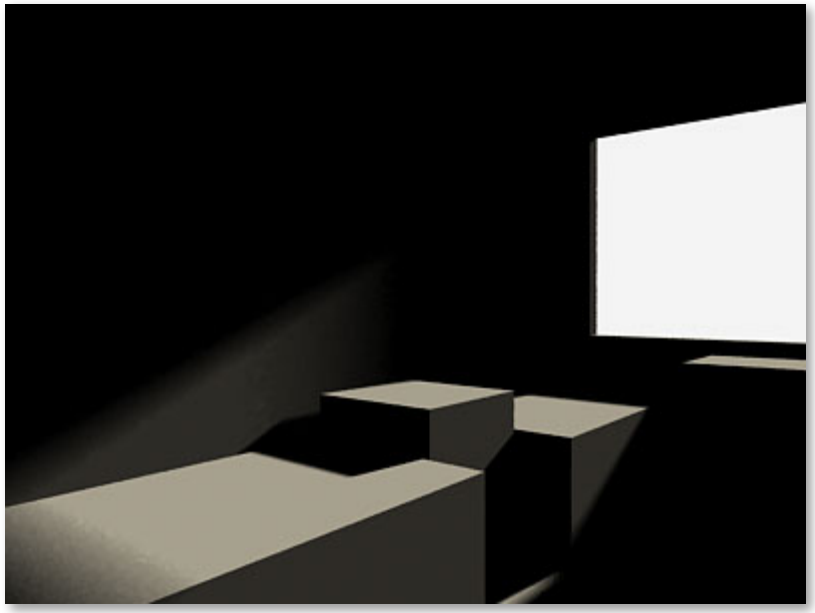
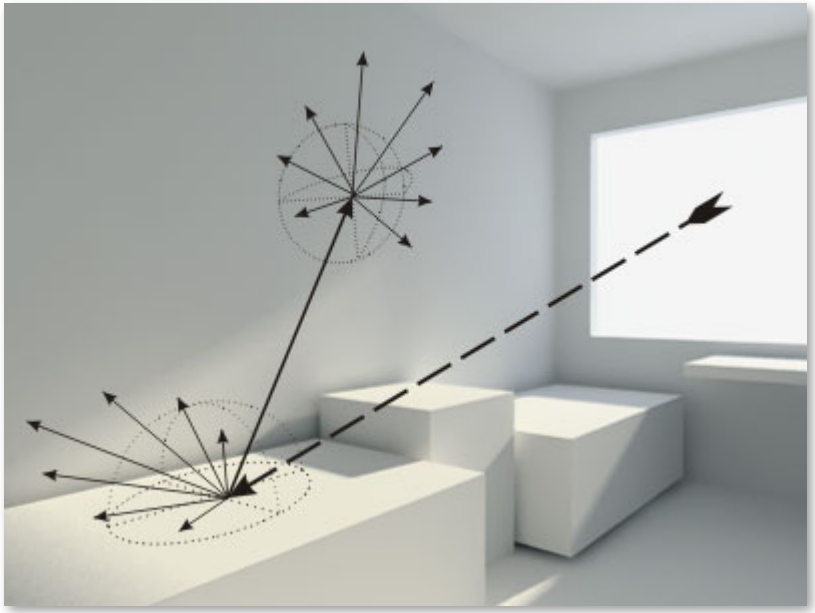
# Ray-tracing
# (Global Illumination part 1)

# Direct and indirect lighting





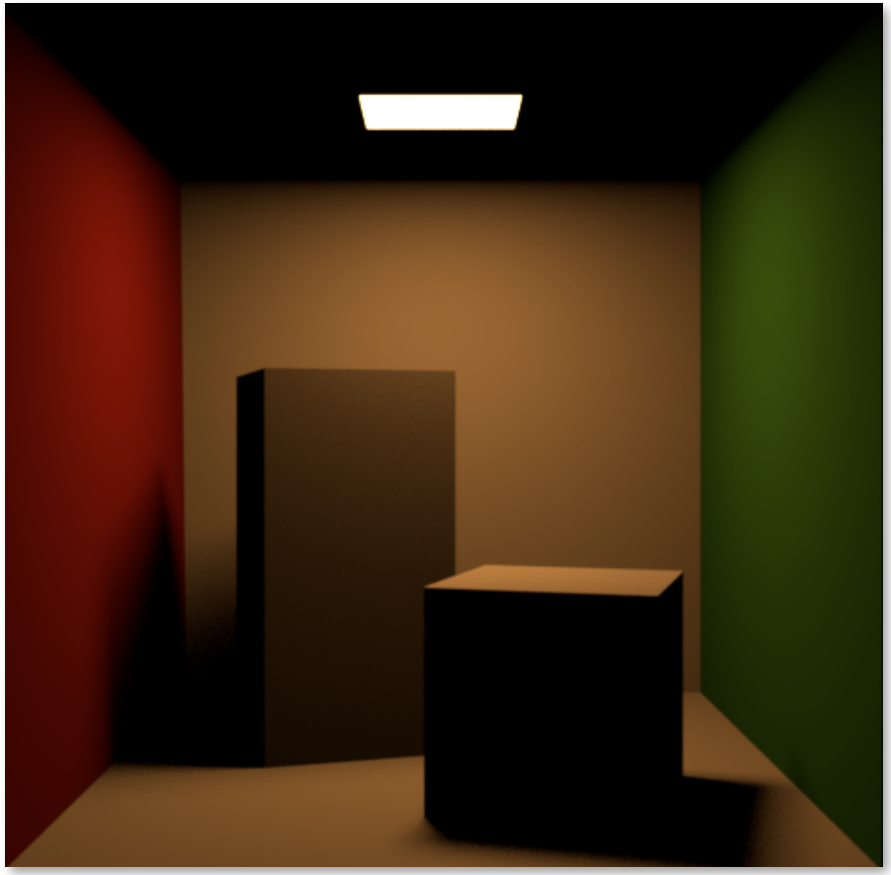
Direct :

local properties

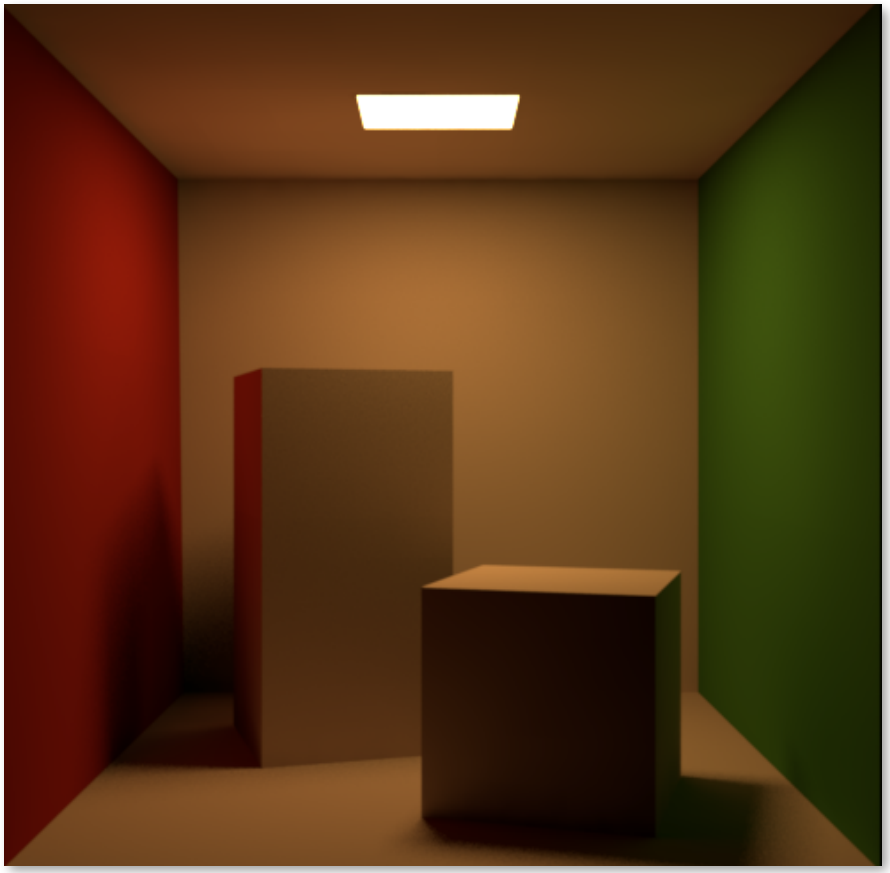
Indirect :

