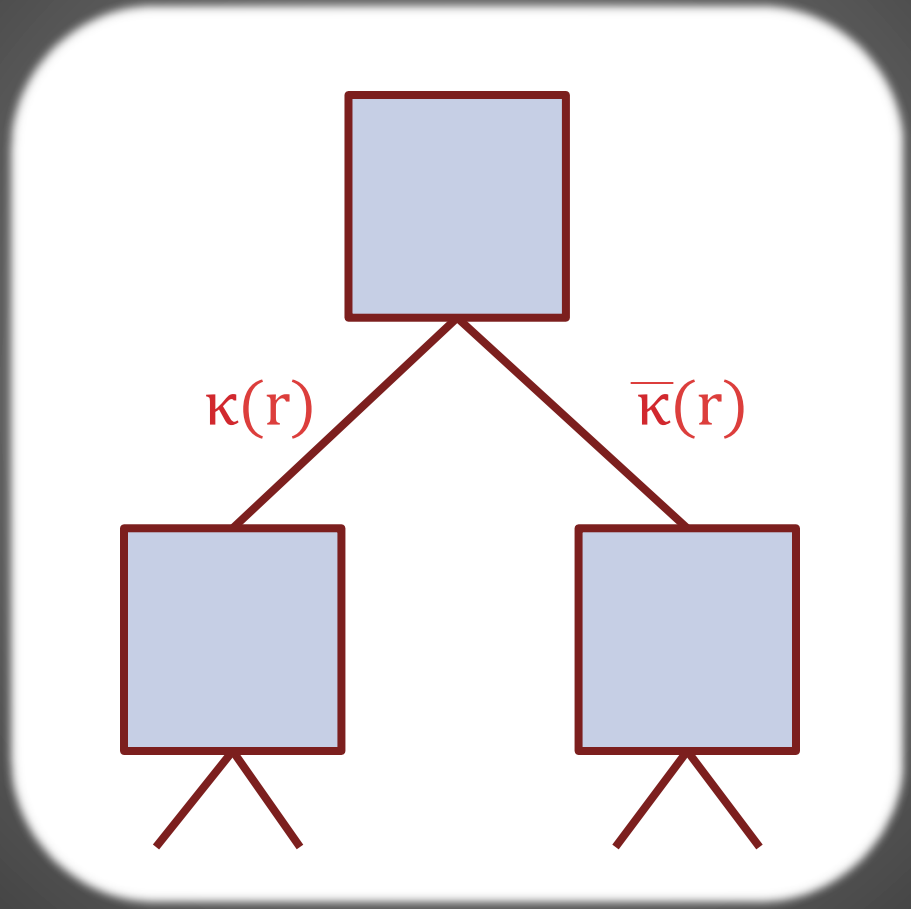
Traversal at a branch



Traversal at a branch



Traversal per node



$\overline{\kappa}(r) = 1 - \kappa(r)$

Specify traversal policy for each node

Left First: $\kappa(r) = 1$

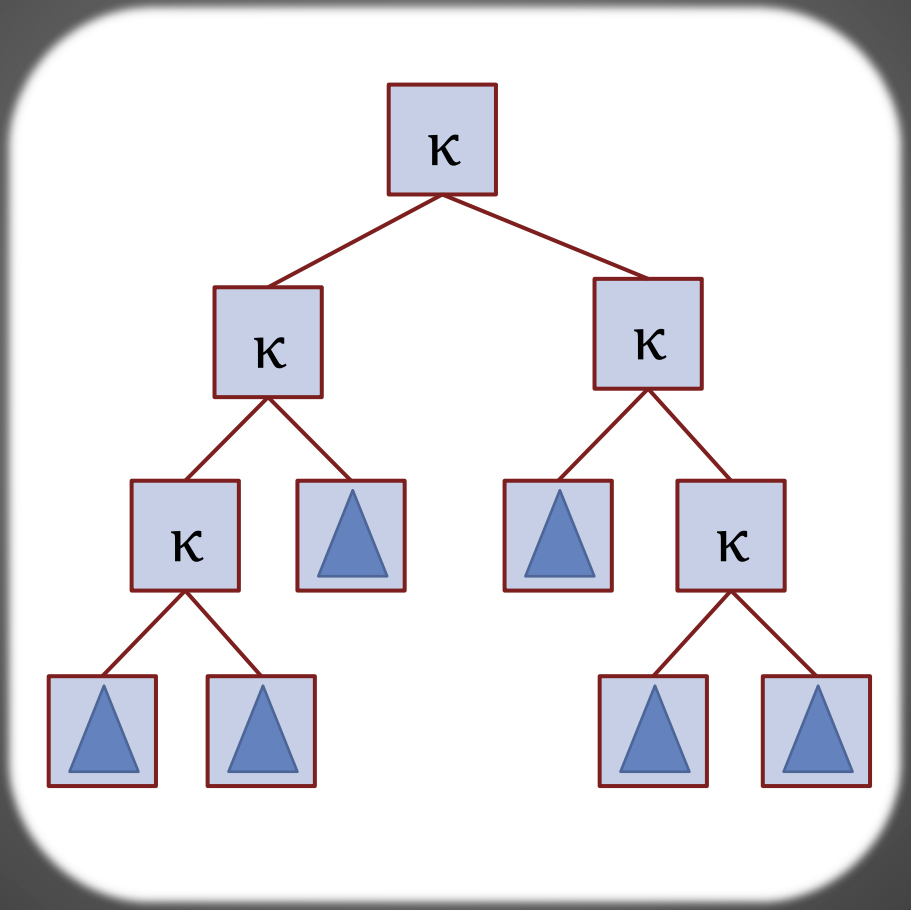
Right First: $\kappa(r) = 0$

Random: $\kappa(r) = 0.5$

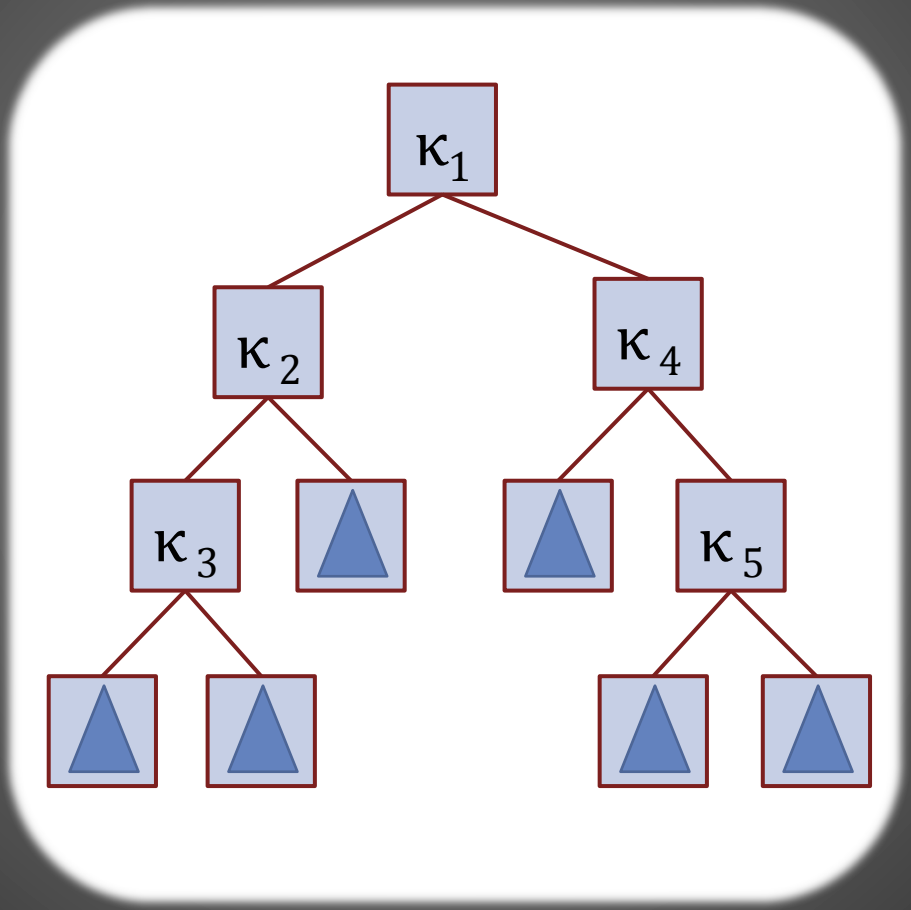
Back to front: $\kappa(r) = \begin{cases} 1 & \text{if r...} \\ 0 & \text{if r...} \end{cases}$

Front to back: $\kappa(r) = \begin{cases} 0 & \text{if r...} \\ 1 & \text{if r...} \end{cases}$

Specify traversal policy for each node



Specify traversal policy for each node

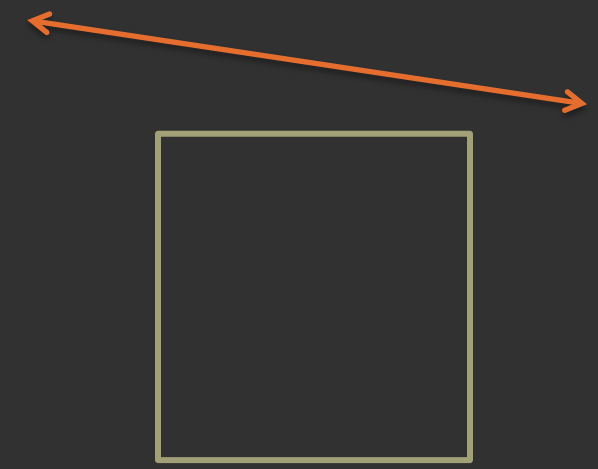


Generalized Kernel Traversal

```
function traverse(node, ray)

   if(ray misses the node's bounds)
      return MISS;

   if(node is a leaf)
      test with n's primitives
        and return the result;

   if(node is a branch)
      evaluate κ_N(r) to determine
        traversal order;
      if(the first child is a hit)
