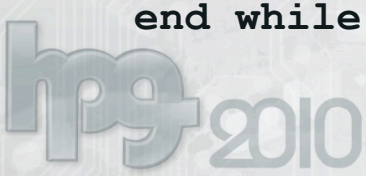
Background

- Ray casting in usual acceleration hierarchies is tree traversal
  - kd-trees
  - Bounding volume hierarchies
- Typically, traversal requires a stack
- Traversal stack is expensive
  - Storage
  - Bandwidth

Closer Look at Traversal

- Basic non-recursive tree traversal

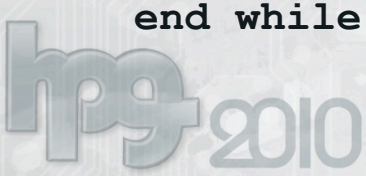
```
while (not terminated)
   if (not a leaf node)
      fetch bounds of children of current node
      check which children are intersected by the ray
      if (intersections)
         sort intersected children according to proximity
         make closest intersected child the current node
         push other children into stack
      else
         pop stack, terminate if empty
      end if
   else
      process primitives in leaf node
      pop stack, terminate if empty
   end if
end while
```



Closer Look at Traversal

- Basic non-recursive tree traversal

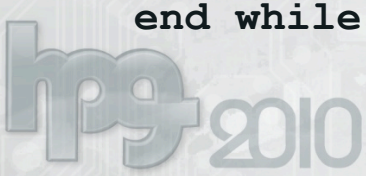
```
while (not terminated)
   if (not a leaf node)
      fetch bounds of children of current node
      check which children are intersected by the ray
      if (intersections)
         sort intersected children according to proximity
         make closest intersected child the current node
         push other children into stack
      else
         pop stack, terminate if empty
      end if
   else
      process primitives in leaf node
      pop stack, terminate if empty
   end if
end while
```



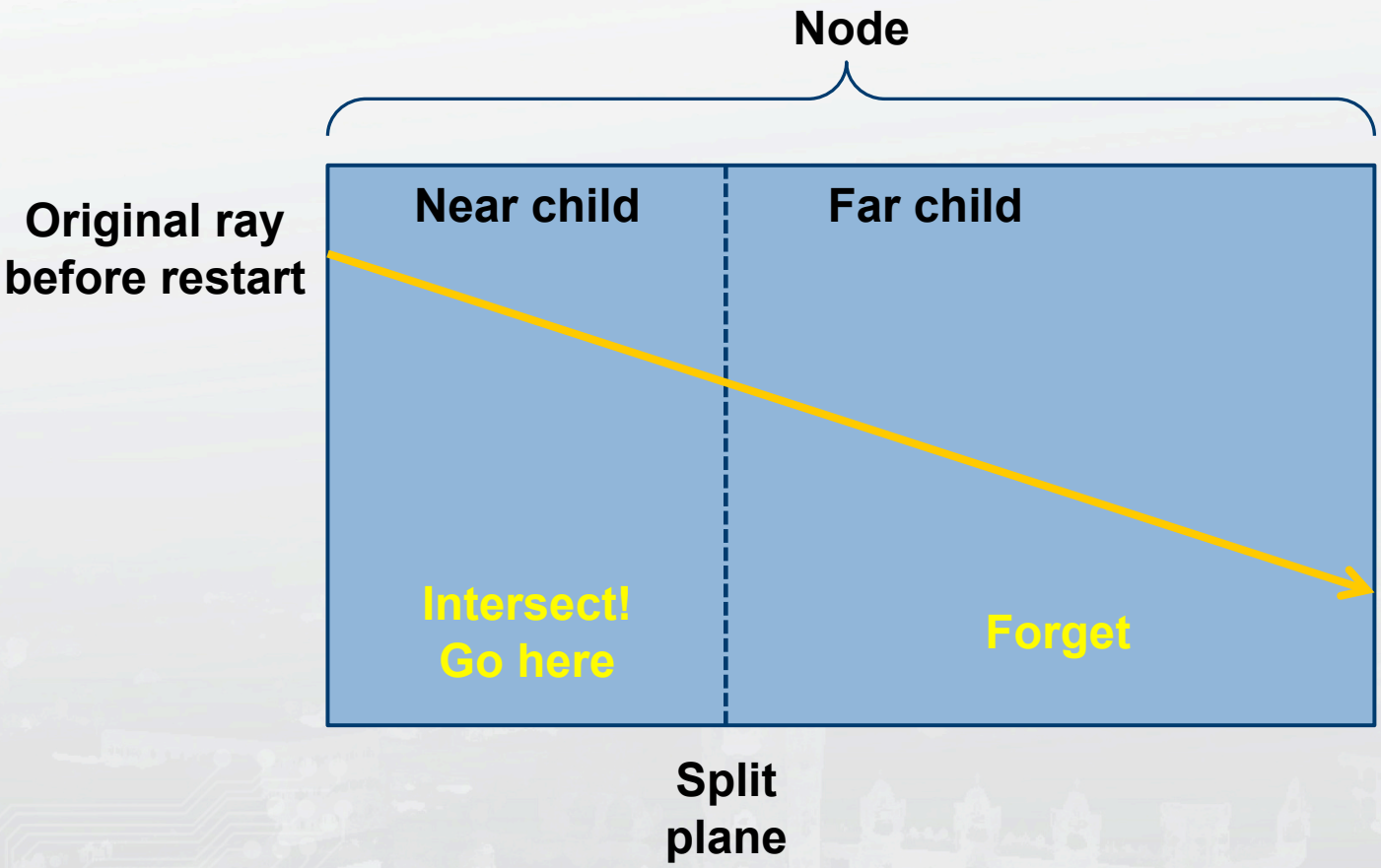
Stackless kd-Tree Traversal

- [Foley and Sugerman 2005]

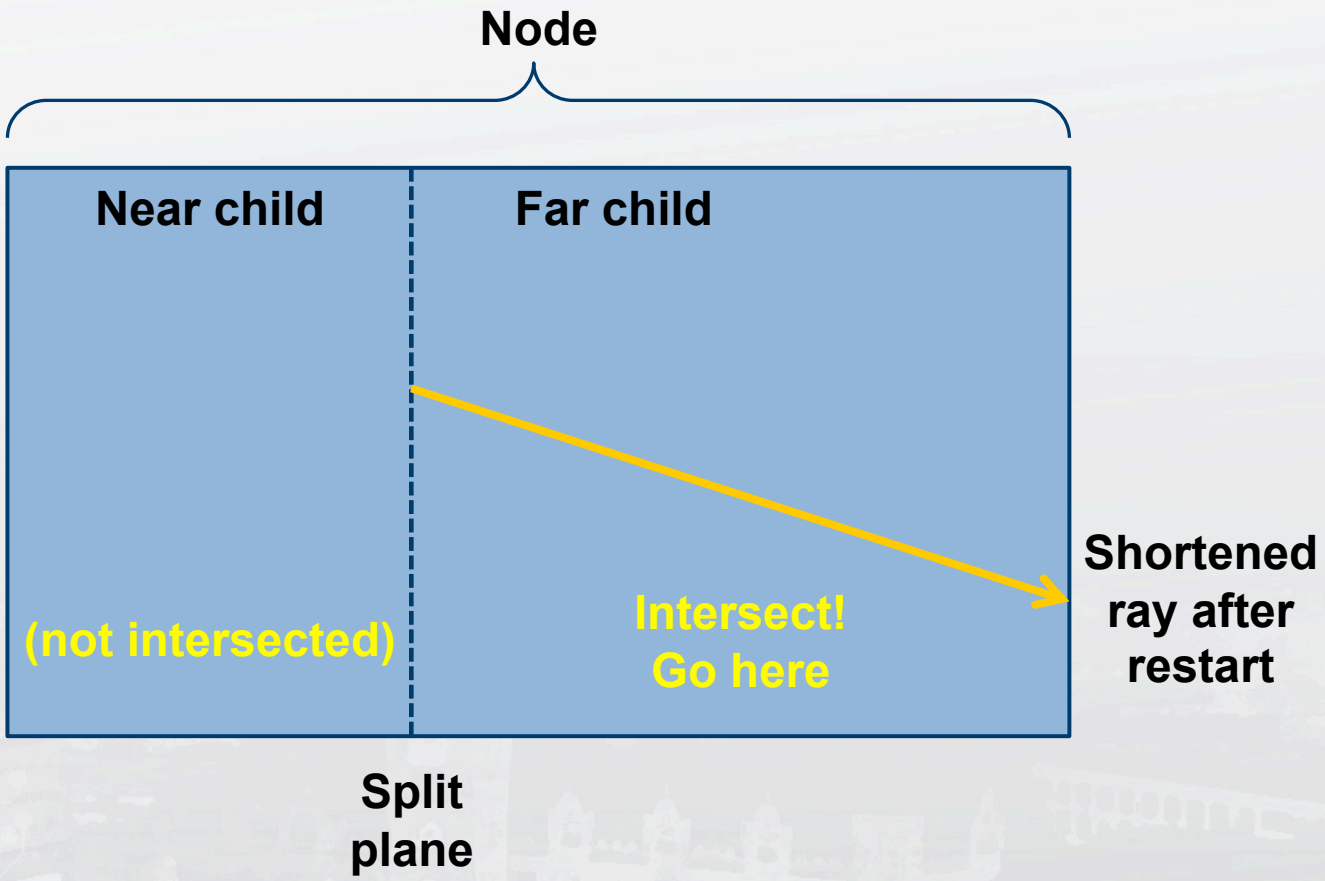
```
while (not terminated) (ray not shrunk to a point)
   if (not a leaf node)
      fetch bounds of children of current node
      check which children are intersected by the ray
      if (intersections)
         sort intersected children according to proximity
         make closest intersected child the current node
         push other children into stack   do nothing
      else
         pop stack, terminate if empty   shorten ray, restart
      end if
   else
      process primitives in leaf node
      pop stack, terminate if empty   shorten ray, restart
   end if
end while
```



