EDAN30 Photorealistic Computer Graphics

Seminar 2, 2012

Bounding Volume Hierarchy

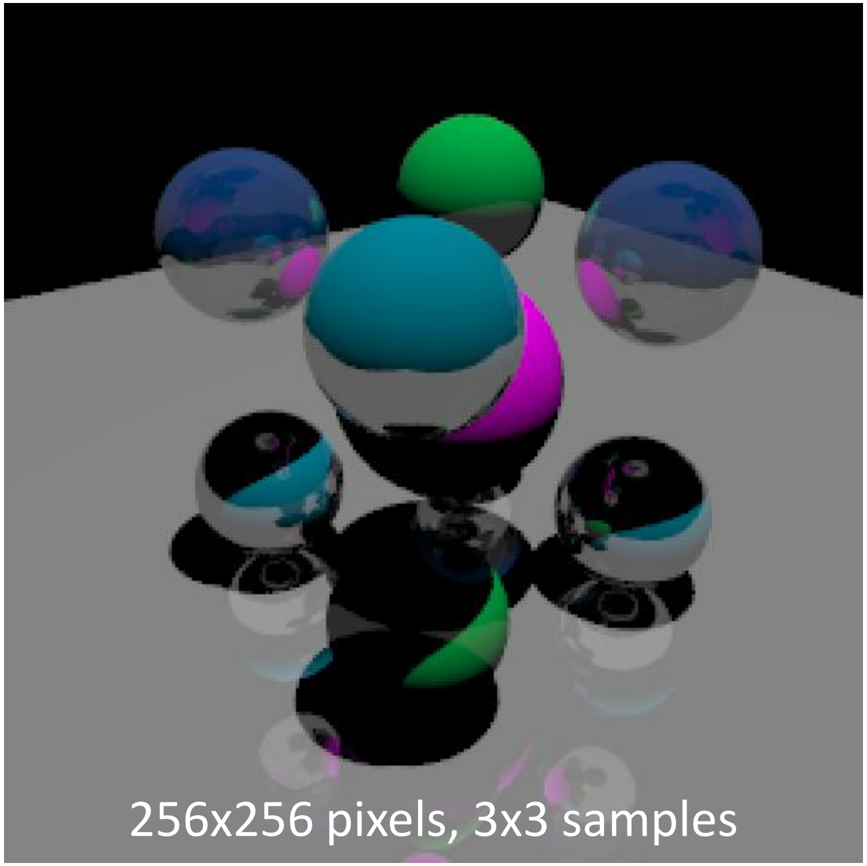
Magnus Andersson, PhD student (magnusa@cs.lth.se)

This seminar

- We want to go from hundreds of triangles to **thousands** (or **millions!**)
- This assignment has *few*, but *tricky*, tasks.

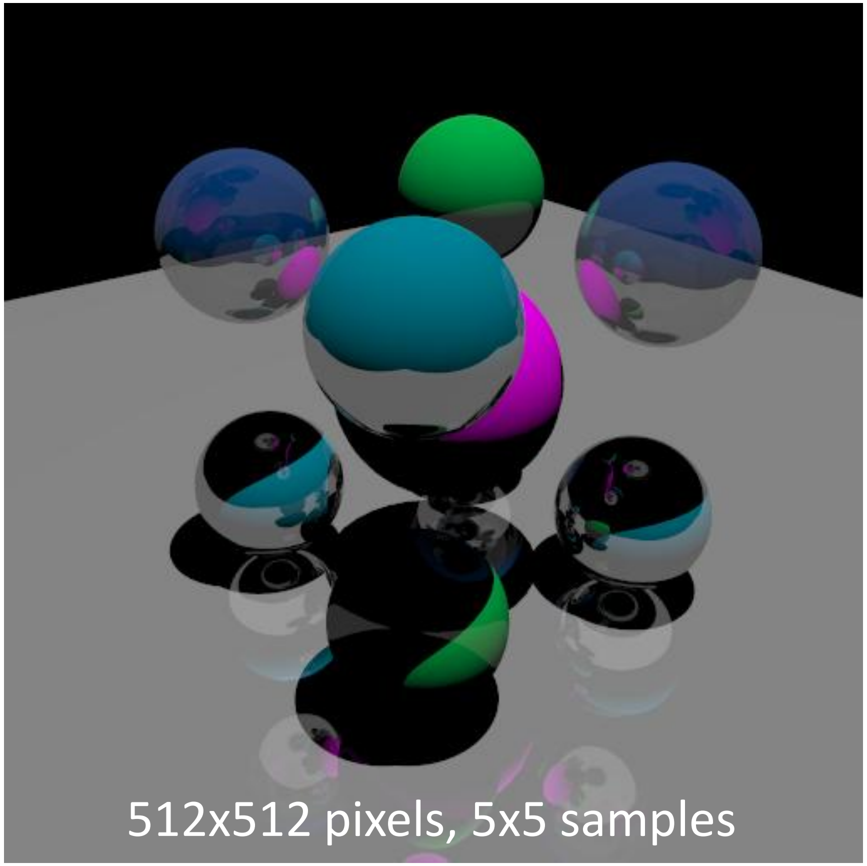
Results

List



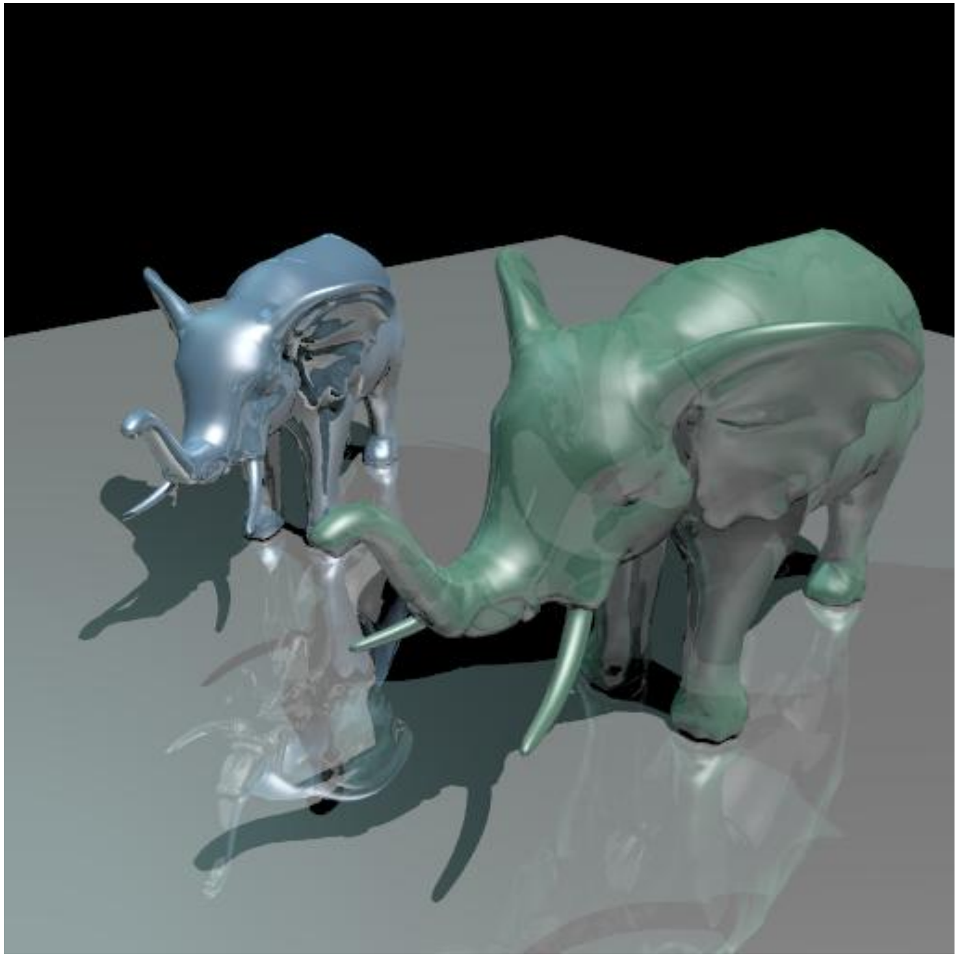
493 s

BVH



16.7 s

Elephants!



Overview

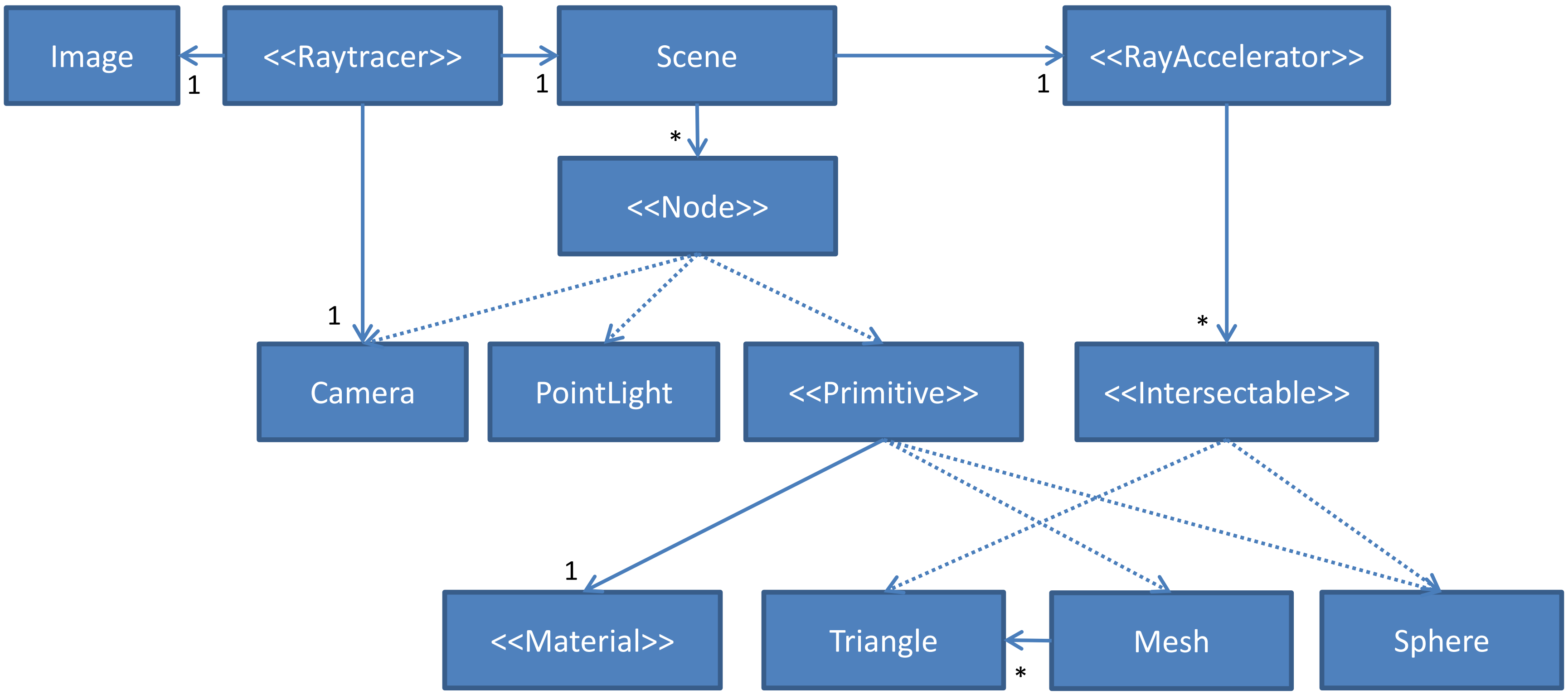


Image
<<Raytracer>>
Scene
<<RayAccelerator>>
1
1
1
*
<<Node>>
1
*
Camera
PointLight
<<Primitive>>
<<Intersectable>>
1
<<Material>>
Triangle
Mesh
Sphere
*

Overview

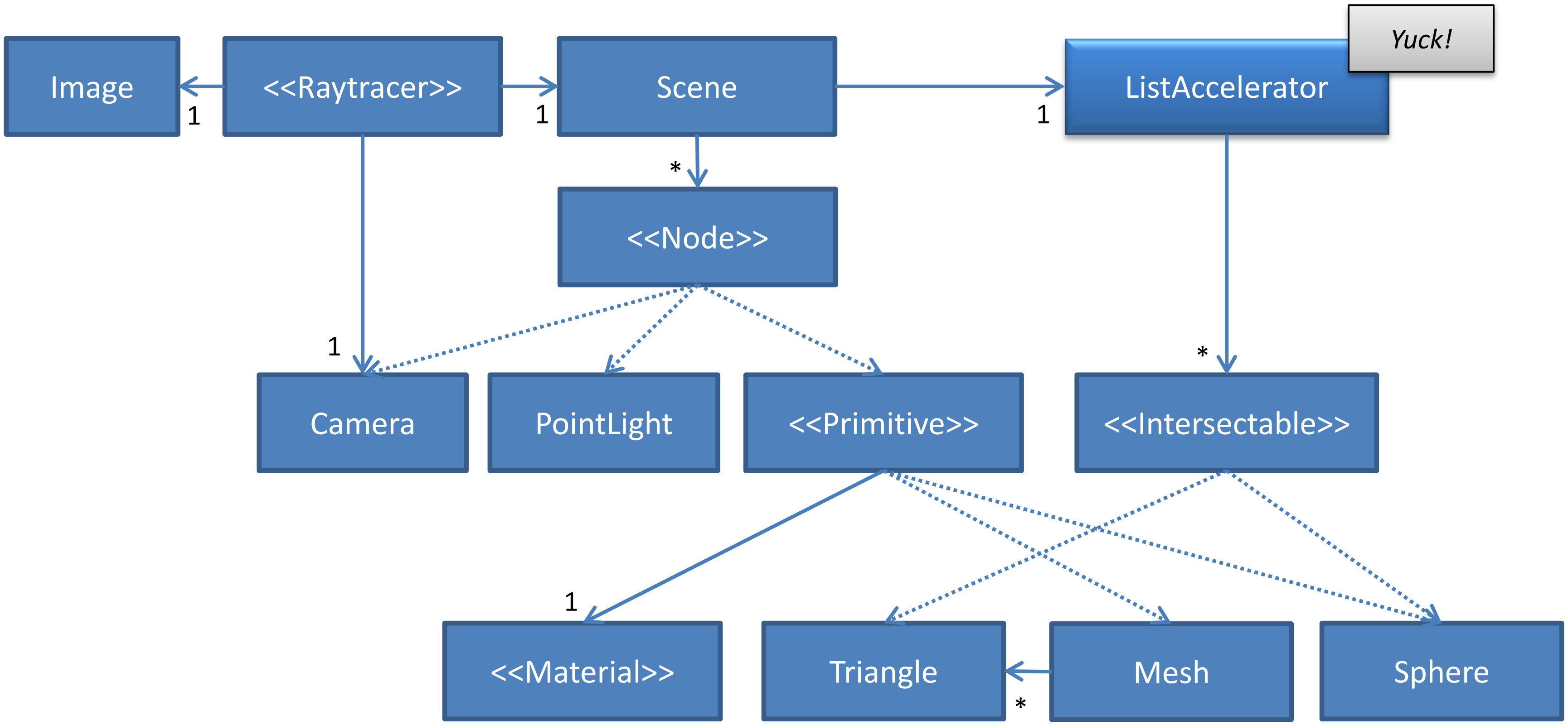


Image
<<Raytracer>>
Scene
ListAccelerator
Yuck!
1
1
1
*
<<Node>>
1
*
Camera
PointLight
<<Primitive>>
<<Intersectable>>
1
<<Material>>
Triangle
Mesh
Sphere
*

Overview

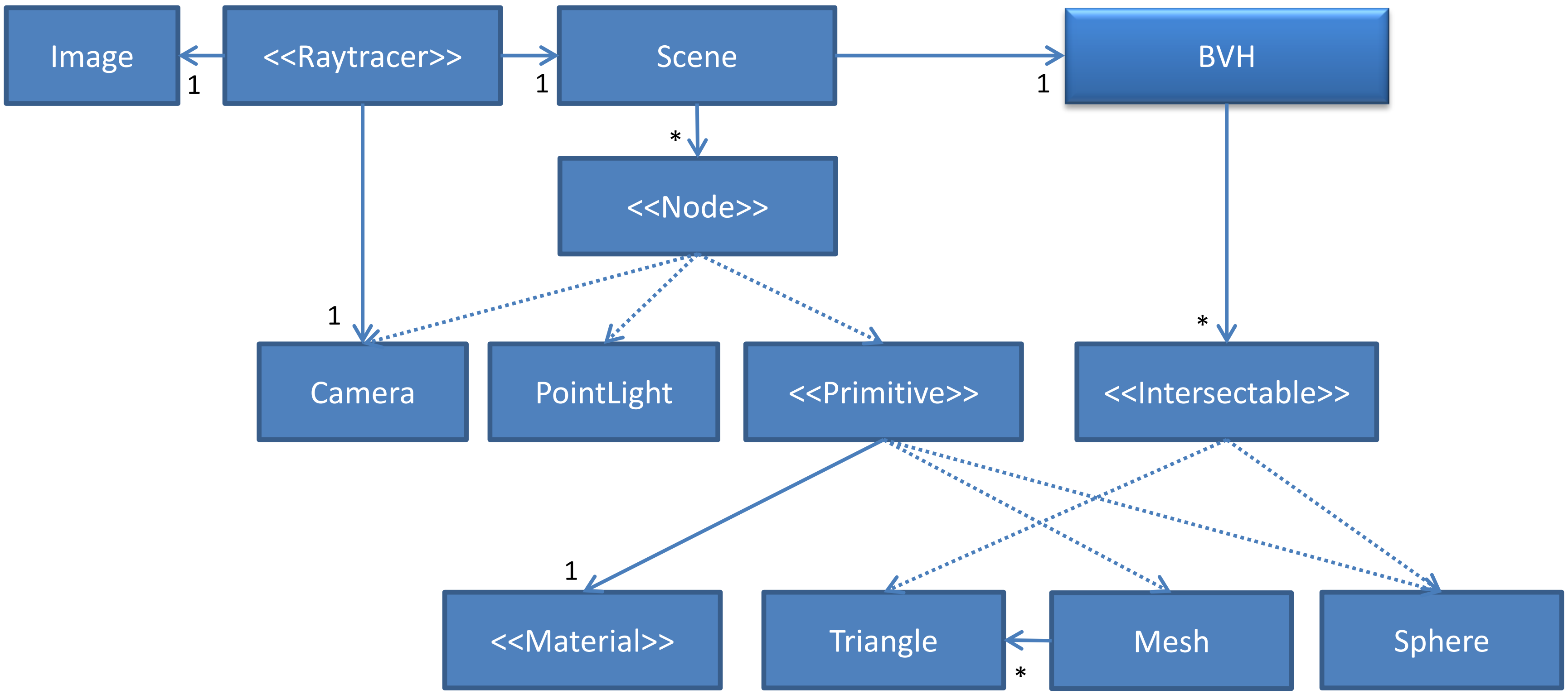


Image
<<Raytracer>>
Scene
BVH
1
1
1
*
<<Node>>
1
*
Camera
PointLight
<<Primitive>>
<<Intersectable>>
1
<<Material>>
Triangle
Mesh
Sphere
*

kD-tree vs. BVH vs. BIH vs. OcTree vs. Uniform Grid vs. …

- What acceleration structure should you choose??

kD-tree vs. BVH vs. BIH vs. OcTree vs. Uniform Grid vs. …

- What acceleration structure should you choose??
- Still an area of active research. It varies what’s in fashion…

kD-tree vs. BVH vs. BIH vs. OcTree vs. Uniform Grid vs. …

- What acceleration structure should you choose??
- Still an area of active research. It varies what's in fashion…
- Short answer = It depends on your application
  - Ray tracing (primary rays only?, GI?, …)
  - Collision detection?
  - Animated?
  - Memory/speed tradeoffs
  - Scene dependent
  - Implementation dependent
  - …

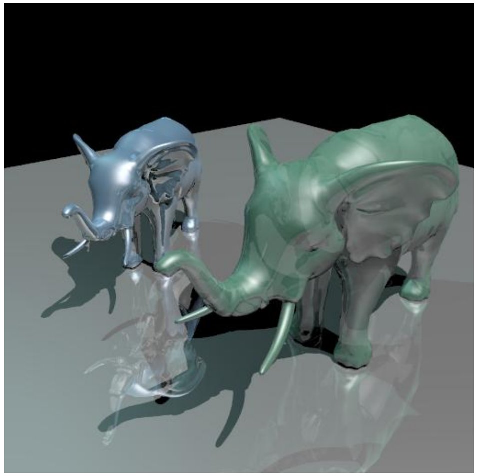
kD-tree vs. BVH vs. BIH vs. OcTree vs. Uniform Grid vs. …

- What acceleration structure should you choose??
- Still an area of active research. It varies what's in fashion…
- Short answer = It depends on your application
- *You'll find yourself playing around with different alternatives before settling on a suitable structure*

kD-tree vs. BVH

- kD-tree (50s)
  - *Implementation from assignments two years ago.*
- BVH (49s)
  - *This year's reference implementation.*

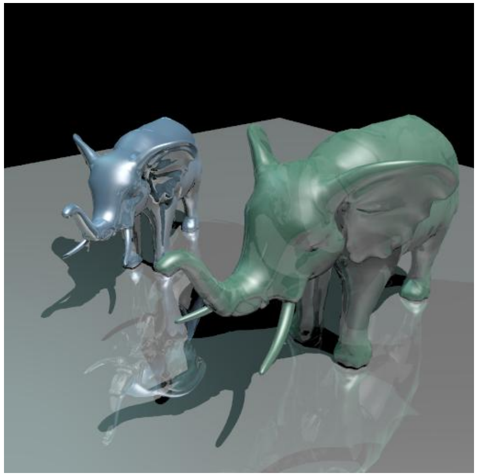
(Only tested on this Elephant scene)



kD-tree vs. BVH

- kD-tree (50s)
  - *Implementation from assignments two years ago.*
- BVH (49s)
  - *This year's reference implementation.*

+ The BVH will be useful for other purposes in a later lab



Assignment 2

- Construction
- Intersection
- Surface Area Heuristic (Optional)
- Further Optimizations (Optional)

Assignment 2

- Construction
- Intersection
- Surface Area Heuristic (Optional)
- Further Optimizations (Optional)

Construction

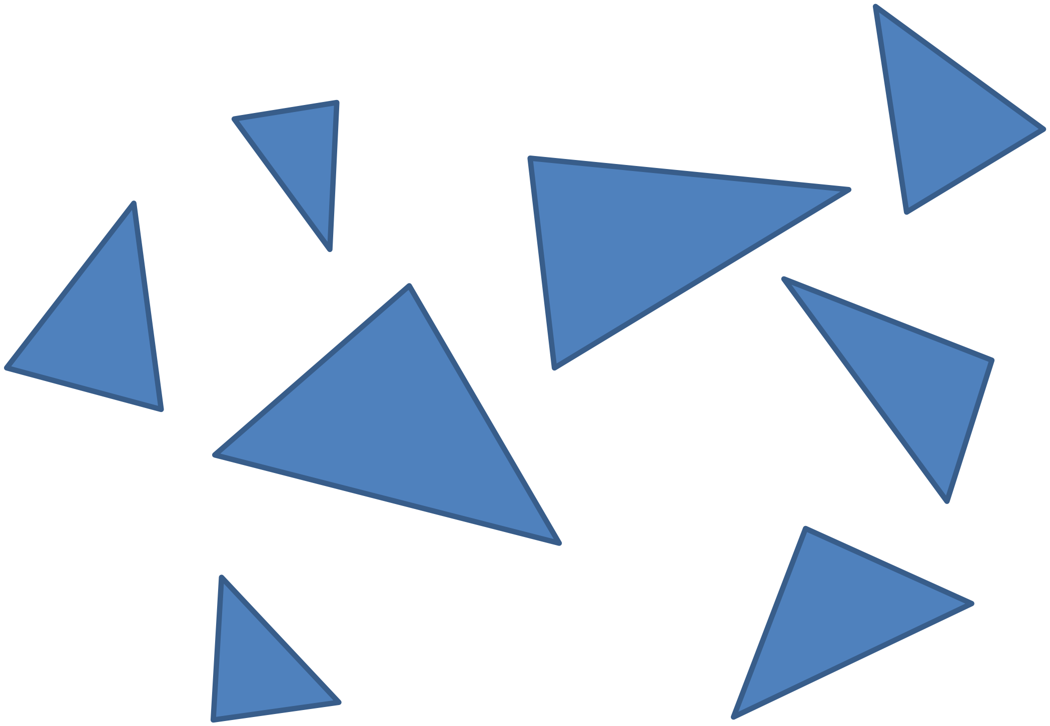
- You will need to create a new class *BVHAccelerator* which inherits from *<<RayAccelerator>>*
- For this assignment you must implement

```
void build(const std::vector<Intersectable *> &objects);
```

- …and you will probably need something like

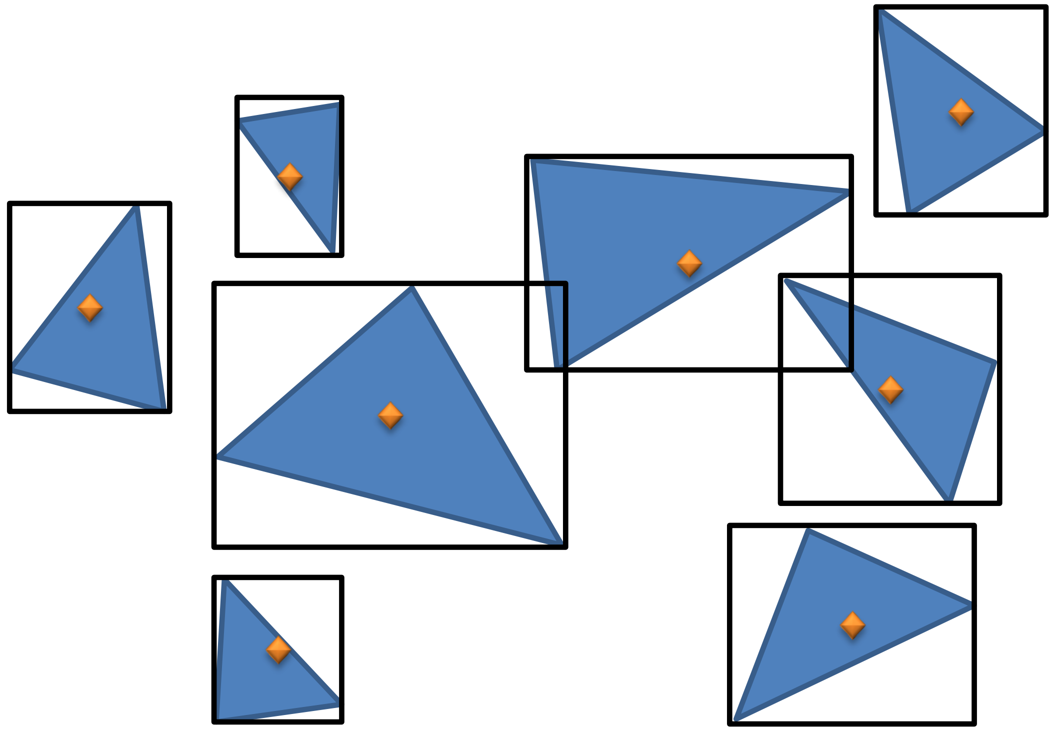
```
void build_recursive(int left_index, int right_index, AABB box,
        BVHNode *node, int depth);
```

Construction, brief overview



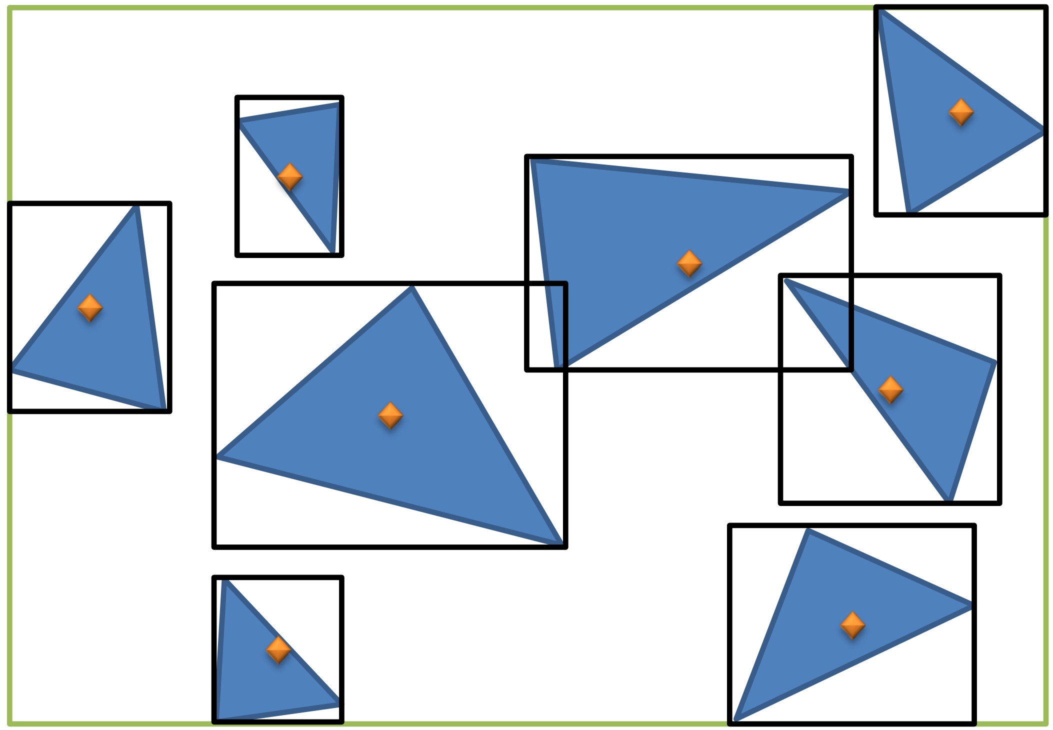
We begin with a bunch of Intersectables

Construction , brief overview



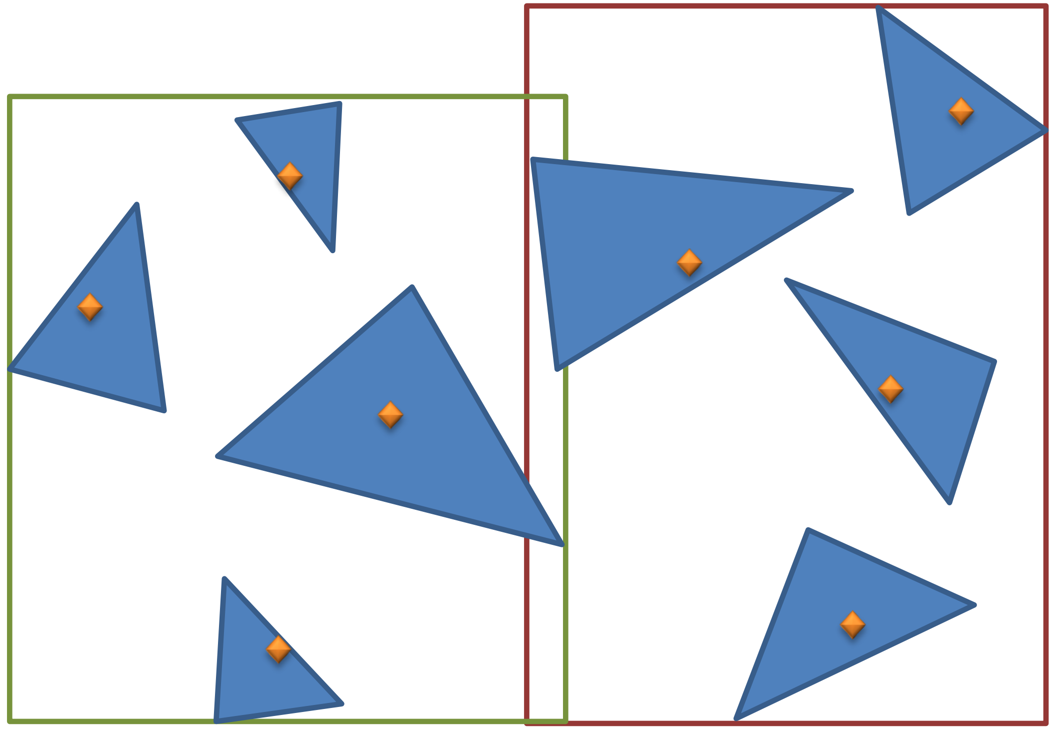
Find bounding box centroids of all intersectables