        return HIT;
      if(the second child is a hit)
        return HIT;
```

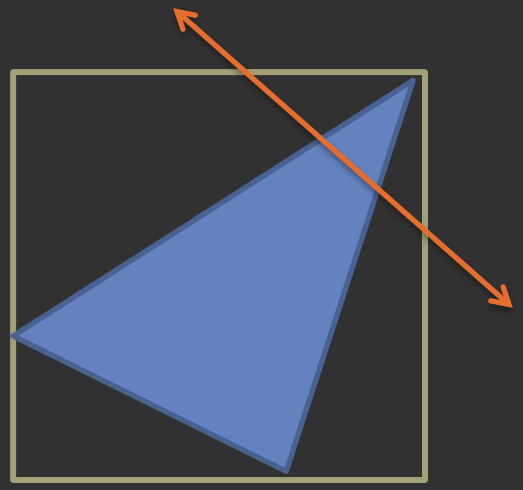
Generalized Kernel Traversal

```
function traverse(node, ray)

   if(ray misses the node's bounds)
      return MISS;

   if(node is a leaf)
      test with n's primitives
        and return the result;

   if(node is a branch)
      evaluate κ_N(r) to determine
        traversal order;
      if(the first child is a hit)
        return HIT;
      if(the second child is a hit)
        return HIT;
```



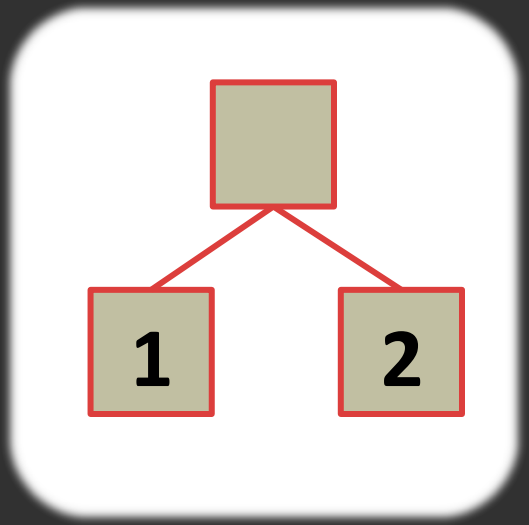
Generalized Kernel Traversal

```
function traverse(node, ray)

   if(ray misses the node's bounds)
      return MISS;

   if(node is a leaf)
      test with n's primitives
        and return the result;

   if(node is a branch)
      evaluate κN(r) to determine
        traversal order;
      if(the first child is a hit)
        return HIT;
      if(the second child is a hit)
        return HIT;
```