global problem

# Direct and indirect lighting

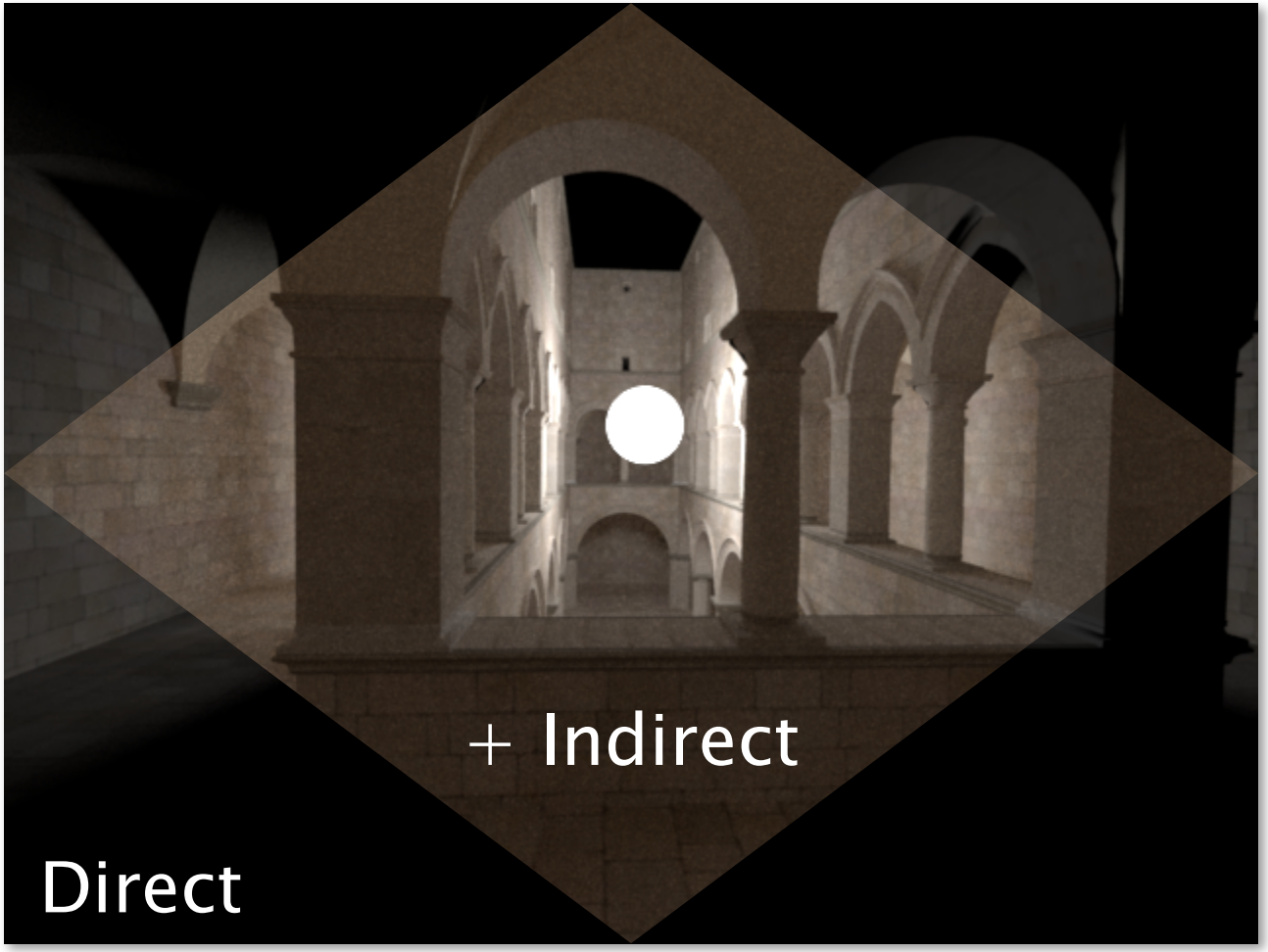


Direct



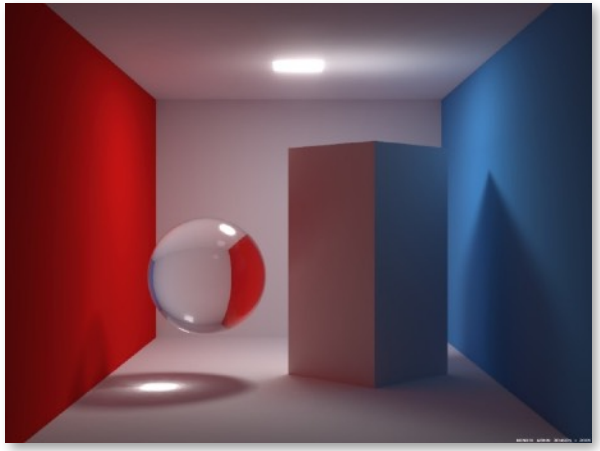
+ Indirect

# Direct and indirect lighting

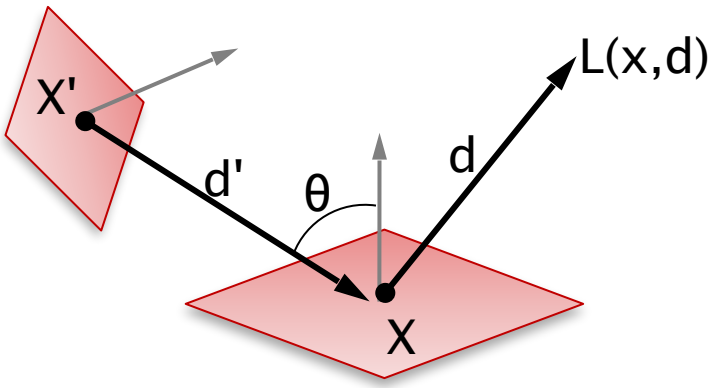


# Global illumination

- Interactions between objects
- Light transport
- Reflections, refraction, diffusion
- Energy Conservation (for light)



# The rendering equation




Radiance unit: Watt/m²/sr

$$L(x,d) = E(x,d) + \int \rho(x,d,d') \, v(x,x') \, L(x',d') \, G(x,x') \, dA$$

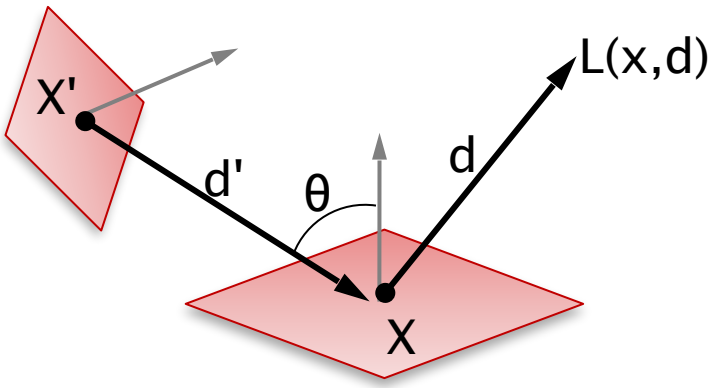
Radiance leaving point x

in the direction d

# Radiance?

- Radiant flux per unit solid angle per unit projected area:
  - $W \cdot sr^{-1} \cdot m^{-2}$
- Invariant along a given ray:
  - $L(x, x \rightarrow y) = L(y, y \leftarrow x)$
  - No attenuation with distance
  - Easier for computations
  - Exception for point light sources

# The rendering equation



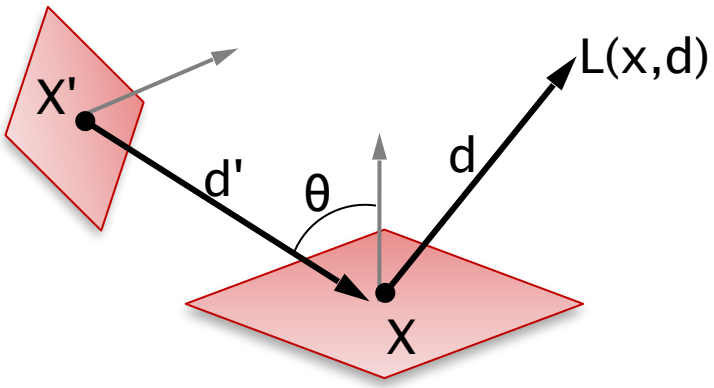

Radiance unit: Watt/m²/sr

$$L(x,d) = E(x,d) + \int \rho(x,d,d') \, v(x,x') \, L(x',d') \, G(x,x') \, dA$$

Radiance emitted from x:

non-zero only if x belongs to a light source

# The rendering equation

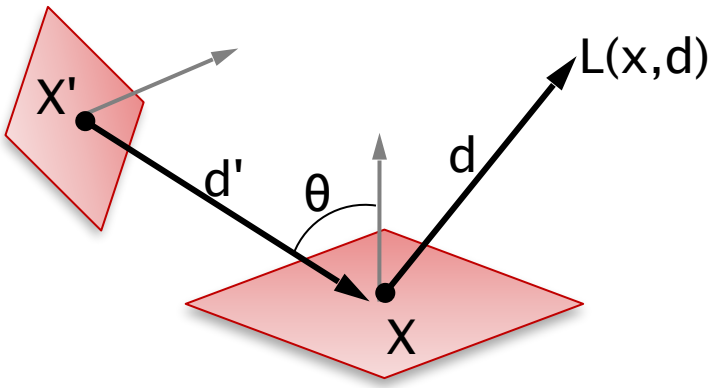



$$L(x,d) = E(x,d) + \int \rho(x,d,d') \, v(x,x') \, L(x',d') \, G(x,x') \, dA$$

Integrating the contribution from all the surfaces

Radiance unit: Watt/m²/sr

# The rendering equation



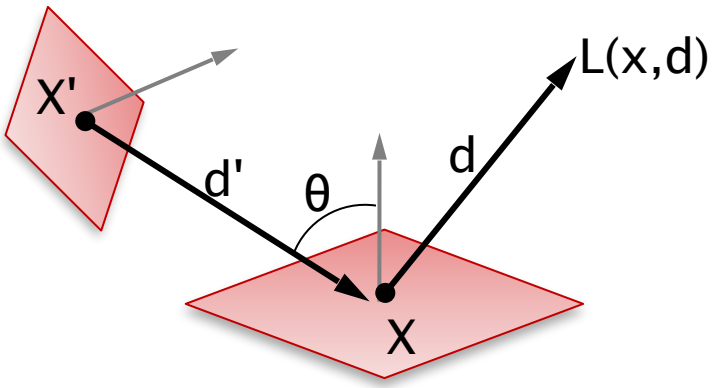

$$L(x,d) = E(x,d) + \int \rho(x,d,d') \, v(x,x') \, L(x',d') \, G(x,x') \, dA$$

Incoming Radiance

from point x' in the direction d'

Radiance unit: Watt/m²/sr

# The rendering equation

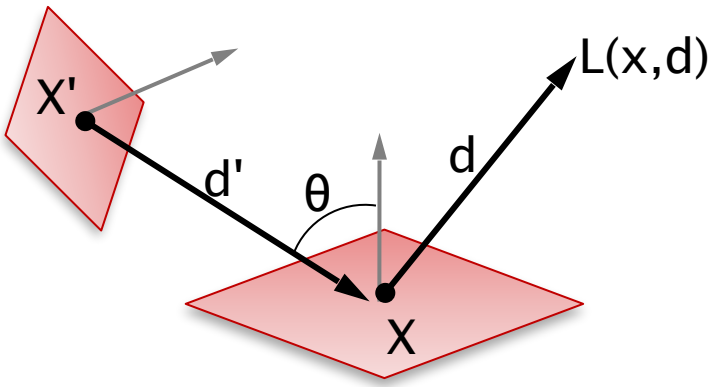



$$L(x,d) = E(x,d) + \int \rho(x,d,d') \, v(x,x') \, L(x',d') \, G(x,x') \, dA$$

Multiplication by the reflectance (BRDF)
of the surface at point x

Radiance unit: Watt/m²/sr

# The rendering equation



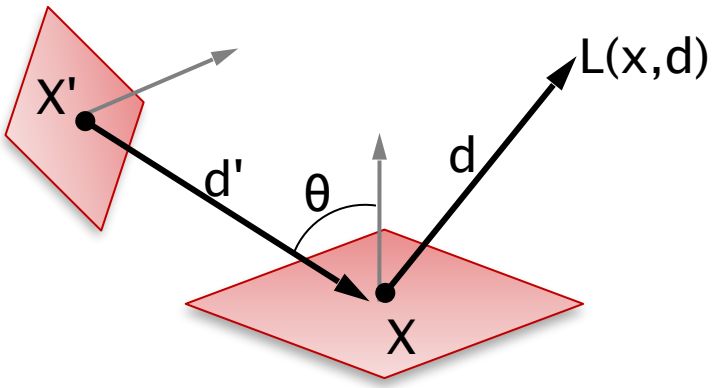

$$L(x,d) = E(x,d) + \int \rho(x,d,d') \, v(x,x') \, L(x',d') \, G(x,x') \, dA$$

Visibility between x and x'

1 when the two points are visible from each other, 0 otherwise

Radiance unit: Watt/m²/sr

# The rendering equation

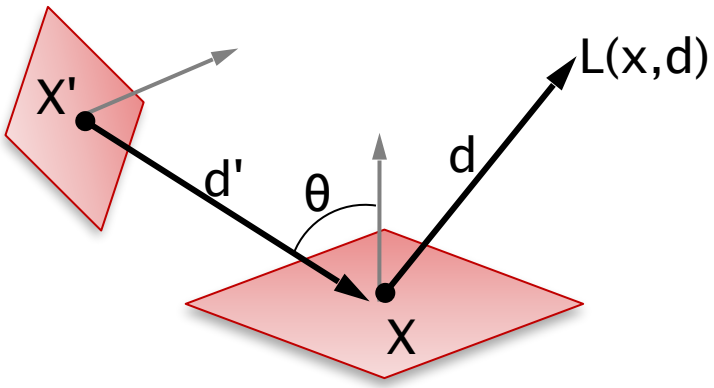



$$L(x,d) = E(x,d) + \int \rho(x,d,d') \, v(x,x') \, L(x',d') \, G(x,x') \, dA$$

Geometric factor depending on the surfaces x and x'

Radiance unit: Watt/m²/sr

# The rendering equation



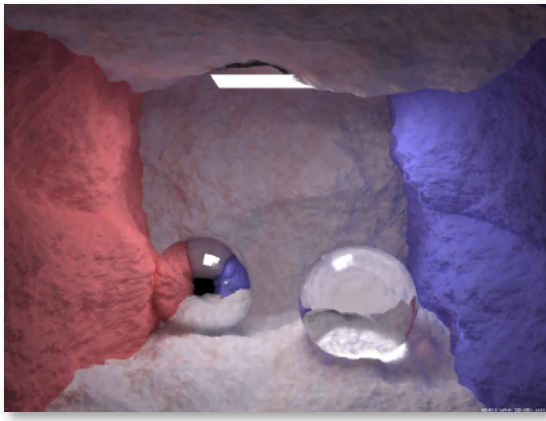

$$L(x,d) = E(x,d) + \int \rho(x,d,d') \, v(x,x') \, L(x',d') \, G(x,x') \, dA$$

Full general analytical solution impossible

Radiance unit: Watt/m²/sr

# Two discretisation

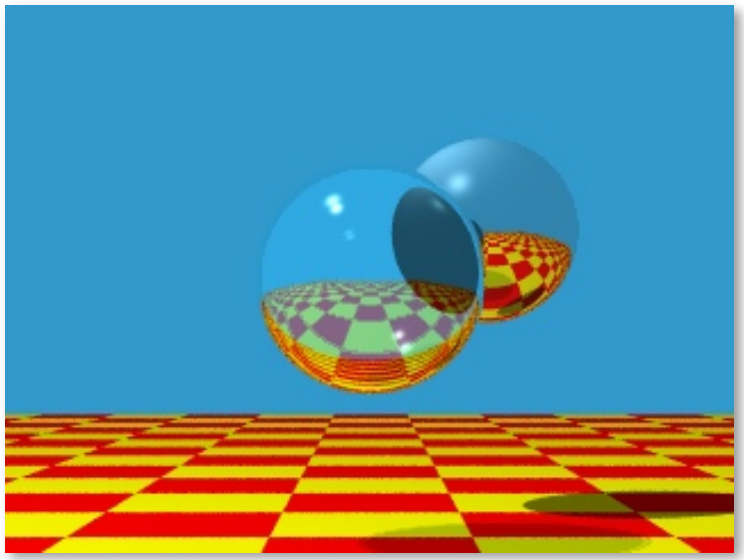
- Ray-tracing and its extensions
  (Monte-Carlo path tracing, Photon mapping…)
  - Sampling the integral
  - Optical laws
- Radiosity
  - Discretize the geometry:
    exchanges between patches
  - All objects are diffuse