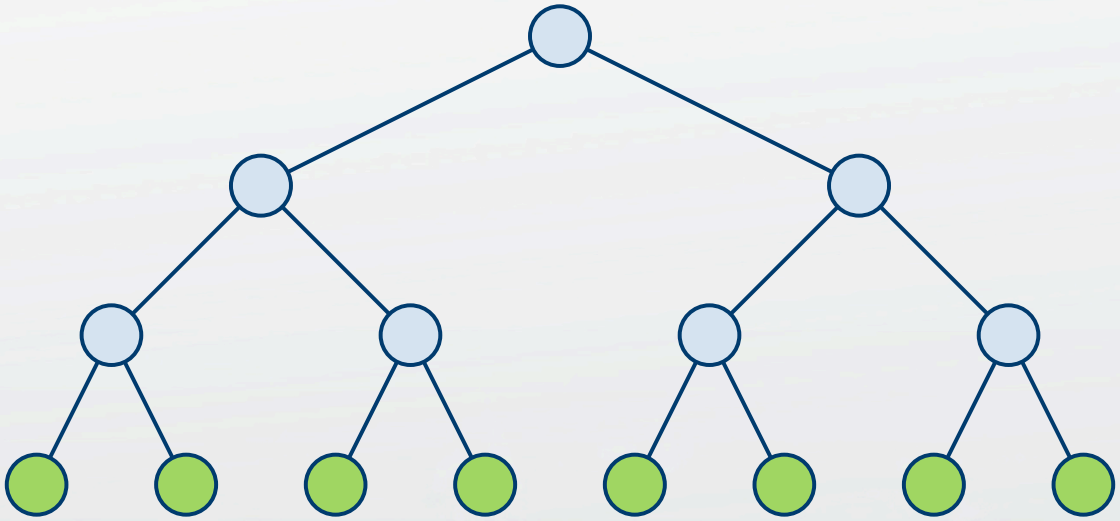
Why Does this Work?



Why Does this Work?

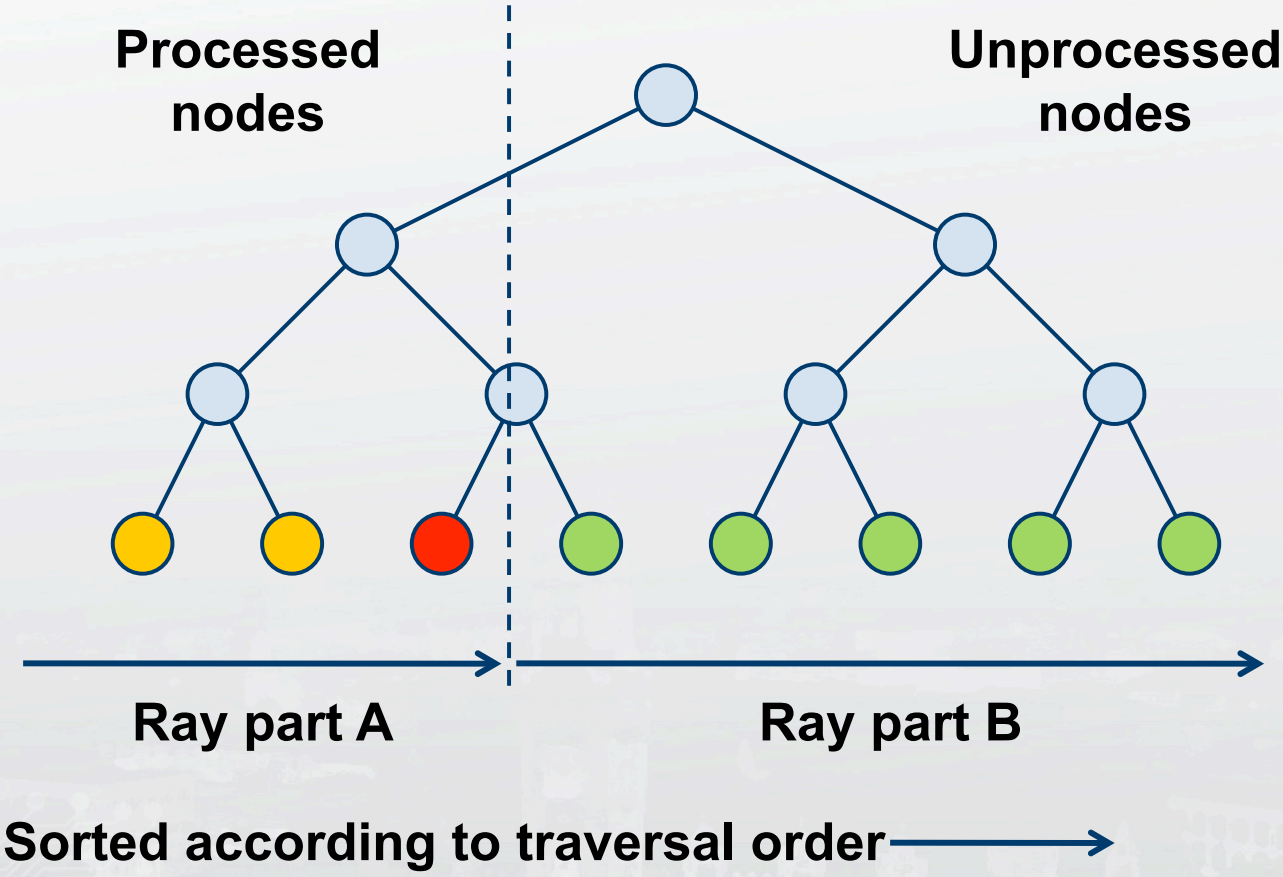


kd-Trees and Other BSP Trees



Sorted according to traversal order →

kd-Trees and Other BSP Trees



Short-Stack kd-tree Traversal

- [Horn et al. 2007]
- Store only *n* topmost stack entries
- Shorten ray and restart when stack is exhausted
- With *n*=3, only 3% too many nodes processed
  - Stackless (*n*=0) roughly doubles the amount of work

What About BVH?