Construction , brief overview



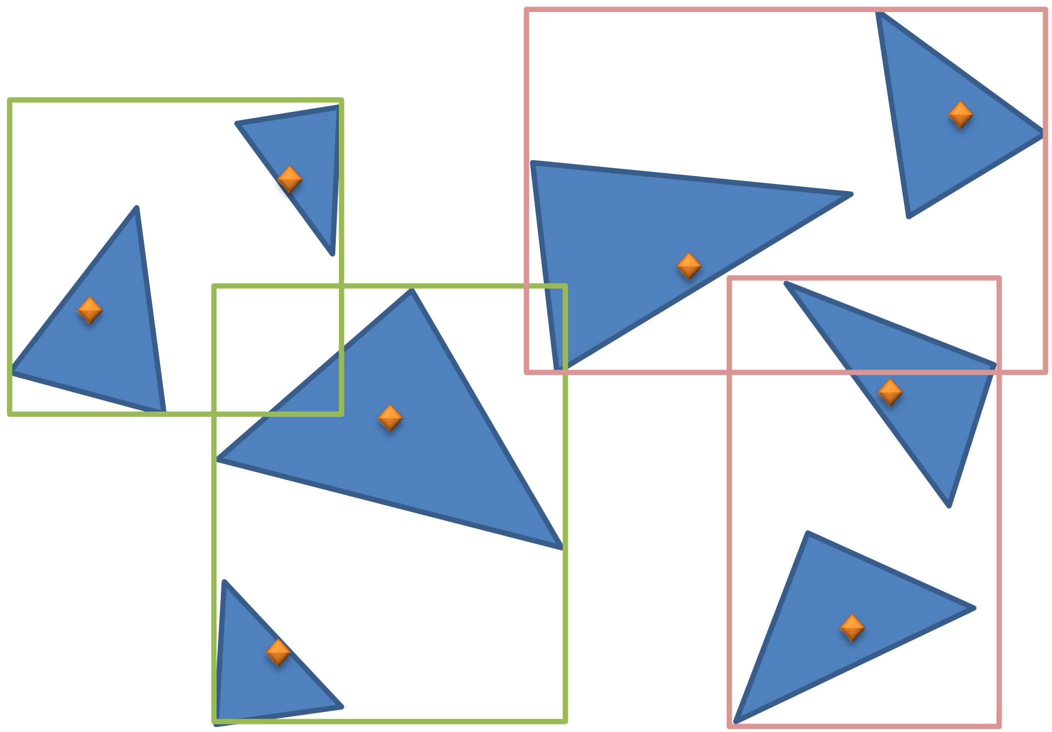
Find the *world* bounding box and create a root node

Construction , brief overview



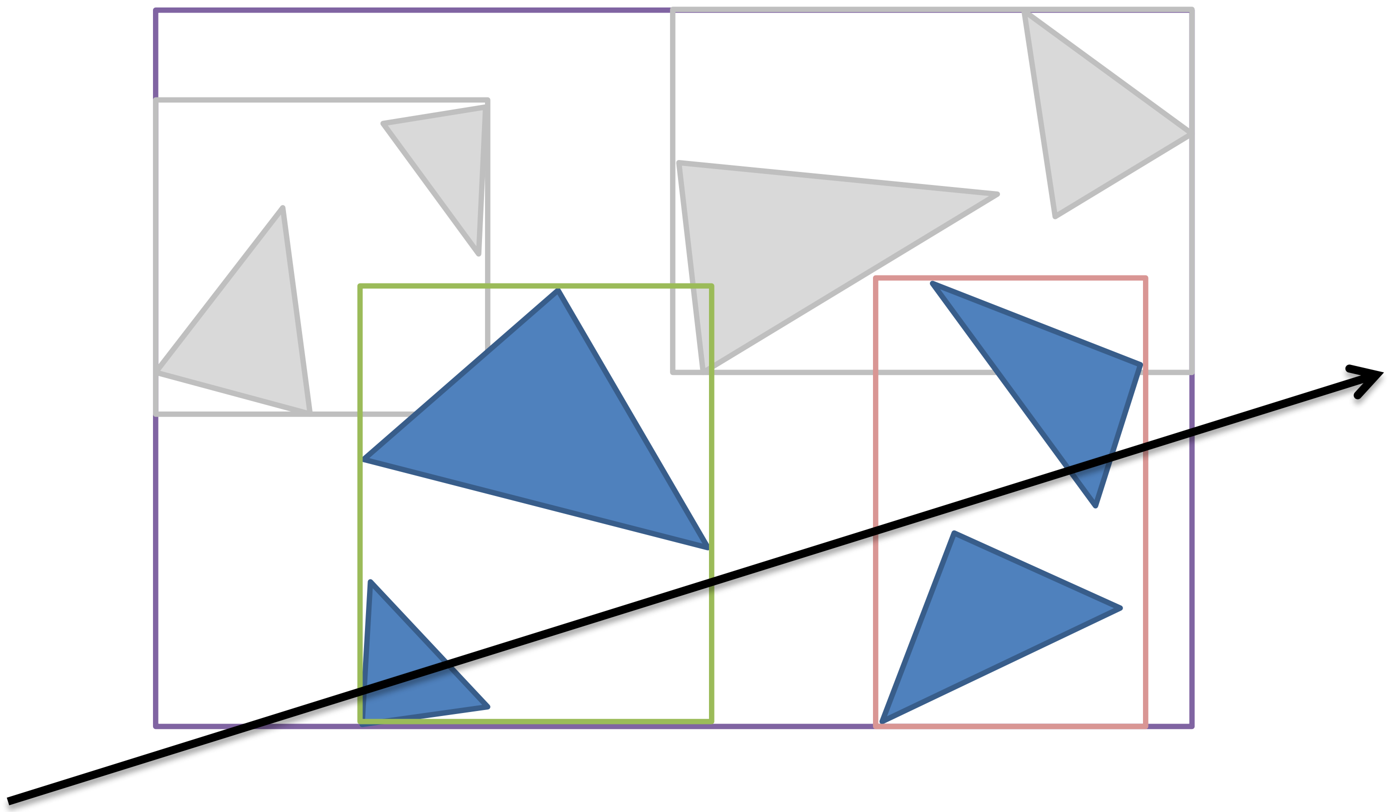
Use some splitting criteria to find a sensible division of the elements into new child nodes

Construction , brief overview



Continue to split recursively until each node contains only *one* or *a few* elements

Construction , brief overview



Now, when shooting rays we don’t have to test all Intersectables anymore!

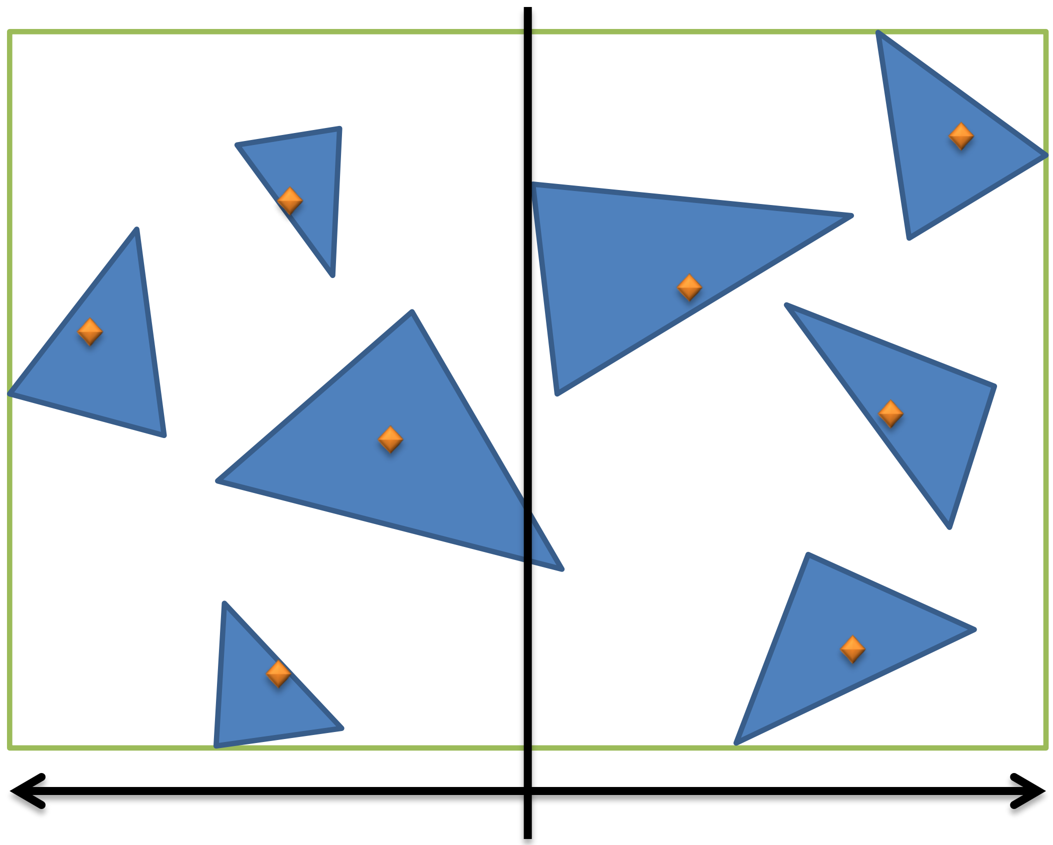
Construction

- So what is a sensible splitting criteria?
- Why not use *mid-point* splitting, since it's easy to understand and implement
  - *Works well when primitives are fairly evenly distributed*

Construction

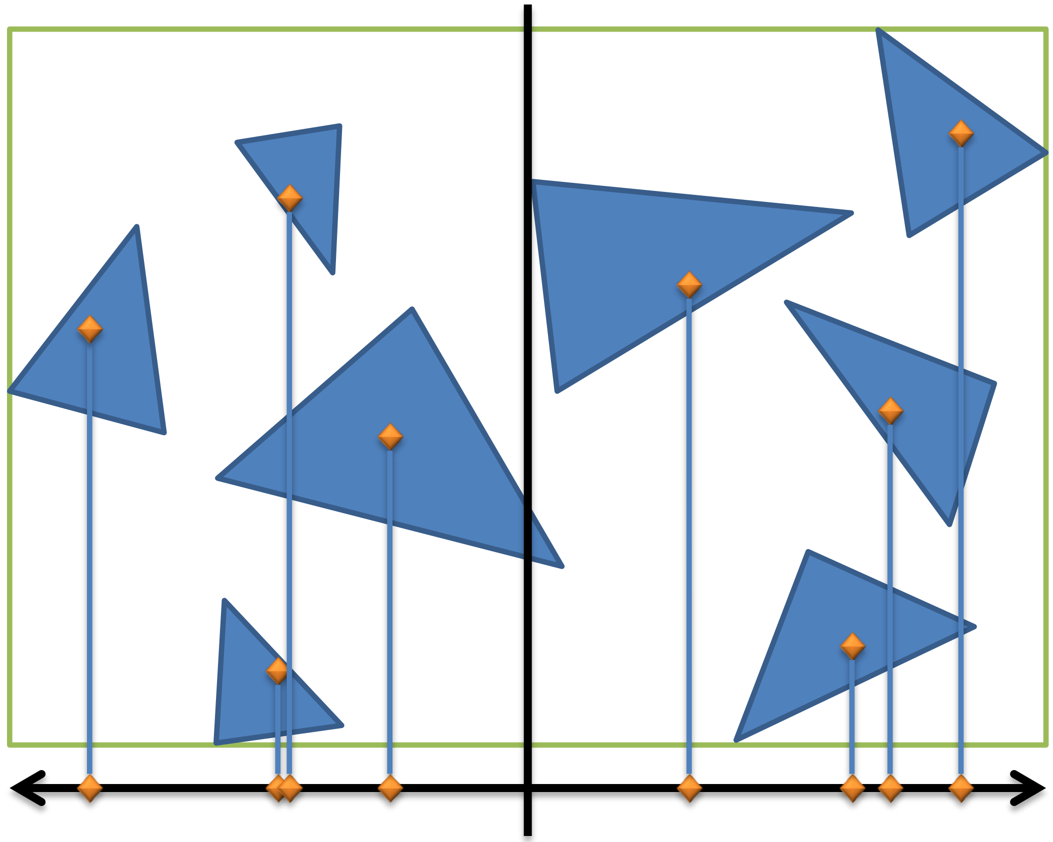
- So what is a sensible splitting criteria?
- Why not use *mid-point* splitting, since it's easy to understand and implement
  - *Works well when primitives are fairly evenly distributed*
- You can try to come up with a different criteria if you want to
  - *I tried splitting on the mean and median. Both were outperformed by mid-point splitting*

Construction



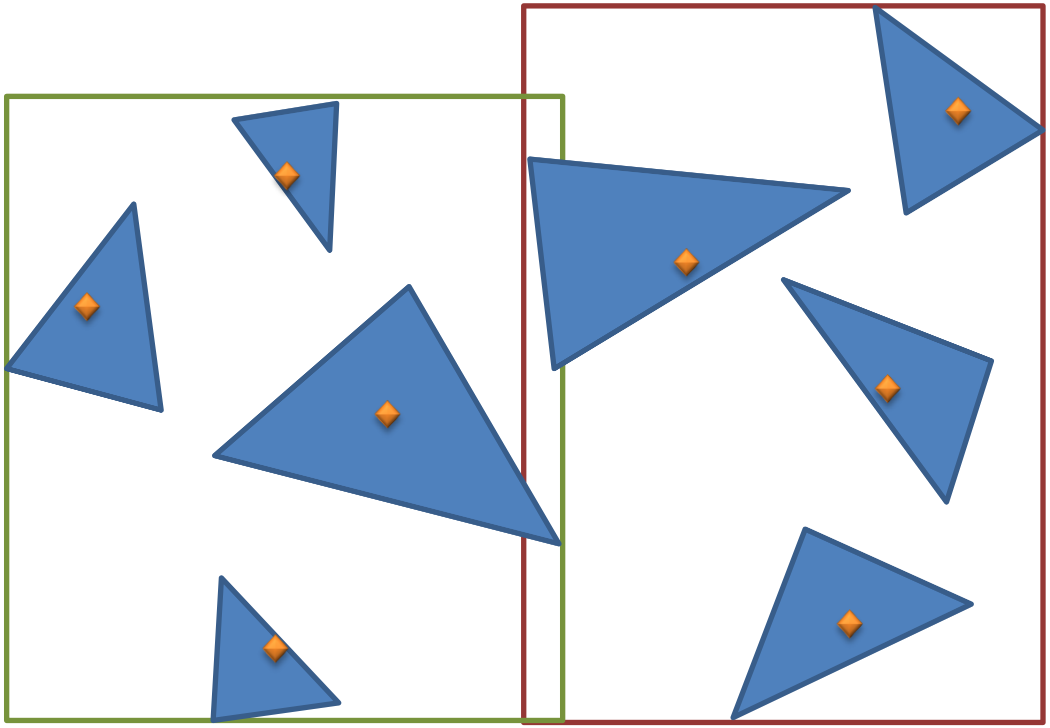
Find the mid point of the largest axis

Construction



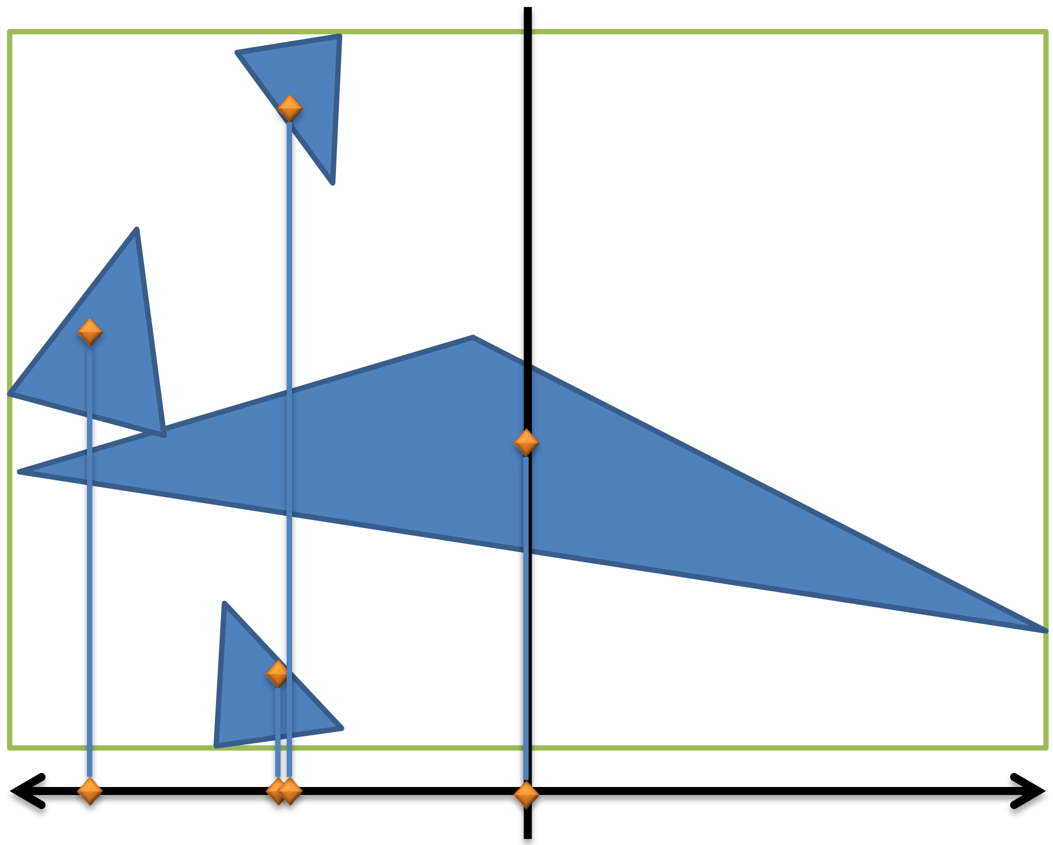
Sort the bounding box *centroids* in the largest axis direction. Then split into a *left* and a *right* side

Construction



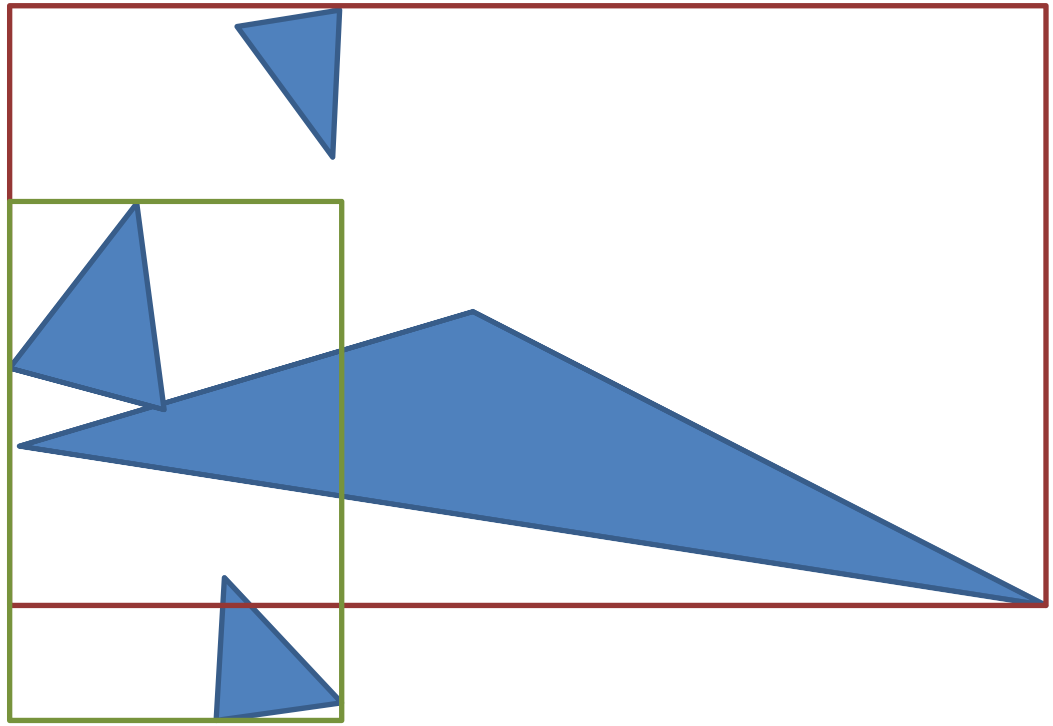
Lather, rinse and repeat. Terminate when a node contains few intersectables (I used 4, which worked well)

Construction



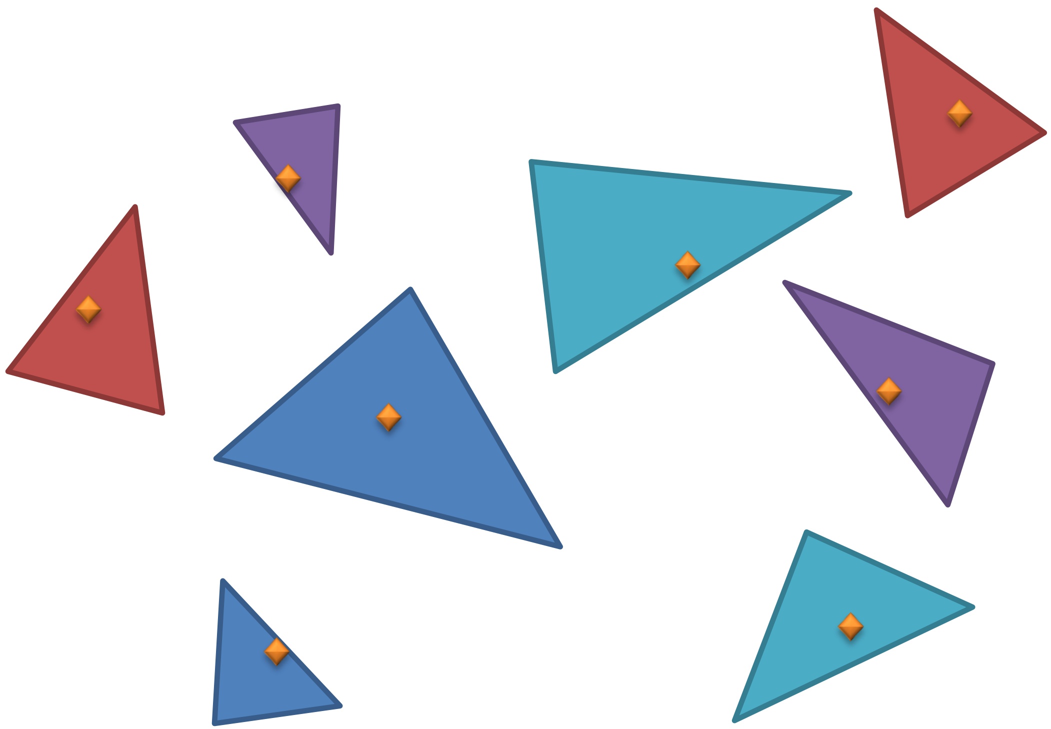
There is a hazard in getting all intersectables on one side – we could end up with empty nodes!

Construction



If this happens, you can, for example revert to median or mean splitting (*median split is depicted above*)

Construction



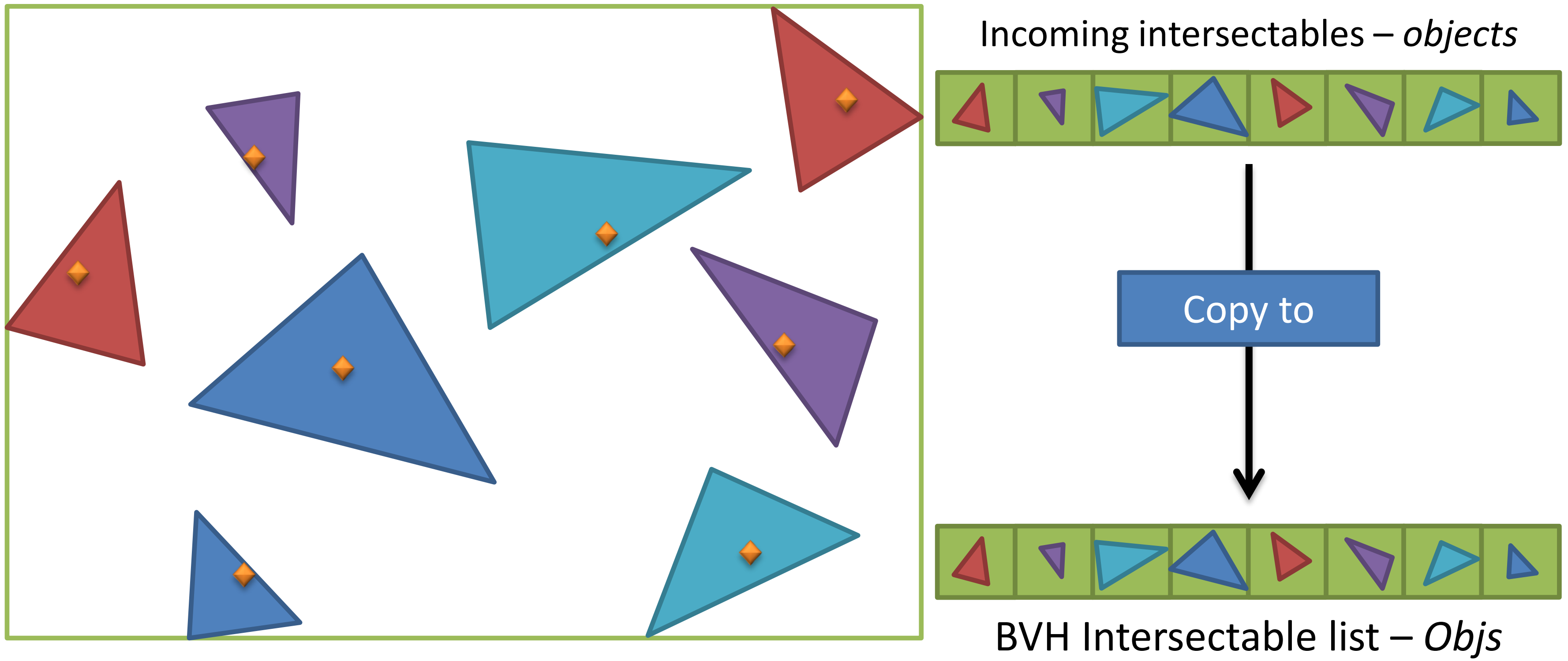
Now that you know the general concepts of a BVH, we will discuss *in-depth* how we keep track of our nodes and intersectables throughout the contruction process.