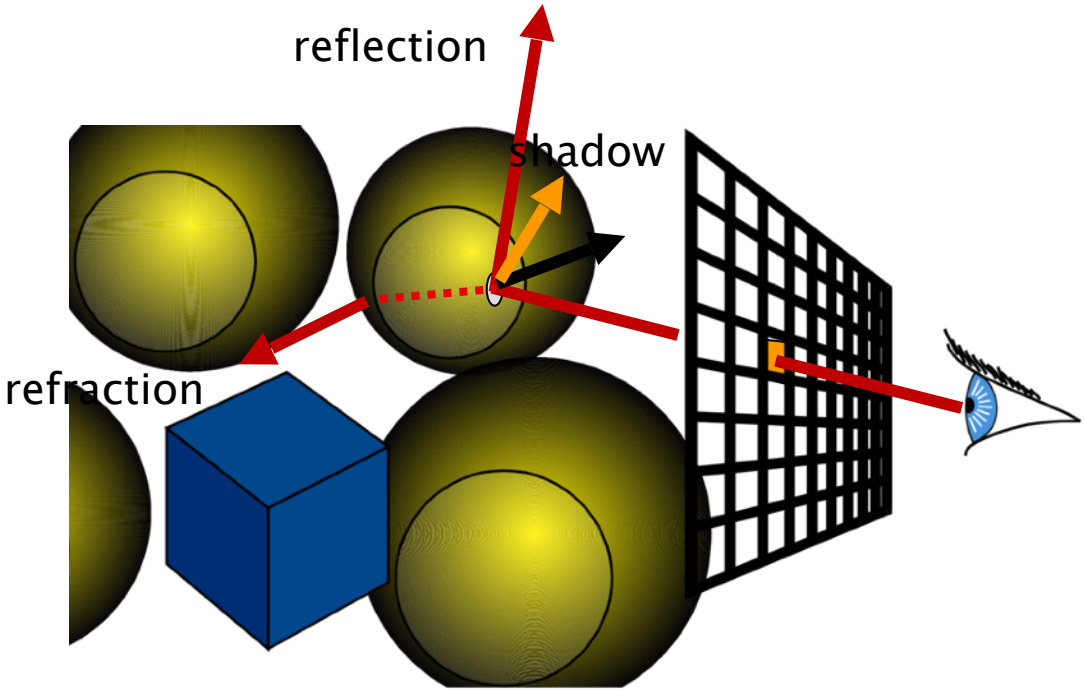


# Ray tracing

# « Whitted Ray Tracing » (1980)

- One ray per pixel
- Three new rays are created



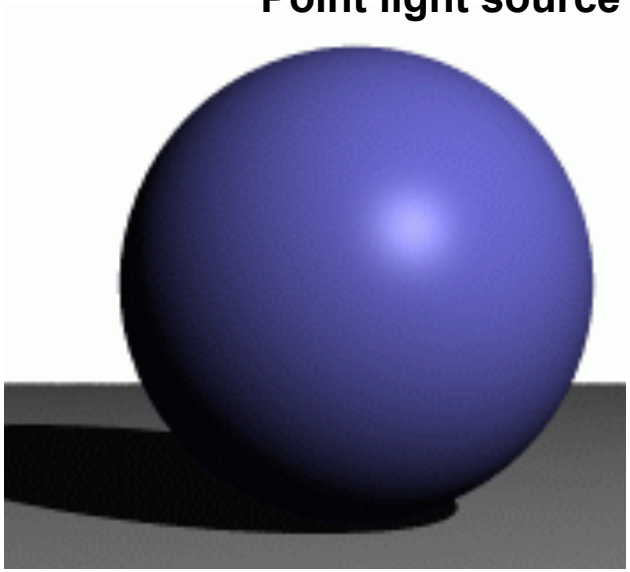


# Even more rays

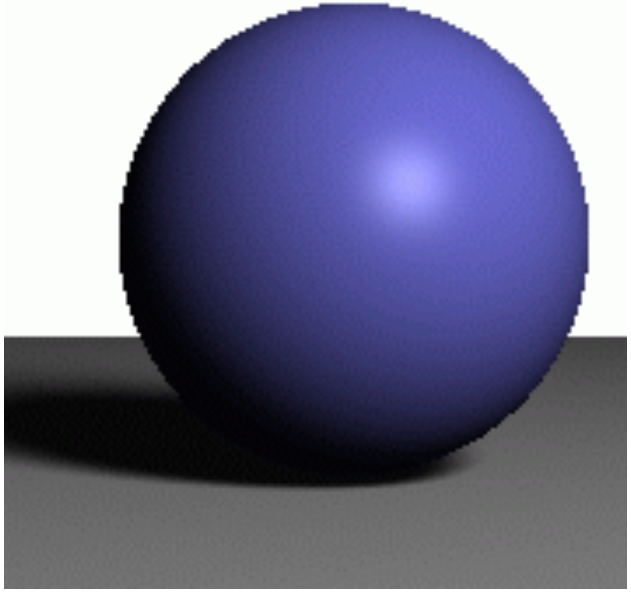
« Distributed Ray Tracing » Cook et al. (1984)

- Soft shadows
  - Several rays for each extended light source

Point light source



Extended light source

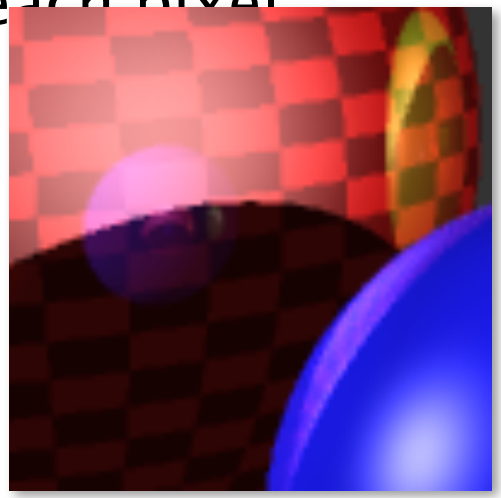


# Even more rays

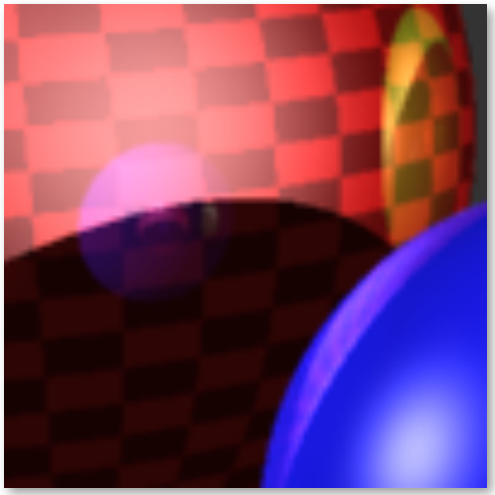
« Distributed Ray Tracing » Cook et al. (1984)

- Soft shadows
  - Several rays for each extended light source
- Anti-aliasing
  - Several rays for each pixel

1 ray



2 rays

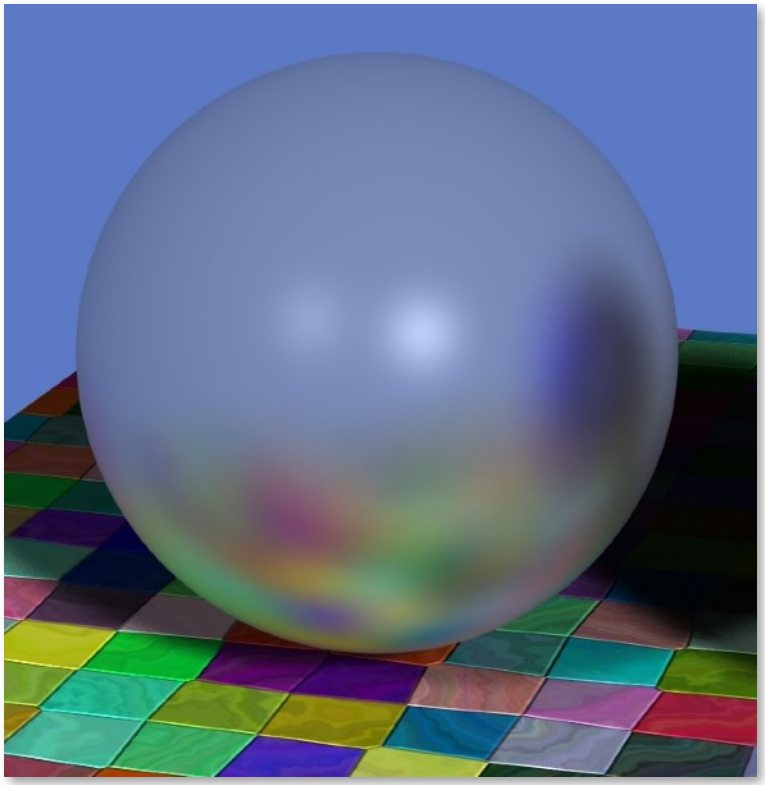


3 rays

# Even more rays

« Distributed Ray Tracing » Cook et al. (1984)

- Soft shadows
  - Several rays for each extended light source
- Anti-aliasing
  - Several rays for each pixel
- Glossy Reflection
  - Several rays are reflected



# Even more rays

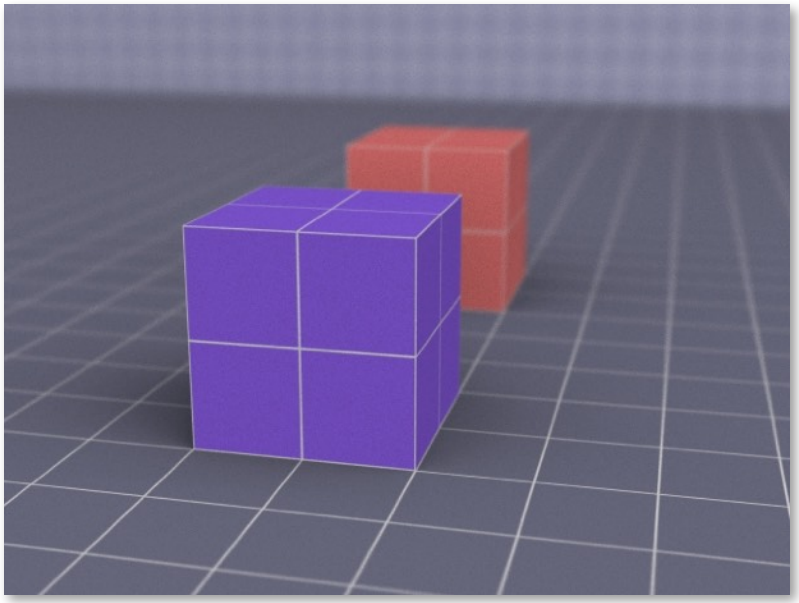
« Distributed ray tracing » Cook et al. (1984)

- Soft shadows
  - Several rays for each extended light source
- Anti-aliasing
  - Several rays for each pixel
- Glossy Reflection
  - Several rays are reflected
- Motion blur
  - Several rays during shutter opening

# Even more rays

« Distributed Ray Tracing » Cook et al. (1984)