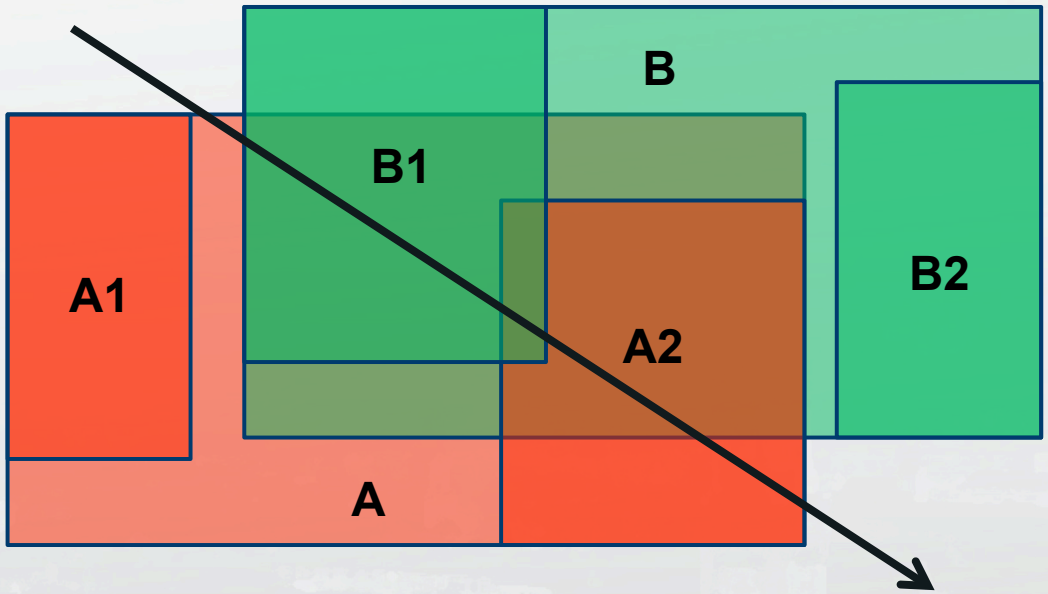
- In BVH, nodes may overlap

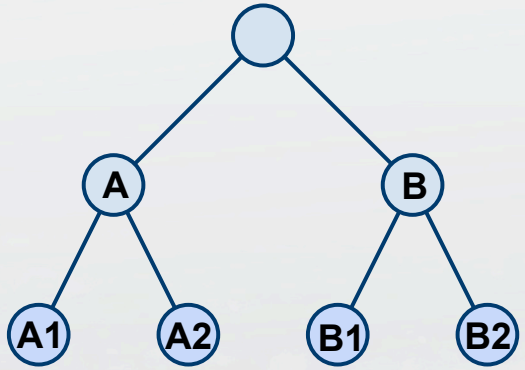
→ Unprocessed part of BVH tree does not correspond to a shortened ray

→ Restarting traversal by shortening the ray is not possible

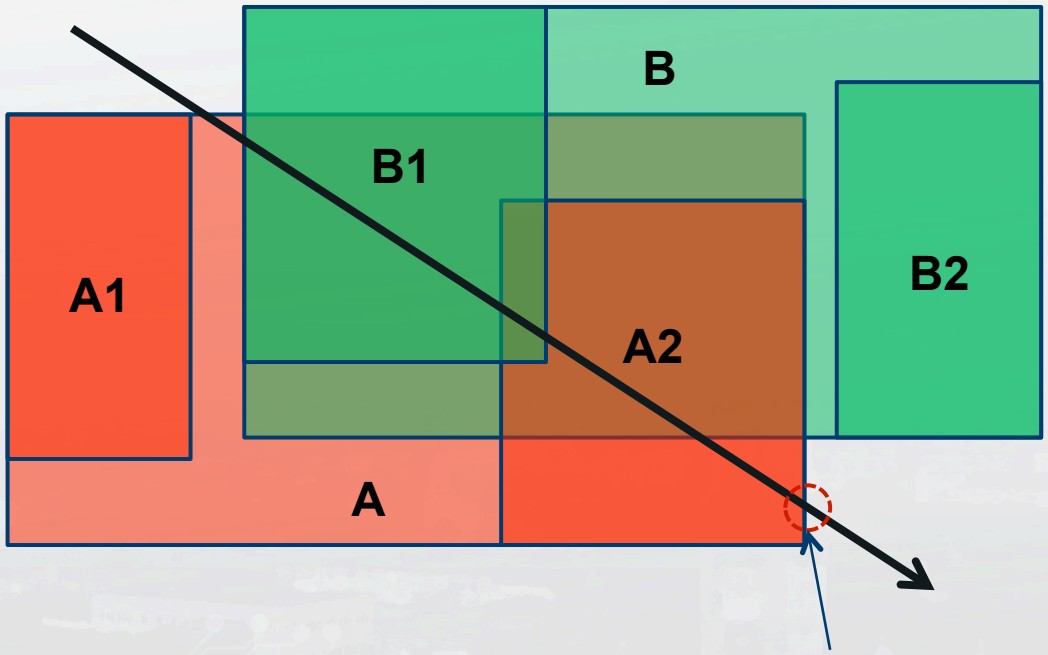
→ Stackless and short-stack traversal cannot be used