Node class

- BVH node class (inner class of *BVHAccelerator*)

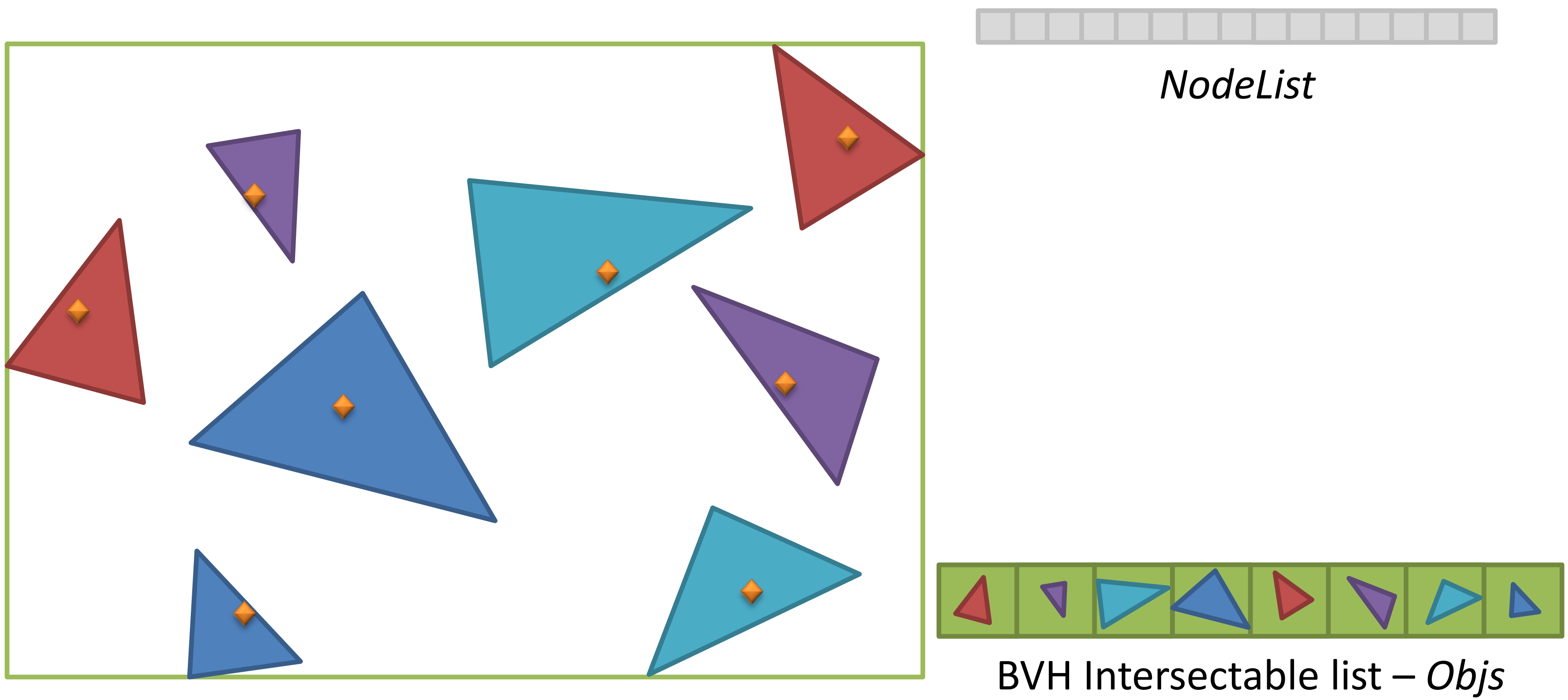
```
class BVHNode {
        private:
        AABB bbox;
        bool leaf;
        unsigned int n_objs;
        unsigned int index;     // if leaf == false: index to left child node,
                                // else if leaf == true: index to first Intersectable in Objs vector
        public:
        void setAABB(AABB &bbox_) {...}
        void makeLeaf(unsigned int index_, unsigned int n_objs_) {...}
        void makeNode(unsigned int left_index_, unsigned int n_objs) {...}
                // n_objs in makeNode is for debug purposes only, and may be omitted later on

        bool isLeaf() { return leaf; }
        unsigned int getIndex() { return index; }
        unsigned int getNObjs() { return n_objs; }
        AABB &getAABB() { return bbox; };
};
```

Construction



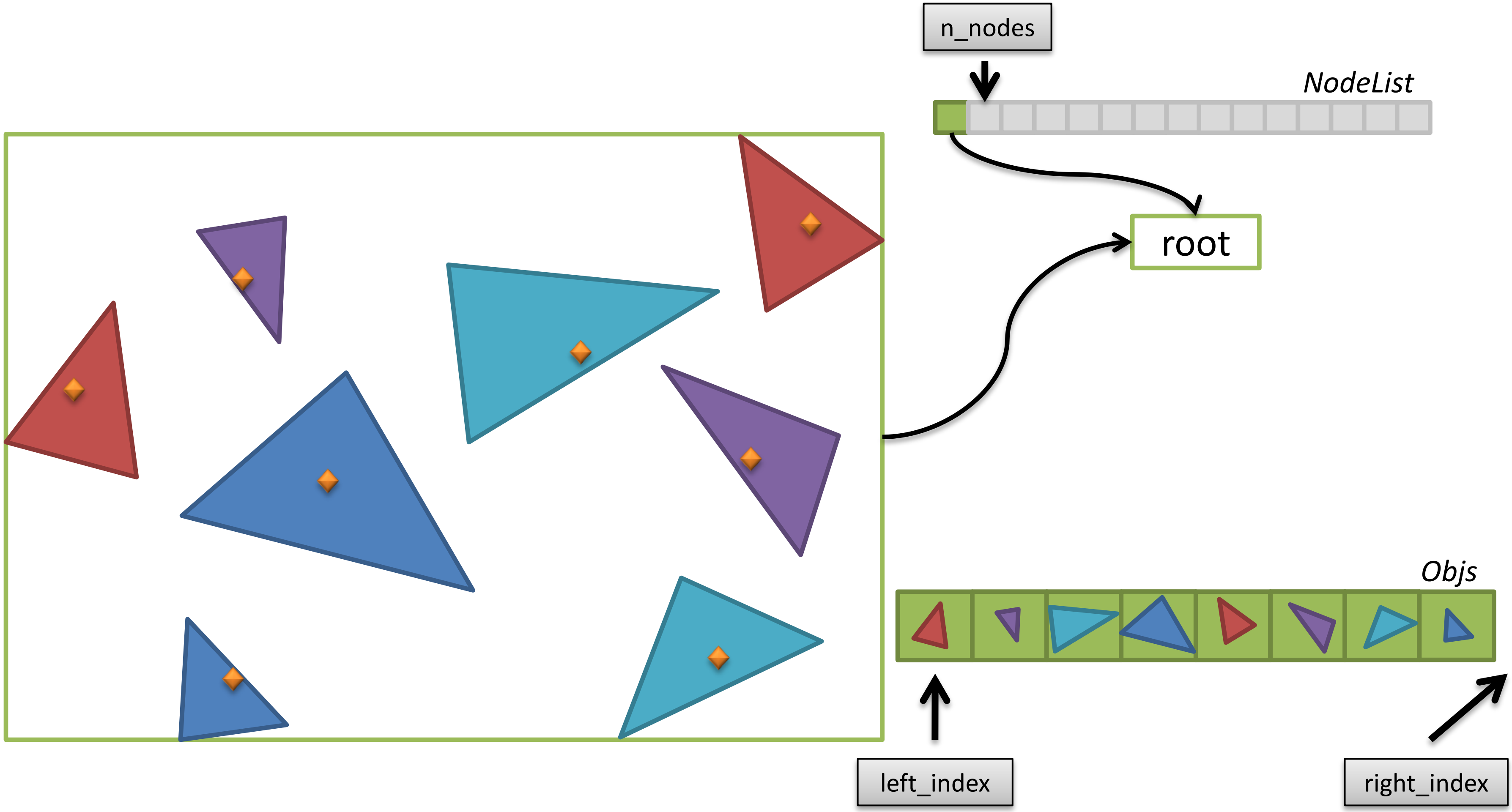
In the **build()**-function we get a list of unsorted *Intersectable* pointers, which we copy to a local vector. At the same time we calculate the world bounding box.

Construction



Set up a *NodeList* vector, which will hold our nodes. (We also happen to know that the number of nodes needed will be at most **2n – 1** nodes, if the leaf nodes contain 1 element each).

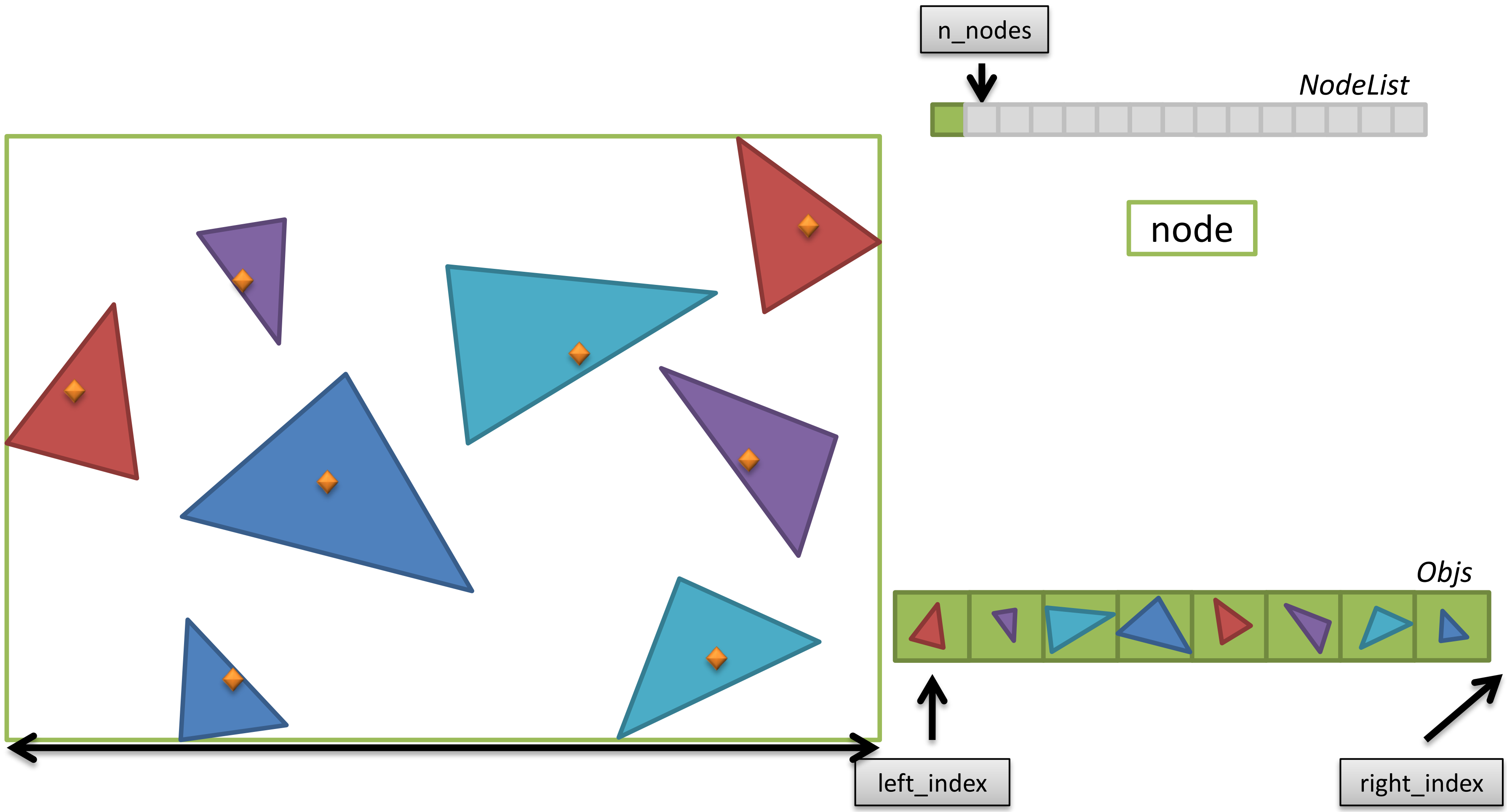
Construction



Set *left_index* = 0, *right_index* = Objs.size() and n_nodes = 1.

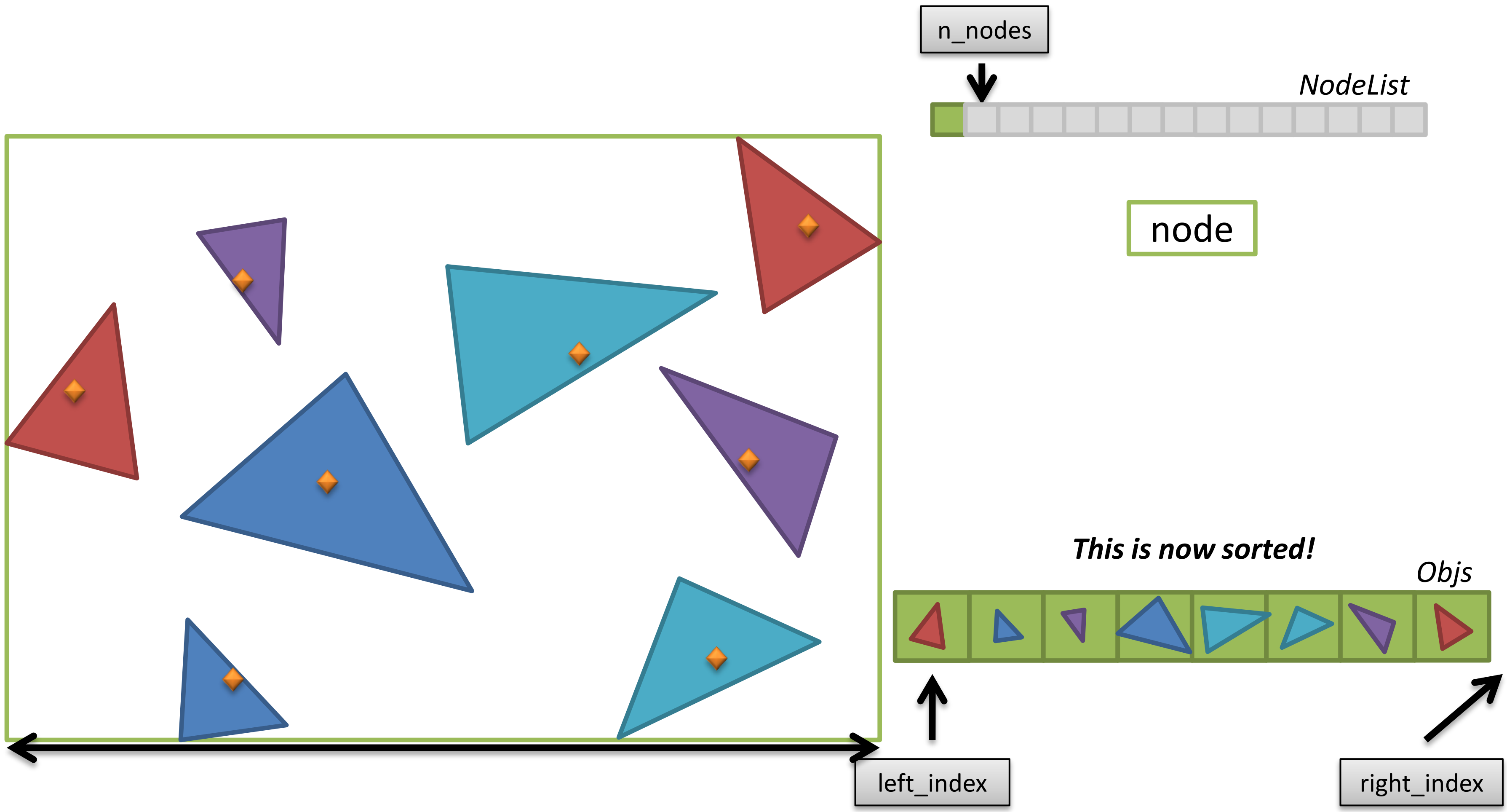
Set the world bounding box to the root node using BVHNode::setAABB(box).
Then start building recursively.

Construction



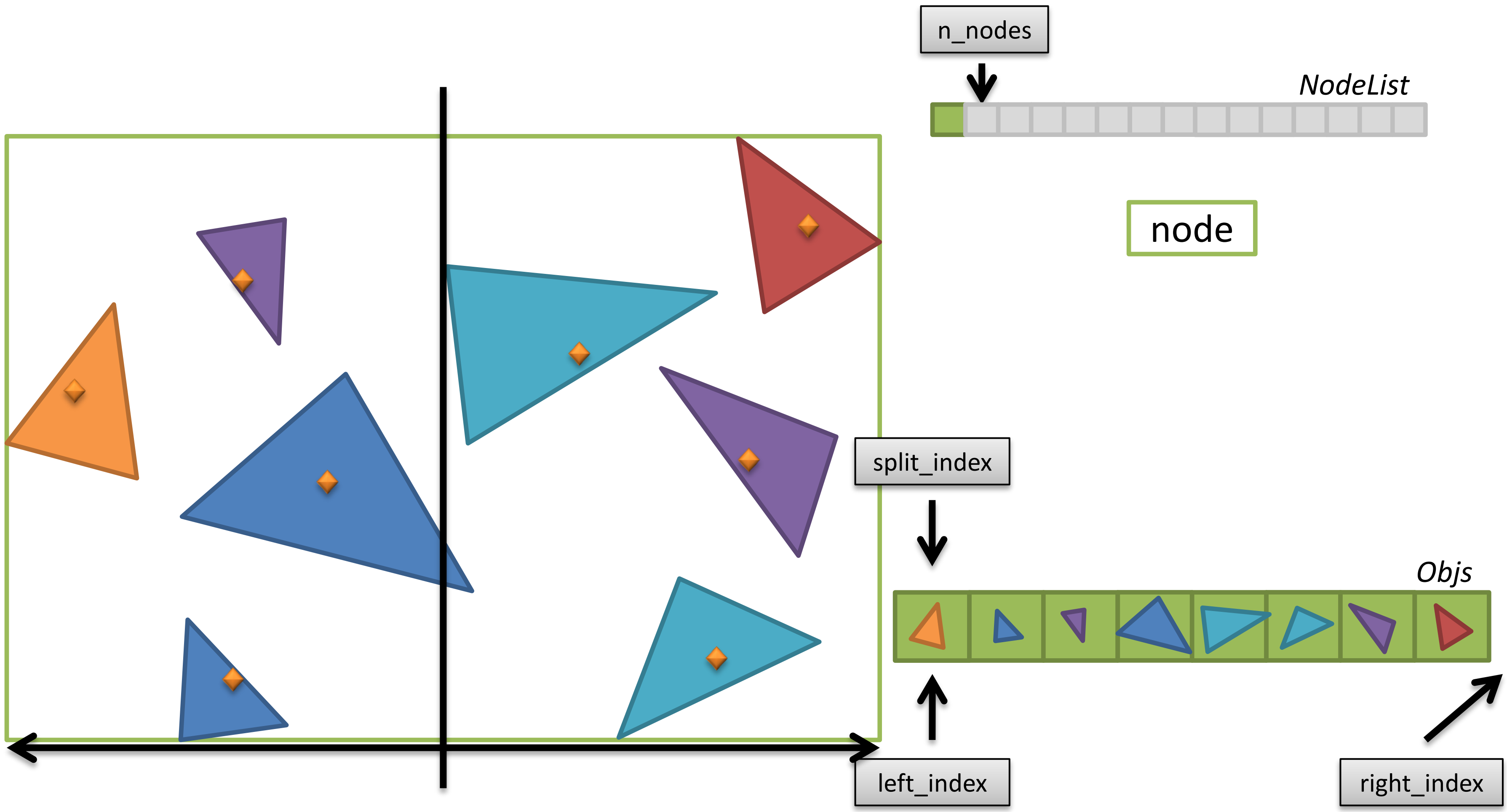
First, check if the number of intersectables is fewer than the threshold (let's say 2 in this case). It isn't.

Construction



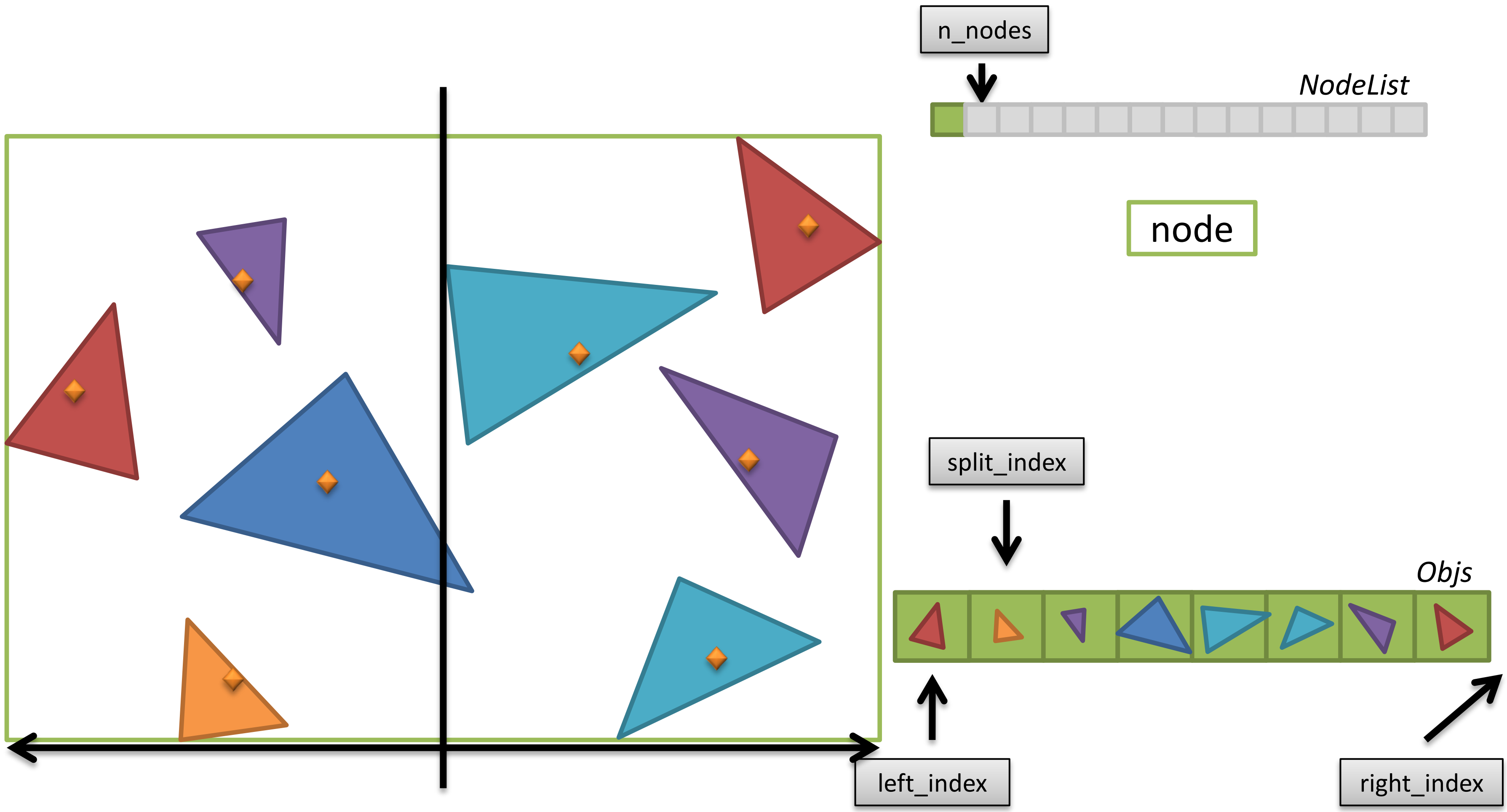
Find largest dimension *d* and sort the elements in that dimension.

Construction



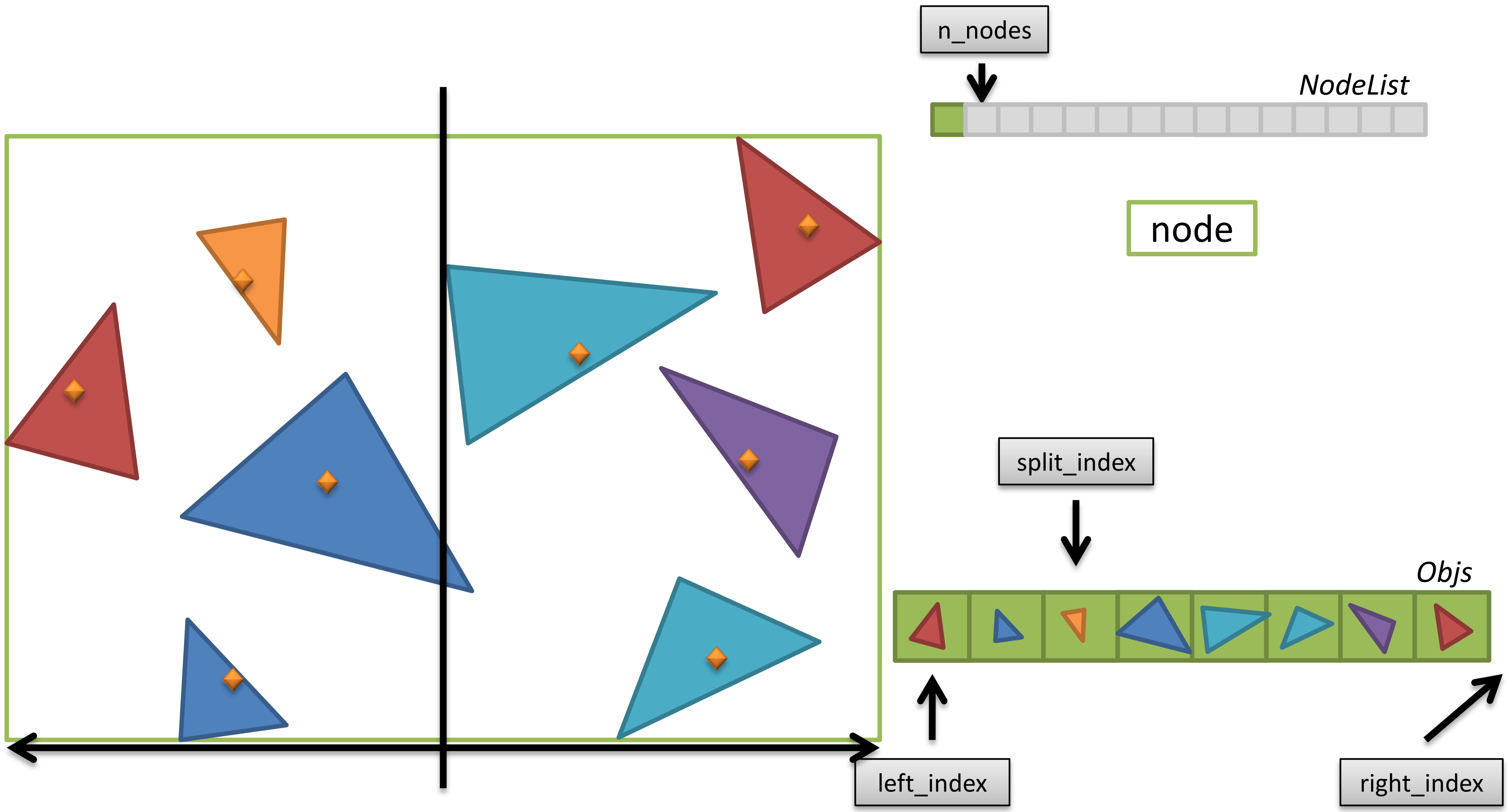
Find the *split_index,* where the mid point divides the primitives in a *left* and *right* side

Construction



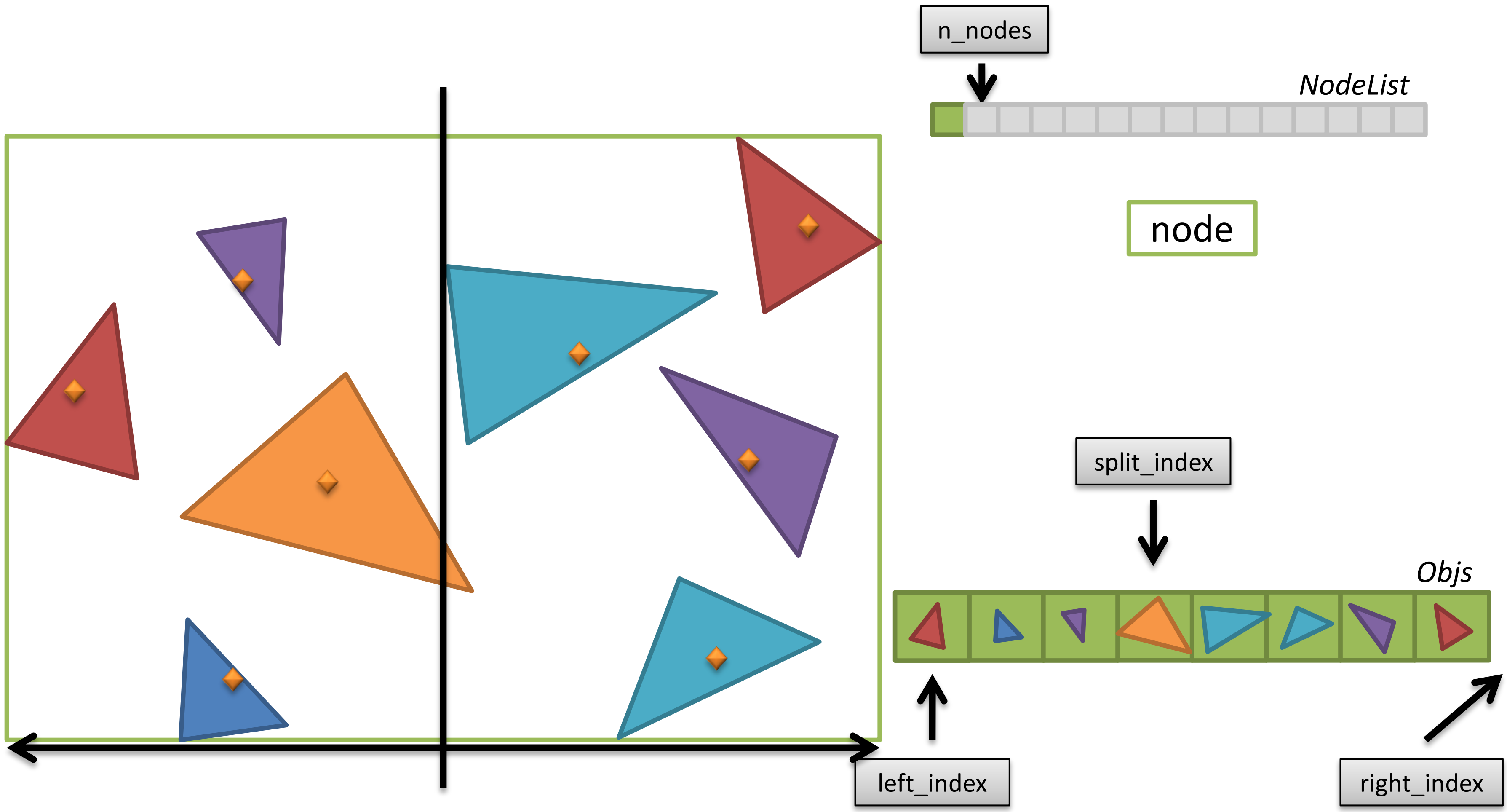
Find the *split_index,* where the mid point divides the primitives in a *left* and *right* side