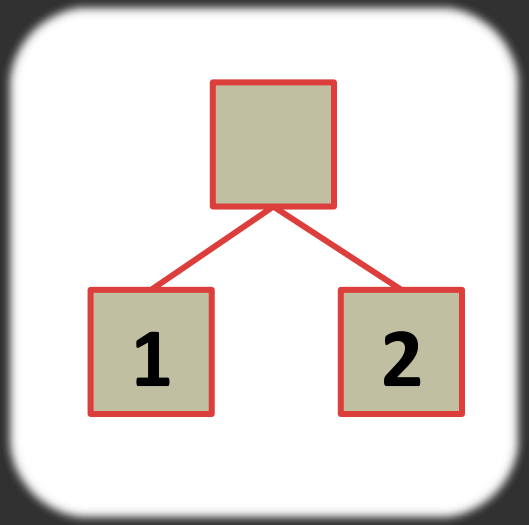
Full Kernel Traversal

```
function traverse(node, ray)

   if(ray misses the node's bounds)
      return MISS;

   if(node is a leaf)
      test with n's primitives
        and return the result;

   if(node is a branch)
      evaluate κ_N(r) to determine
        traversal order;
      if(the first child is a hit)
        return HIT;
      if(the second child is a hit)
        return HIT;
```



Generalized Kernel Traversal

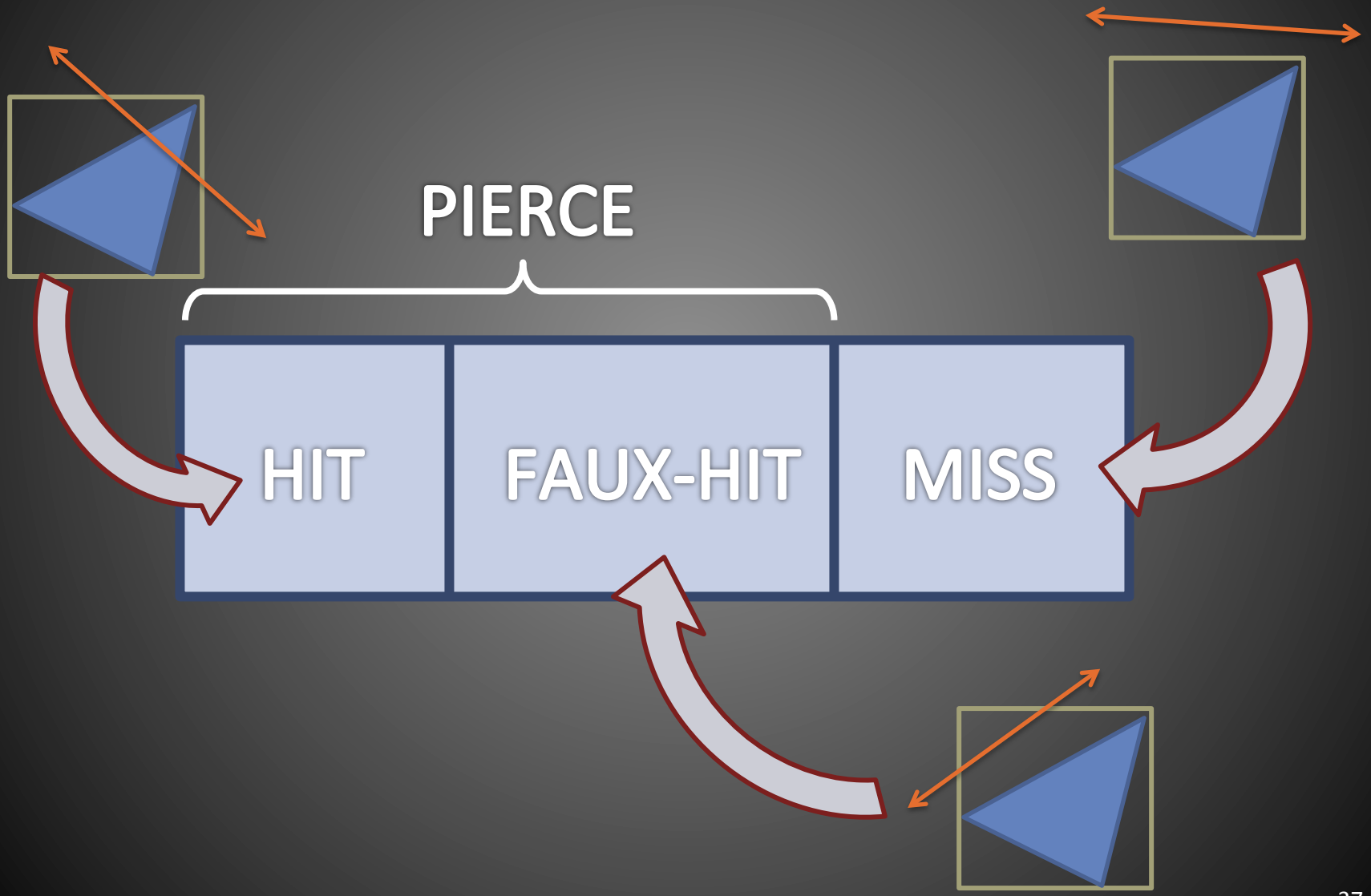
```
function traverse(node, ray)

   if(ray misses the node's bounds)
      return MISS;

   if(node is a leaf)
      test with n's primitives
        and return the result;

   if(node is a branch)
      evaluate κ_N(r) to determine
        traversal order;
      if(the first child is a hit)
        return HIT;
      if(the second child is a hit)
        return HIT;
```