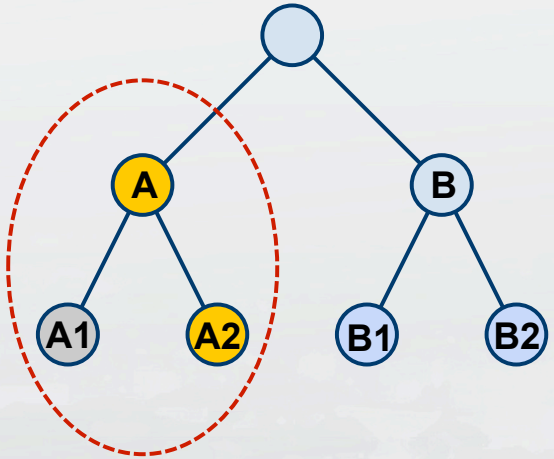
Example of Impossible Restart





Example of Impossible Restart

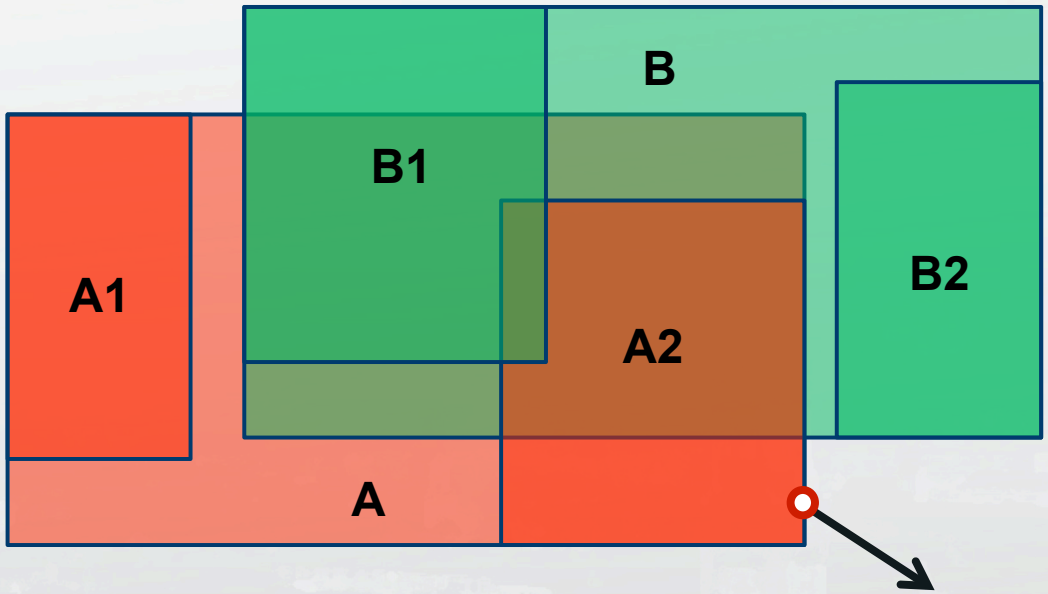




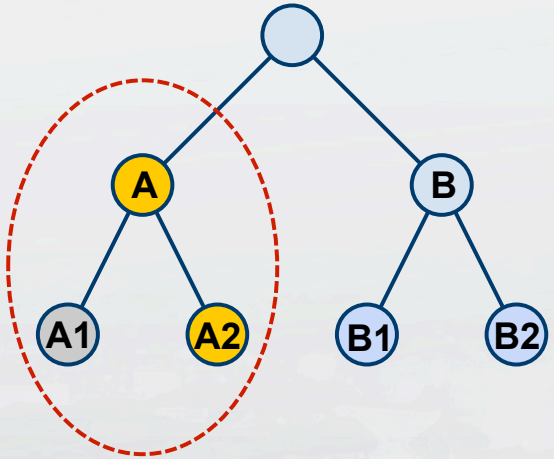
Processed nodes

Run out of stack here, need to restart after processing A2

Example of Impossible Restart

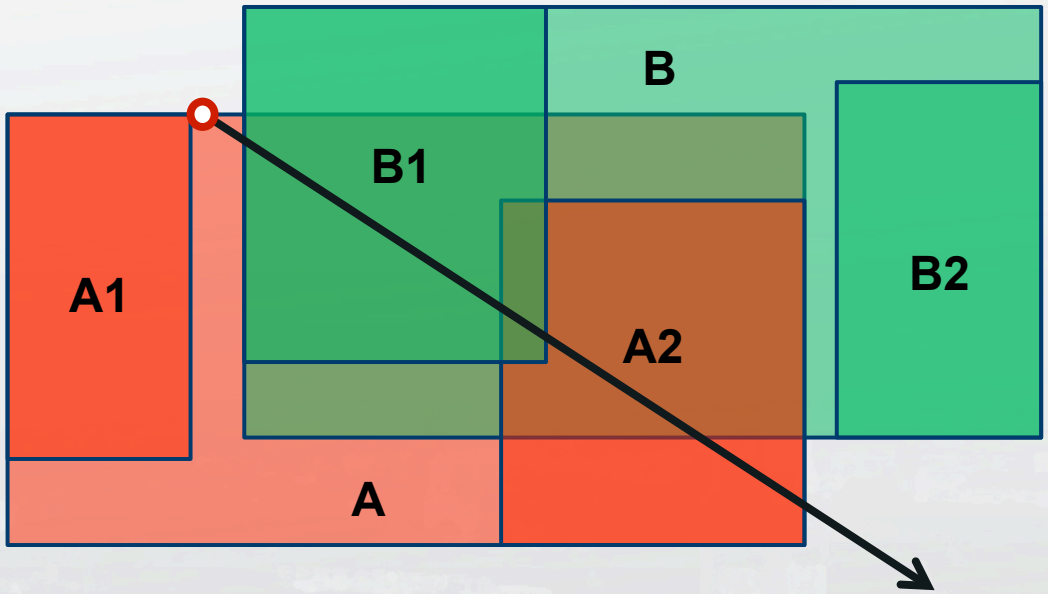


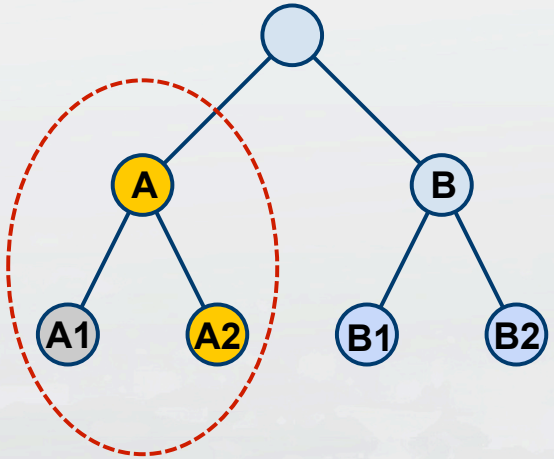
Does not work,
misses B entirely



Processed
nodes

Example of Impossible Restart

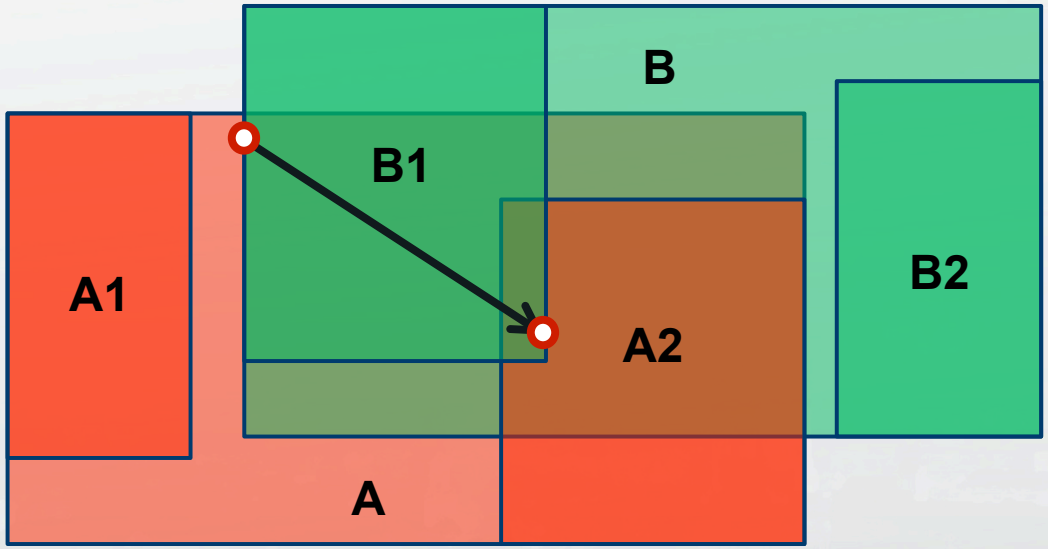




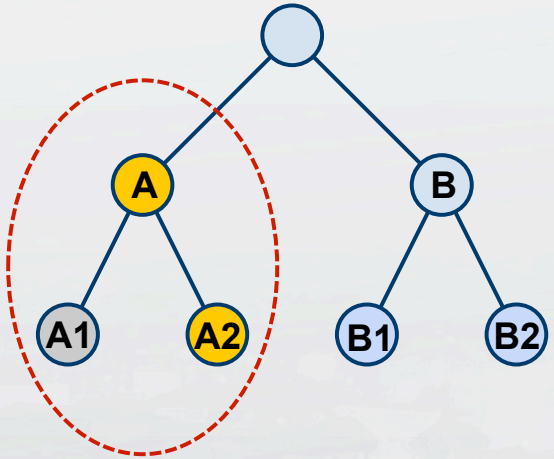
Processed nodes

Does not work, revisits A and A1

Example of Impossible Restart



Problem: Any ray that covers
B1 will revisit A and A2

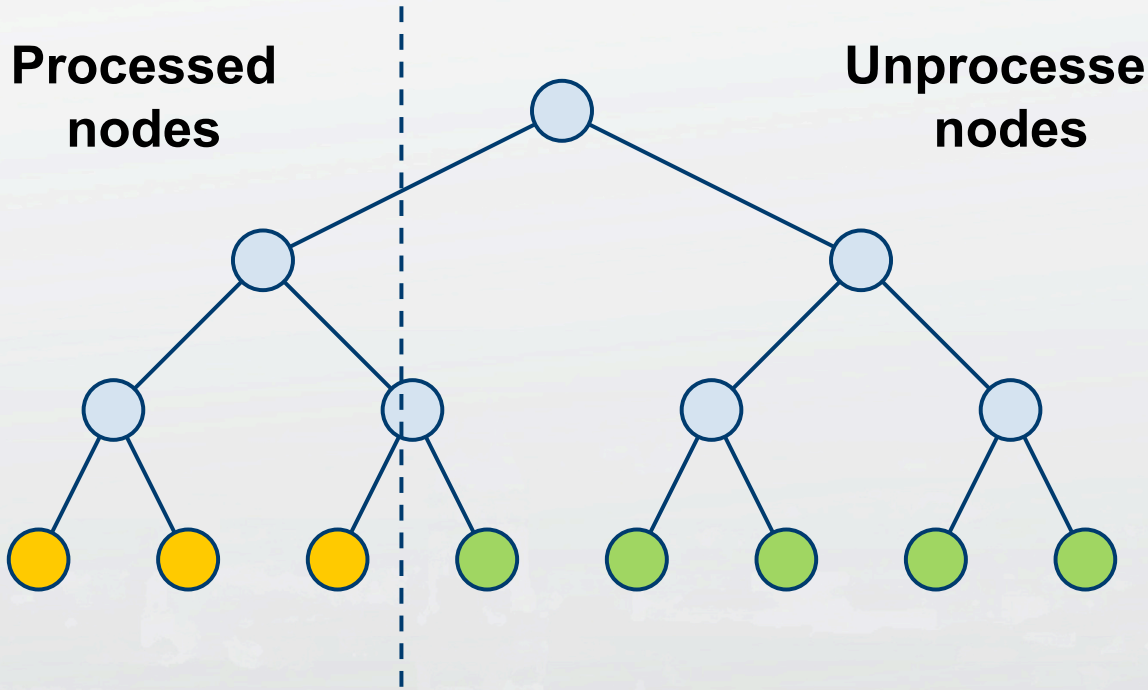


Processed
nodes

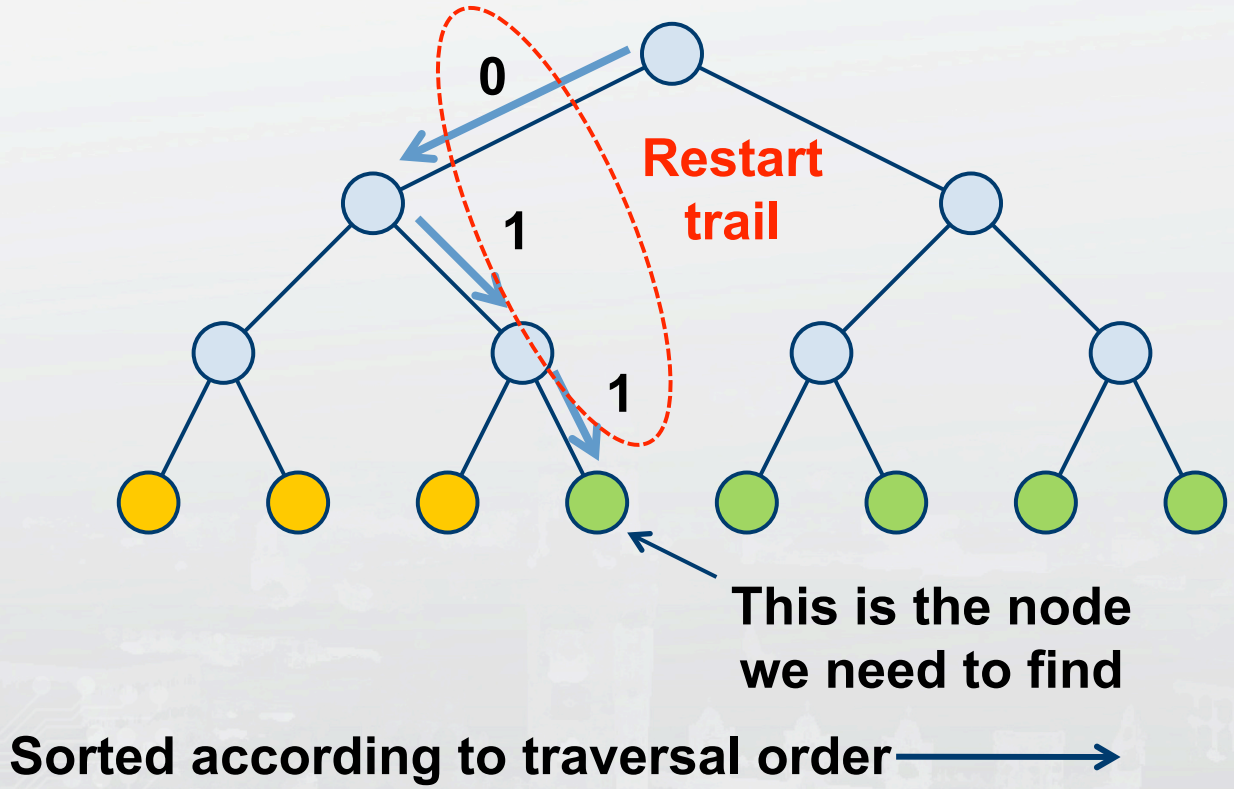
Solution: Restart Trail

- If we sort a BVH according to ray traversal order, we can partition it into processed and unprocessed parts
  - Unlike in kd-trees, these parts do not correspond to ray segments, but there's no need to worry about that
- Therefore, we can cheaply store which part of the hierarchy has been processed
- Upon restart, consult the information to steer the traversal to the correct part of the tree

Processed vs Unprocessed Nodes



Finding the Next Unprocessed Node



Restart Trail Encoding, Issues

- One bit per level is enough
  - This leaves many options for the actual encoding
- Must remain consistent during traversal
  - What if only one child node is intersected?