Construction



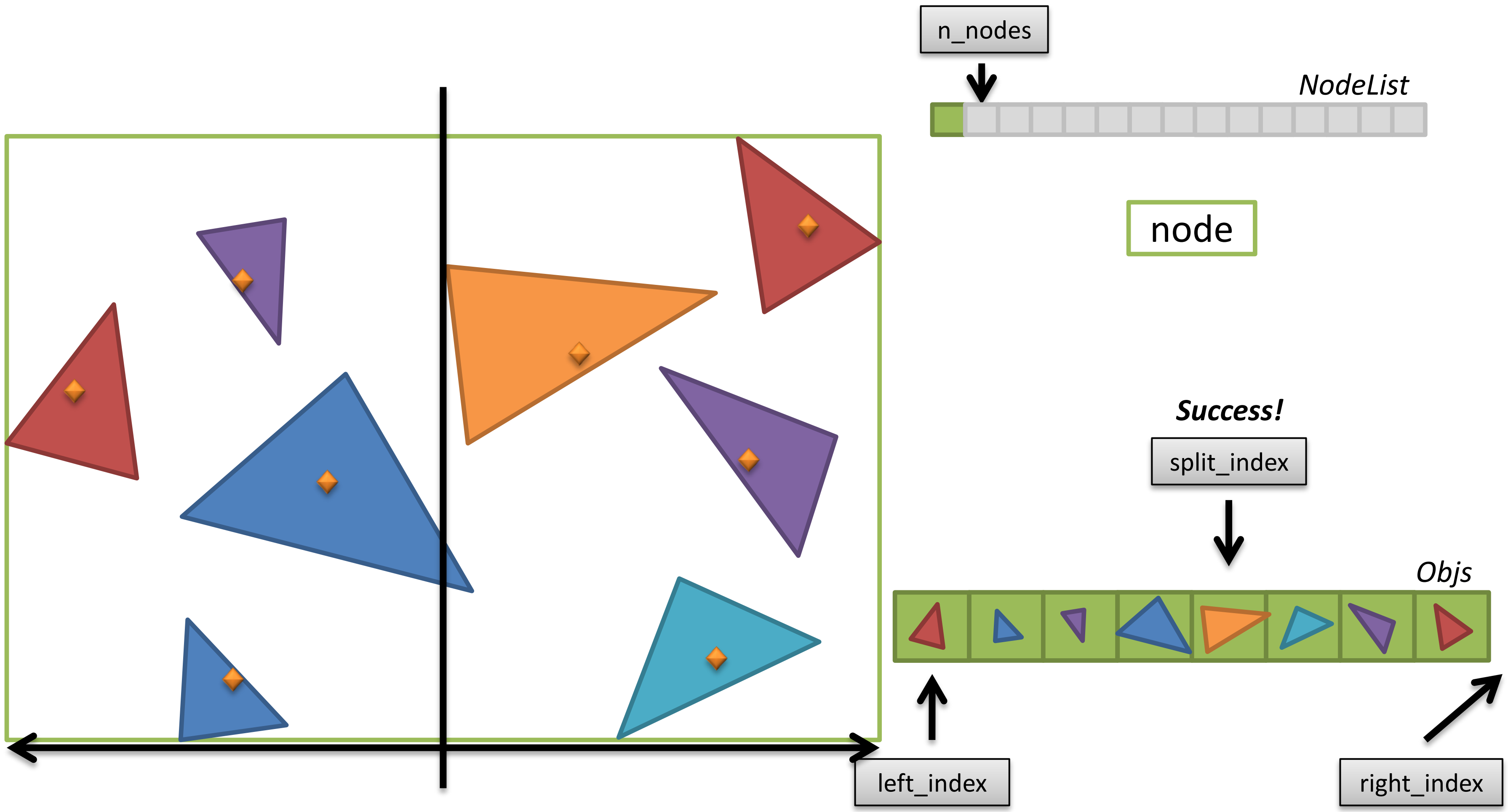
Find the *split_index,* where the mid point divides the primitives in a *left* and *right* side

Construction



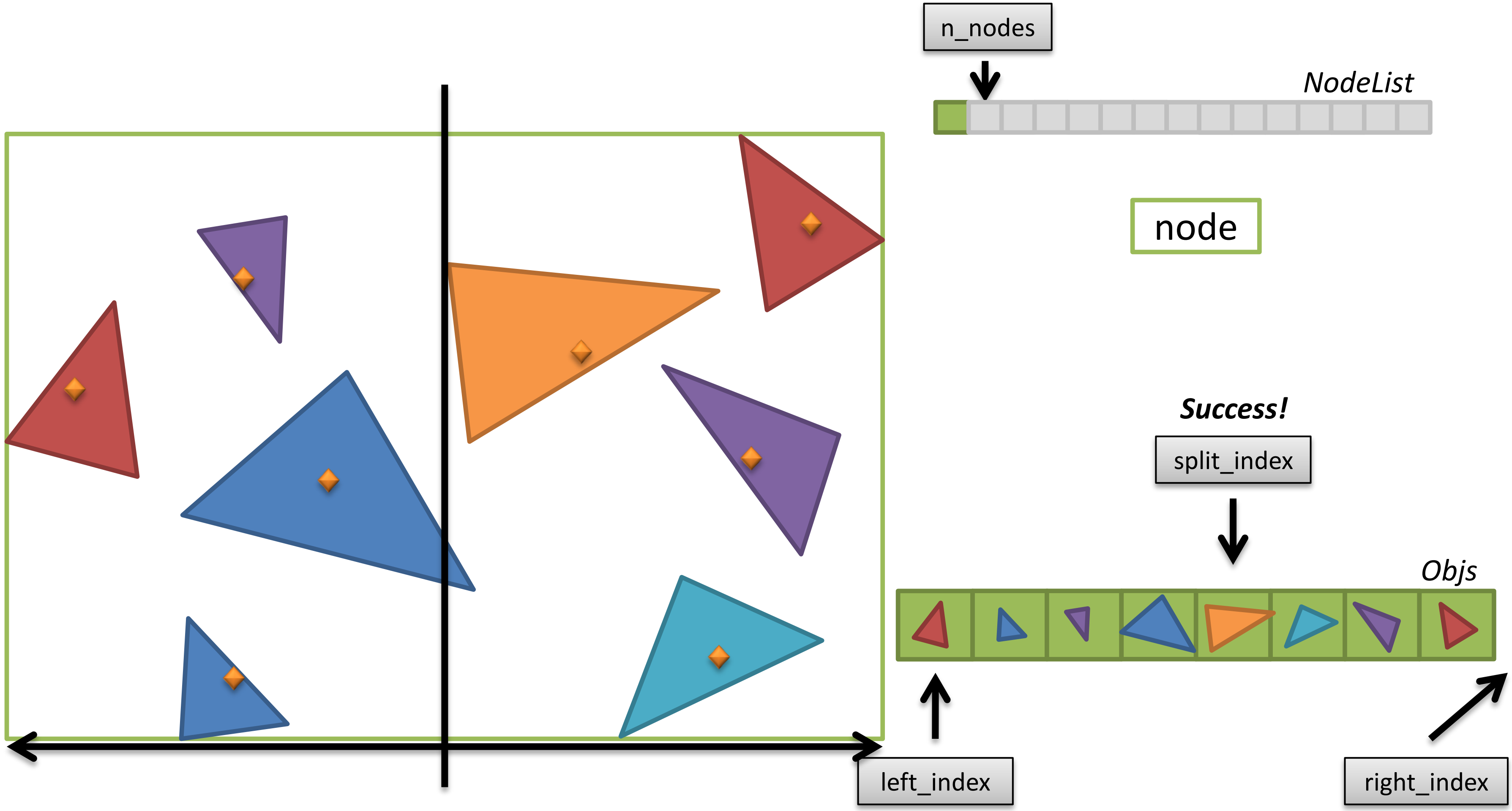
Find the *split_index,* where the mid point divides the primitives in a *left* and *right* side

Construction



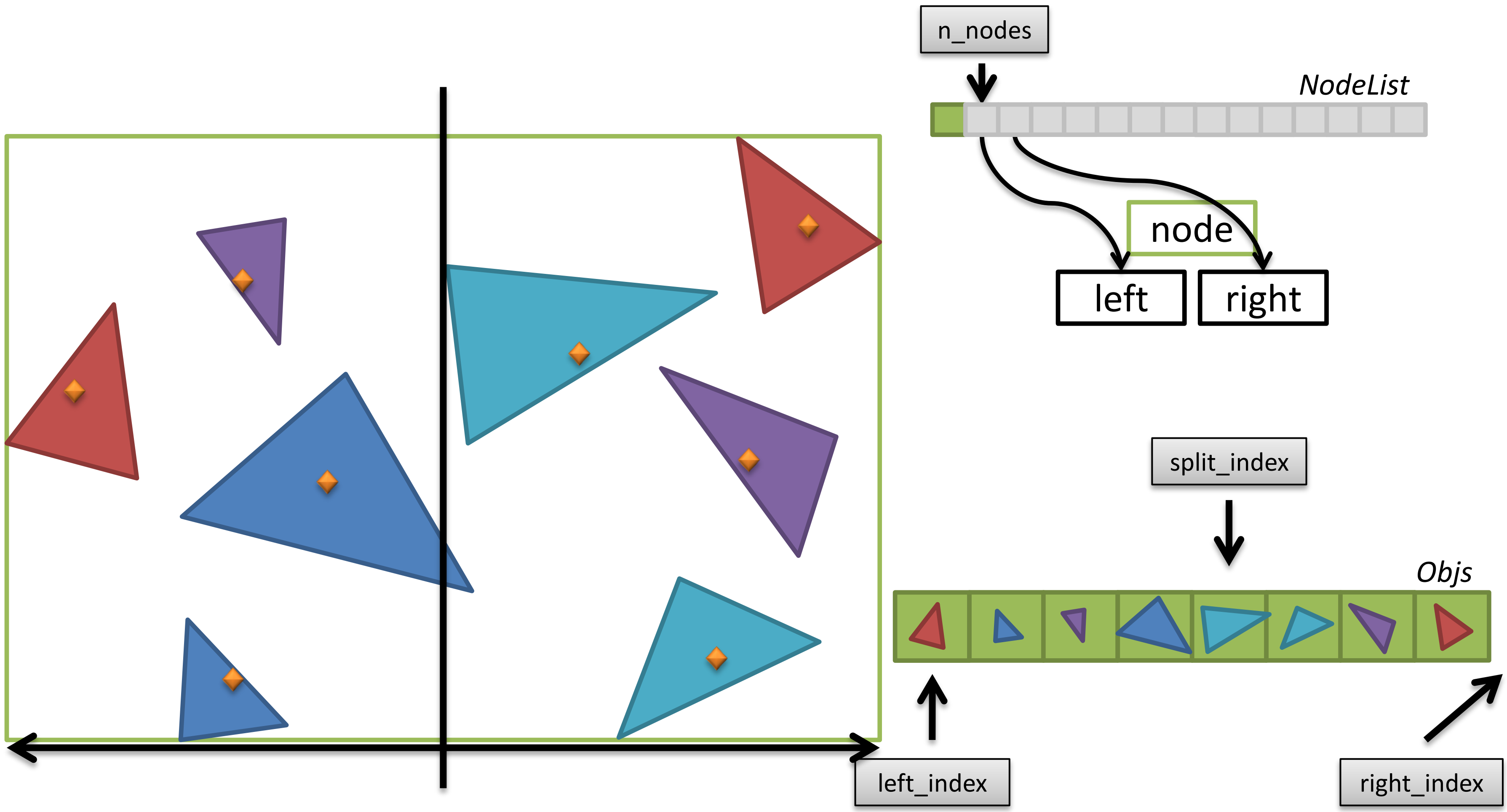
Find the *split_index,* where the mid point divides the primitives in a *left* and *right* side

Construction



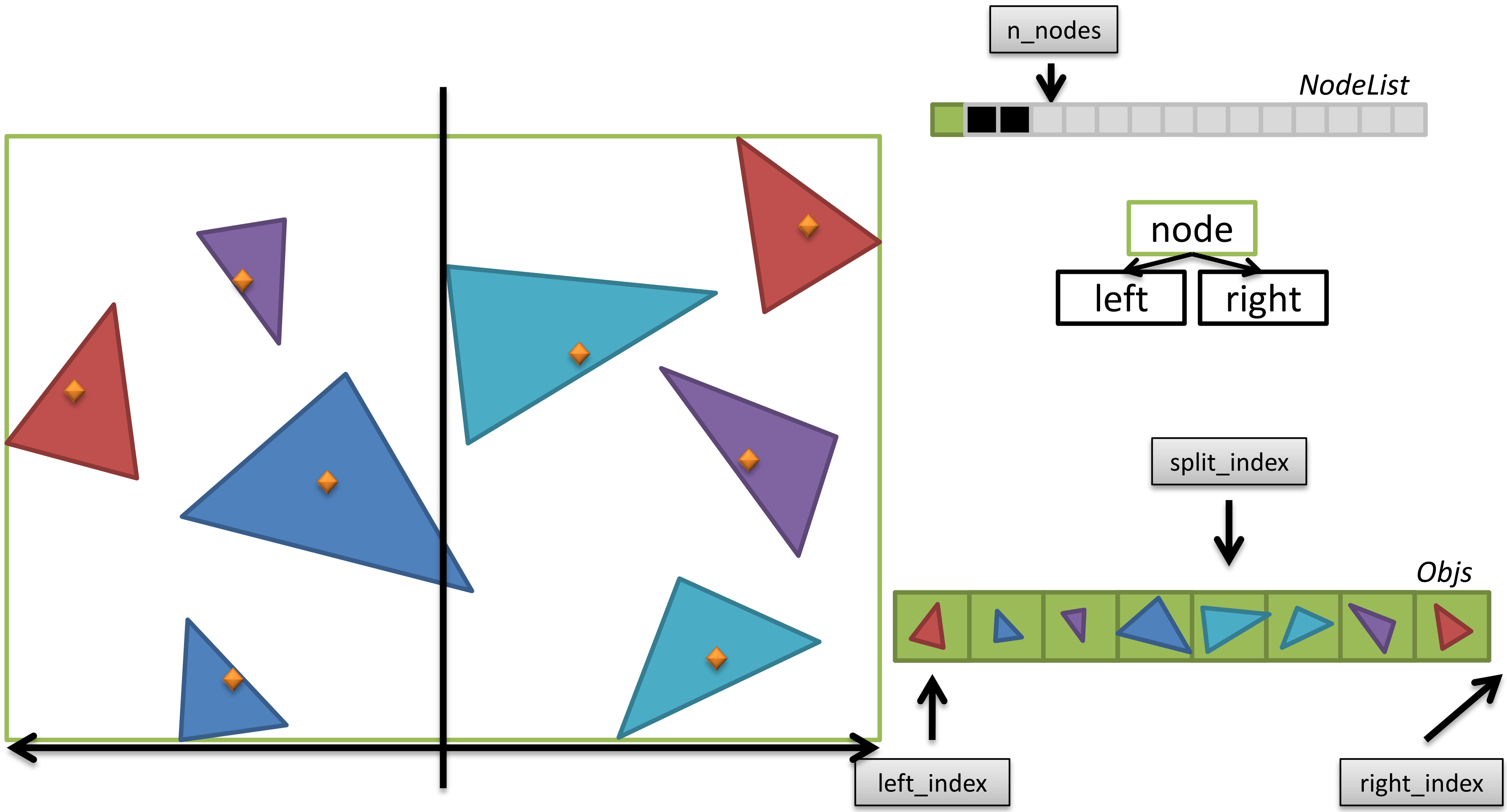
(We could have used binary search)

Construction



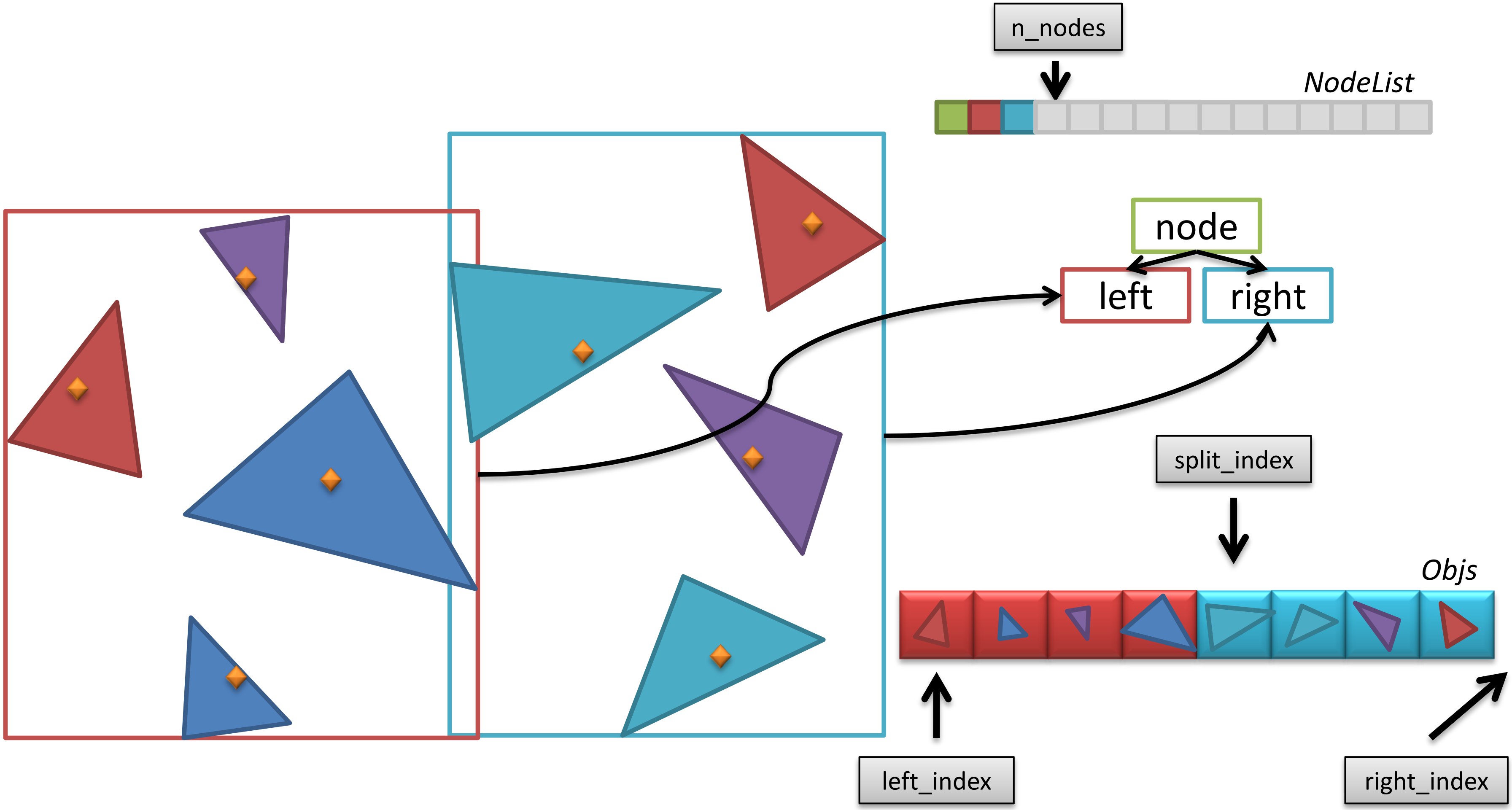
Allocate two new nodes (*left* and *right*) from the NodeList. The left node will have index ***n_nodes*** and the right one ***n_nodes + 1***.

Construction



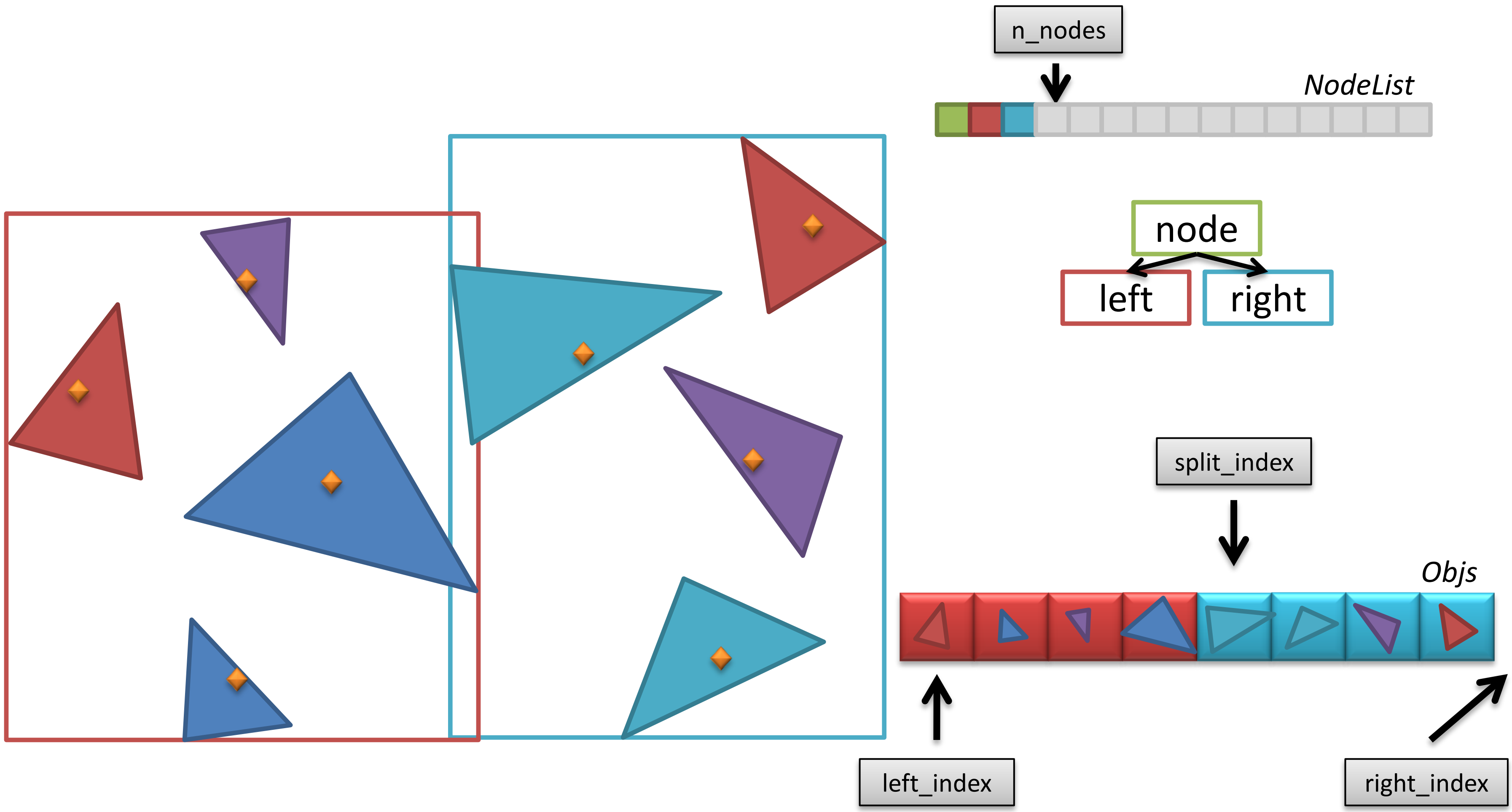
Initiate **node** (which is currently the root node) with
BVHNode::makeNode(*n_nodes*).
(The index to the right node is always n_nodes + 1)

Construction



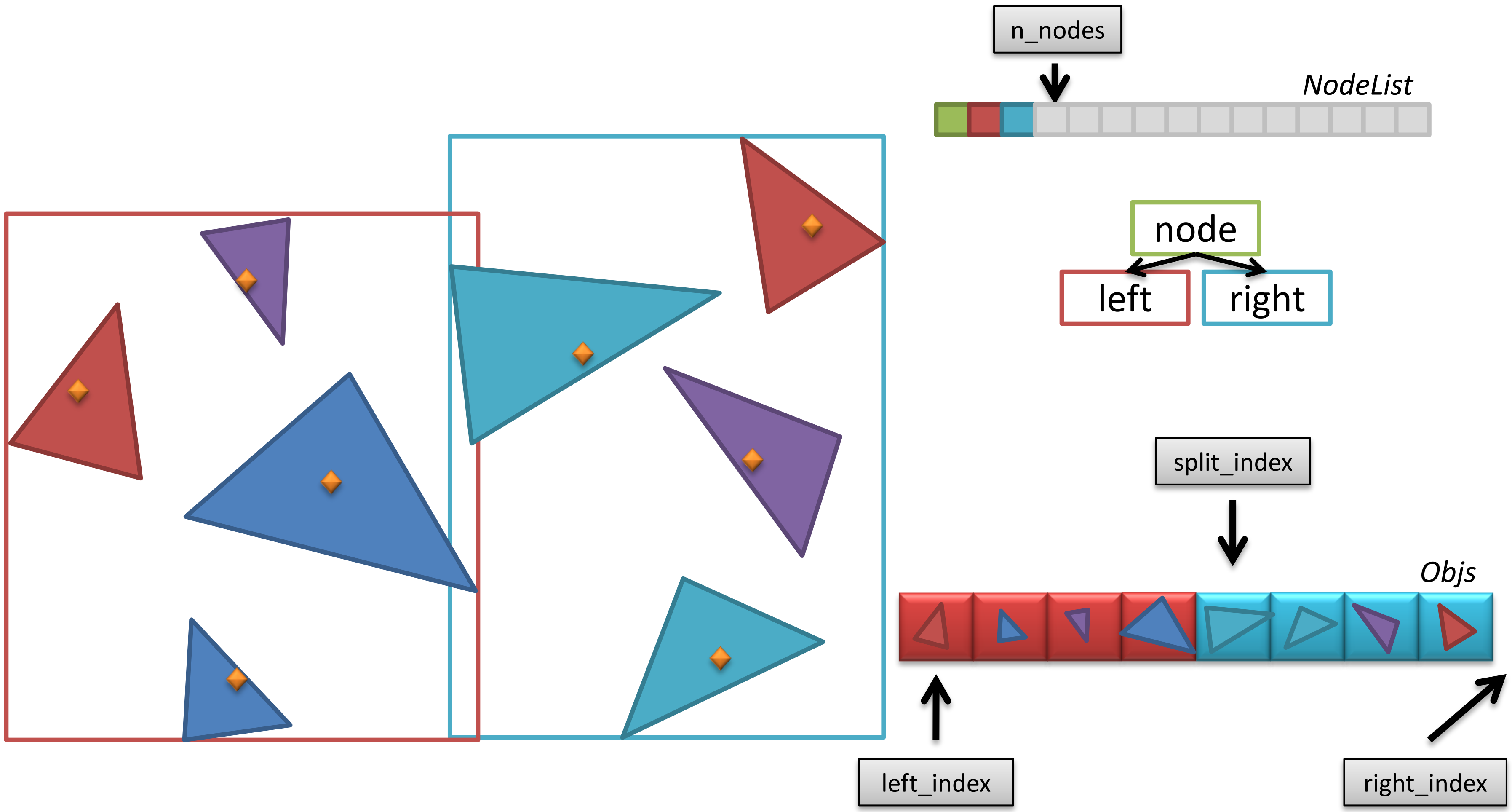
Calculate the bounding boxes for *left* and *right* and assign them to the two newly created nodes

Construction



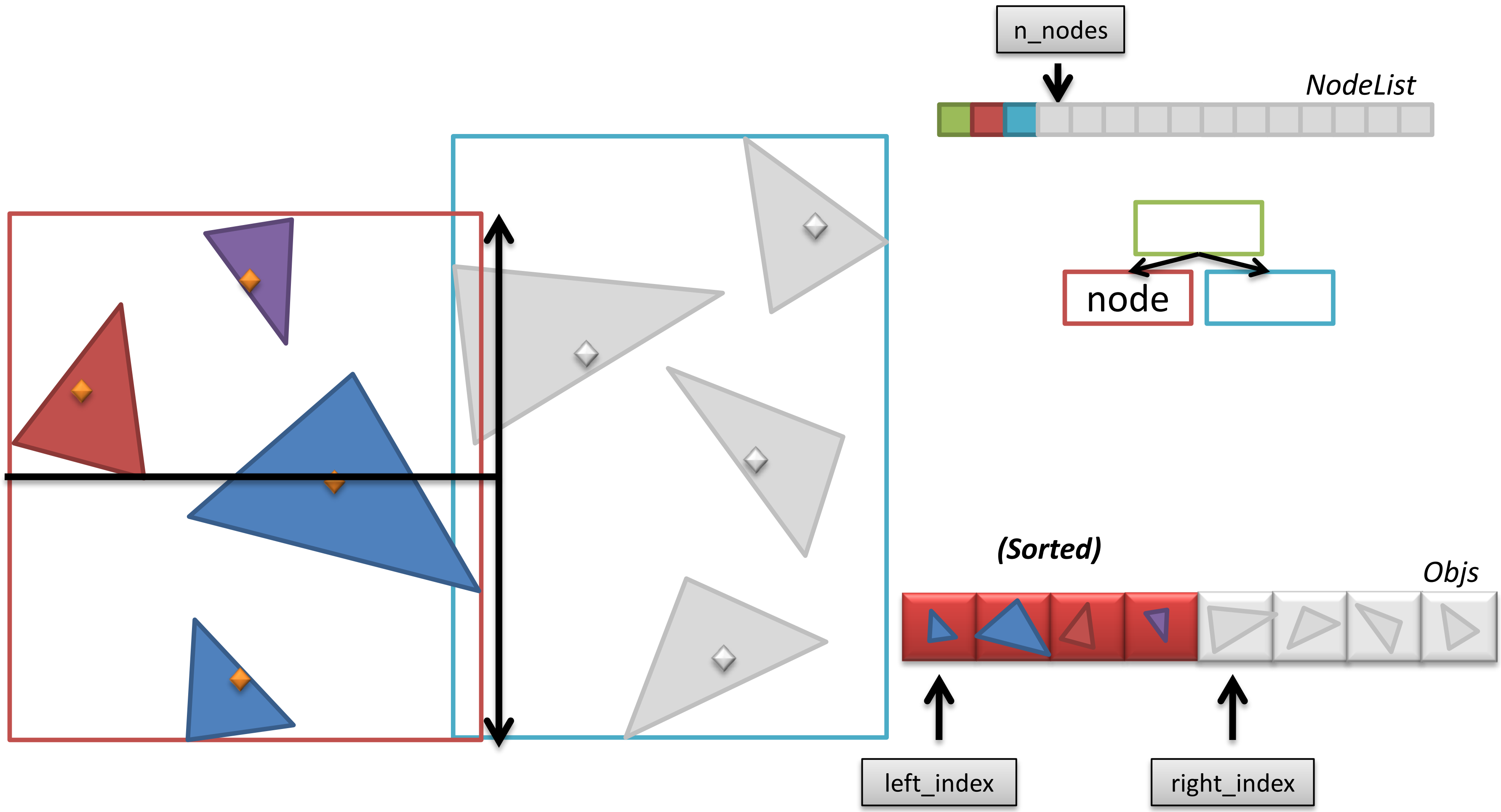
Call the **build_recursive**()-function for the *left* and then the *right* node.