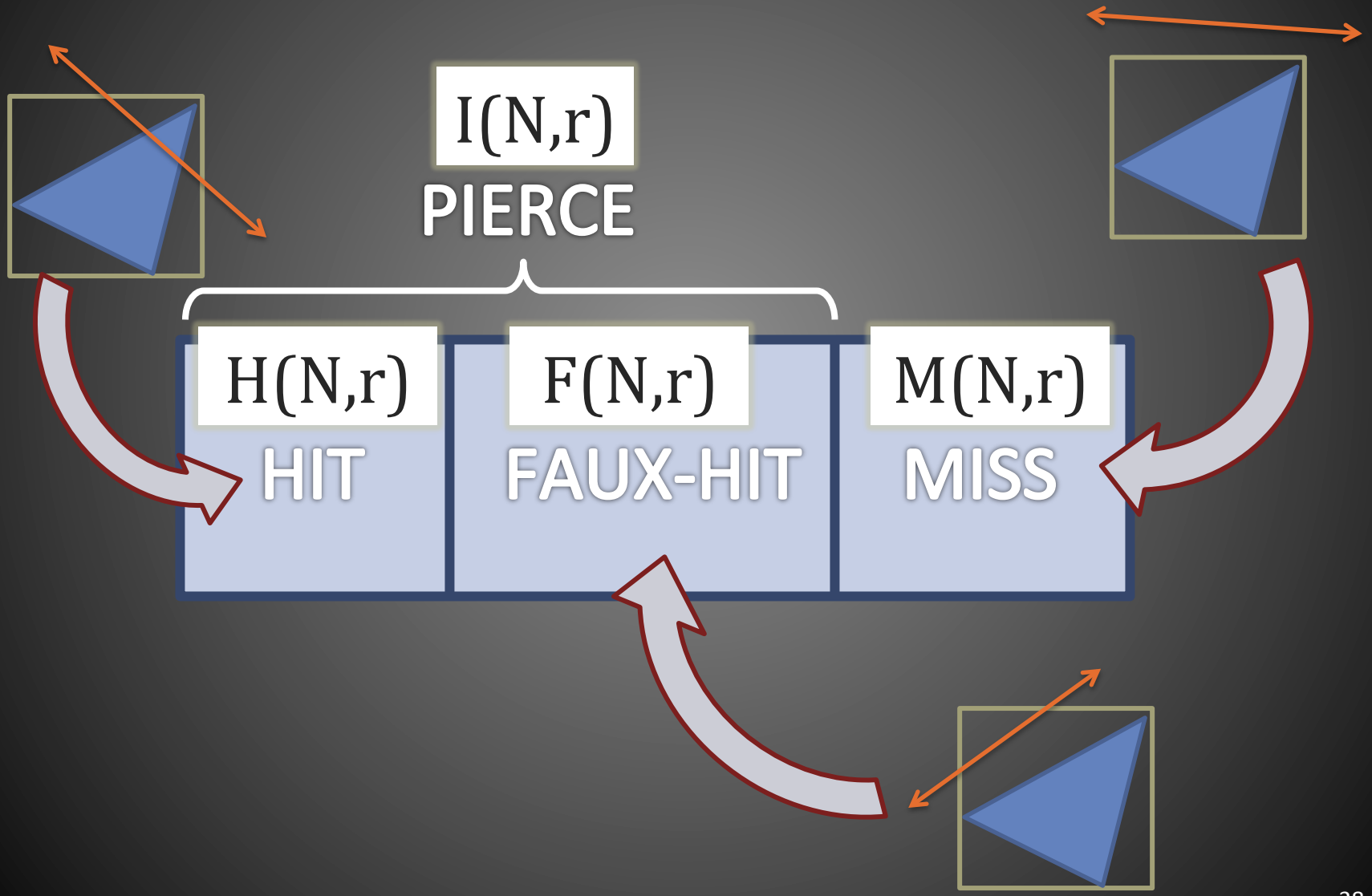
Steps to deriving SRDH:

- Define generalized traversal for shadow rays
- Develop cost metric for our general traversal routine and a *known* ray set.
- Use this cost metric to build BVHs.

Rays and nodes interact in three ways



Encode states with binary indicator functions



The cost metric follows from the traversal algorithm

```
function traverse(node, ray)
   if(ray misses the node's bounds)
      return MISS;
   if(node is a leaf)
      test with n's primitives
        and return the result;
   if(node is a branch)
      evaluate κ_N(r) to determine
        traversal order;
      if(the first child is a hit)
        return HIT;
      if(the second child is a hit)
        return HIT;
```

The cost metric follows from the traversal algorithm

```
function traverse(node, ray)
   if(ray misses the node's bounds)
      return MISS;
   if(node is a leaf)
      test with n's primitives
        and return the result;
   if(node is a branch)
      evaluate κ_N(r) to determine
        traversal order;
      if(the first child is a hit)
        return HIT;
      if(the second child is a hit)
        return HIT;
```

Leaf case:

$$C(N,r)=c_b+I(N,r)c_p$$

The cost metric follows from the traversal algorithm

```
function traverse(node, ray)
   if(ray misses the node's bounds)
      return MISS;
   if(node is a leaf)
      test with n's primitives
        and return the result;
   if(node is a branch)
      evaluate κ_N(r) to determine
        traversal order;
      if(the first child is a hit)
        return HIT;
      if(the second child is a hit)
        return HIT;
```

Branch case:

$$C(N,r)=c_b+I(N,r)C'(N,r)$$

$$C'(N,r)=\kappa(r)Ord(L,R) + \ldots$$

$$Ord(L,R)=C(L,r) + (1-H(L,r))C(R,r)$$

The cost metric follows from the traversal algorithm

```
function traverse(node, ray)
   if(ray misses the node's bounds)
      return MISS;
   if(node is a leaf)
      test with n's primitives
        and return the result;
   if(node is a branch)
      evaluate κ_N(r) to determine
        traversal order;
      if(the first child is a hit)
        return HIT;
      if(the second child is a hit)
        return HIT;
```

Branch case:

$$C(N,r)=c_b+I(N,r)C'(N,r)$$

$$C'(N,r)=\kappa(r)\mathrm{Ord}(L,R) + \ldots$$

$$\mathrm{Ord}(L,R)=C(L,r)+(1-H(L,r))C(R,r)$$

The cost metric follows from the traversal algorithm

```
function traverse(node, ray)
   if(ray misses the node's bounds)
      return MISS;
   if(node is a leaf)
      test with n's primitives
        and return the result;
   if(node is a branch)
      evaluate κ_N(r) to determine
        traversal order;
      if(the first child is a hit)
        return HIT;
      if(the second child is a hit)
        return HIT;
```

Branch case:

$$C(N,r)=c_b+I(N,r)C'(N,r)$$

$$C'(N,r)=\kappa(r)Ord(L,R)+\bar{\kappa}(r)Ord(R,L)$$

$$Ord(L,R)=C(L,r)+(1-H(L,r))C(R,r)$$

Branch case: canonical recursive form

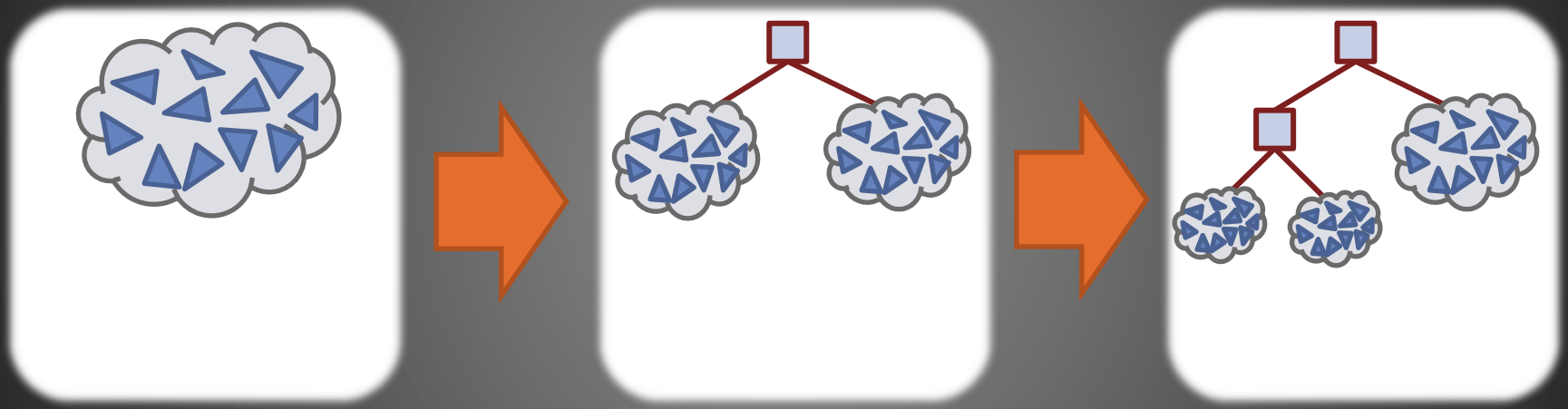
$$C(N,r)=c_b+I(N,r)C'(N,r)$$

$$C'(N,r)=(1-\bar{\kappa}(r)H(R,r))C(L,r)$$
$$+(1-\kappa(r)H(L,r))C(R,r)$$

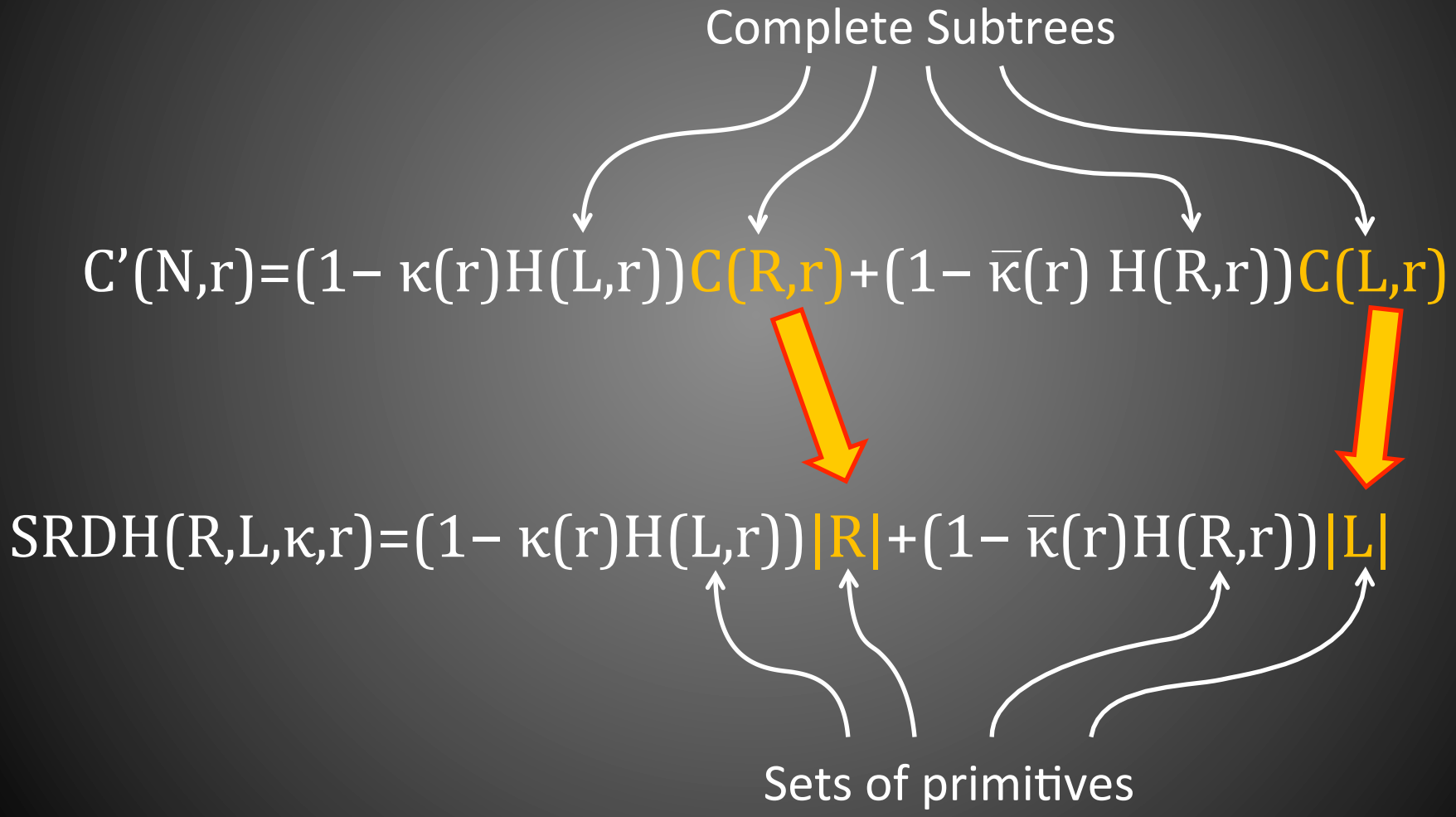
Steps to deriving SRDH:

- Define generalized traversal for shadow rays
- Develop cost metric for our general traversal routine and a *known* ray set
- Use this cost metric to build BVHs

Top-down and greedy BVH build



A heuristic estimates the cost metric



Greedy SRDH build optimizes over partitions and traversal policies

SAH:

```
forall(partitions in set-of-partitions)
        …evaluate SAH and pick min…
```

SRDH:

```
forall(partitions in set-of-partitions)
        forall(traversalKernels in set-of-kernels)
                …evaluate SRDH and pick min…
```

Greedy SRDH build optimizes over partitions and traversal policies

SAH:

```
forall(partitions in set-of-partitions)
        …evaluate SAH and pick min…
```

SRDH:

```
forall(partitions in set-of-partitions)
        forall(traversalKernels in set-of-kernels)
                …evaluate SRDH and pick min…
```

$$SRDH(R,L,\kappa,r)=(1-\kappa(r)H(L,r))|R|+(1-\kappa(r)H(R,r))|L|$$

Greedy SRDH build optimizes over partitions and traversal policies

SAH:

```
forall(partitions in set-of-partitions)
        …evaluate SAH and pick min…
```

SRDH:

```
forall(partitions in set-of-partitions)
        forall(traversalKernels in set-of-kernels)
                …evaluate SRDH and pick min…
```

$$SRDH(R,L,\kappa,r)=(1-\kappa(r)H(L,r))|R|+(1-\kappa(r)H(R,r))|L|$$

Putting it all together: three stage build

- STEP 1: Build a BVH with SAH (BVH can also be used for radiance rays too)
- STEP 2: Do full trace of the whole representative ray set, finding EVERY hit for each ray.
- STEP 3: Use representative ray set to build a specialized BVH with the SRDH.