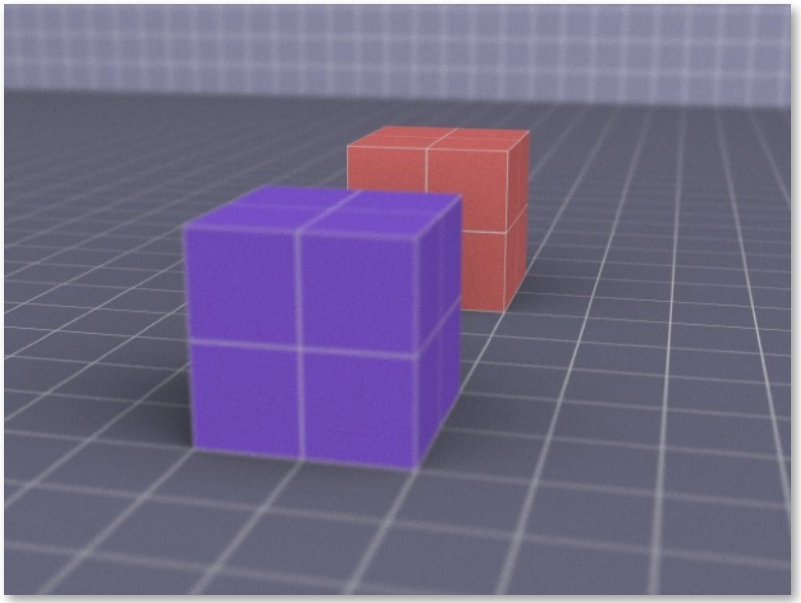
- Soft shadows
  - Several rays for each extended light source
- Anti-aliasing
  - Several rays for each pixel
- Glossy Reflection
  - Several rays are reflected
- Motion blur
  - Several rays during time
- Depth of field
  - Several rays per pixel, focusing with a lens



# Even more rays

« Distributed Ray Tracing » Cook et al. (1984)

- Soft shadows
  - Several rays for each extended light source
- Anti-aliasing
  - Several rays for each pixel
- Glossy Reflection
  - Several rays are reflected
- Motion blur
  - Several rays during time
- Depth of field
  - Several rays per pixel, focusing with a lens



# Ray-tracing

- Main operation:
  - Ray-primitive intersection
- Speed-up:
  - Scene structure

# Ray-Primitive intersection

- Ray: starting point $P$, direction $\boldsymbol{u}$

- Primitives:
  - Sphere
  - Cylinder
  - Plane (infinite)
  - Triangle

# Ray-sphere intersection

- Ray: starting point $P$, direction $\boldsymbol{u}$
- Sphere : center $C$, radius $r$

- $\exists$ intersection: Boolean

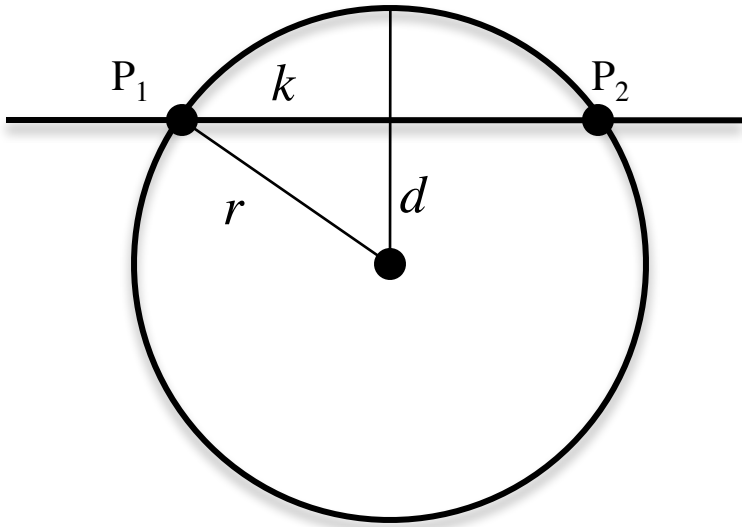
$$\Delta = (\vec{u} \cdot \overrightarrow{CP})^2 - CP^2 + r^2 > 0$$

- Intersection point: $Q = P + \lambda \boldsymbol{u}$
  - Several potential intersection points
  - $\lambda > 0$
  - First point only

# Ray-cylinder intersection

- Ray: starting point $P$, direction $\boldsymbol{u}$
- Cylinder: axis $D\,(Q, \boldsymbol{v})$, radius $r$

- $\exists$ intersection : Boolean
  - $\boldsymbol{n} = (\boldsymbol{u} \times \boldsymbol{v}) \,/\, \| \boldsymbol{u} \times \boldsymbol{v} \|$
  - $d = | PQ . \boldsymbol{n} | < r$




- 2 intersection points:
  - $\lambda_{\text{middle}} = (PQ \times \boldsymbol{v}) \,.\, \boldsymbol{n} \,/\, \| \boldsymbol{u} \times \boldsymbol{v} \|$
  - $+$ Pythagore for $\mathrm{P}_1, \mathrm{P}_2 : k = \mathrm{sqrt}(r^2 - d^2)$

"Intersecting a ray with a cylinder", Joseph M. Cychosz & Warren N. Waggenspack, Jr., Graphics Gems IV, 1994

# Ray-plane intersection

- Ray: starting point $P$, direction $\boldsymbol{u}$
- Plane: point $Q$, normal $\boldsymbol{n}$

- Intersection point:
  - $\lambda = -(PQ \cdot \boldsymbol{n}) / (\boldsymbol{u} \cdot \boldsymbol{n})$

# Ray-triangle intersection

- Ray: starting point $P$, direction $\boldsymbol{u}$
- Triangle $(V_0, V_1, V_2)$

- [Badouel 90]:
  - Ray-plane intersection
  - Projection on 2D plane
  - (remove one coordinate)
  - Compute barycentric coordinates in 2D

# Ray-triangle intersection

- Ray: starting point $P$, direction $\boldsymbol{u}$
- Triangle $(V_0, V_1, V_2)$
- [Möller-Trumbore 97]: fast+minimal storage
- Solve for $(\lambda, \alpha, \beta)$ in a single step:

$$P+\lambda\vec{u} = (1-\alpha-\beta)V_0 + \alpha V_1 + \beta V_2$$

$$\begin{pmatrix} \lambda \\ \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} -\vec{u} & V_1 - V_0 & V_2 - V_0 \end{pmatrix}^{-1} (P - V_0)$$

http://www.graphics.cornell.edu/pubs/1997/MT97.html

# Ray-triangle intersection

- Ray: starting point $P$, direction $\boldsymbol{u}$
- [Möller-Trumbore 97]: fast+minimal storage
- $\boldsymbol{e}_i = V_i - V_0 \qquad \boldsymbol{t} = P - V_0$

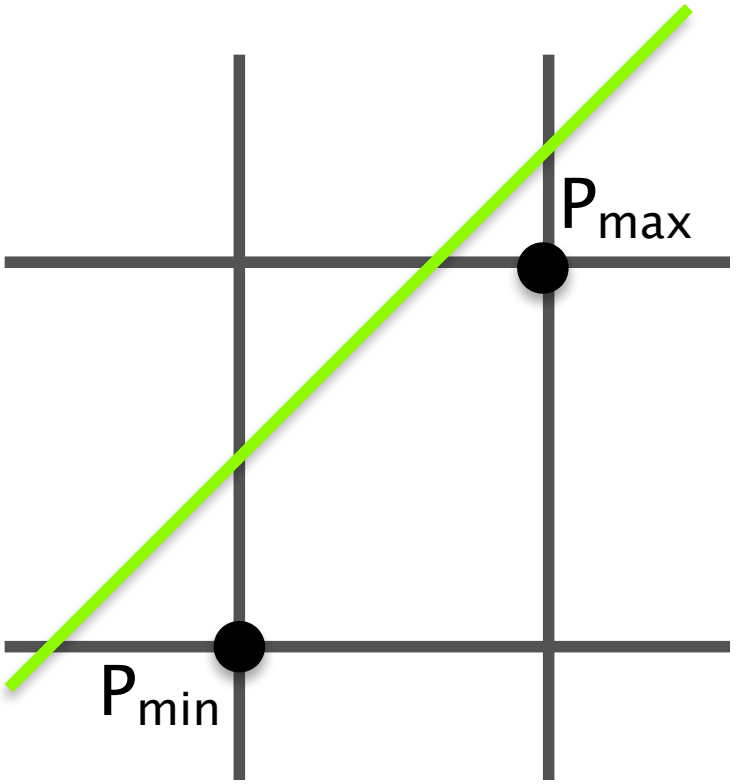
$$\begin{pmatrix} \lambda \\ \alpha \\ \beta \end{pmatrix} = \frac{1}{(\vec{u} \times \vec{e}_2) \cdot \vec{e}_1} \begin{pmatrix} (\vec{t} \times \vec{e}_1) \cdot \vec{e}_2 \\ (\vec{u} \times \vec{e}_2) \cdot \vec{t} \\ (\vec{t} \times \vec{e}_1) \cdot \vec{u} \end{pmatrix} = \frac{1}{\vec{p} \cdot \vec{e}_1} \begin{pmatrix} \vec{q} \cdot \vec{e}_2 \\ \vec{p} \cdot \vec{t} \\ \vec{q} \cdot \vec{u} \end{pmatrix}$$

Speed/memory tradeoff: precompute the $\boldsymbol{e}_i$

Still the fastest algorithm, even with GPUs

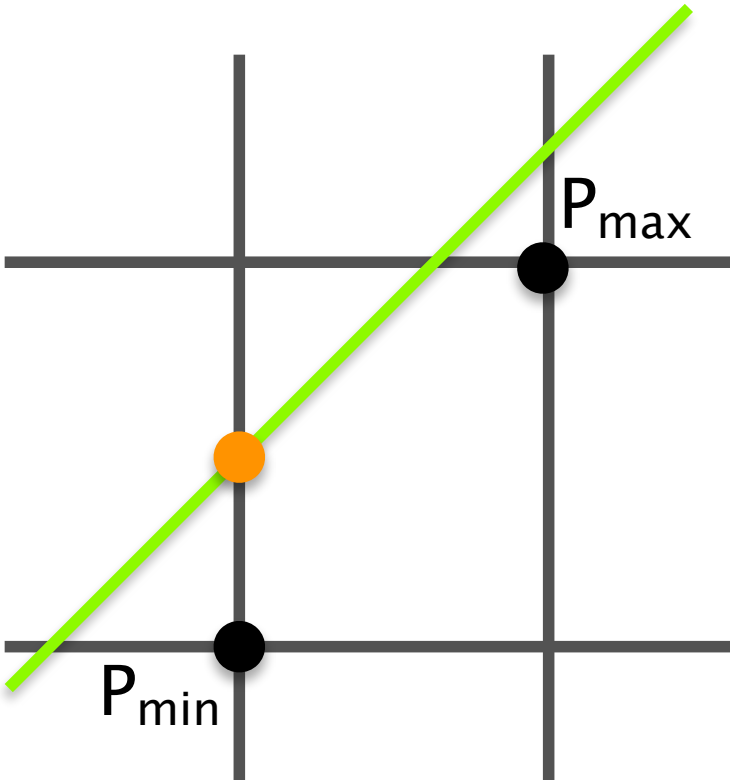
# Axis-Aligned Bounding Box

- Bounding box defined by $P_{\min}$, $P_{\max}$
- Ray intersection?
  - $\lambda_{x\min} = (x_{\min} - P_x) / \boldsymbol{u}_x$
- Think parallel!
  - $t_{\min} = (P_{\min} - P) / \boldsymbol{u}$
  - $t_{\max} = (P_{\max} - P) / \boldsymbol{u}$
  - Component-wise div
  - 2 GPU operations