Construction



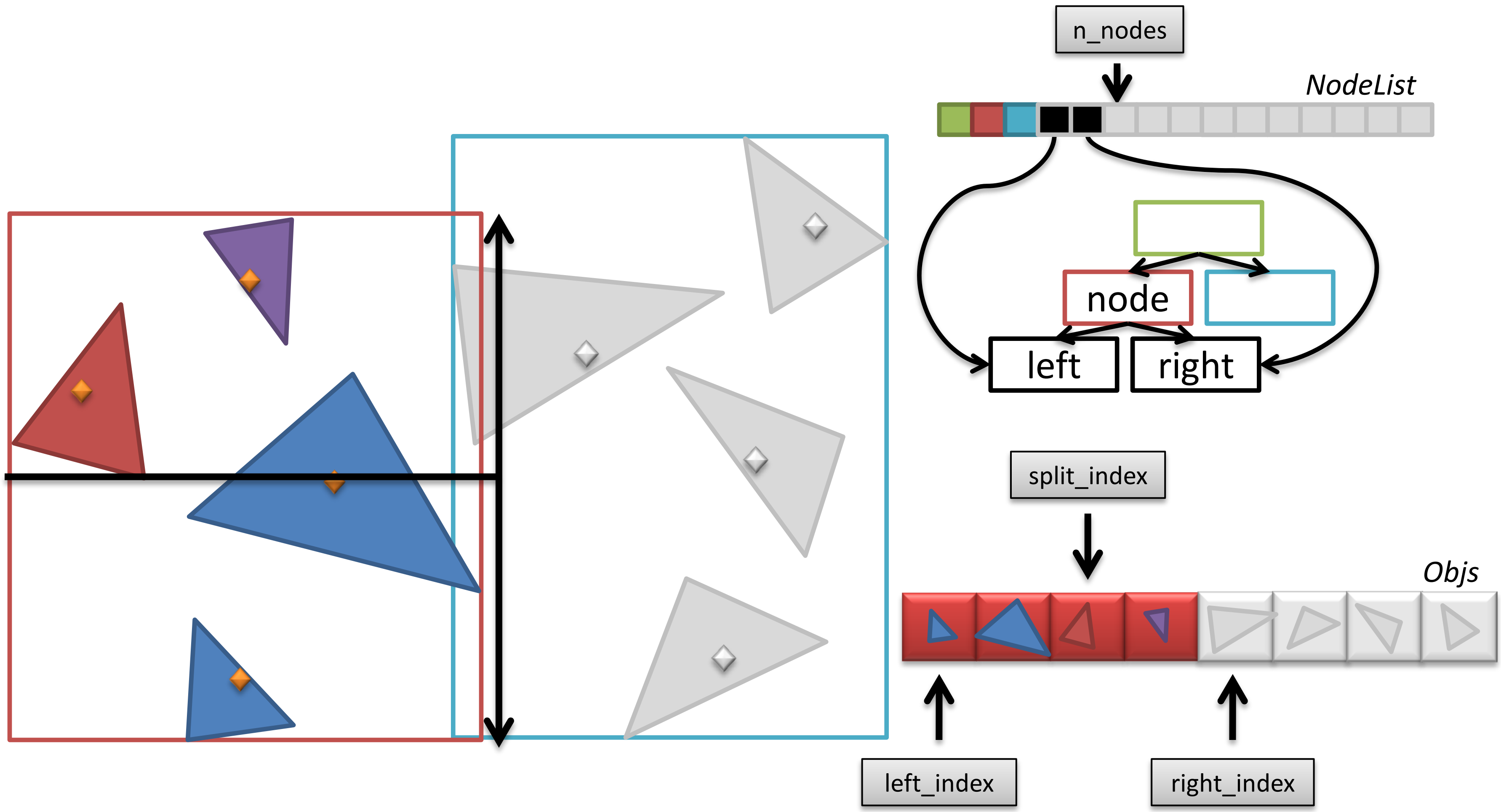
Be sure to pass the correct **Objs-vector indices** to the *left* and *right* nodes. The *left* node is now responsible for *[left_index, split_index)* and the *right* node for *[split_index, right_index)*.

Construction

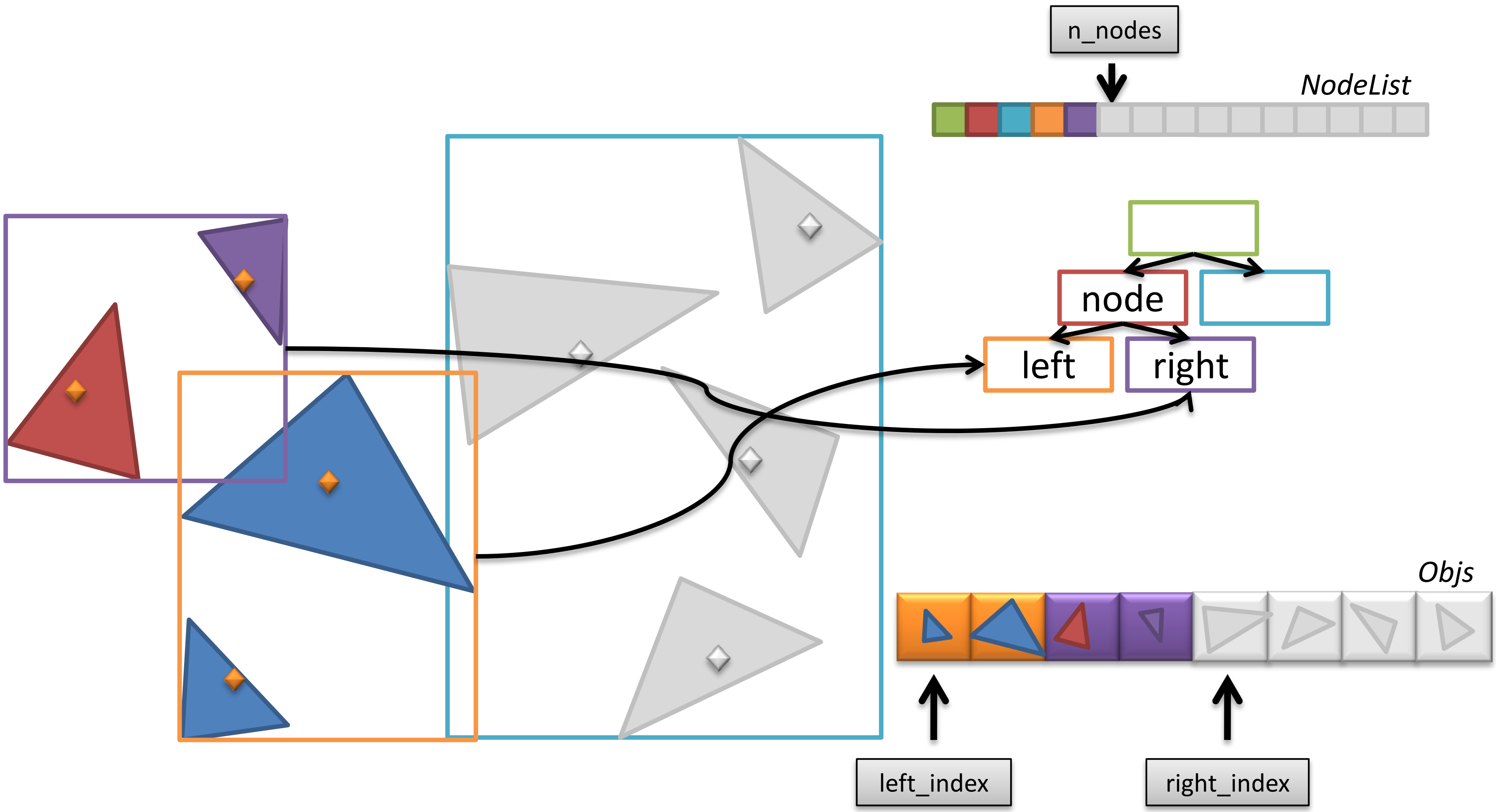


Processing of the left node yields the following result…

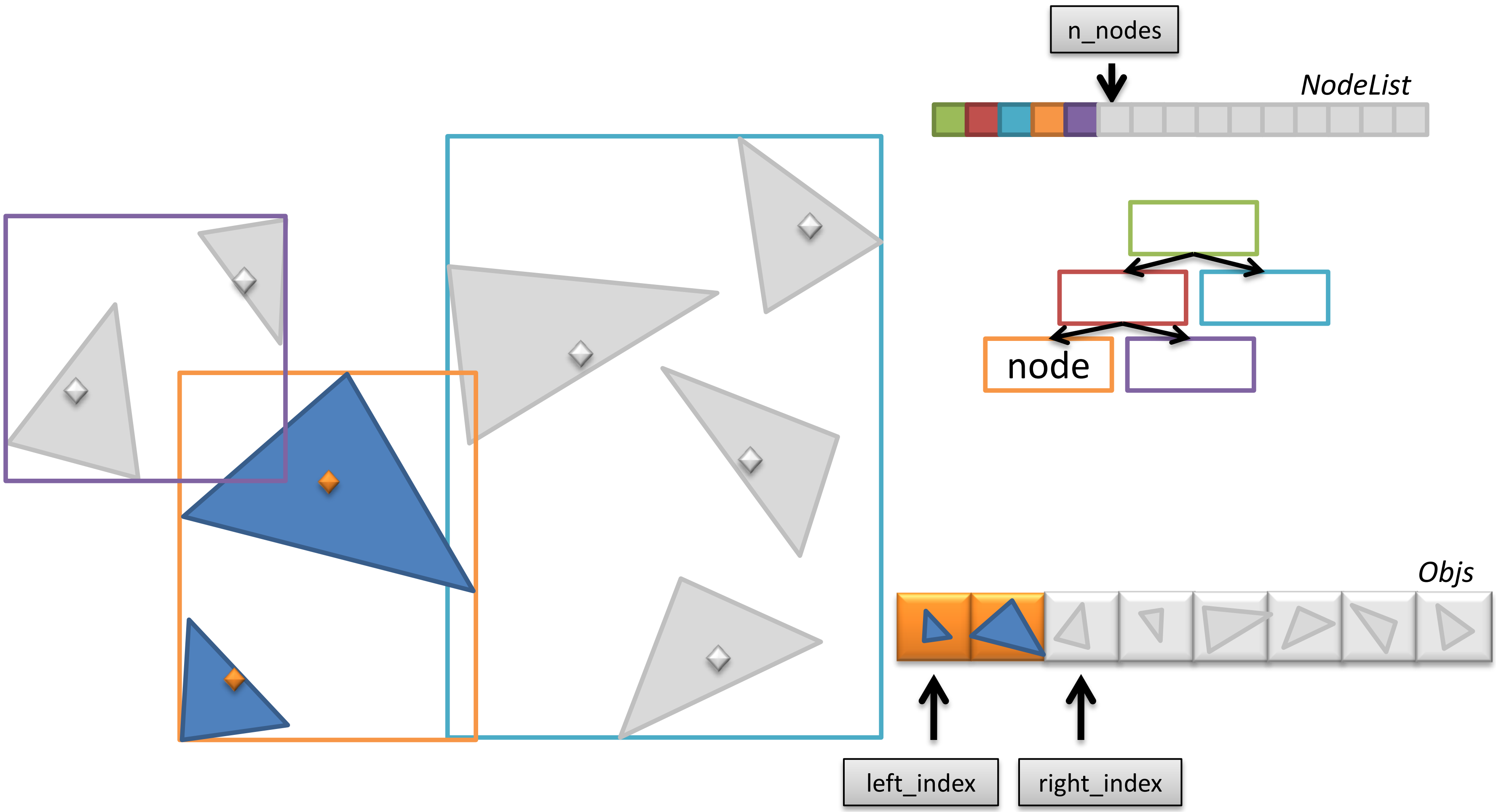
Construction



Construction



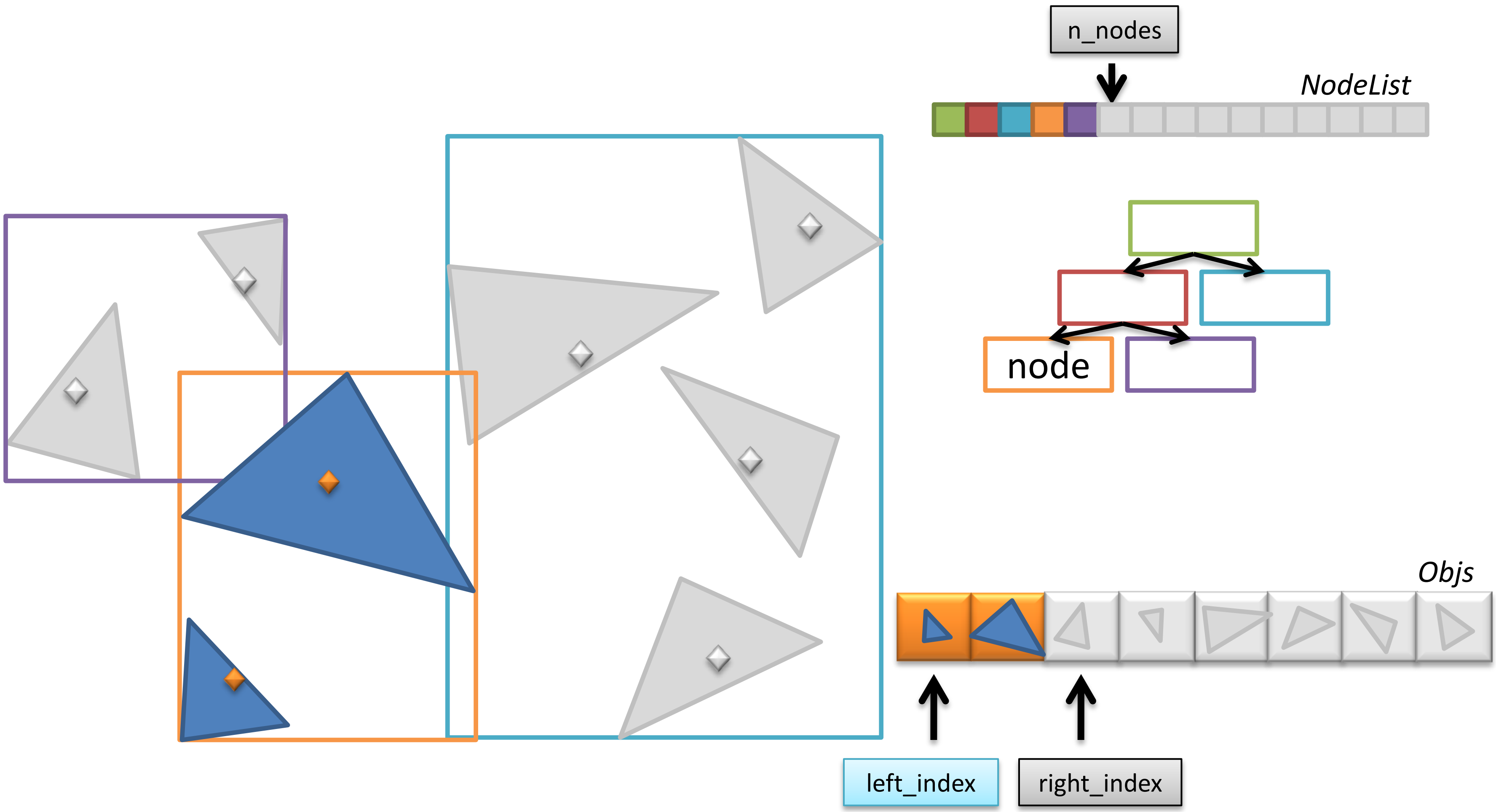
Construction



Finally we have only 2 primitives in the node:

(right_index – left_index <= 2)

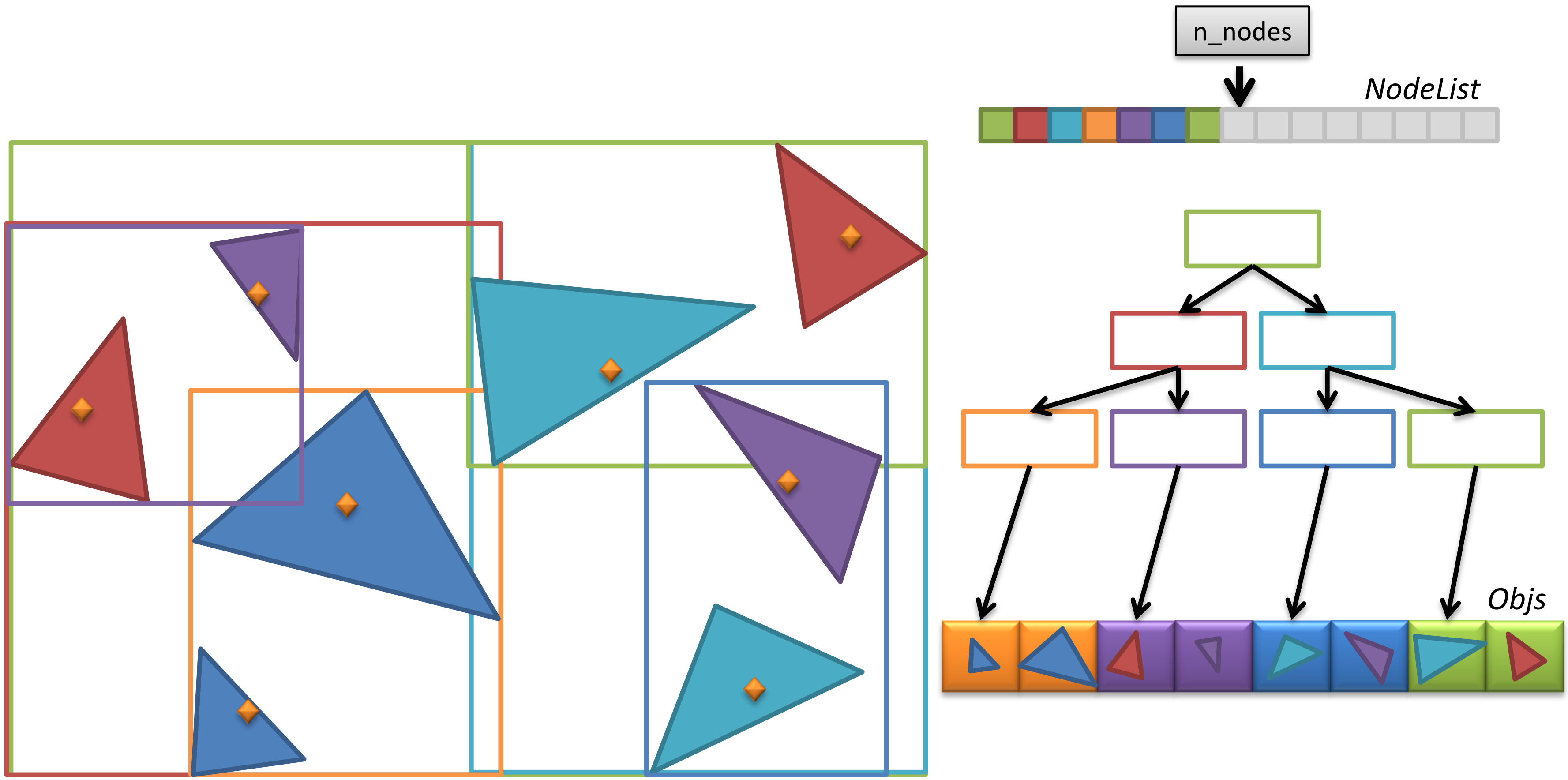
Construction



We initiate the current node as a leaf using

void BVHNode::makeLeaf(**left_index**, **right_index – left_index**);

Construction



This is what we end up with when we're done.

Construction, Pseudo code

Setup

void build(const std::vector<Intersectable *> &objects)

- Create new vector for *Intersectable* pointer copies
- Create new vector for the nodes
- Create Root node
- worldBox = AABB();   // world bounding box
- For each intersectable[i] in *objects*
  - worldBox.include(intersectable[i] bounding box)
  - Objs.push_back(intersectable[i])
- EndFor
- Set world bounding box to root node
- build_recursive(0, Objs.size(), root, 0);

*The declaration was: void build_recursive(int left_index, int right_index, BVHNode *node, int depth);*

Construction, Pseudo code

Recursion

void build_recursive(int left_index, int right_index, BVHNode *node, int depth)

- If ((right_index – left_index) <= Threshold || (*other termination criteria)*)
  - Initiate current node as a leaf with primitives from Objs[left_index] to Objs[right_index]
- Else
  - Split intersectables into *left* and *right* by finding a *split_index*
    - Make sure that neither *left* nor *right* is completely empty
  - Calculate bounding boxes of *left* and *right* sides
  - Create two new nodes, *leftNode* and *rightNode* and assign bounding boxes
  - Initiate current node as an interior node with *leftNode* and *rightNode* as children
  - build_recursive(left_index, split_index, leftNode, depth + 1)
  - build_recursive(split_index, right_index, rightNode, depth + 1)
- EndIf

Construction

- Sorting in C++
  - *This is what I did at least…*

```
#include <algorithm>
// …
ComparePrimitives cmp;
cmp.sort_dim = 0;            // x = 0, y = 1, z = 2
std::sort(objs.begin() + from_index, objs.begin() + to_index, cmp);
```

  - *ComparePrimitives??*

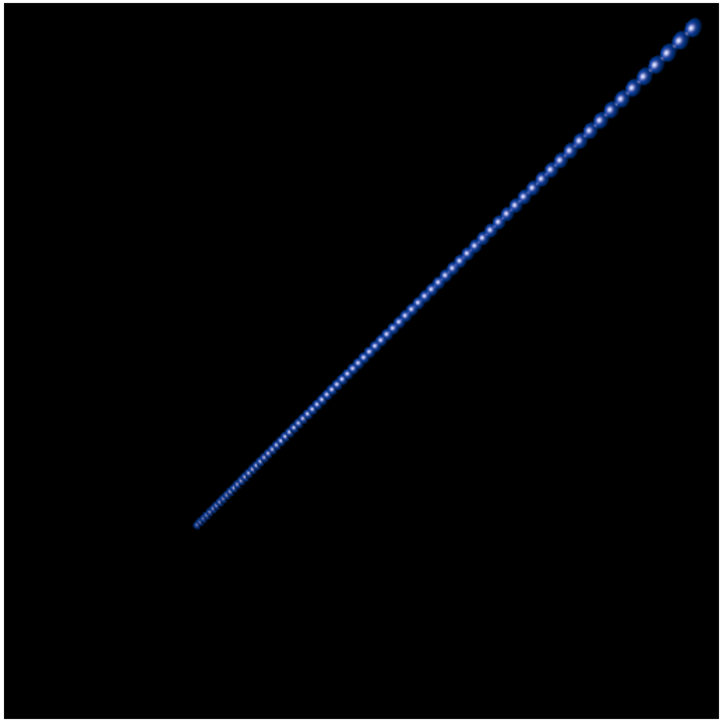
Construction

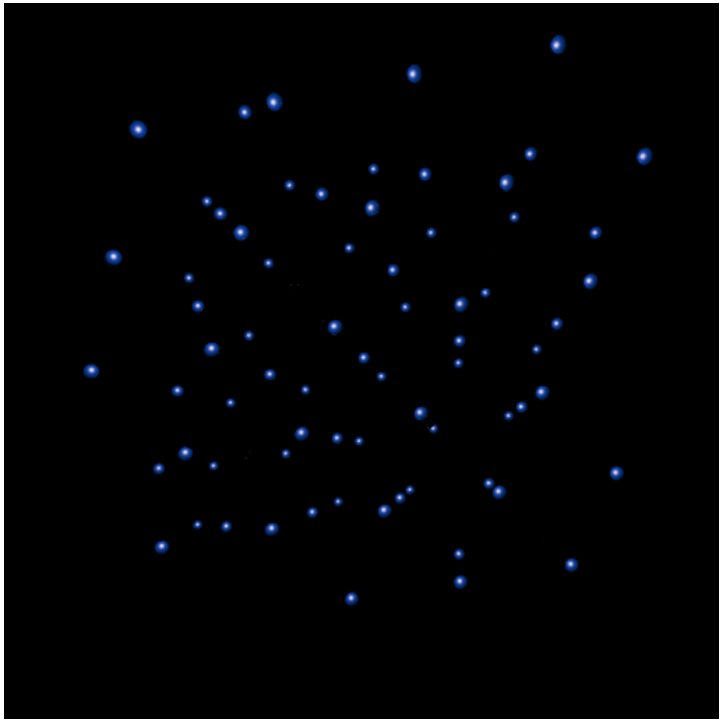
- Sorting in C++

```
class ComparePrimitives {
        public:
        bool operator() (Intersectable *a,  Intersectable *b) {
                AABB box;
                a->getAABB(box);
                float ca = (box.mMax(sort_dim) + box.mMin(sort_dim)) * 0.5f;
                b->getAABB(box);
                float cb = (box.mMax(sort_dim) + box.mMin(sort_dim)) * 0.5f;
                return ca < cb;
        }

        int sort_dim;
};
```

Debug Scenes

Test scenes used to verify your implementation

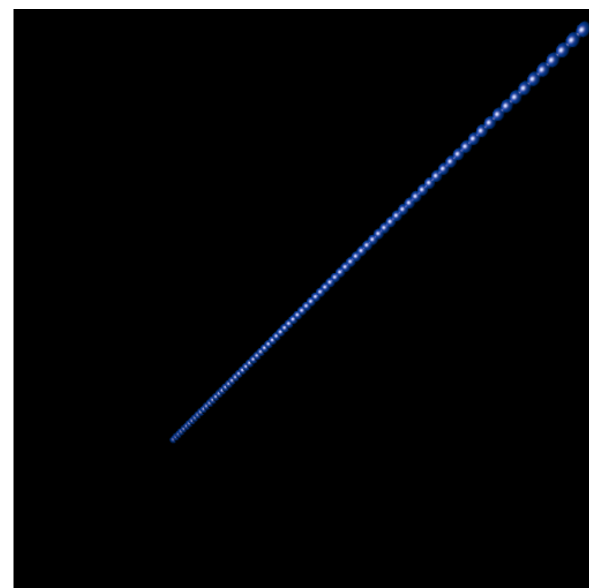

Non-scrambled positions


Scrambled positions

Debug Scenes

```
Node<Primitives: 80>
  Node<Primitives: 40>
   Node<Primitives: 20>
    Node<Primitives: 10>
     Node<Primitives: 5>
      Leaf<Primitives: 3, First primitive: 0>
      Leaf<Primitives: 2, First primitive: 3>
     Node<Primitives: 5>
      Leaf<Primitives: 3, First primitive: 5>
      Leaf<Primitives: 2, First primitive: 8>
    Node<Primitives: 10>
     Node<Primitives: 5>
      Leaf<Primitives: 3, First primitive: 10>
      Leaf<Primitives: 2, First primitive: 13>
     Node<Primitives: 5>
      Leaf<Primitives: 3, First primitive: 15>
      Leaf<Primitives: 2, First primitive: 18>
   Node<Primitives: 20>
    Node<Primitives: 10>
     Node<Primitives: 5>
      Leaf<Primitives: 3, First primitive: 20>
      Leaf<Primitives: 2, First primitive: 23>
     Node<Primitives: 5>
      Leaf<Primitives: 3, First primitive: 25>
      Leaf<Primitives: 2, First primitive: 28>
    Node<Primitives: 10>
     Node<Primitives: 5>
      Leaf<Primitives: 3, First primitive: 30>
      Leaf<Primitives: 2, First primitive: 33>
     Node<Primitives: 5>
      Leaf<Primitives: 3, First primitive: 35>
      Leaf<Primitives: 2, First primitive: 38>
```

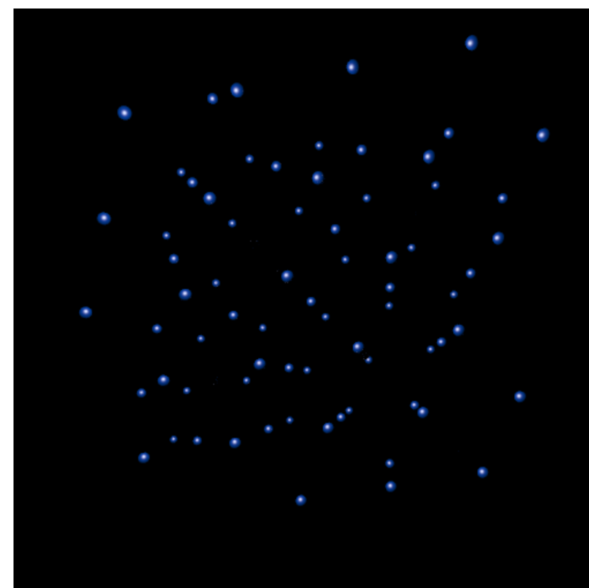
```
Node<Primitives: 40>
  Node<Primitives: 20>
   Node<Primitives: 10>
    Node<Primitives: 5>
     Leaf<Primitives: 3, First primitive: 40>
     Leaf<Primitives: 2, First primitive: 43>
    Node<Primitives: 5>
     Leaf<Primitives: 3, First primitive: 45>
     Leaf<Primitives: 2, First primitive: 48>
   Node<Primitives: 10>
    Node<Primitives: 5>
     Leaf<Primitives: 3, First primitive: 50>
     Leaf<Primitives: 2, First primitive: 53>
    Node<Primitives: 5>
     Leaf<Primitives: 3, First primitive: 55>
     Leaf<Primitives: 2, First primitive: 58>
  Node<Primitives: 20>
   Node<Primitives: 10>
    Node<Primitives: 5>
     Leaf<Primitives: 3, First primitive: 60>
     Leaf<Primitives: 2, First primitive: 63>
    Node<Primitives: 5>
     Leaf<Primitives: 3, First primitive: 65>
     Leaf<Primitives: 2, First primitive: 68>
   Node<Primitives: 10>
    Node<Primitives: 5>
     Leaf<Primitives: 3, First primitive: 70>
     Leaf<Primitives: 2, First primitive: 73>
    Node<Primitives: 5>
     Leaf<Primitives: 3, First primitive: 75>
     Leaf<Primitives: 2, First primitive: 78>
```



Debug Scenes

```
Node<Primitives: 80>
  Node<Primitives: 40>
    Node<Primitives: 21>
      Node<Primitives: 8>
        Leaf<Primitives: 4, First primitive: 0>
        Leaf<Primitives: 4, First primitive: 4>
      Node<Primitives: 13>
        Node<Primitives: 7>
          Leaf<Primitives: 4, First primitive: 8>
          Leaf<Primitives: 3, First primitive: 12>
        Node<Primitives: 6>
          Leaf<Primitives: 3, First primitive: 15>
          Leaf<Primitives: 3, First primitive: 18>
    Node<Primitives: 19>
      Node<Primitives: 7>
        Leaf<Primitives: 4, First primitive: 21>
        Leaf<Primitives: 3, First primitive: 25>
      Node<Primitives: 12>
        Node<Primitives: 6>
          Leaf<Primitives: 3, First primitive: 28>
          Leaf<Primitives: 3, First primitive: 31>
        Node<Primitives: 6>
          Leaf<Primitives: 3, First primitive: 34>
          Leaf<Primitives: 3, First primitive: 37>
```

```
Node<Primitives: 40>
  Node<Primitives: 19>
    Node<Primitives: 6>
      Leaf<Primitives: 3, First primitive: 40>
      Leaf<Primitives: 3, First primitive: 43>
    Node<Primitives: 13>
      Node<Primitives: 6>
        Leaf<Primitives: 4, First primitive: 46>
        Leaf<Primitives: 2, First primitive: 50>
      Node<Primitives: 7>
        Leaf<Primitives: 4, First primitive: 52>
        Leaf<Primitives: 3, First primitive: 56>
  Node<Primitives: 21>
    Node<Primitives: 6>
      Leaf<Primitives: 3, First primitive: 59>
      Leaf<Primitives: 3, First primitive: 62>
    Node<Primitives: 15>
      Node<Primitives: 8>
        Leaf<Primitives: 4, First primitive: 65>
        Leaf<Primitives: 4, First primitive: 69>
      Node<Primitives: 7>
        Leaf<Primitives: 4, First primitive: 73>
        Leaf<Primitives: 3, First primitive: 77>
```