Results

Two key questions:

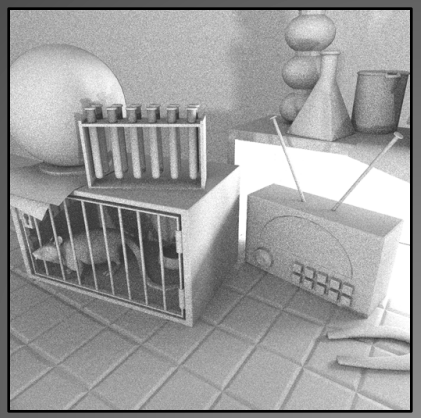
- How much does SRDH improve traversal cost when perfect information about shadow rays is present?

- How does the benefit of the SRDH decrease as less shadow ray information is known a priori? (Is a practical implementation possible?)

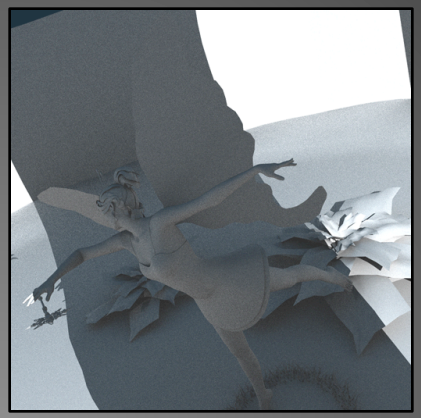
Test Scenes / Experimental Setup

Bedroom
(93% occluded)

Mad Science
(80% occluded)



Fairy
(56% occluded)



Sponza
(14% occluded)

Arcade
(38% occluded)

- 1024x1024 renderings
- Four possible traversal kernels:
  - back-to-front, front-to-back, left-first, right-first
- BVH build: up to 32 partitioning buckets in each dimension
- Use count of nodes traversed as a proxy for traversal cost

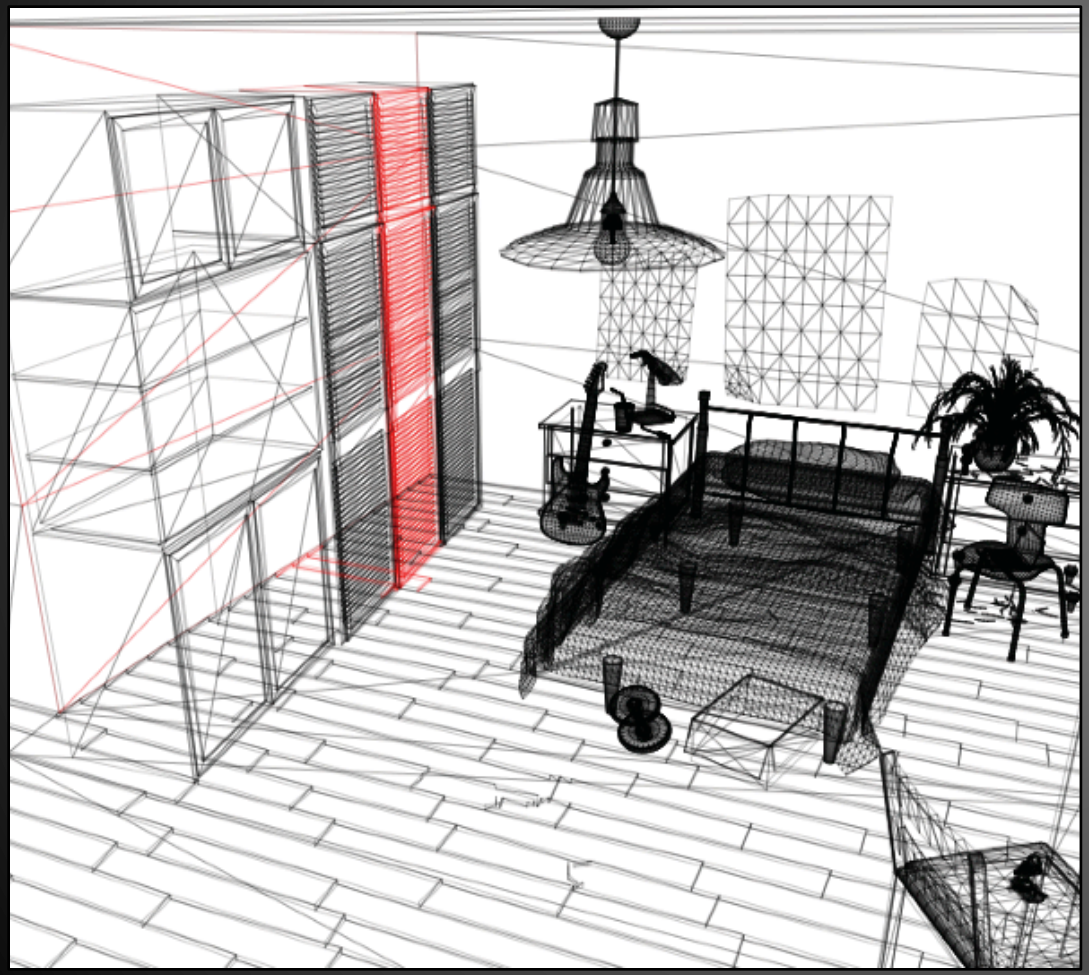
How much does SRDH improve traversal cost when perfect information about shadow rays is present?

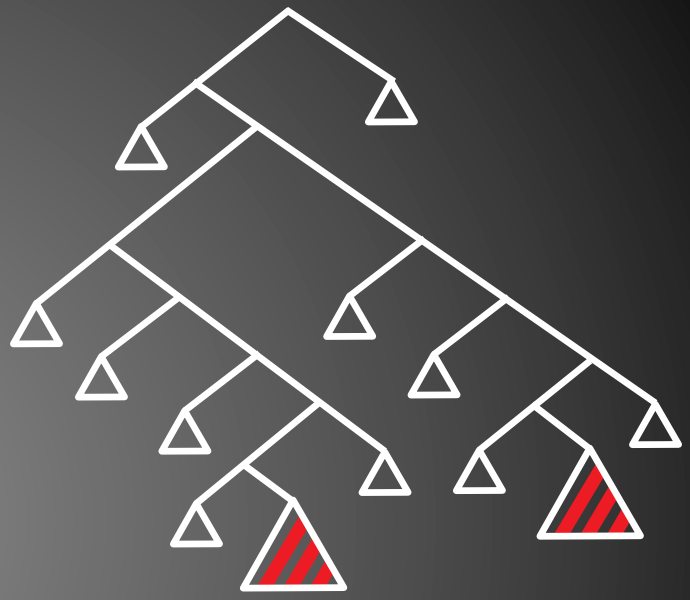
- Optimize over all traversal kernels
  - Left-first, right-first, front-to-back, back-to-front
- Representative ray set includes all shadow rays (best-case scenario)