  - What if the ray is shortened from the end during traversal?
- Needs to be updated efficiently
  - After processing a node, we want to update the trail without remembering anything about our ancestors
- Should be transparent during traversal
  - No splitting into before-restart and after-restart codepaths

Our Encoding Scheme

- **0** = Node not visited yet OR node has two children to be traversed, subtree under near child not fully traversed

- **1** = Node has one child to be traversed OR node has two children to be traversed, subtree under near child has been fully traversed

- One sentinel bit on top of actual trail
  - Allows easy detection of when to terminate the traversal

Use of Trail During Traversal

- If node has two children
  - **0** = Go to near child
  - **1** = Go to far child
- When initialized to all zeros, does not affect usual traversal order
- Always set bit when going through one-child node
  - Does not affect traversal, but enables efficient updates!

Efficient Updates

- Other view of encoding:
  - **0** = I have a far child that is not processed yet
  - **1** = otherwise
- Trail update must make trail point to next unprocessed node
- Find last zero bit, toggle it to one, clear the rest
  - Sounds suspiciously like binary addition

Traversal Example

0

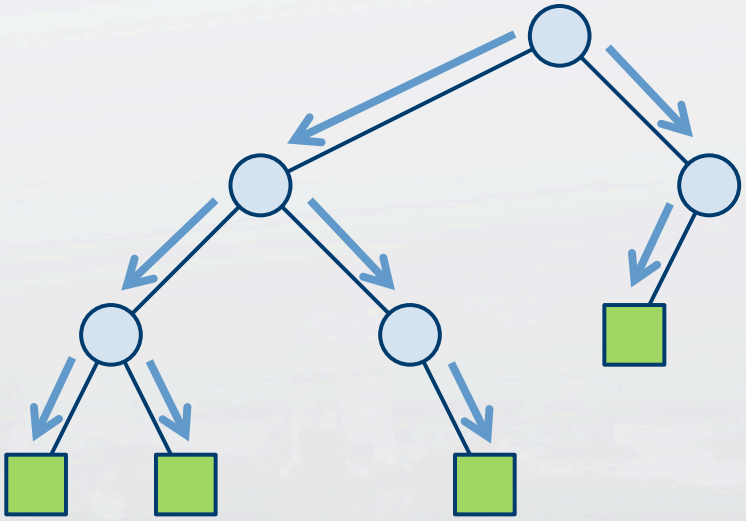
0

0

0

0

...



Traversal Example

0

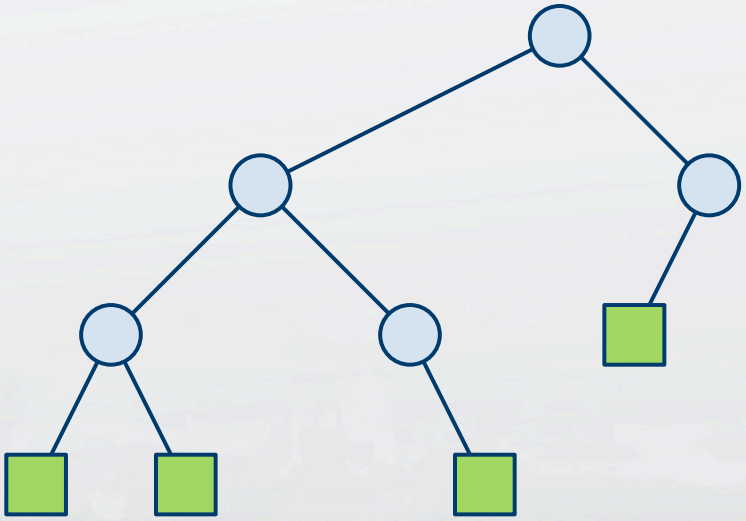
0

0

0

0

...



Traversal Example

0

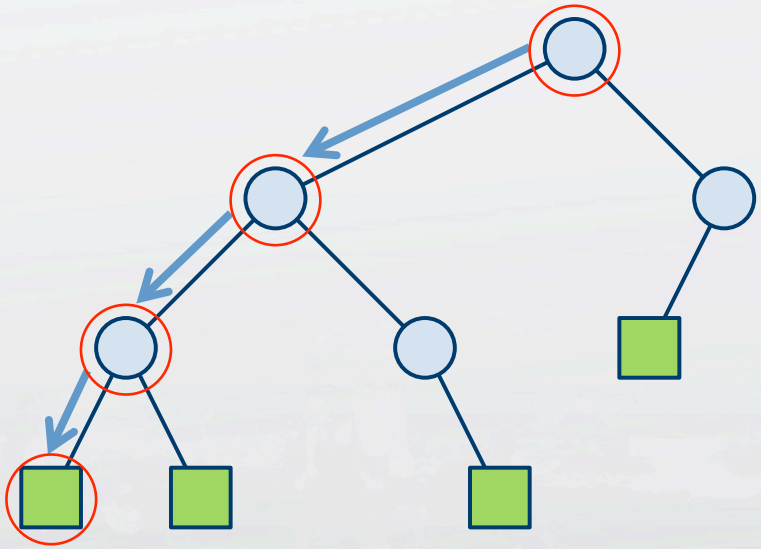
0

0

0

0

...