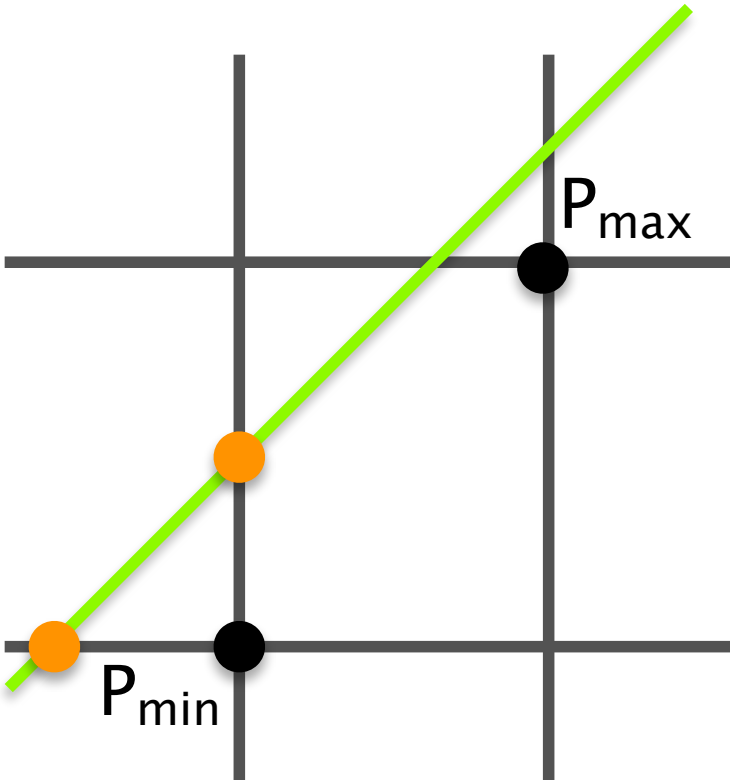
# Axis-Aligned Bounding Box

- Bounding box defined by $P_{\min}$, $P_{\max}$
- Ray intersection?
  - $\lambda_{x\min} = (x_{\min} - P_x) / \boldsymbol{u}_x$
- Think parallel!
  - $t_{\min} = (P_{\min} - P) / \boldsymbol{u}$
  - $t_{\max} = (P_{\max} - P) / \boldsymbol{u}$
  - Component-wise div
  - 2 GPU operations



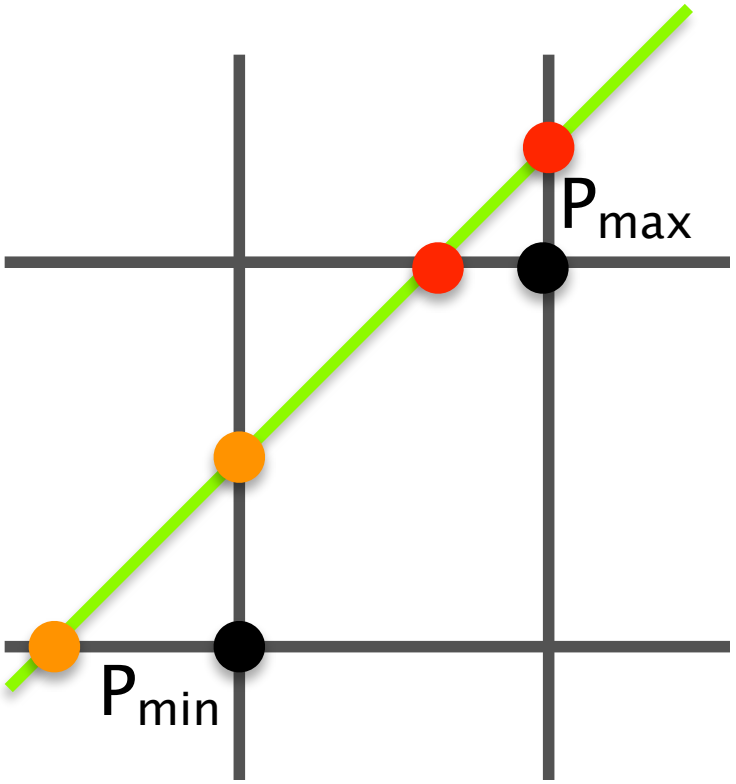
# Axis-Aligned Bounding Box

- Bounding box defined by $P_{\min}$, $P_{\max}$
- Ray intersection?
  - $\lambda_{x\min} = (x_{\min} - P_x) / \boldsymbol{u}_x$
- Think parallel!
  - $t_{\min} = (P_{\min} - P) / \boldsymbol{u}$
  - $t_{\max} = (P_{\max} - P) / \boldsymbol{u}$
  - Component-wise div
  - 2 GPU operations



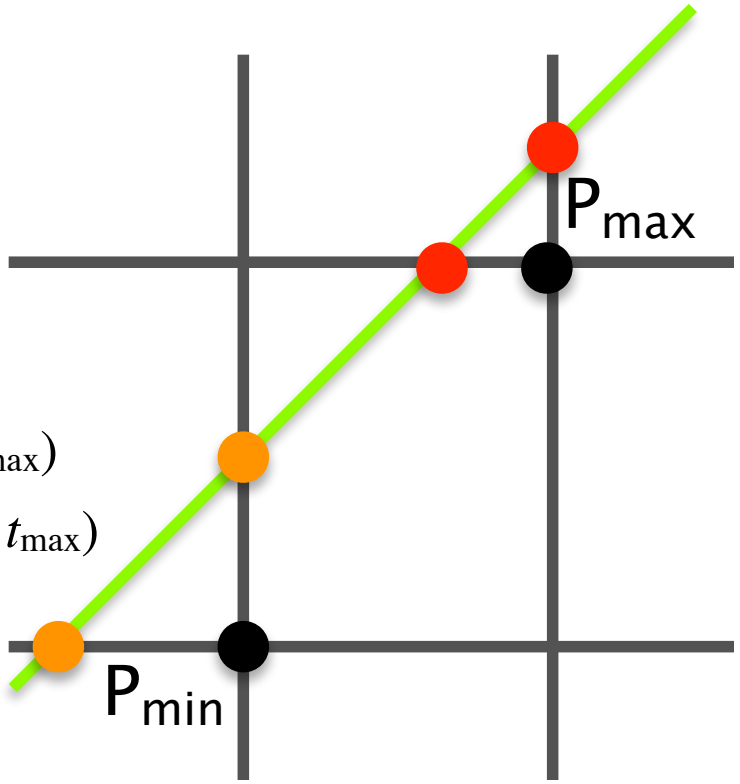
# Axis-Aligned Bounding Box

- Bounding box defined by $P_{\min}$, $P_{\max}$
- Ray intersection?
  - $\lambda_{x\min} = (x_{\min} - P_x) / \boldsymbol{u}_x$
- Think parallel!
  - $t_{\min} = (P_{\min} - P) / \boldsymbol{u}$
  - $t_{\max} = (P_{\max} - P) / \boldsymbol{u}$
  - Component-wise div
  - 2 GPU operations



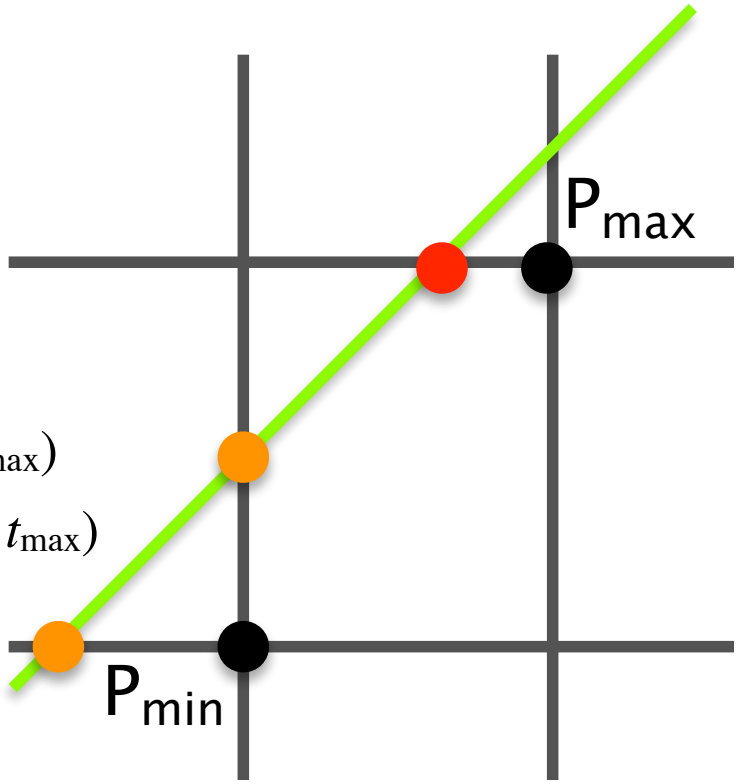
# Axis-Aligned Bounding Box

- Bounding box defined by $P_{\min}$, $P_{\max}$
- Ray intersection?
  - $t_{\min} = (P_{\min} - P) / \boldsymbol{u}$
  - $t_{\max} = (P_{\max} - P) / \boldsymbol{u}$
  - Needs sorting:
    - In = max value of $\min(t_{\min}, t_{\max})$
    - Out = min value of $\max(t_{\min}, t_{\max})$
  - Out > In = intersection
    - Otherwise: no intersect



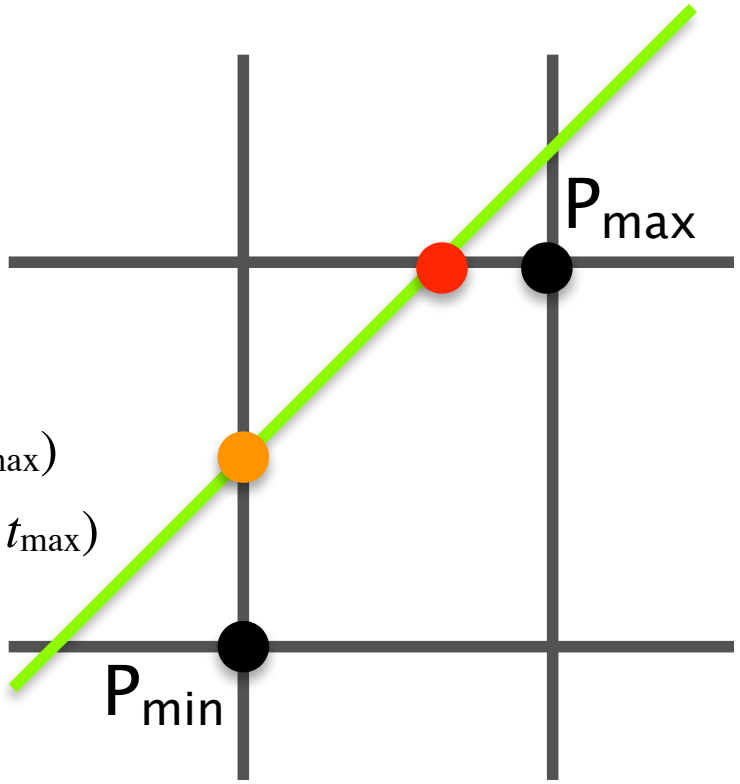
# Axis-Aligned Bounding Box

- Bounding box defined by $P_{\min}$, $P_{\max}$
- Ray intersection?
  - $t_{\min} = (P_{\min} - P) / \boldsymbol{u}$
  - $t_{\max} = (P_{\max} - P) / \boldsymbol{u}$
  - Needs sorting:
    - In = max value of $\min(t_{\min}, t_{\max})$
    - Out = min value of $\max(t_{\min}, t_{\max})$
  - Out > In = intersection
    - Otherwise: no intersect