Assignment 2

- Construction
- Intersection
- Surface Area Heuristic (Optional)
- Further Optimizations (Optional)

Intersection

- For this assignment you must implement
  - *Boolean test*

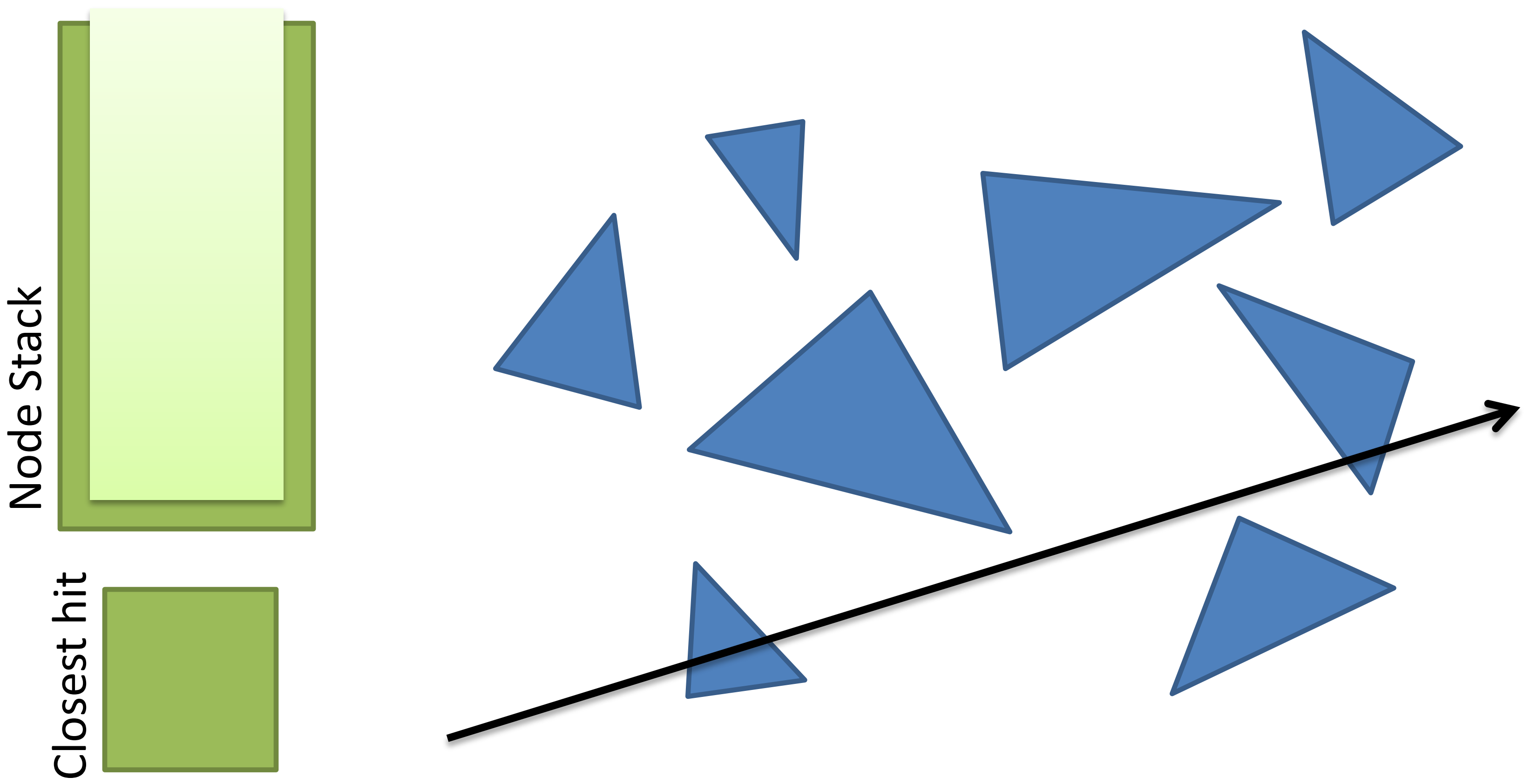
    ```
    bool BVHAccelerator::intersect(const Ray& ray);
    ```

  - *Closest hit*

    ```
    bool BVHAccelerator::intersect(const Ray& ray, Intersection& is);
    ```

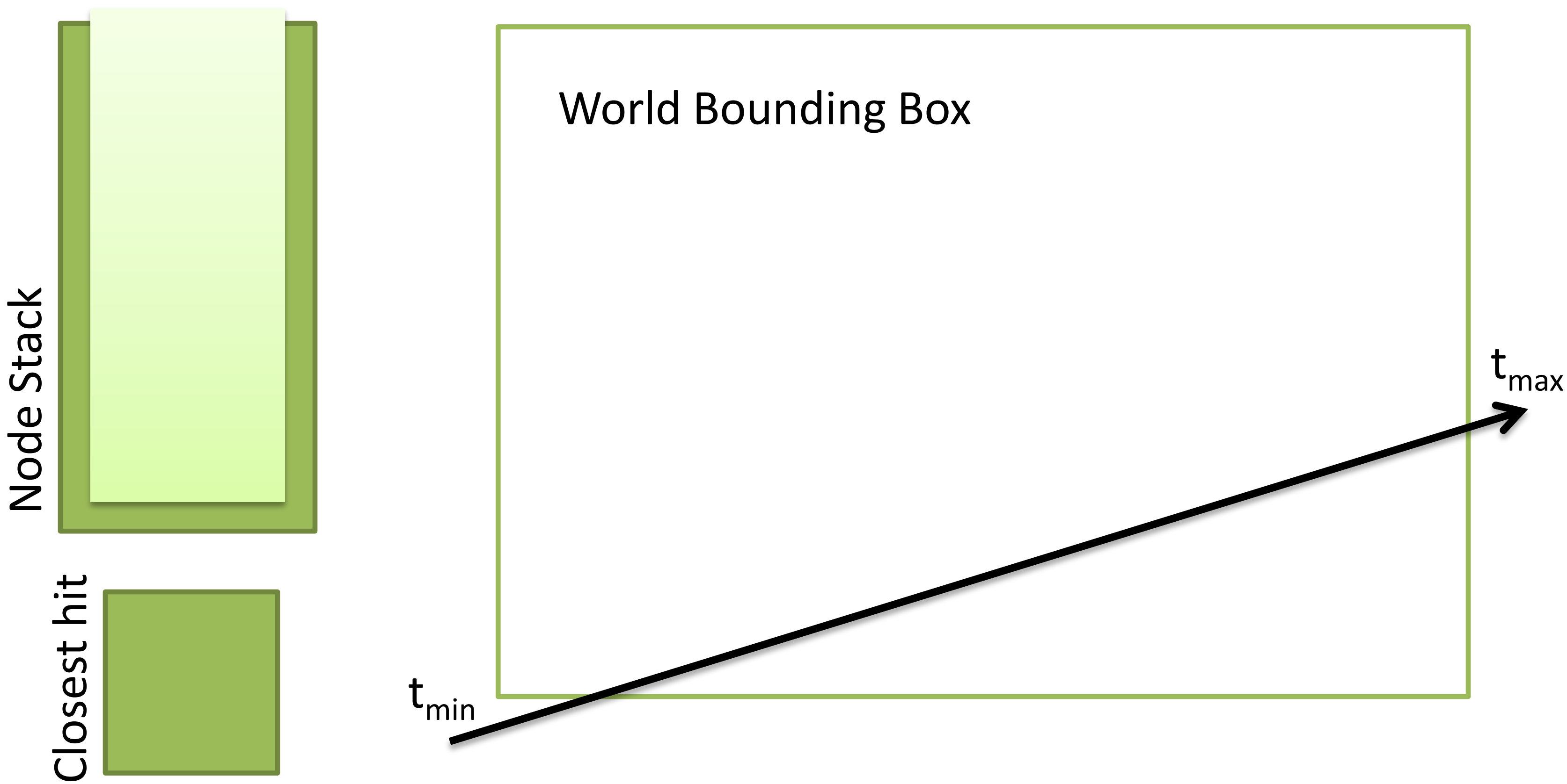
- The two functions are very similar. If you have one of them, you can easily implement the other.

Closest-Hit Intersection



Find closest intersection point

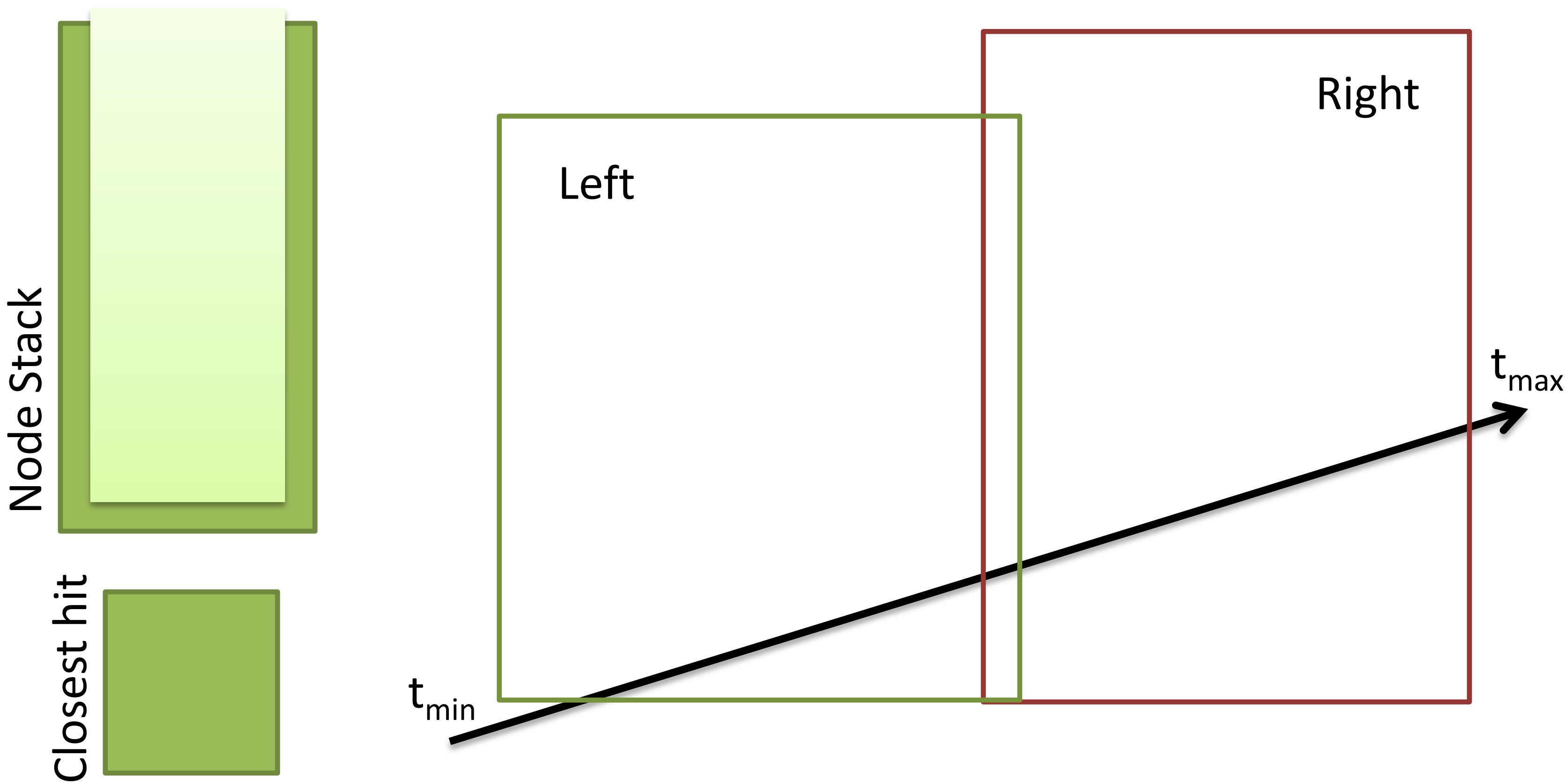
Closest-Hit Intersection



First check if we even hit the world bounding box.

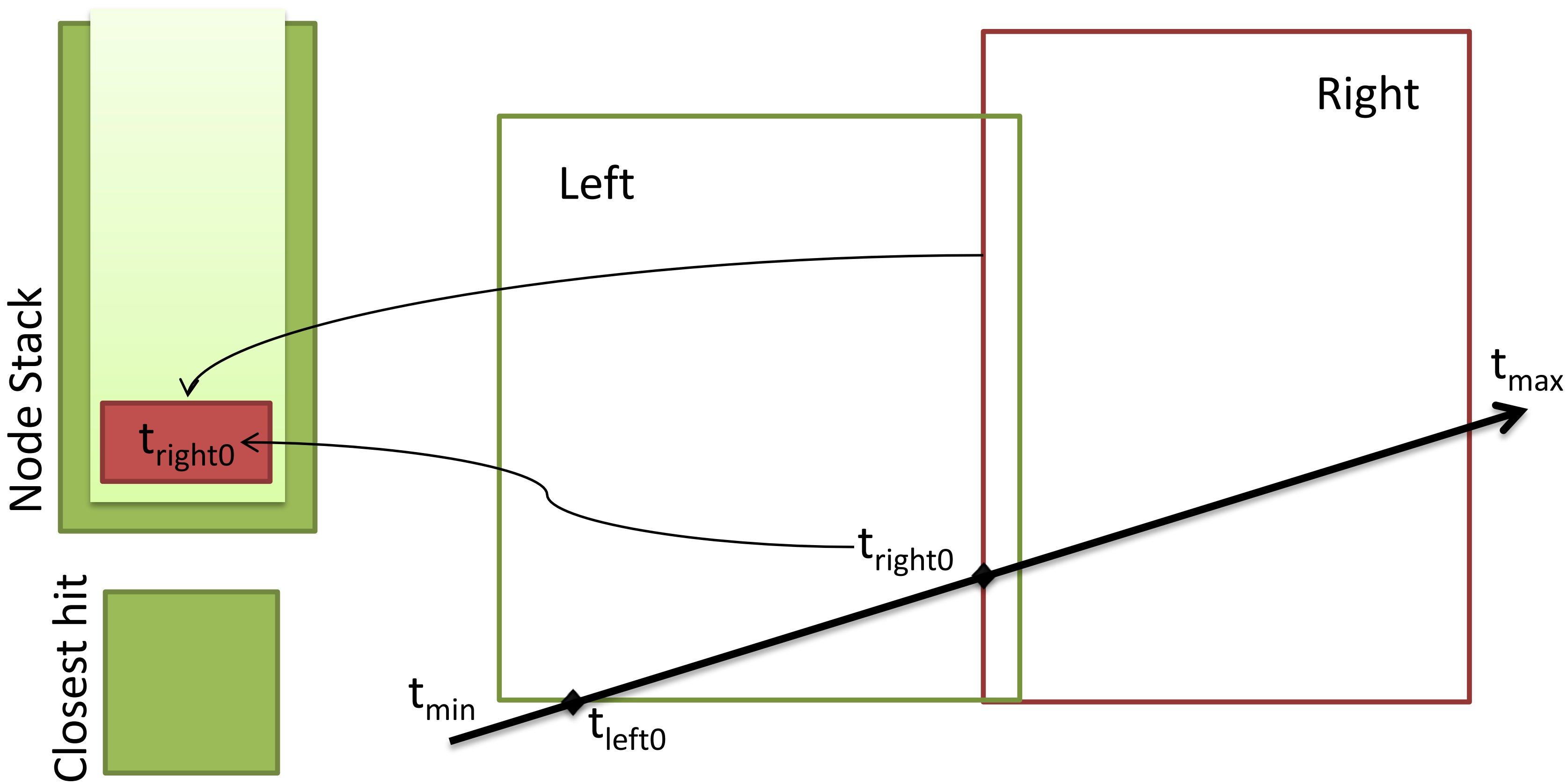
```
bool AABB::intersect(const Ray& r, float& tmin, float& tmax) const;
```

Closest-Hit Intersection



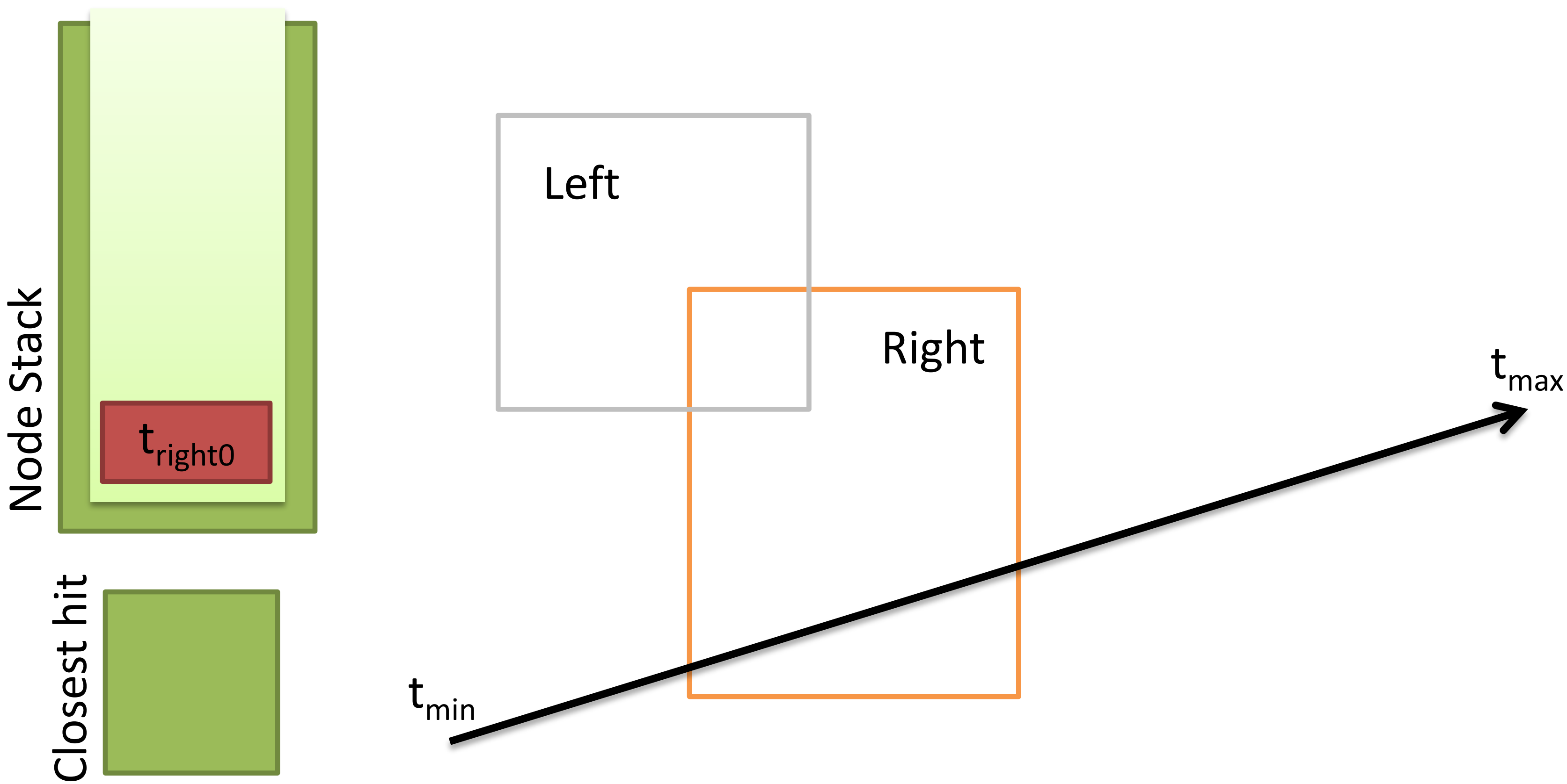
Check the two children for intersection (again using **AABB::intersect(…)**). In this case, both boxes were hit.

Closest-Hit Intersection



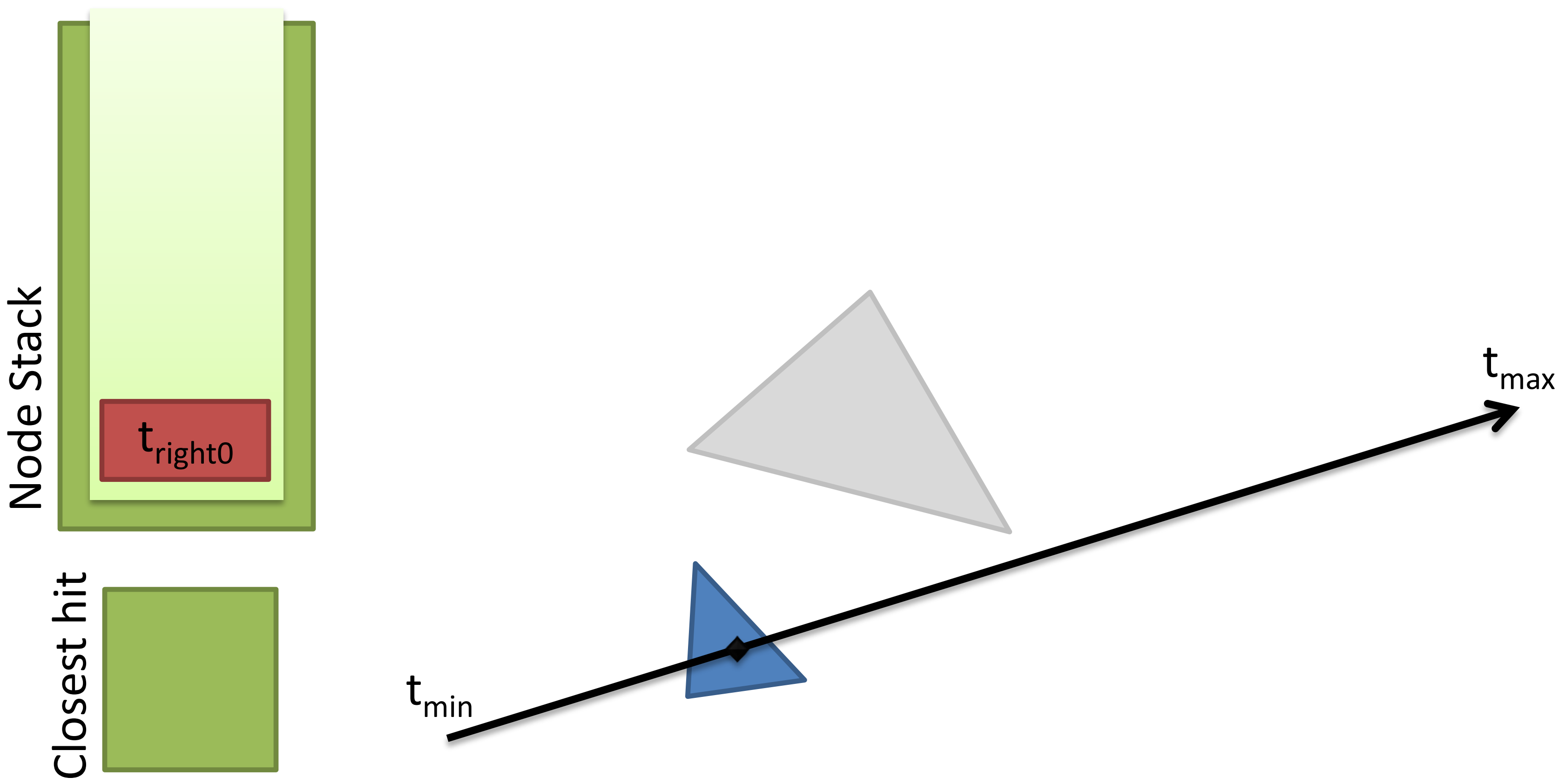
Put the node furthest away on the stack along with it's hit parameter *t*. Traverse the closest node

Closest-Hit Intersection



This time we only hit one node, which happens to be a *leaf node*

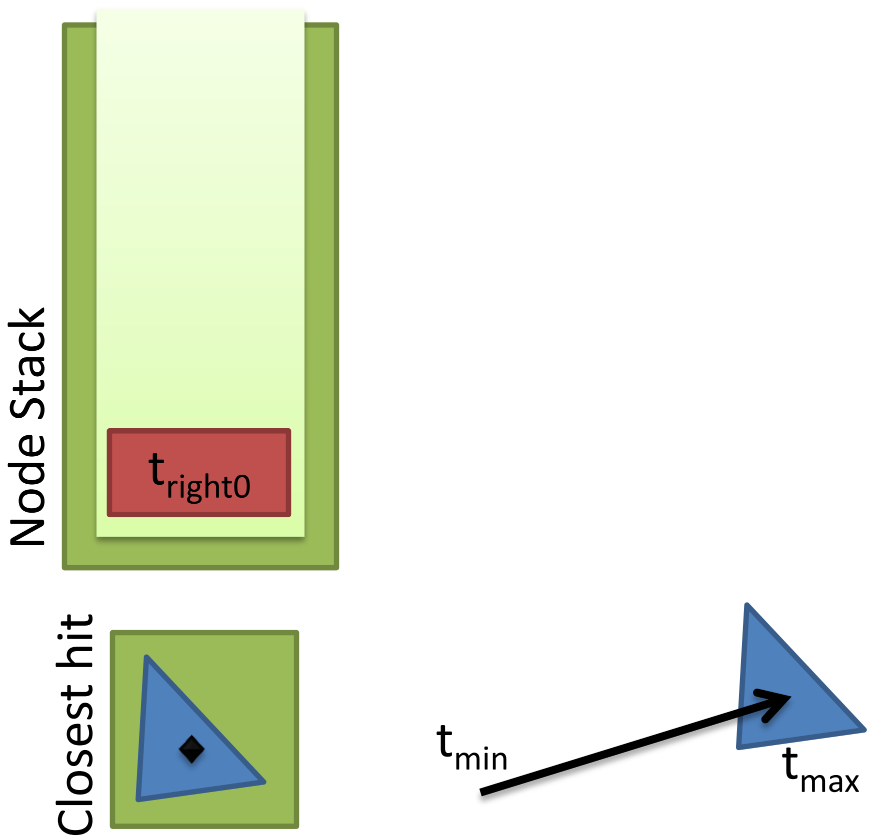
Closest-Hit Intersection



Intersection test with each primitive in the leaf.