Traversal Example

0

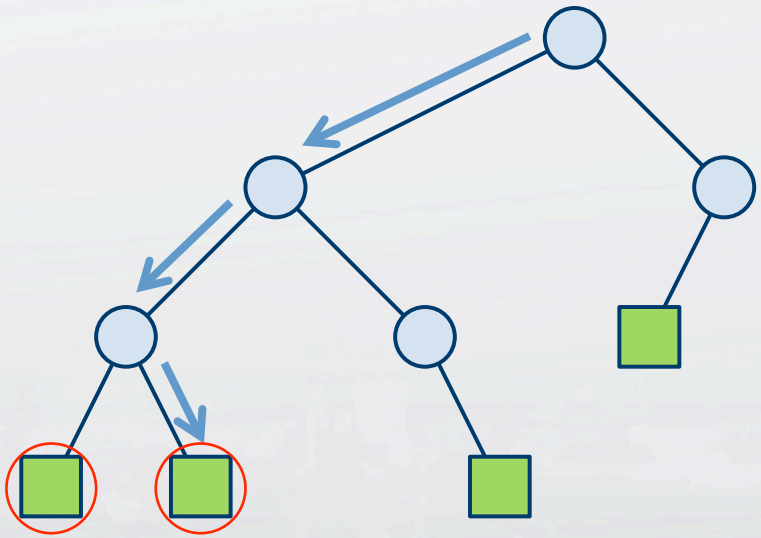
0

0

1

0

...



Traversal Example

0

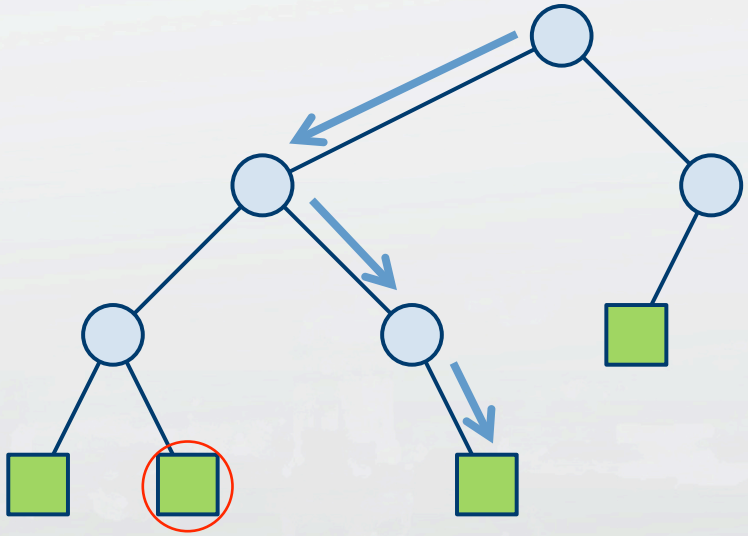
0

1

0

0

...



Traversal Example

0

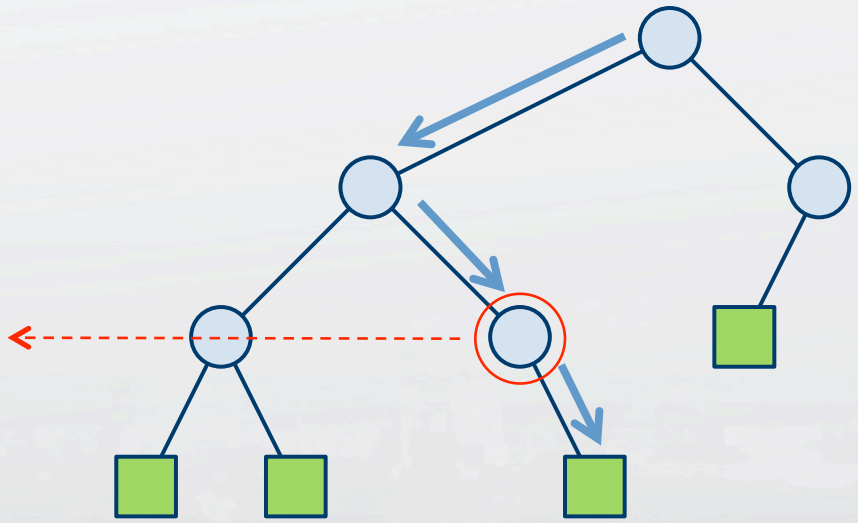
0

1

1

0

...



Traversal Example

0

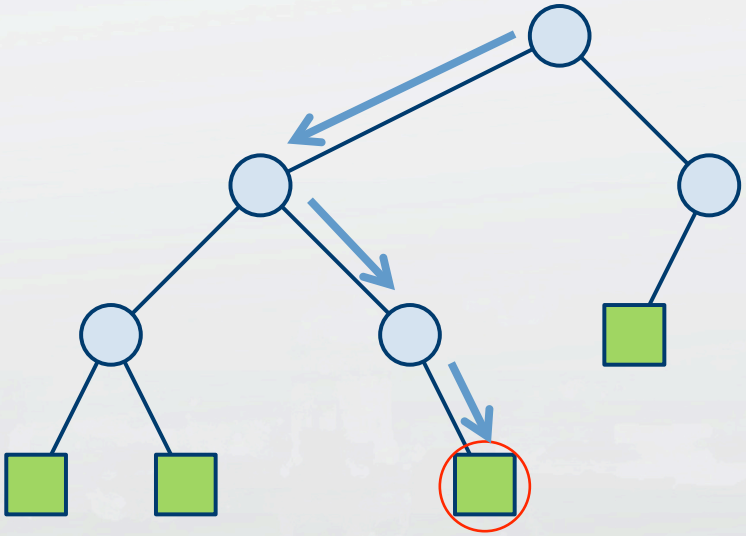
0

1

1

0

...



Traversal Example

0

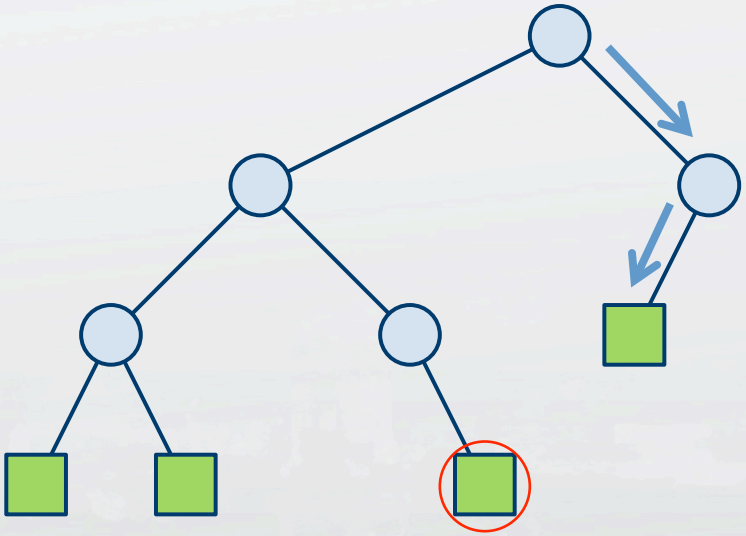
1

0

0

0

...



Traversal Example

0

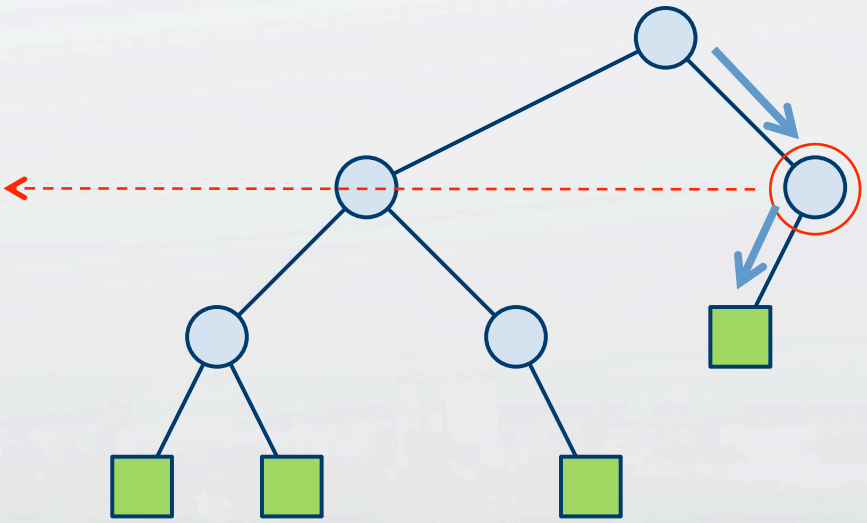
1

1

0

0

...