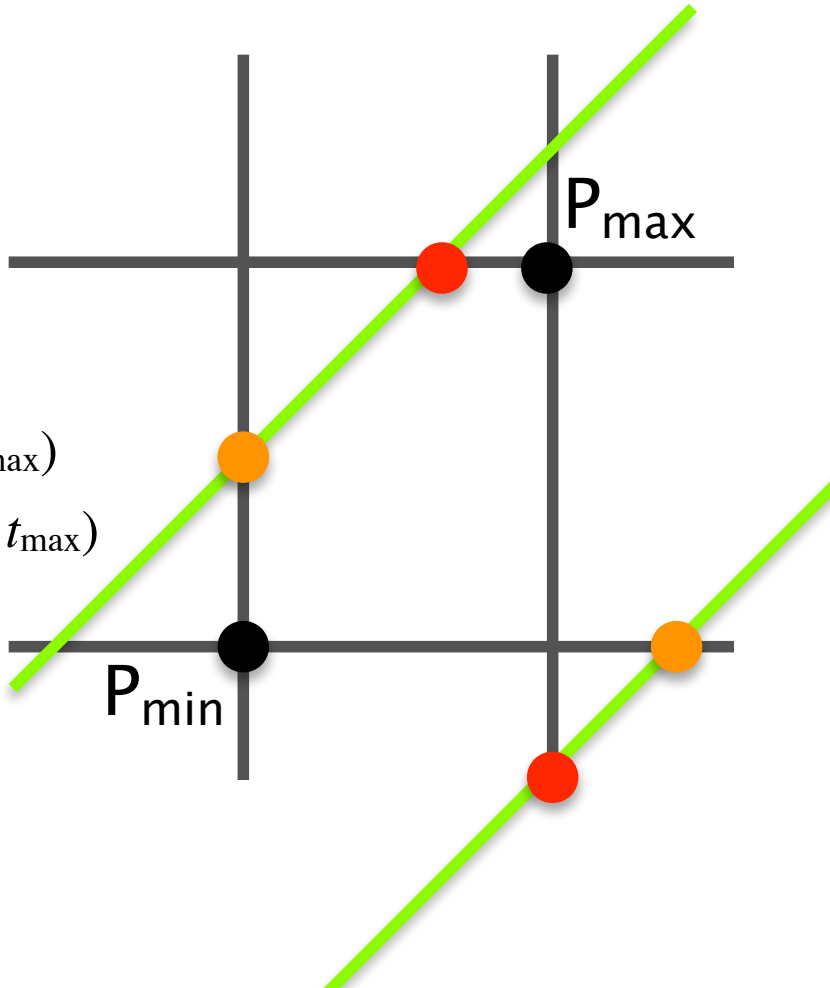
# Axis-Aligned Bounding Box

- Bounding box defined by $P_{\min}$, $P_{\max}$
- Ray intersection?
  - $t_{\min} = (P_{\min} - P) / \boldsymbol{u}$
  - $t_{\max} = (P_{\max} - P) / \boldsymbol{u}$
  - Needs sorting:
    - In = max value of $\min(t_{\min}, t_{\max})$
    - Out = min value of $\max(t_{\min}, t_{\max})$
  - Out > In = intersection
    - Otherwise: no intersect



# Axis-Aligned Bounding Box

- Bounding box defined by $P_{\min}$, $P_{\max}$
- Ray intersection?
  - $t_{\min} = (P_{\min} - P) / \boldsymbol{u}$
  - $t_{\max} = (P_{\max} - P) / \boldsymbol{u}$
  - Needs sorting:
    - In = max value of $\min(t_{\min}, t_{\max})$
    - Out = min value of $\max(t_{\min}, t_{\max})$
  - Out > In = intersection
    - Otherwise: no intersect



# Counting operations

- Ray-sphere intersection:
  - 2 dot products, 3 vec4 ops, 3 float ops
- Ray-triangle intersection:
  - 4 dot products, 2 cross products, 3 vec4 ops, 3 float ops
- Ray-AABB intersection:
  - 6 vec4 ops, 5 float ops

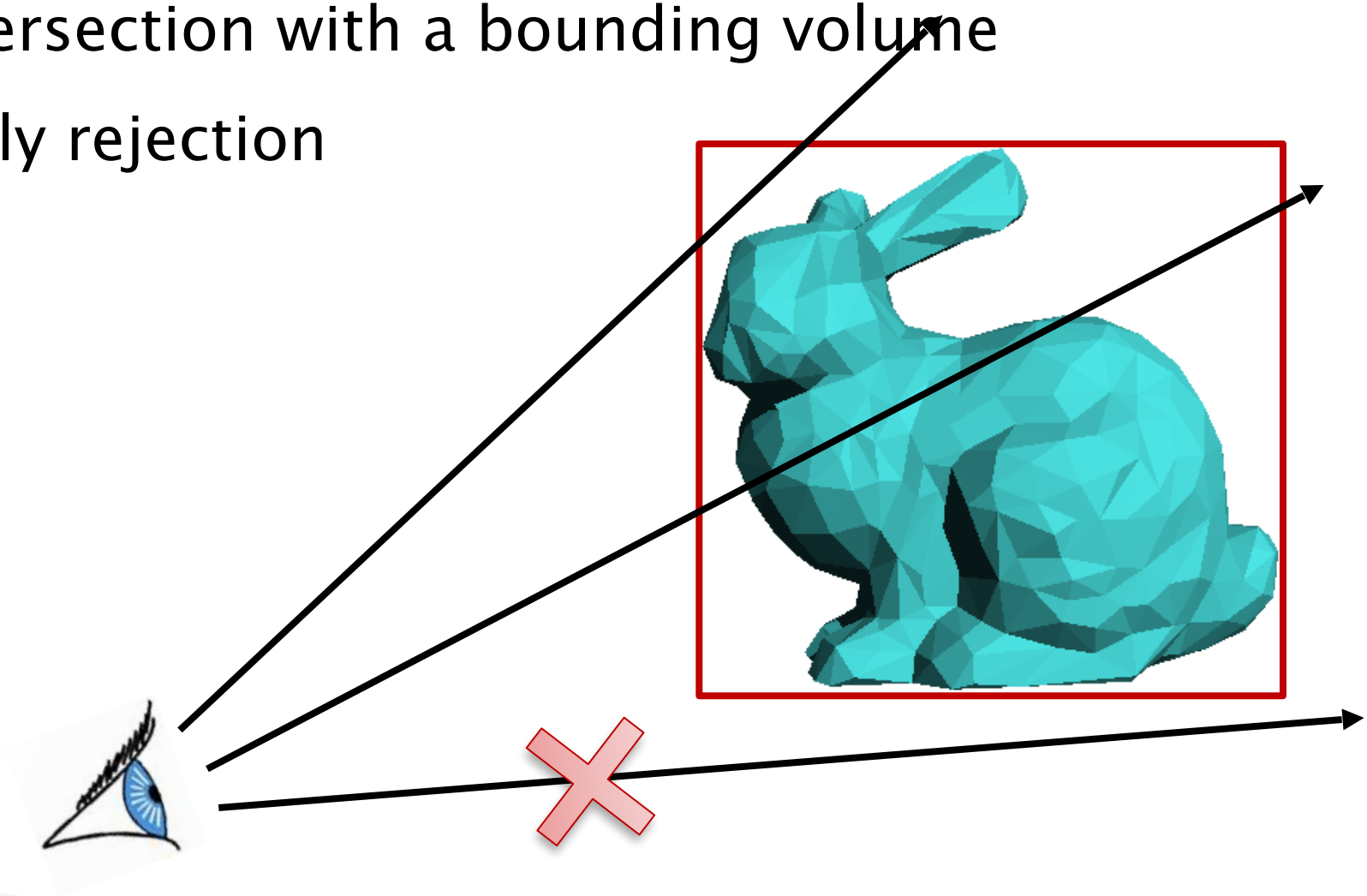
- Ray-bounding volume $\cong$ 1 ray-triangle

# Ray-scene intersection

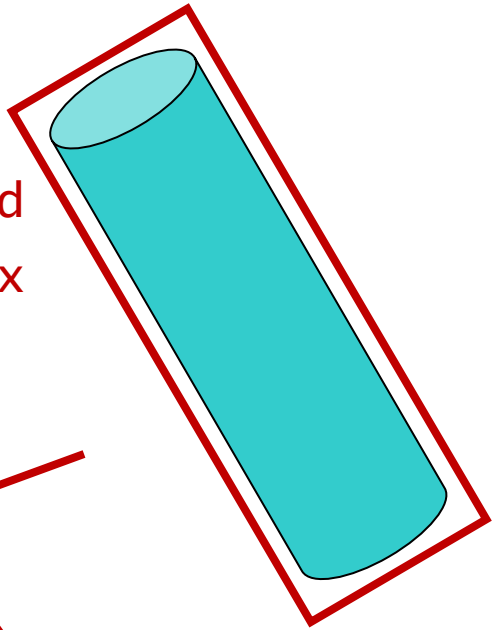
- 99 % of the time is spent doing intersections
- >100 K triangles
- Need for accelerations:
  - Bounding volumes
  - Hierarchies of bounding volumes
  - kD-tree
  - Octree, uniform grid
- *Building* hierarchies: offline, CPU
- *Intersecting* hierarchies: GPU

# Bounding volumes

- Intersection with a bounding volume
- Early rejection

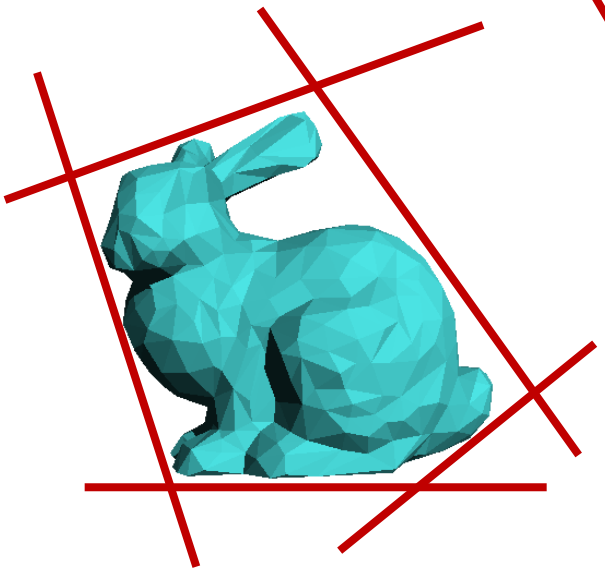


# Bounding volumes

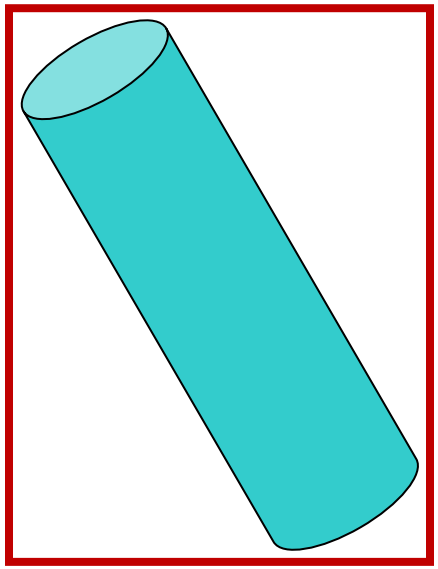


Not-axis-aligned bounding box

Bounding sphere



Arbitrary convex region



Axis-aligned bounding box

# Spatial hierarchies

- 1 object = 100K+ triangles
- Need spatial hierarchy beyond AABB
- At the hierarchy leaves:
  - Intersect leaf, then list of triangles
  - 1 triangle per leaf: wasteful
  - 10 triangles per leaf: too much
  - Threshold: usually 4 triangles
- BVH, kD-trees, octrees, uniform grid…

# Bounding Volume Hierarchy

- Top-down or bottom-up
- Top-down:
  - Start with object AABB
  - Subdivide recursively until limit
  - Easier to implement
- Bottom-up:
  - Start with triangles
  - Find spatially close triangles
  - Group and recurse
  - More compact

# BVH top-down construction

- Slides from Magnus Andersson (LTH)
- http://fileadmin.cs.lth.se/cs/Education/EDAN30/lectures/S2-bvh.pdf