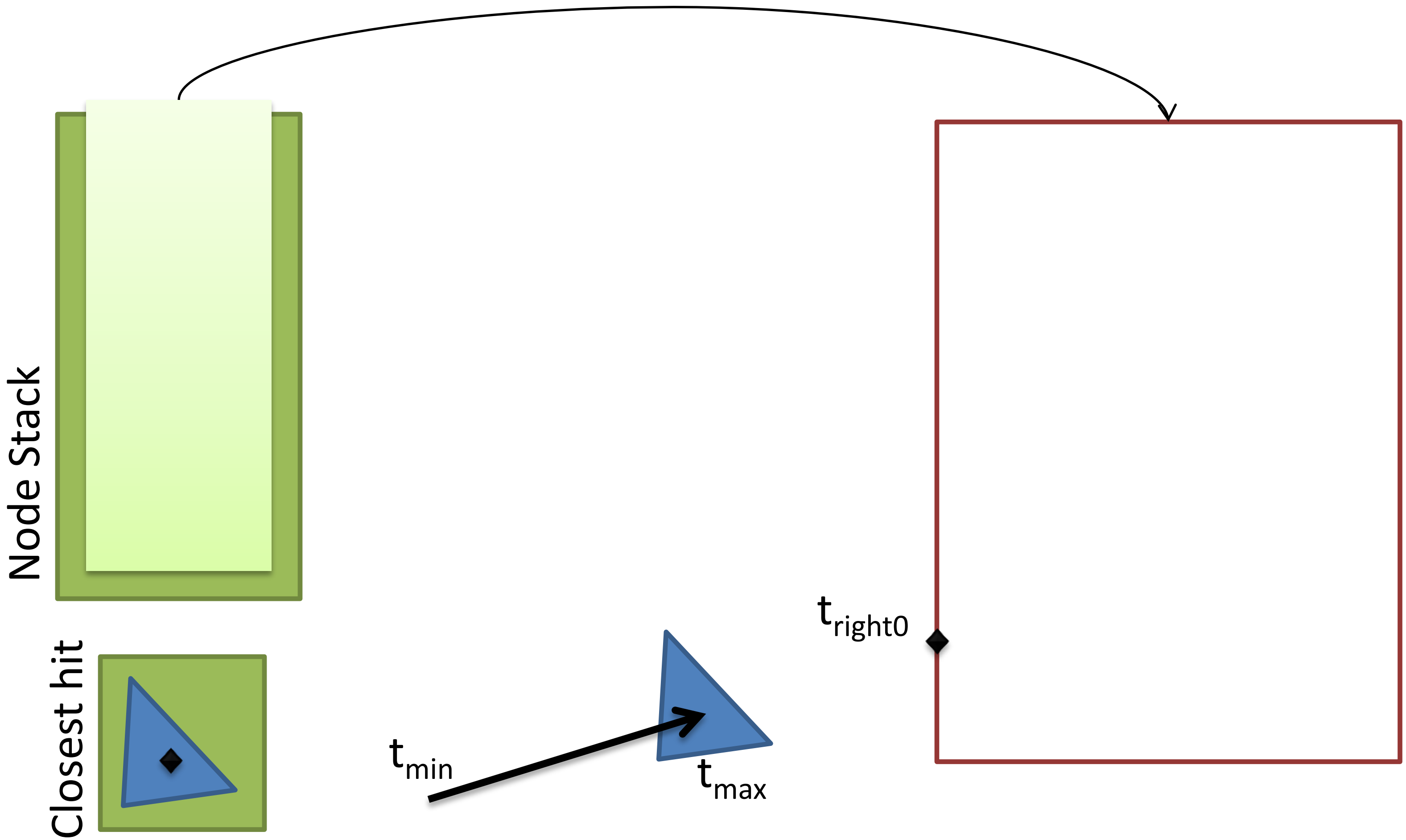
```
bool Intersectable::intersect(const Ray& ray, Intersection& is) const;
```

Closest-Hit Intersection



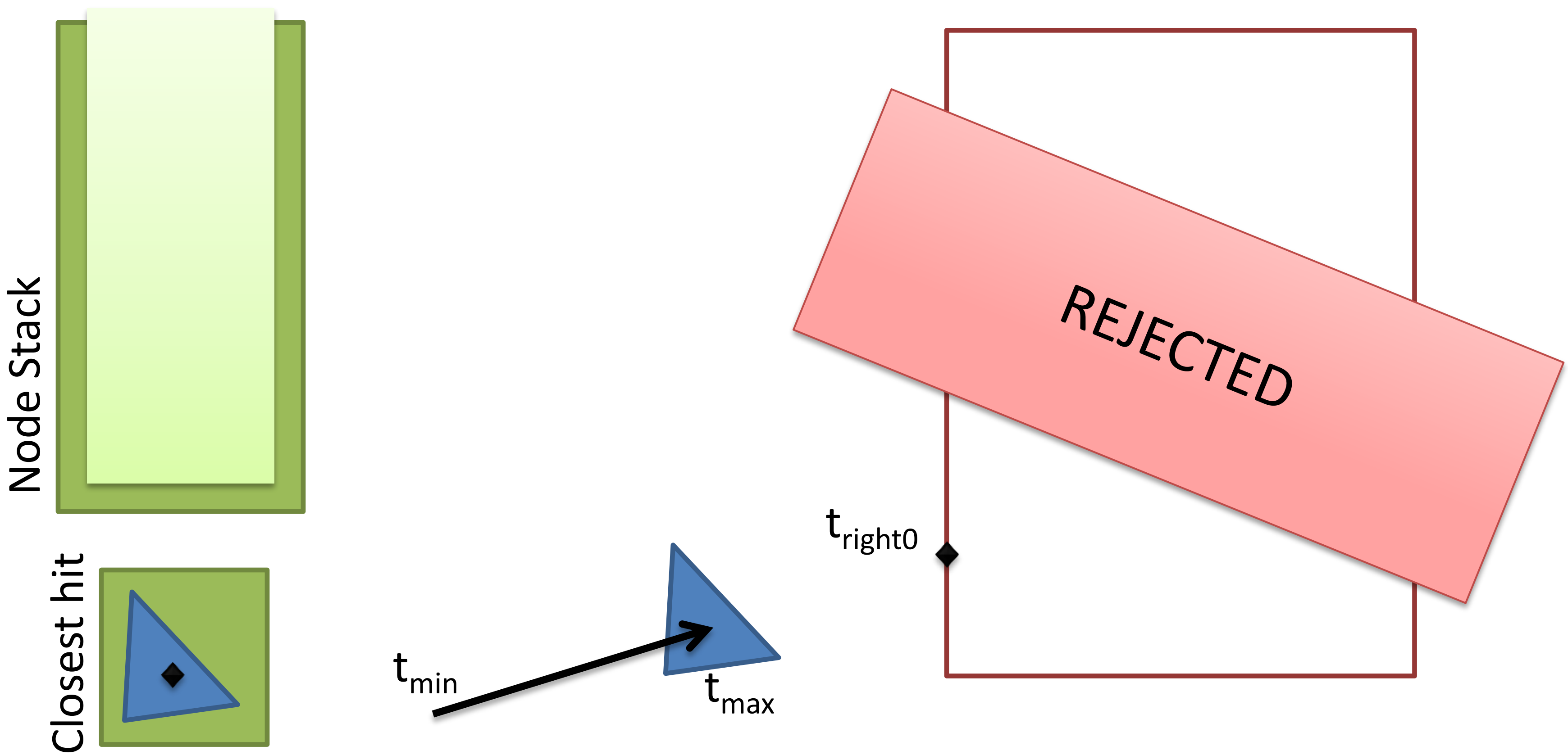
Store intersection and shorten ray.

Closest-Hit Intersection



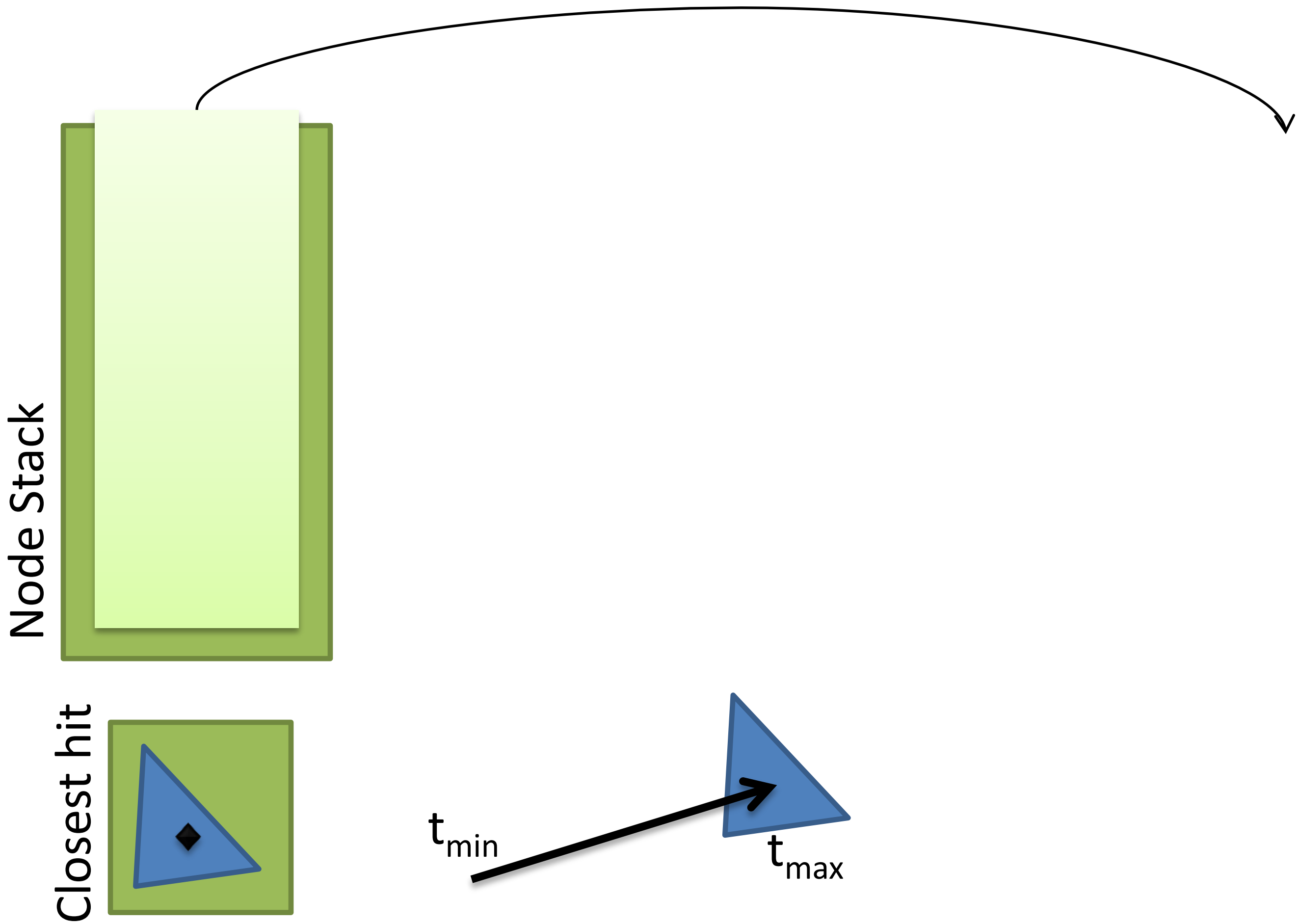
Pop the stack and recursively intersection test with the node.

Closest-Hit Intersection



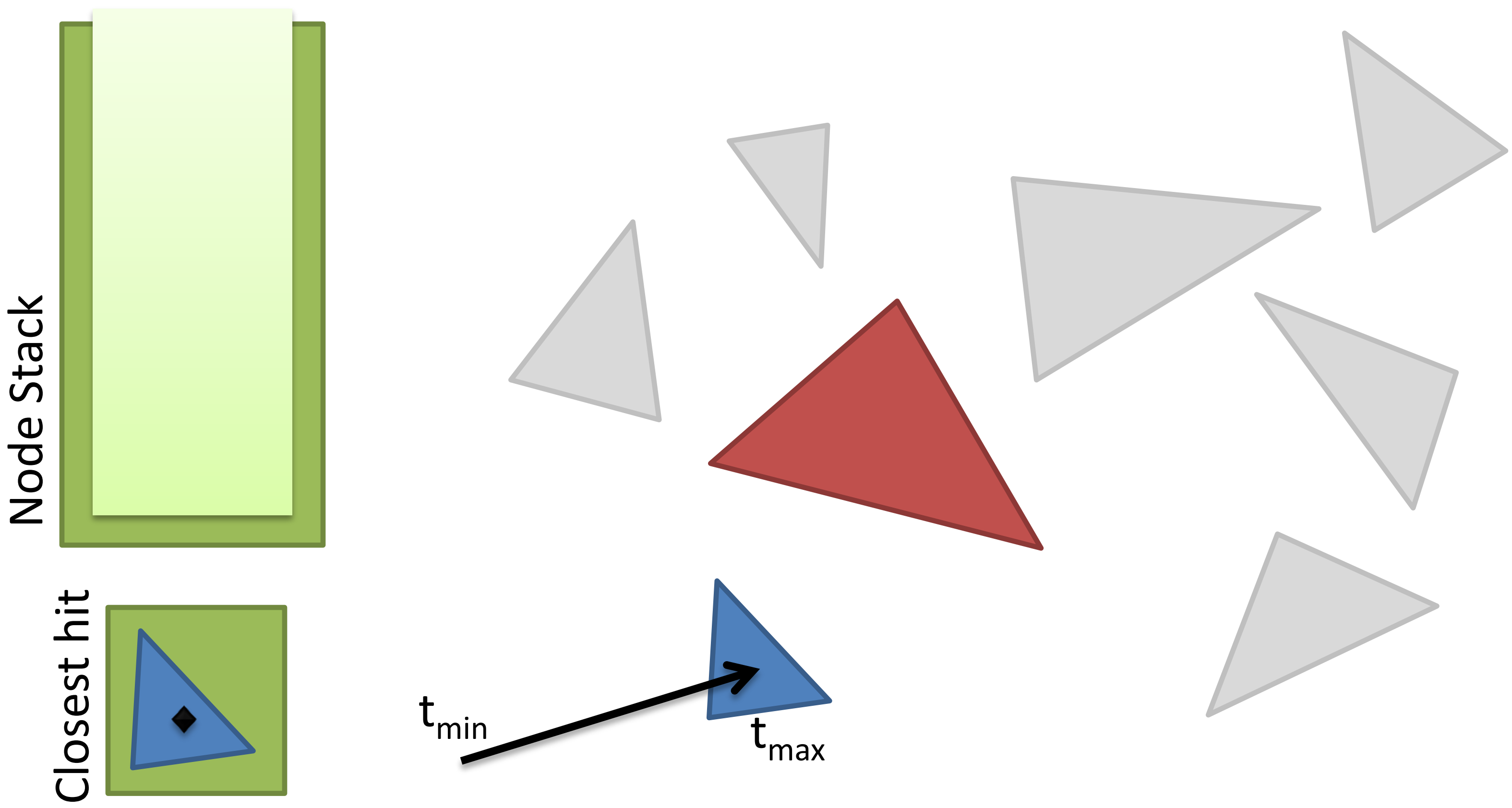
Optimization – We can trivially reject the pop'd node since its *t*-value is now further away than t_{max} of the ray.

Closest-Hit Intersection



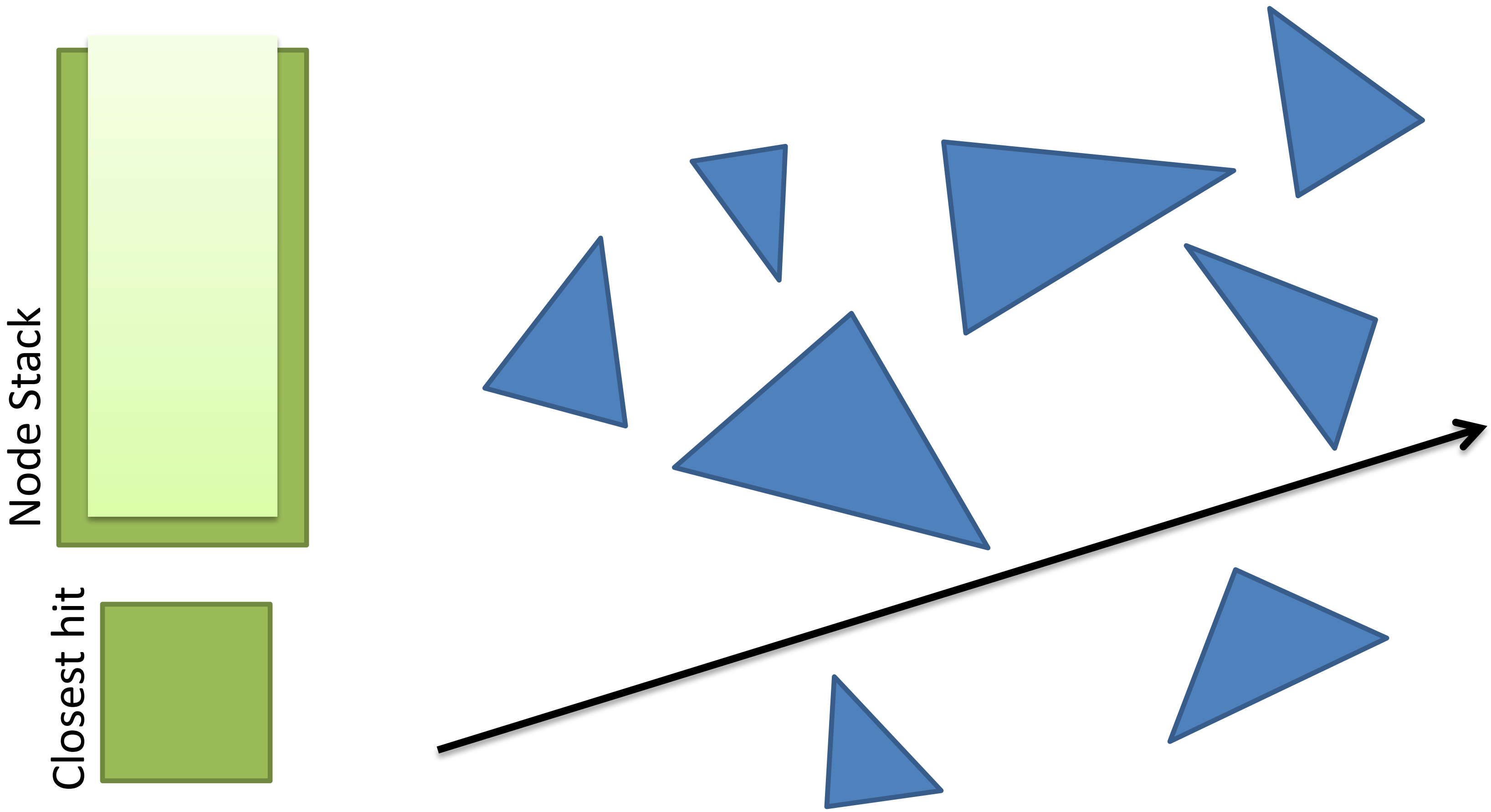
Try to pop the stack again to fetch the next node… but now it's empty, which means we're done!

Closest-Hit Intersection



We found the closest hit with little effort!

No Intersection



If there is no intersection, the *Closest hit* will of course be empty – return *false*

Closest-Hit Intersection, Pseudo code

- LocalRay = Ray, CurrentNode = Root
- Check LocalRay intersection with Root (world box)
  - No hit => return *false*
- For (infinity)
  - If (***NOT*** CurrentNode.isLeaf())
    - Intersection test with both child nodes
      - Both nodes hit => Put the one furthest away on the stack. *CurrentNode* = *closest* node
        - » continue
      - Only one node hit => *CurrentNode = hit* node
        - » continue
      - No Hit: Do nothing (let the stack-popping code below be reached)
  - Else // Is leaf
    - For each primitive in leaf perform intersection testing
      - Intersected => update *LocalRay.maxT* and store *ClosestHit*
  - EndIf
  - Pop stack until you find a node with *t* < *LocalRay.maxT* => *CurrentNode* = pop'd
    - Stack is empty? => return *ClosestHit* (no closest hit => return false, otherwise return true
- EndFor

Intersection

- Stack element

```
struct StackItem {
        BVHNode         *ptr;
        float           t;
};
```

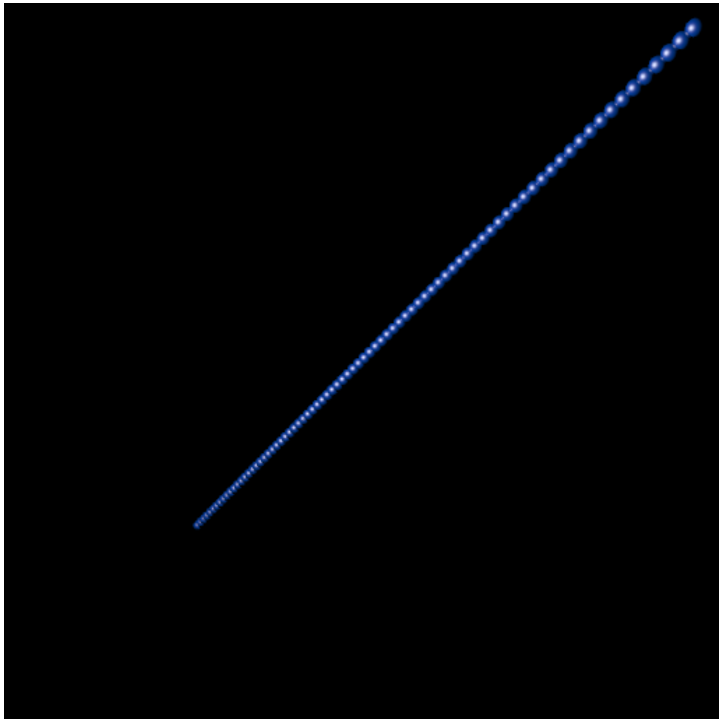
- Use either a C-style vector or C++ Stack class
  - StackItem stack[MAX_STACK_DEPTH];
  - Stack<StackItem> stack;

Boolean Intersection, Pseudo code

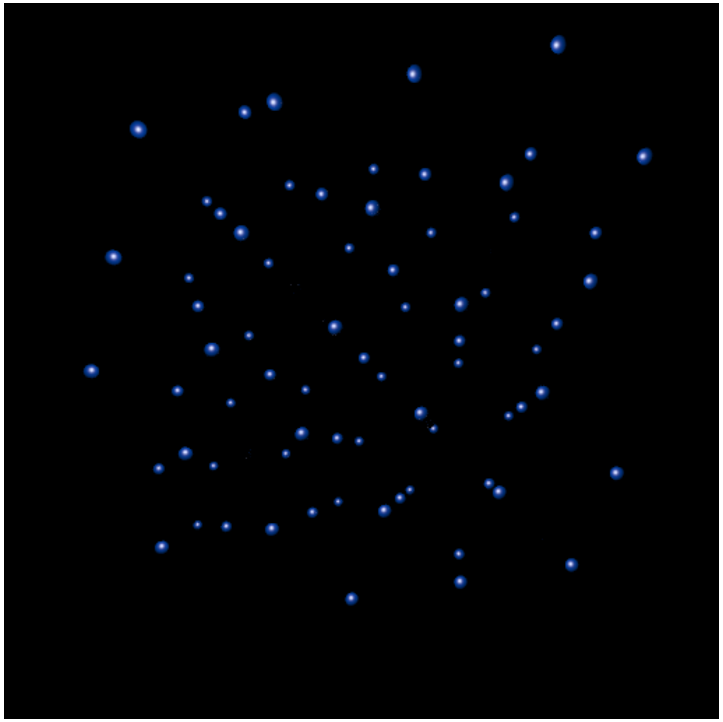
- LocalRay = Ray, CurrentNode = Root
- Check LocalRay intersection with Root (world box)
  - No hit => return *false*
- For (infinity)
  - If (***NOT*** CurrentNode.isLeaf())
    - Intersection test with both child nodes
      - Both nodes hit => Put **right** one on the stack. *CurrentNode* = ***left*** node
        - » Goto LOOP;
      - Only one node hit => *CurrentNode = hit* node
        - » Goto LOOP;
      - No Hit: Do nothing (let the stack-popping code below be reached)
  - Else // Is leaf
    - For each primitive in leaf perform intersection testing
      - Intersected => **return true;**
  - EndIf
  - Pop stack, *CurrentNode* = pop'd node
    - Stack is empty => **return false**
- EndFor

Debug Scenes

Make these scenes work first…



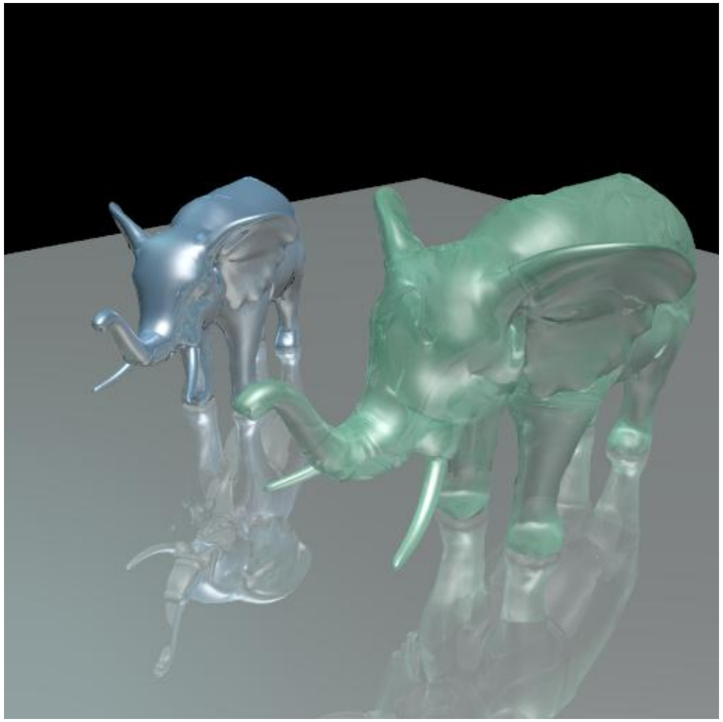
Non-scrambled positions



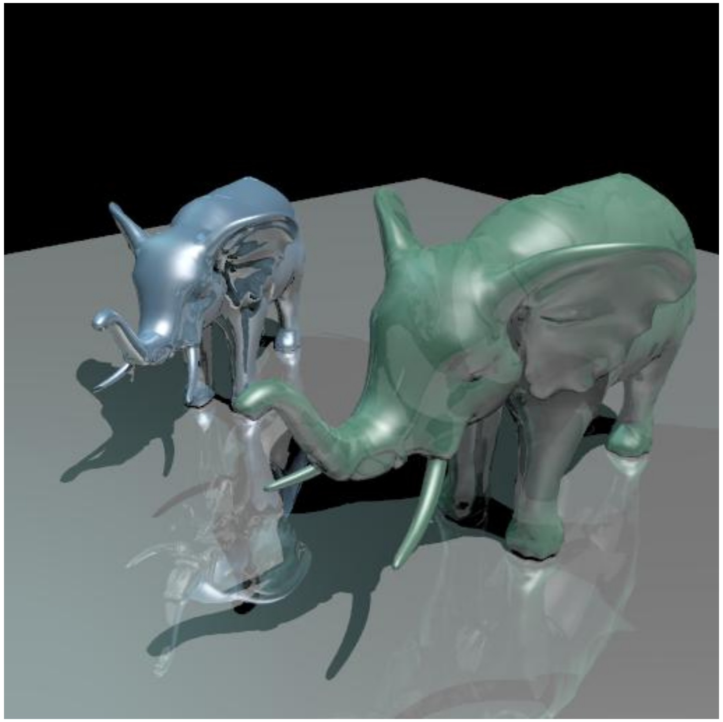
Scrambled positions

Intersection

Elephants, without and with shadows



Without shadows



With shadows

Assignment 2

- Construction
- Intersection
- Surface Area Heuristic (Optional)
- Further Optimizations (Optional)

Surface Area Heuristic

$$c = c_t + \frac{S(B_l)}{S(B_p)} n_l c_i + \frac{S(B_r)}{S(B_p)} n_r c_i$$

- c = estimated cost of traversing *p* and its children (*l*, *r*)
- c_t = ~cost of performing one traversal iteration
- c_i = ~cost of performing one intersection test
- $n_{l,\, r}$ = number of elements in child node
- $S(B_{l,\, r})$ = surface area of child node
- $S(B_p)$ = surface area of parent node

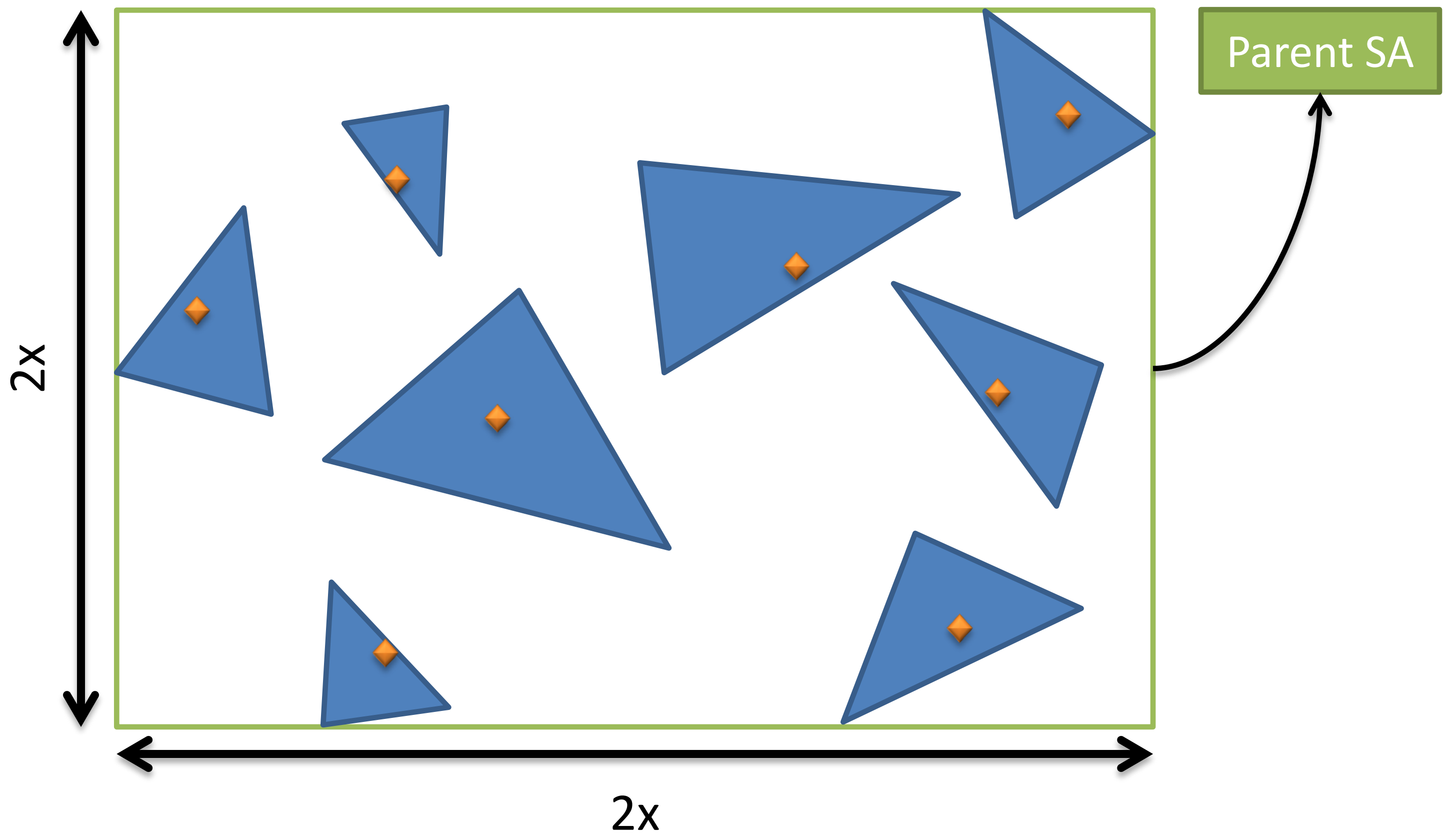
Surface Area Heuristic

Continue to split if $c < n_p c_i$

- c = estimated cost of traversing p and its children (l, r)
- c_i = ~cost of performing one intersection test
- n_p = number of elements in parent node

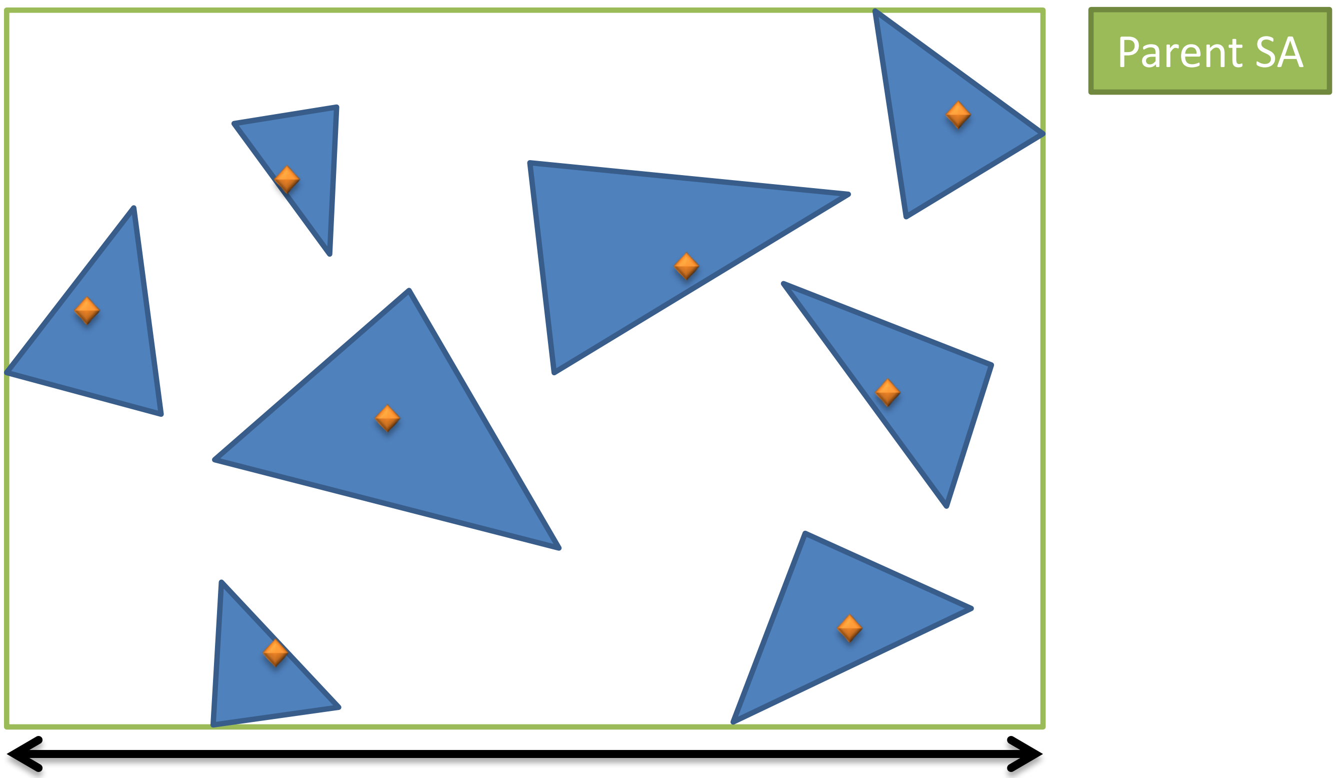
We stop splitting and create a leaf when it's cheaper to intersect all the *Intersectables*, than to split the node further.