Bedroom: SRDH reduces traversal steps by 56%

Bedroom
(93% occlusion)

SRDH groups occluders and places them high in the BVH

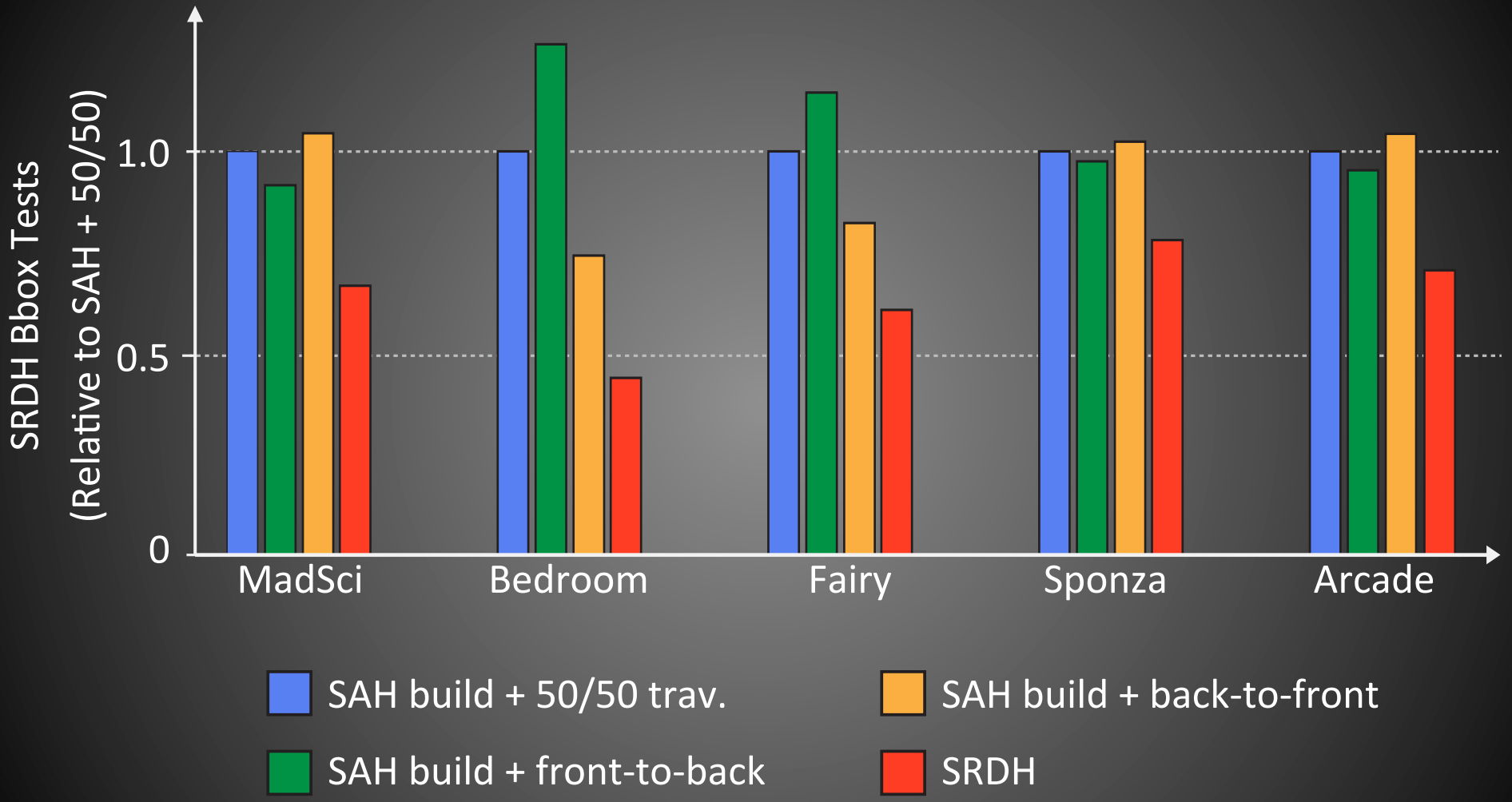




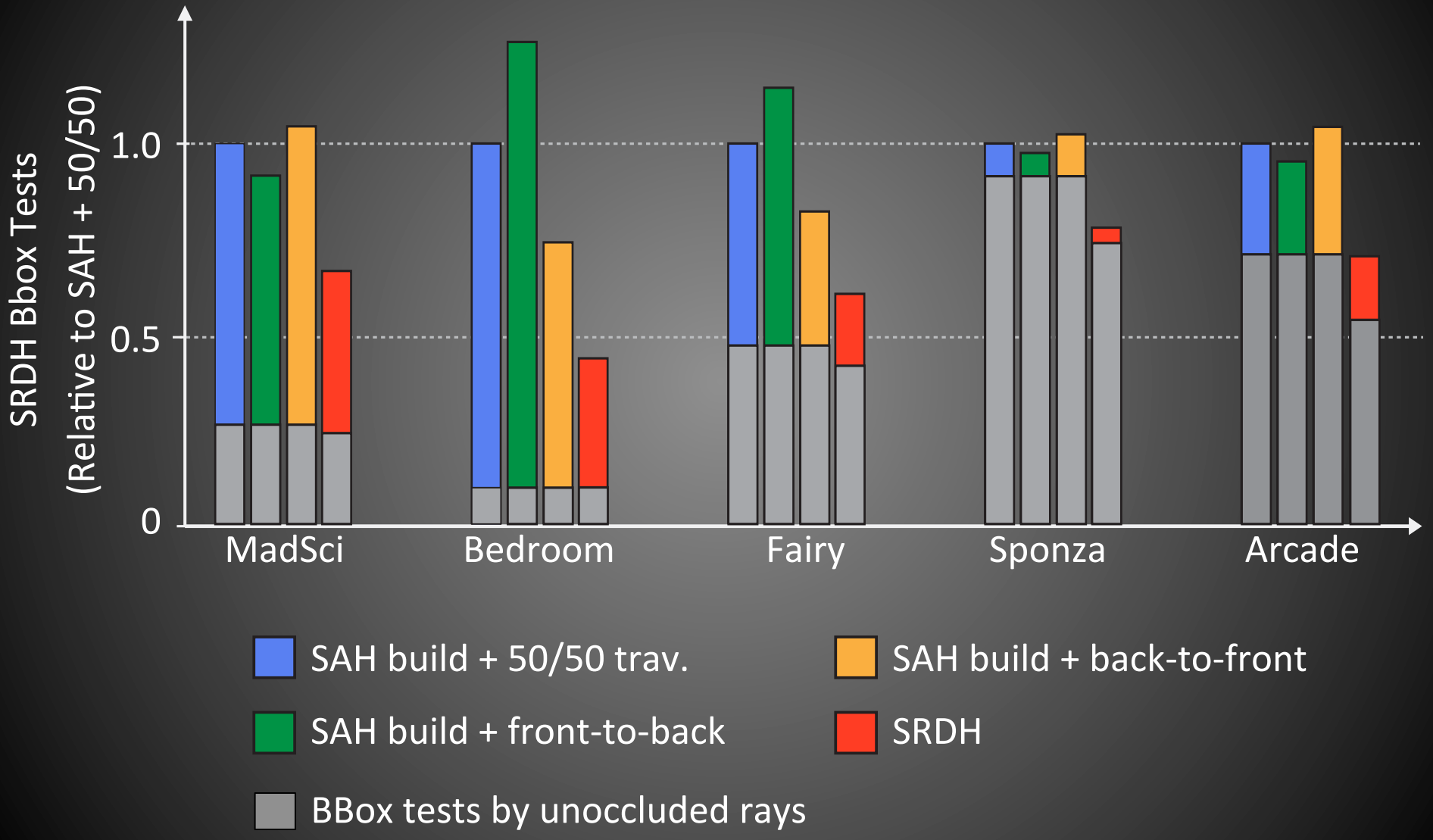
SAH-built BVH

SRDH-built BVH

SRDH reduces traversal steps 22% to 56%

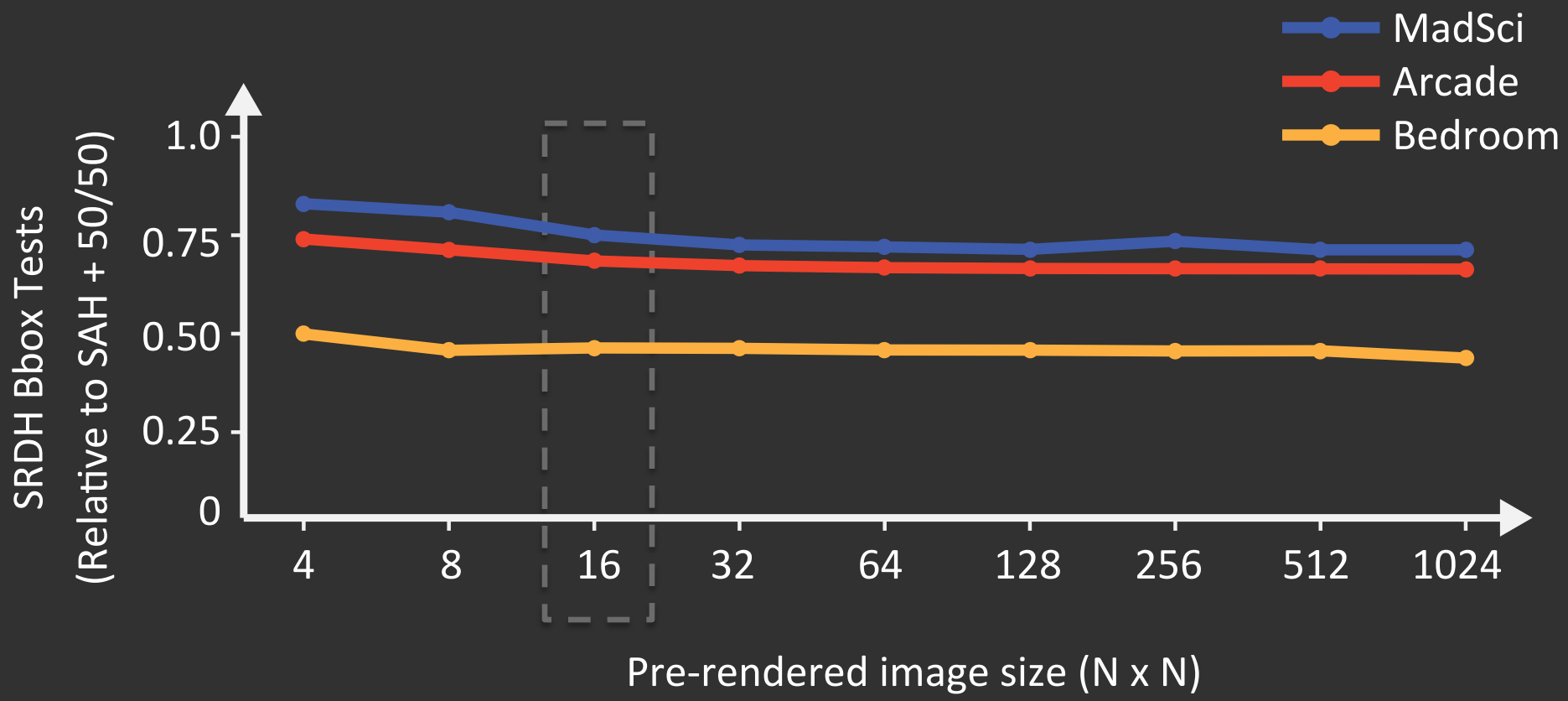


Smallest benefit in low occlusion conditions



How does the benefit of the SRDH decrease as less shadow ray information is known a priori? (Is a practical implementation possible?)

SRDH is robust to sparse representative ray sets



Representative ray set from 16x16 pre-rerendering always within 6% of optimal
(ray set is less than 1% size of all shadow rays)

Bottom Line

- SRDH increases preprocessing costs (it builds two BVHs)
  - SRDH tree build (with a 16x16 ray set) takes 1.6x time of simple SAH build
  - Overall preprocessing 2.6x that of SAH build
- SRDH reduces shadow ray traversal costs 22% to 56%
  - Benefit greatest in scenes with high occlusion
- We found that only employing the always-left and always-right kernels was sufficient for our kernel choices
  - No runtime overhead to implement traversal policy

SRDH Summary

- The SRDH simultaneously determines BVH structure and traversal order based on knowledge of shadow rays and scene geometry

- Enabled by new cost metric for a generalized model of traversal

Final Thoughts

- By dreaming up data structures with greater freedom, we can minimize over the freedoms during build, and create better algorithms

- Rendering algorithms can do better to specialize themselves to specific machines and workloads