# Construction

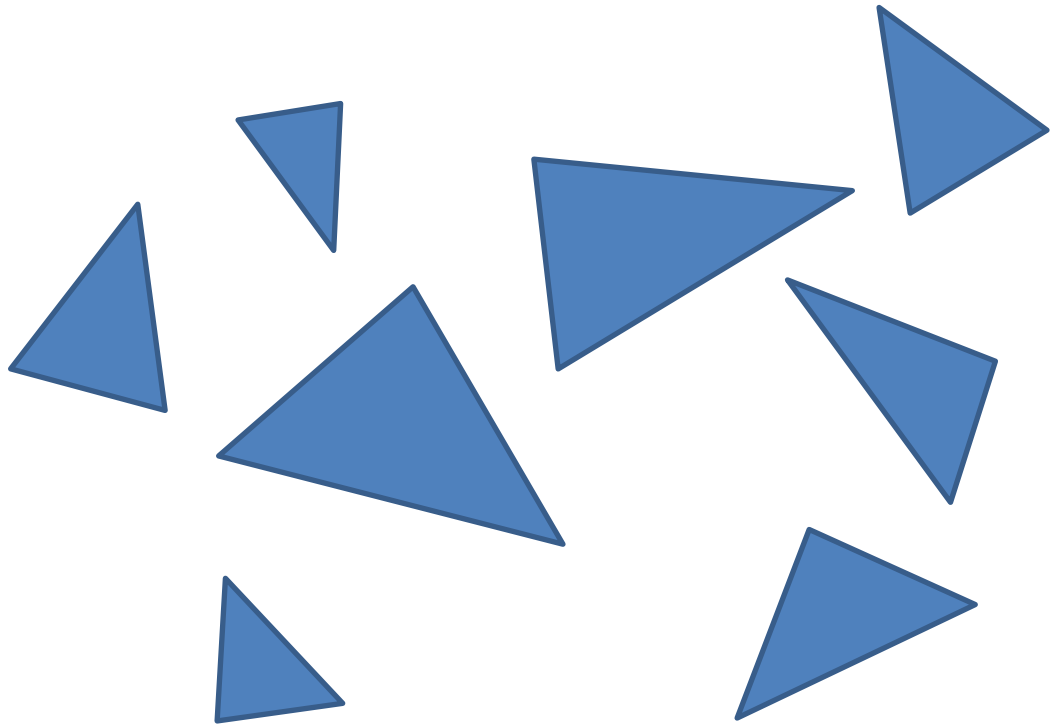
- You will need to create a new class *BVHAccelerator* which inherits from *<<RayAccelerator>>*
- For this assignment you must implement

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

- …and you will probably need something like

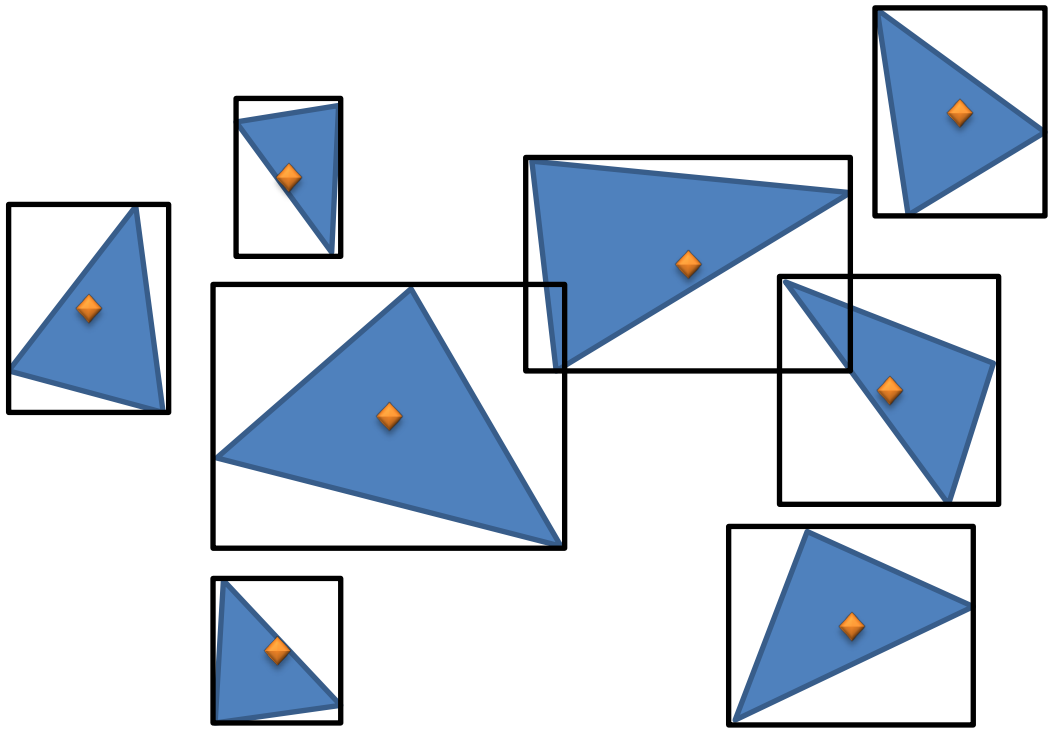
```
void build_recursive(int left_index, int right_index, AABB box,
                     BVHNode *node, int depth);
```

# Construction, brief overview



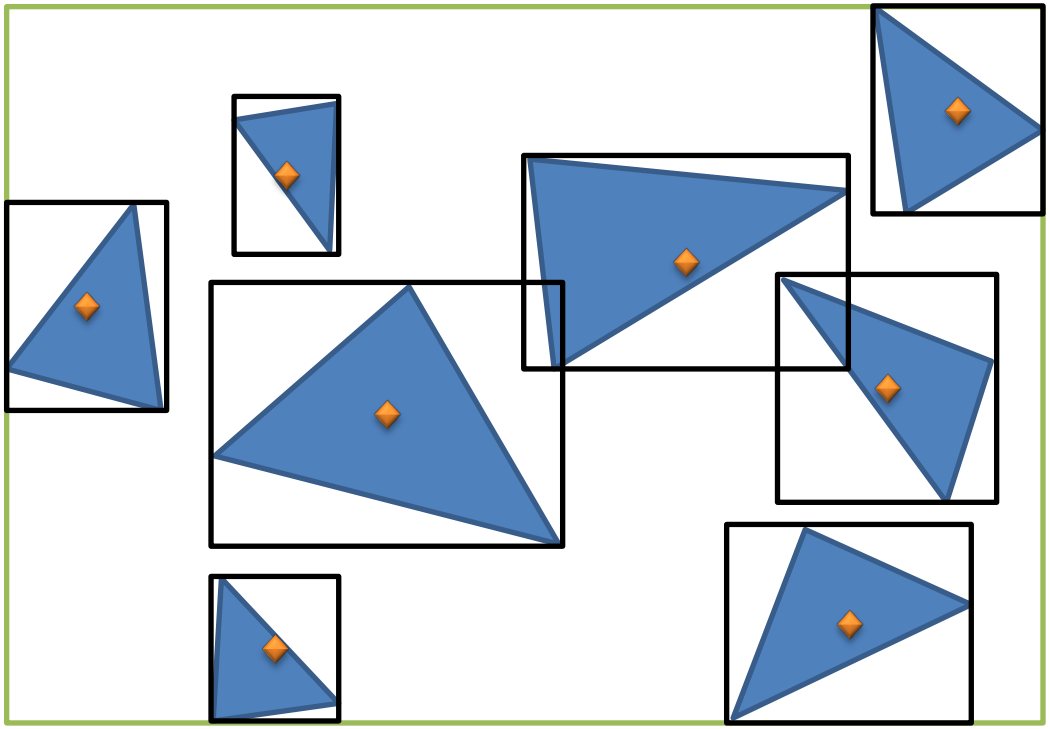
We begin with a bunch of Intersectables

# Construction , brief overview



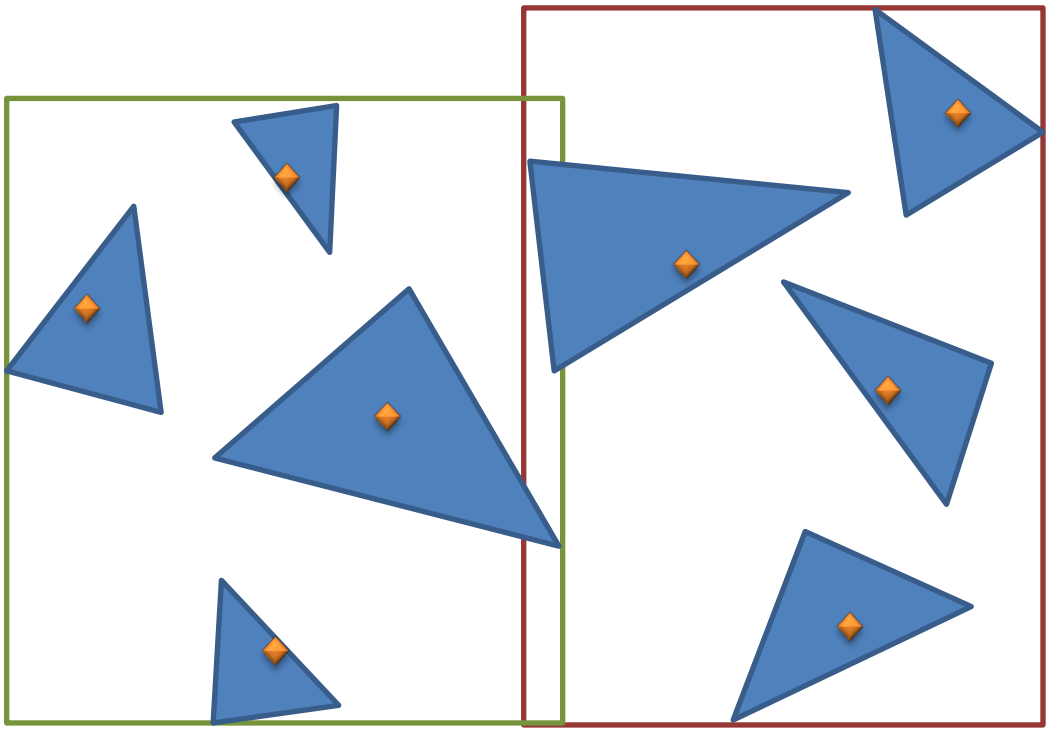
Find bounding box centroids of all intersectables

# Construction , brief overview



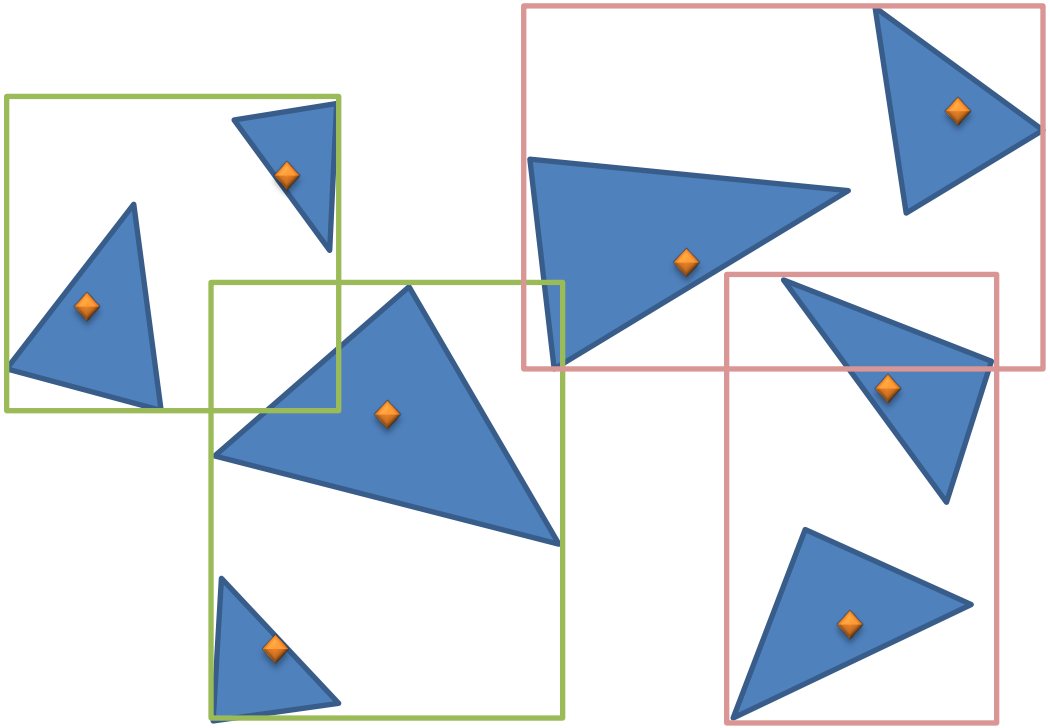
Find the *world* bounding box and create a root node