Surface Area Heuristic



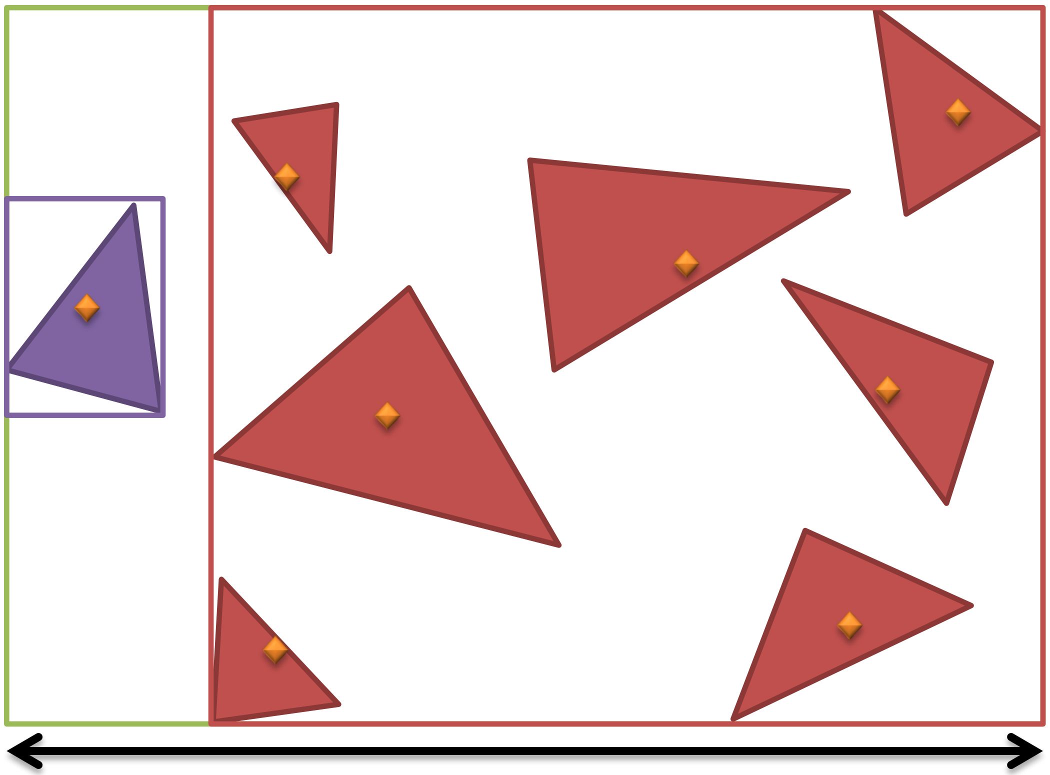
The parent surface area is passed by previous recursion iteration.

Surface Area Heuristic



For each axis. Begin by sorting elements. Then calculate cost *c* of each potential split

Surface Area Heuristic



$$c = c_t + \frac{S(B_l)}{S(B_p)} n_l c_i + \frac{S(B_r)}{S(B_p)} n_r c_i$$

$n_l = 1$

$n_r = 7$

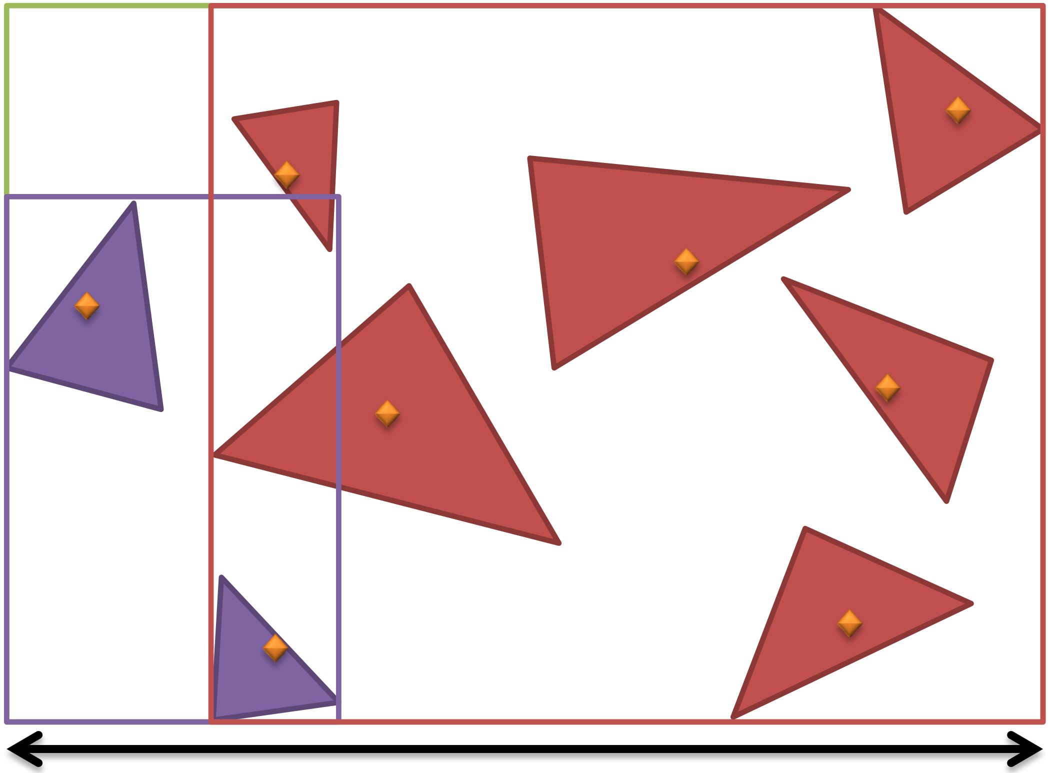
$S(B_l) =$ Left SA

$S(B_r) =$ Right SA

$S(B_p) =$ Parent SA

For each axis. Begin by sorting elements. Then calculate cost *c* of each potential split

Surface Area Heuristic



$$c = c_t + \frac{S(B_l)}{S(B_p)} n_l c_i + \frac{S(B_r)}{S(B_p)} n_r c_i$$

$n_l = 2$

$n_r = 6$

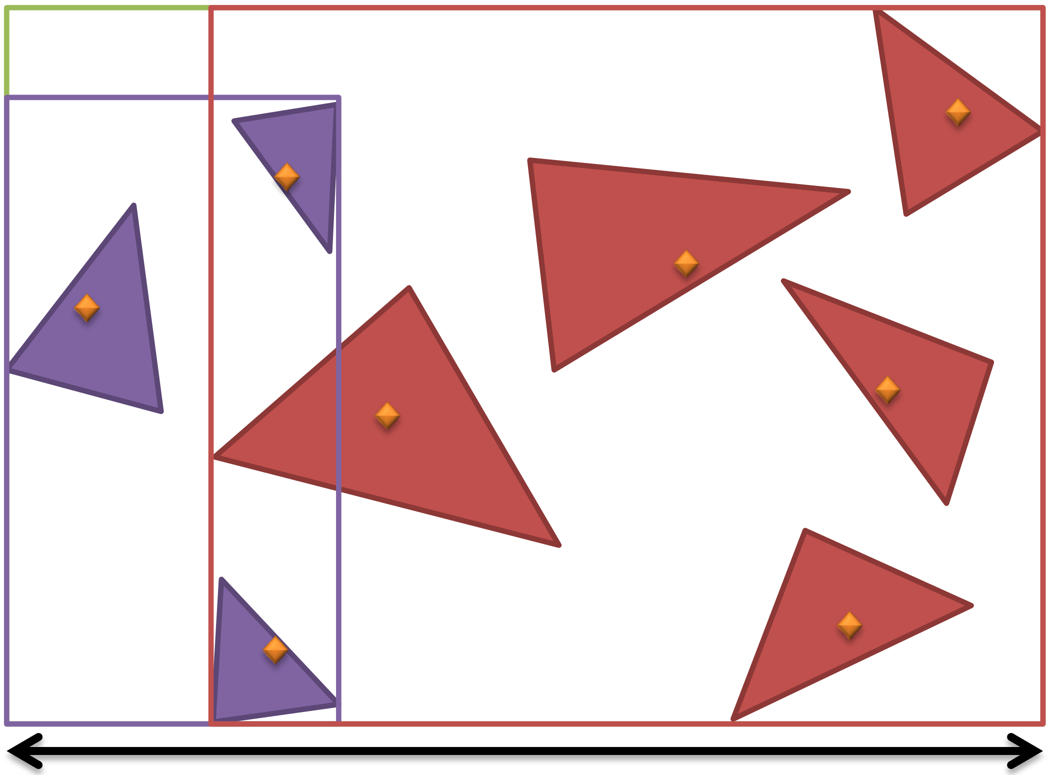
$S(B_l) =$ Left SA

$S(B_r) =$ Right SA

$S(B_p) =$ Parent SA

For each axis. Begin by sorting elements. Then calculate cost *c* of each potential split

Surface Area Heuristic



$$c = c_t + \frac{S(B_l)}{S(B_p)} n_l c_i + \frac{S(B_r)}{S(B_p)} n_r c_i$$

$n_l = 3$

$n_r = 5$

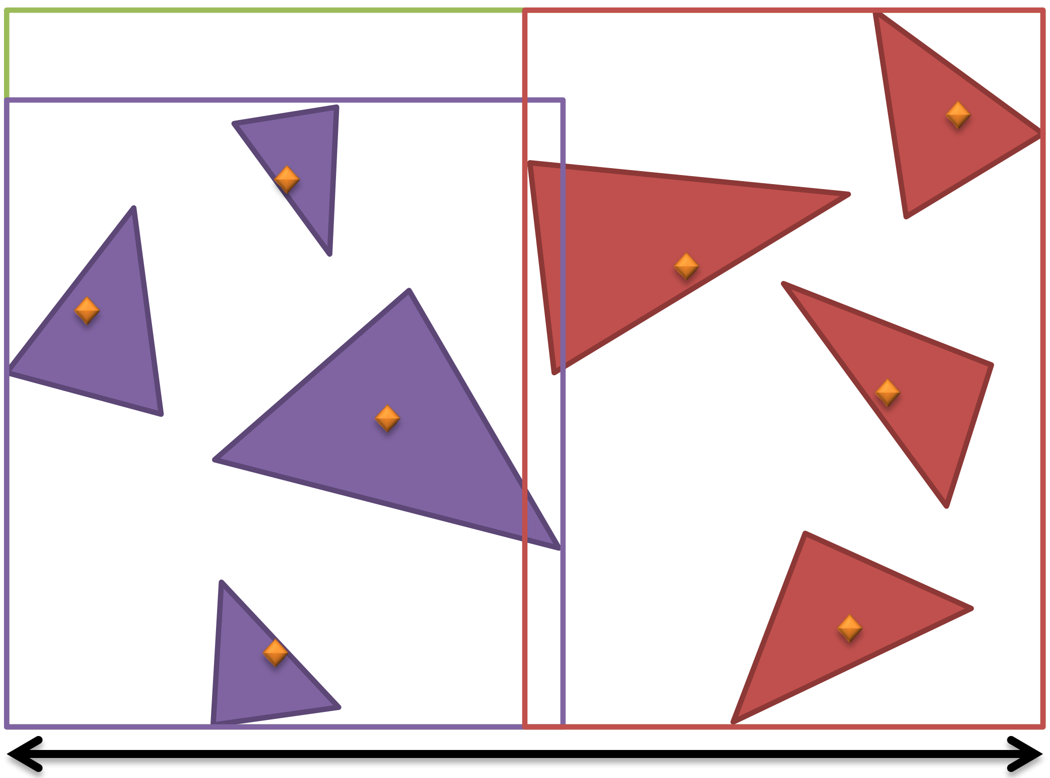
$S(B_l) =$ Left SA

$S(B_r) =$ Right SA

$S(B_p) =$ Parent SA

For each axis. Begin by sorting elements. Then calculate cost c of each potential split

Surface Area Heuristic



$$c = c_t + \frac{S(B_l)}{S(B_p)} n_l c_i + \frac{S(B_r)}{S(B_p)} n_r c_i$$

$n_l = 4$

$n_r = 4$

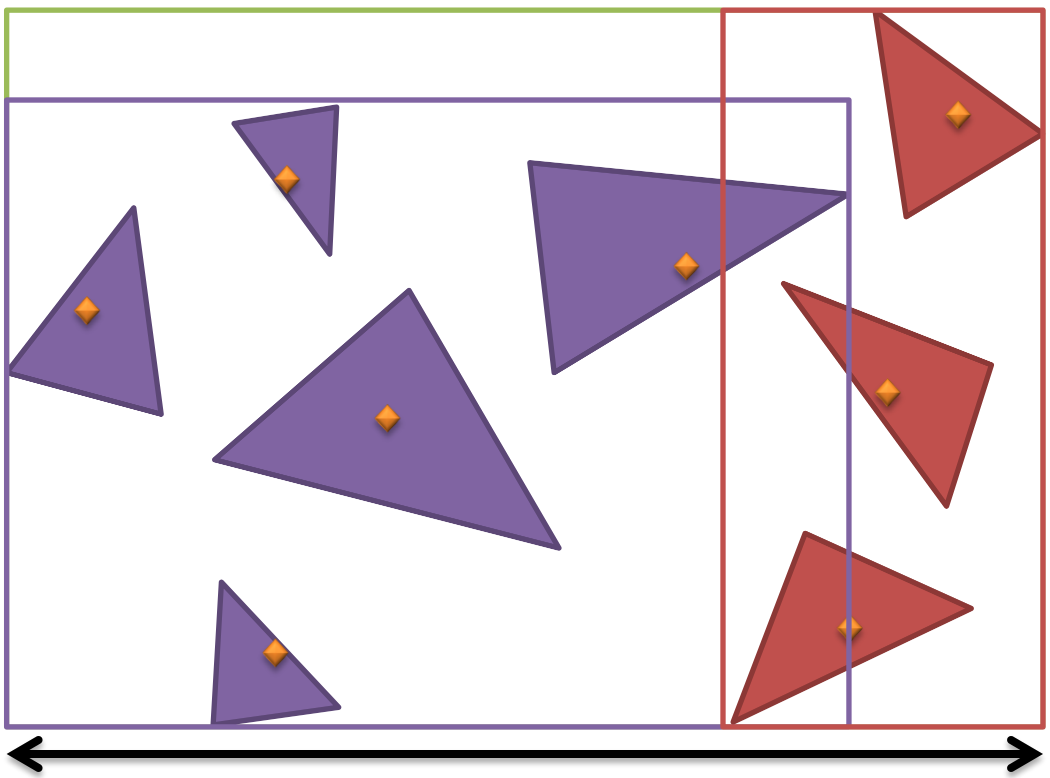
$S(B_l) =$ Left SA

$S(B_r) =$ Right SA

$S(B_p) =$ Parent SA

For each axis. Begin by sorting elements. Then calculate cost *c* of each potential split

Surface Area Heuristic



$$c = c_t + \frac{S(B_l)}{S(B_p)} n_l c_i + \frac{S(B_r)}{S(B_p)} n_r c_i$$

$n_l = 5$

$n_r = 3$

$S(B_l) =$ Left SA

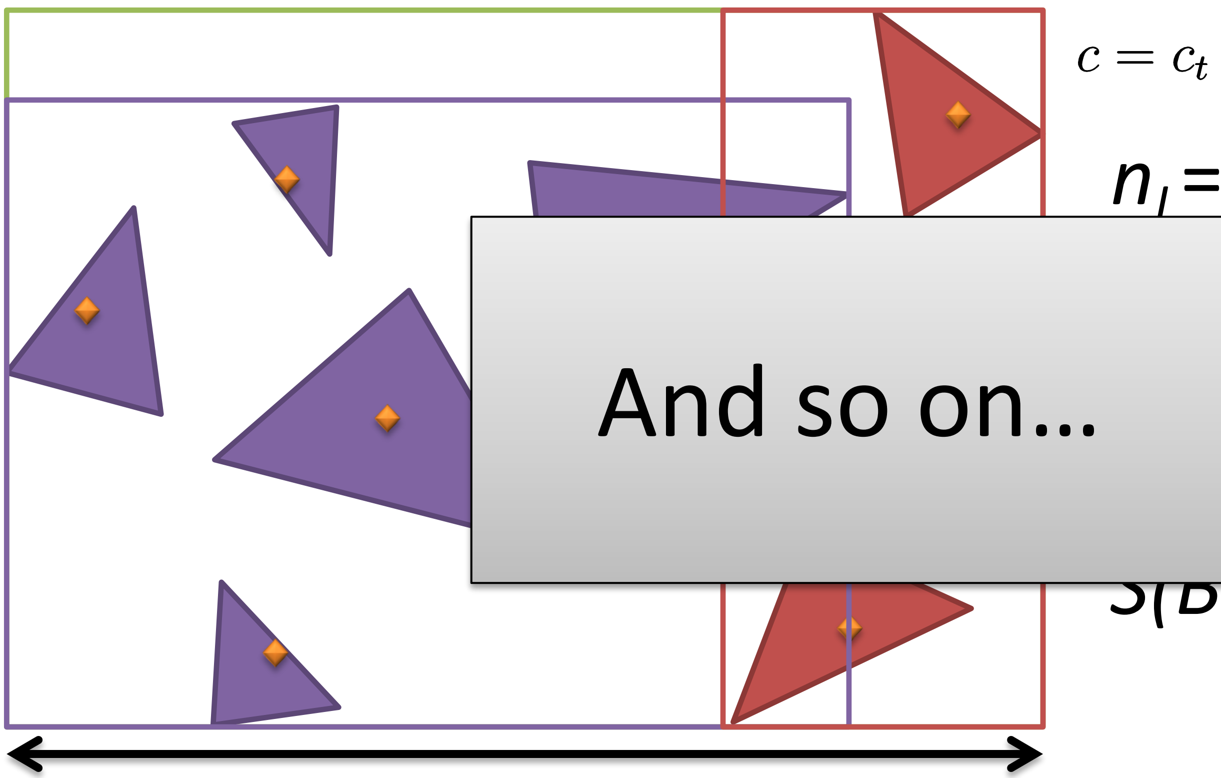
$S(B_r) =$ Right SA

$S(B_p) =$ Parent SA

For each axis. Begin by sorting elements. Then calculate cost *c* of each potential split

Surface Area Heuristic

$$c = c_t + \frac{S(B_l)}{S(B_p)} n_l c_i + \frac{S(B_r)}{S(B_p)} n_r c_i$$



$n_l = 5$

3

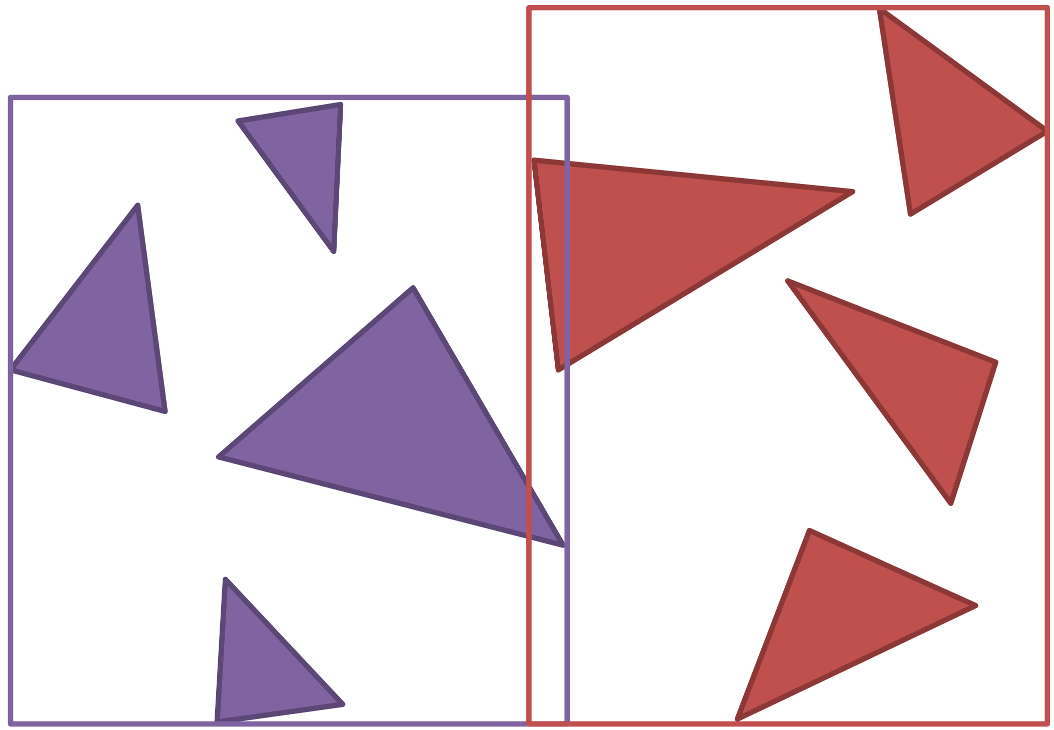
Left SA

Right SA

Parent SA

For each axis. Begin by sorting elements. Then calculate cost *c* of each potential split

Surface Area Heuristic



Keep the best split (lowest *c*) over all three axes.

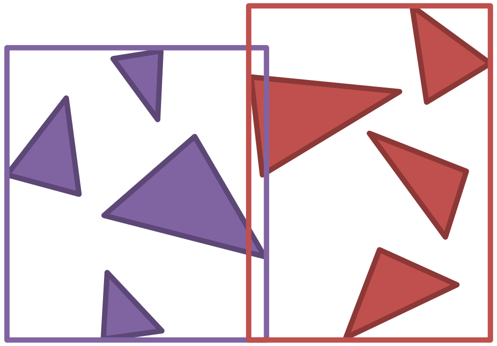
Continue splitting for as long as it pays off ($c < n_p c_i$)

Surface Area Heuristic

Can be slow for large scenes...

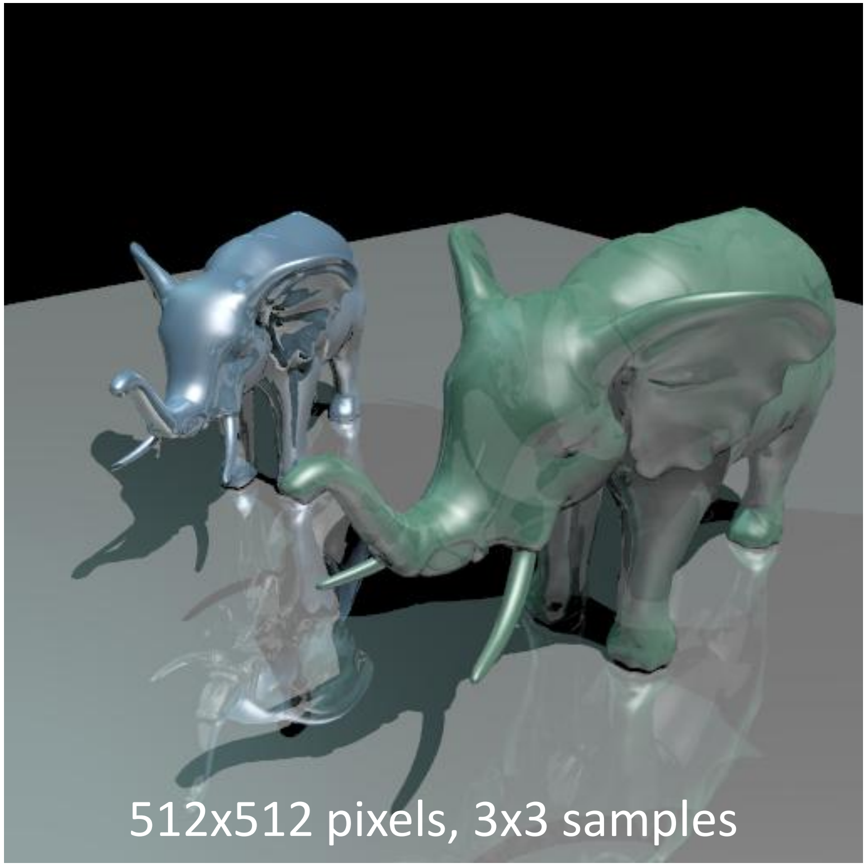
Optimize:

- Binned sorting
- Try only a few split planes
- Try selectively enabling/disabling SAH calculation at different levels
- Etc, etc...



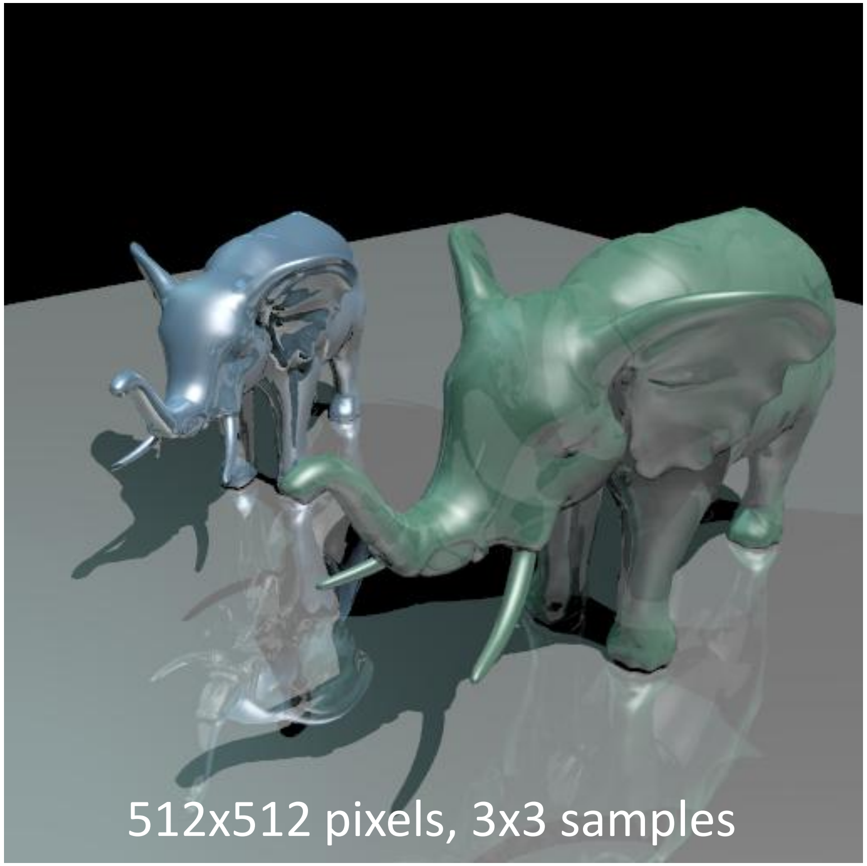
Results

Mid-point split



60 s

SAH



49 s

Assignment 2

- Construction
- Intersection
- Surface Area Heuristic (Optional)
- Further Optimizations (Optional)

Further Optimizations (Optional)

- Take a look inside

  `bool AABB::intersect(const Ray &r, float &tmin, float &tmax) const;`

  - *There is one expensive computation that can be pre-computed…*
  - *Loop unrolling* **may** *help slightly...*
- The *BVHNode* class takes 36 bytes (if your implementation matches mine). This can be reduced (and more nicely aligned)
- Make sure that your <<*Intersectable*>>-intersection functions are optimized
- Avoid recursion and pay careful attention to inner loops
- …be creative! (and/or use Google “BVH Construction” ;)

  - There are other “hard-core” optimizations as well which are beyond the scope of this course..

Further Optimizations (Optional)

Typical work distribution for a BVH

CS:EIP	Symbol + Offset	Timer samples
▷ 0x211210	AABB::intersectFast	54,09
▷ 0x21c6f0	Triangle::intersect	9,39
▷ 0x212140	BVHAccelerator::intersect	8,22
▷ 0x21c450	Triangle::intersect	5,66
▷ 0x211090	AABB::include	5,32
▷ 0x211fb0	BVHAccelerator::intersect	5,3
▷ 0x21d570	WhittedTracer::trace	1,71
▷ 0x21cf70	Triangle::calculateNormalDifferential	1,66
▷ 0x21cc10	Triangle::getAABB	1,17
▷ 0x212920	std::_Unguarded_partition<Intersect...	1,06
▷ 0x215200	Intersection::getReflectedRay	0,83
▷ 0x213fb0	Diffuse::evalBRDF	0,64
▷ 0x212f80	std::_Insertion_sort1<Intersectable *...	0,61
▷ 0x215030	Intersection::calculatePositionDifferen...	0,54
▷ 0x215570	Intersection::getRefractedRay	0,47
▷ 0x213450	Ray::Ray	0,41
▷ 0x213900	Camera::getRay	0,41
▷ 0x2113b0	Intersection::Intersection	0,38
▷ 0x214f80	Ray::Ray	0,37

That is all.

The second assignment is out now!

As usual, we’ll be active on the forum, so be sure to check in if you have any comments or questions!

Fin