Traversal Example

0

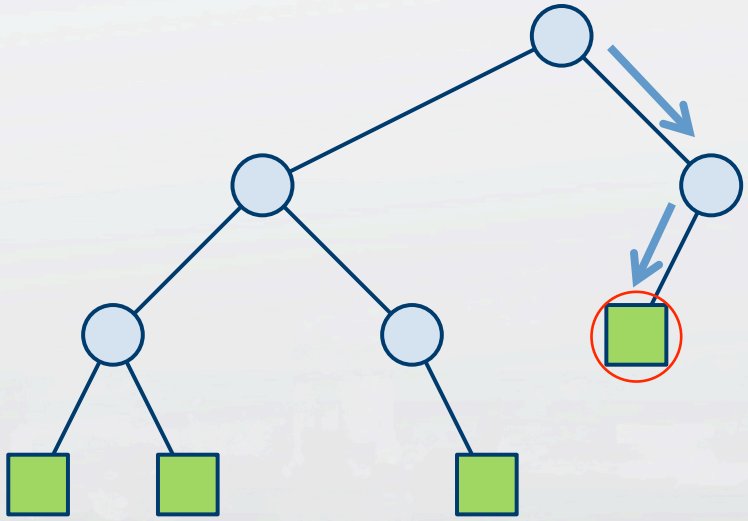
1

1

0

0

...



Traversal Example

1

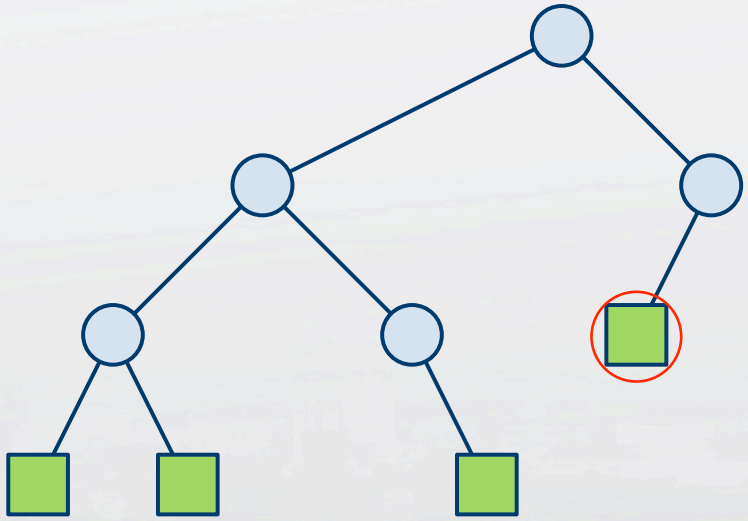
0

0

0

0

...



Traversal Example

1

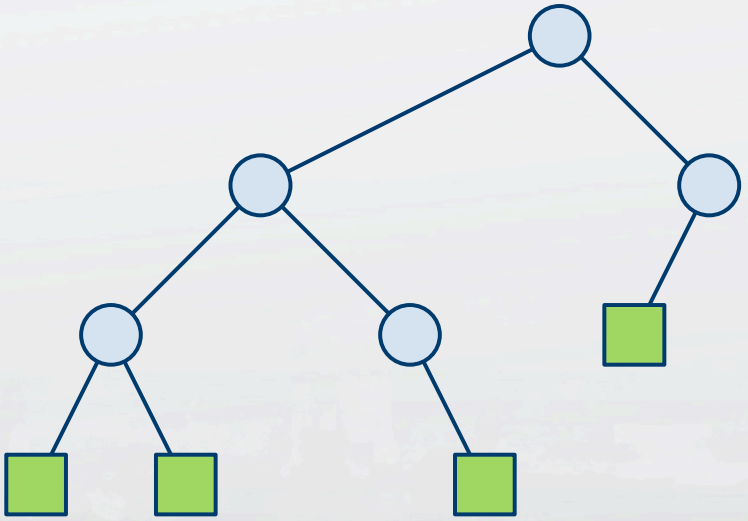
0

0

0

0

...



Gotcha: Rays Shortened at the End

- During traversal, primitive intersection may shorten the ray at the end
  - Makes it possible to terminate traversal before ray exits scene
- Can change two-child node into one-child node
  - Node has two children during descent, but only one during restart
  - If not handled properly, this can cause multiple traversal

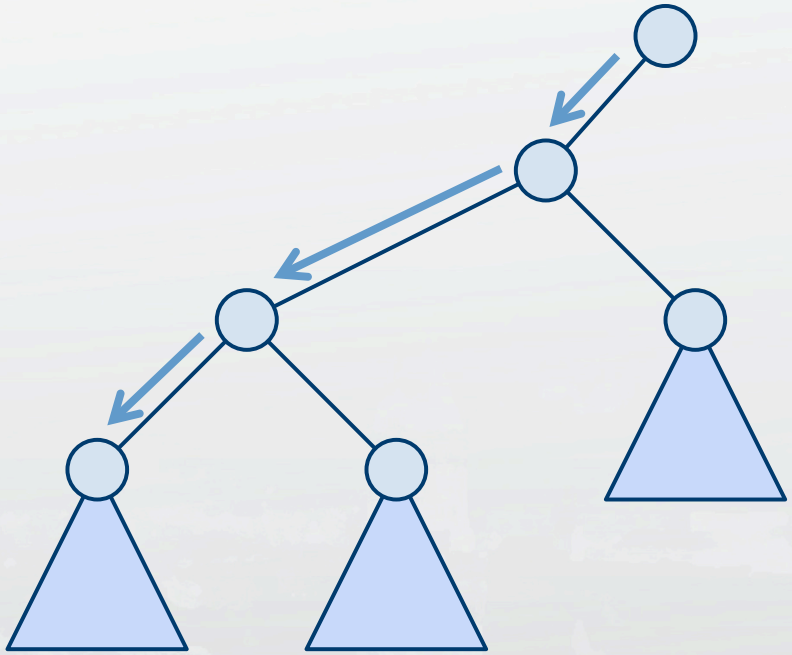
Trouble with Shortened Ray

...

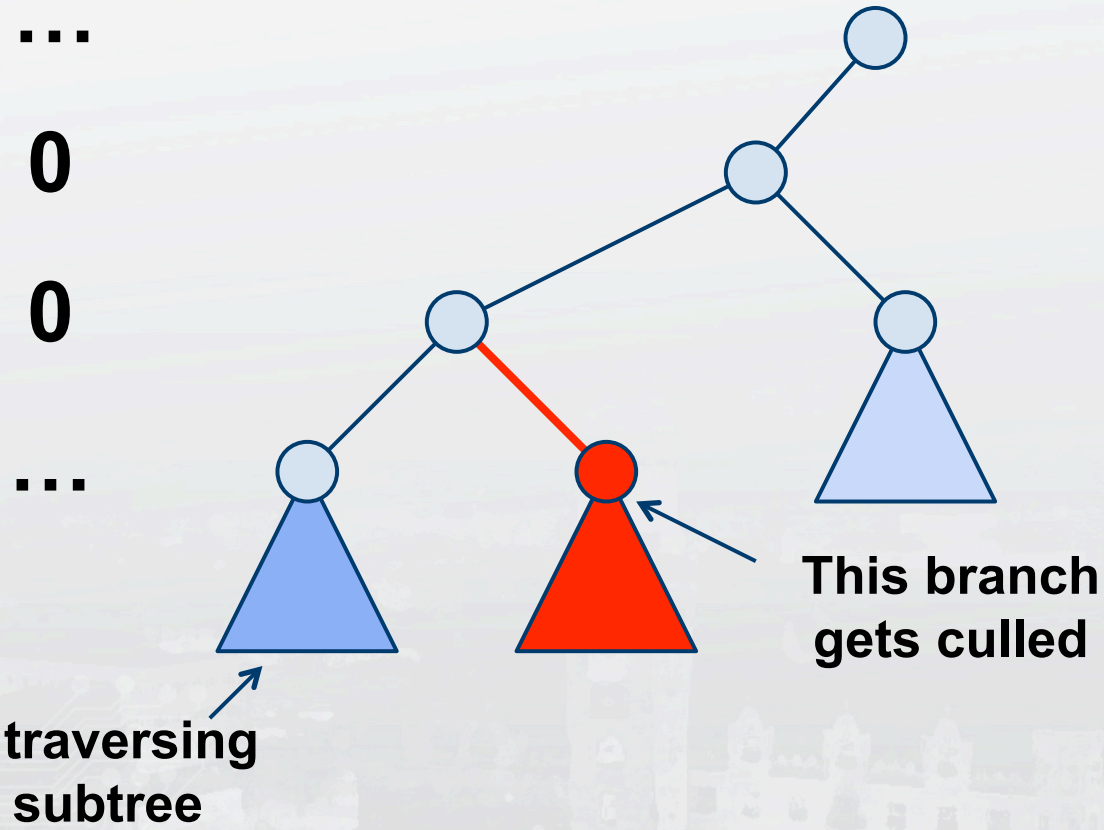
0

0

...



Trouble with Shortened Ray



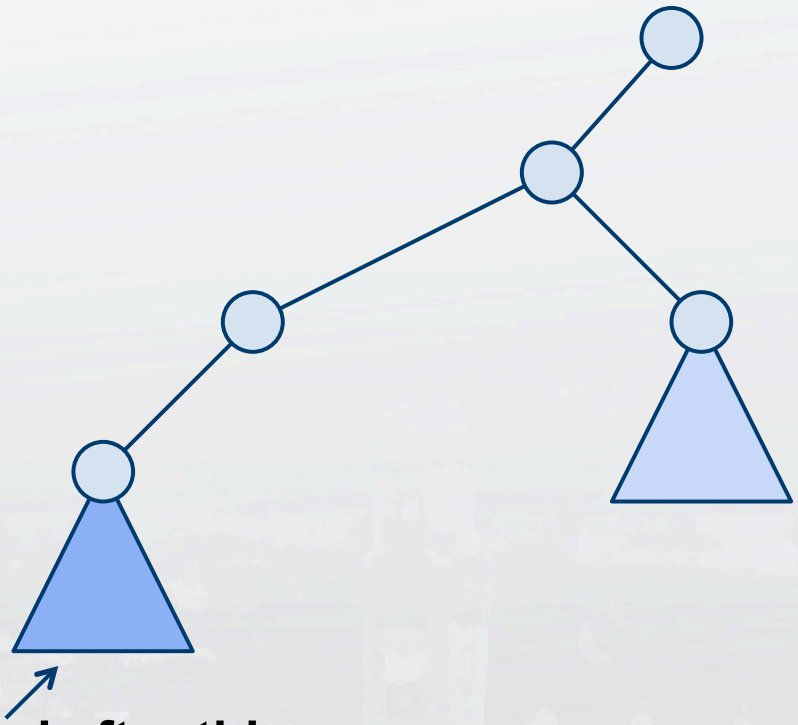
Trouble with Shortened Ray

…

0

1

…



Trail is updated after this subtree is processed

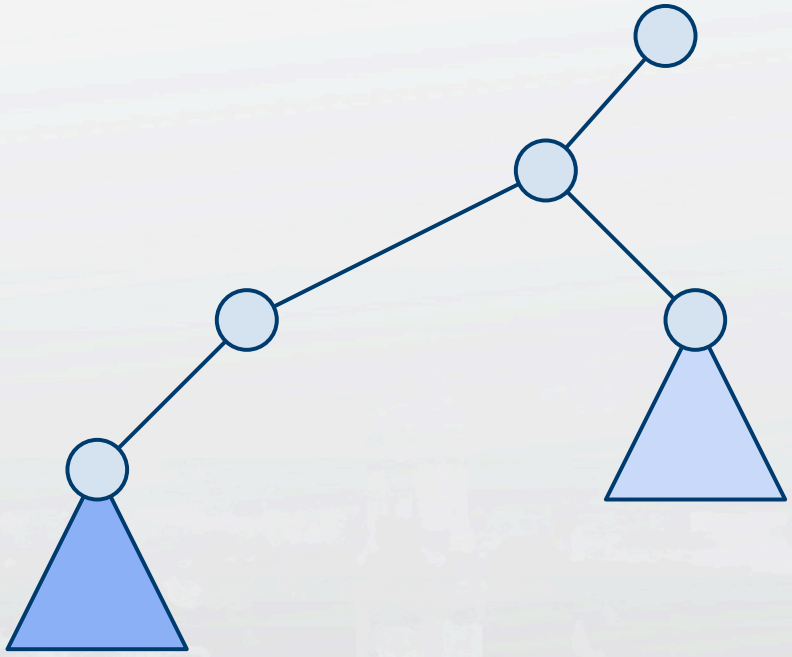
Trouble with Shortened Ray

...

0

1

...



Stack is exhausted, and a restart is initiated

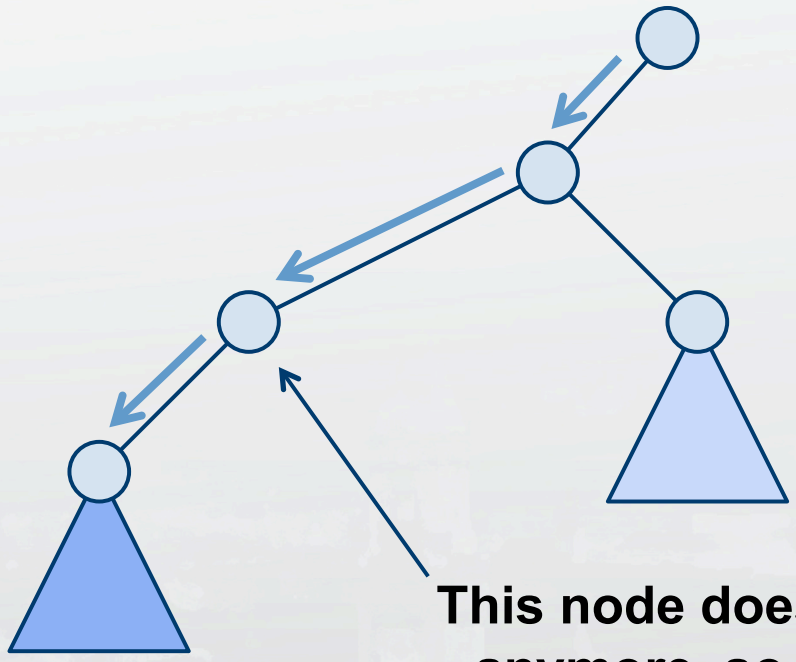
Trouble with Shortened Ray

...

0

1

...



This node does not have a far child anymore, so we cannot go there

The same subtree is processed again

Solving the Shortened Ray Problem

- Notice which bit was toggled to one when updating trail
- When traversing, detect if there is a one-child node on this level
- If so, the only explanation is that the far child was culled
- Proper action is to update trail and restart again

Practical Implementation

- Keep trail and current level bit in registers
  - Set and query bits trivially using Boolean operations
- Trail update
  - Find last zero above current level, toggle to one, clear the rest

```
TrailReg &= -LevelReg;
TrailReg += LevelReg;
```

- Determining level pointer after trail update
  - Find last one

```
temp = TrailReg >> 1;
LevelReg = ((temp – 1) ^ temp) + 1;
```

Results

Fairy Forest, 174K tris

Conference, 282K tris

- Overvisit factor of **2.2–2.4** for stackless traversal
  - Similar to results of Foley and Sugerman [2005]
- Three-entry short stack causes **5–8%** overvisit
  - Compared to 3% reported by Horn et al. [2007]
  - Possibly because two-child intersections are more frequent in BVHs than kd-trees, yielding more restarts

Results on Hardware

- Restart trails were originally implemented in BVH ray cast kernels
- Used to improve performance for a long while, but did not pay off in the most optimized variants
- Not tested on Fermi hardware yet
  - Cache trashing caused by stack traffic could make stackless (or register-only) traversal attractive again

Possible Use Cases

- Situations where
  - Not enough local storage for storing full traversal stack
  - Not enough memory bandwidth for stack traffic
- Custom traversal hardware?
  - Cleaner memory interface
  - Read-only memory client
- Pays off whenever cost of extra node traversal is less than cost of stack pushes and pops

Extensions, Future Work

- Storing two bits per level to avoid doing node intersection tests again
  - Avoiding intersection tests might not pay off on wide SIMD
- Extension to higher branching factors
- Could a similar approach be applied in other kinds of traversals?
  - e.g. closest-point and kNN searches for density estimation

Thank You

- Questions