# Construction , brief overview



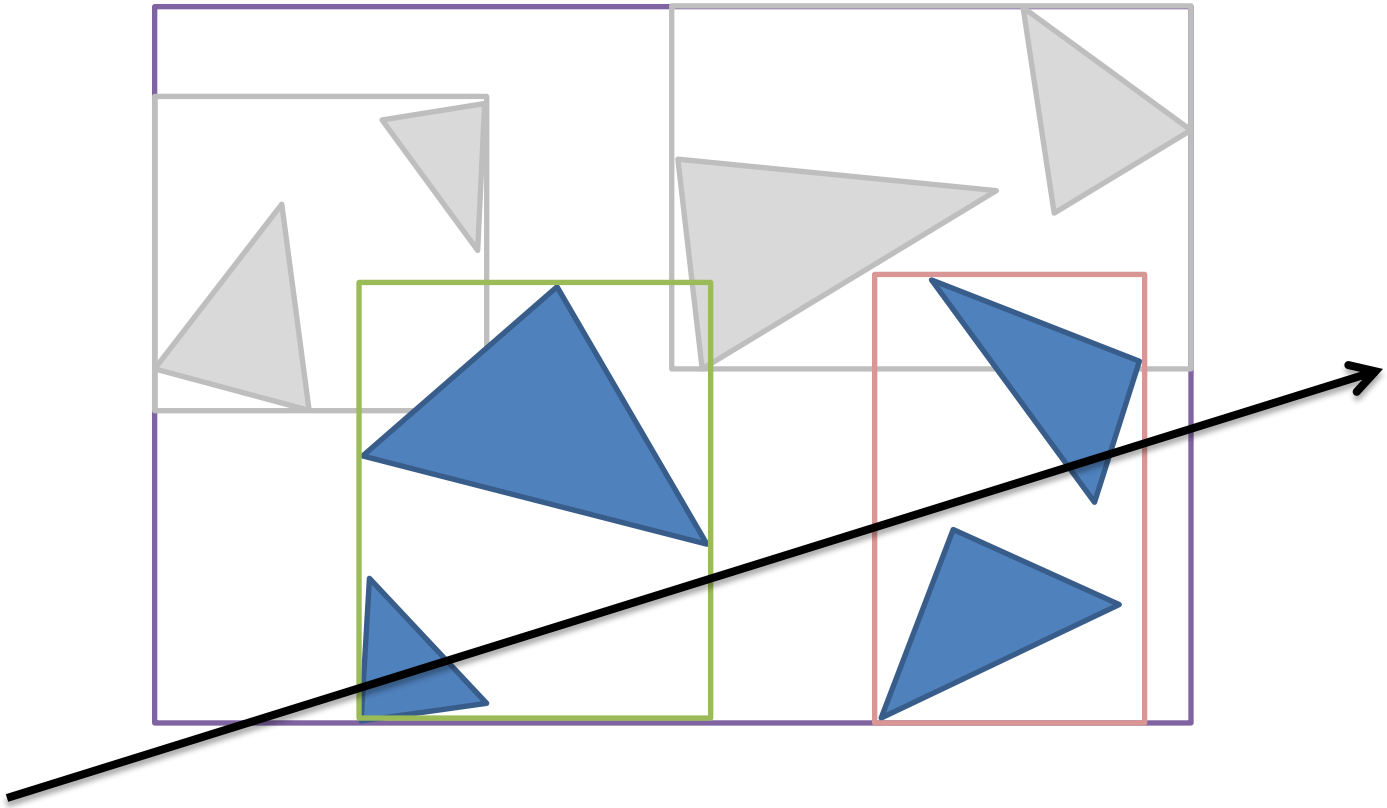
Use some splitting criteria to find a sensible division of the elements into new child nodes

# 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!

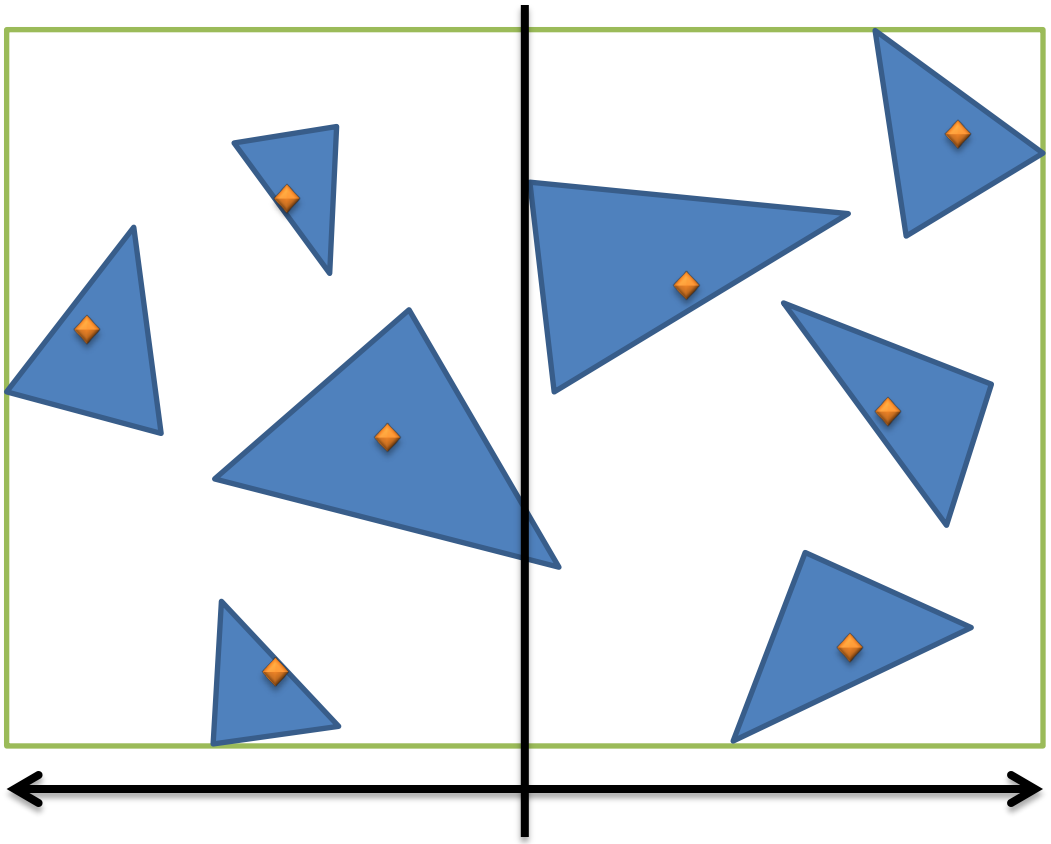
# Construction

- 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

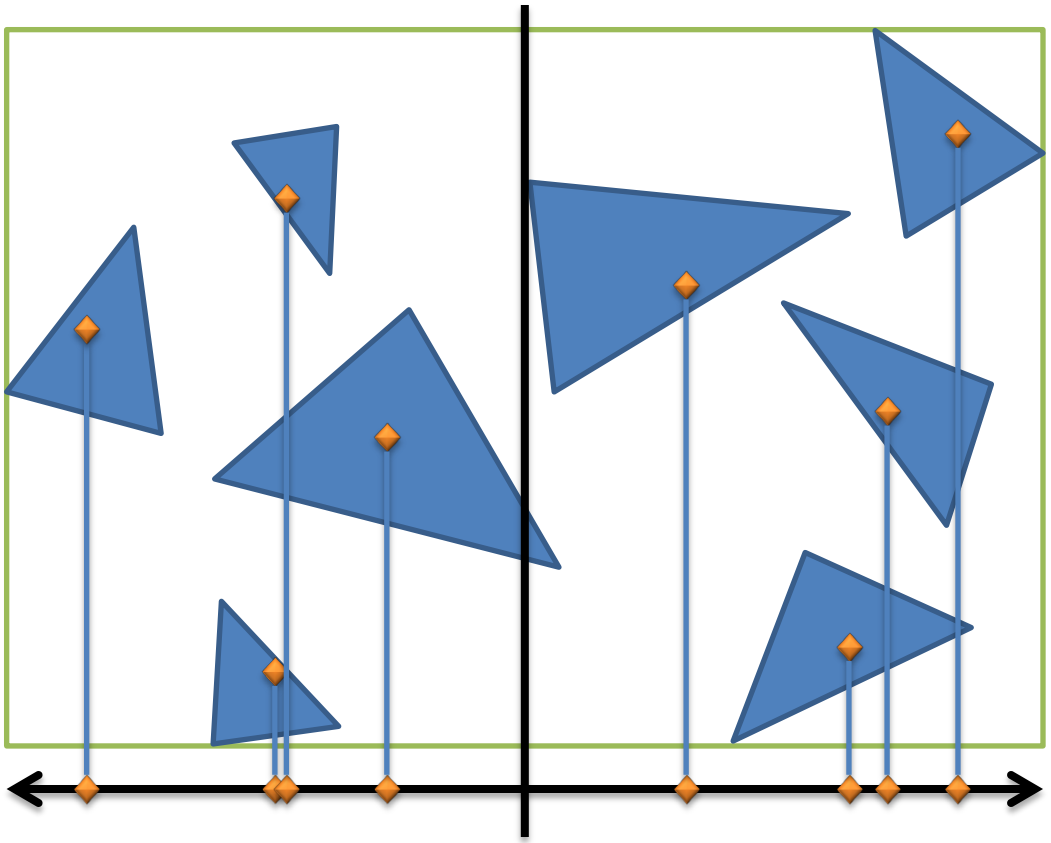
- So what is a sensible splitting criteria?
- Why not use *mid-point* splitting, since it's easy to understand and implement
  - *Works well when primitives are fairly evenly distributed*
- You can try to come up with a different criteria if you want to
  - *I tried splitting on the mean and median.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

# Construction



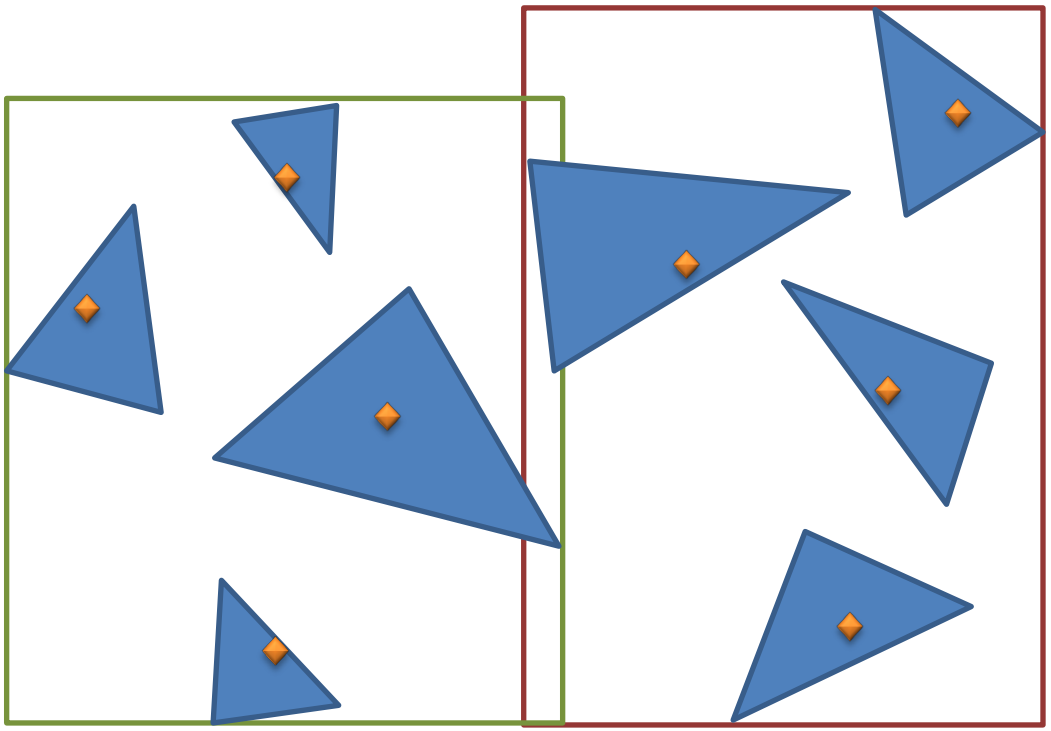
Find the mid point of the largest axis

# Construction



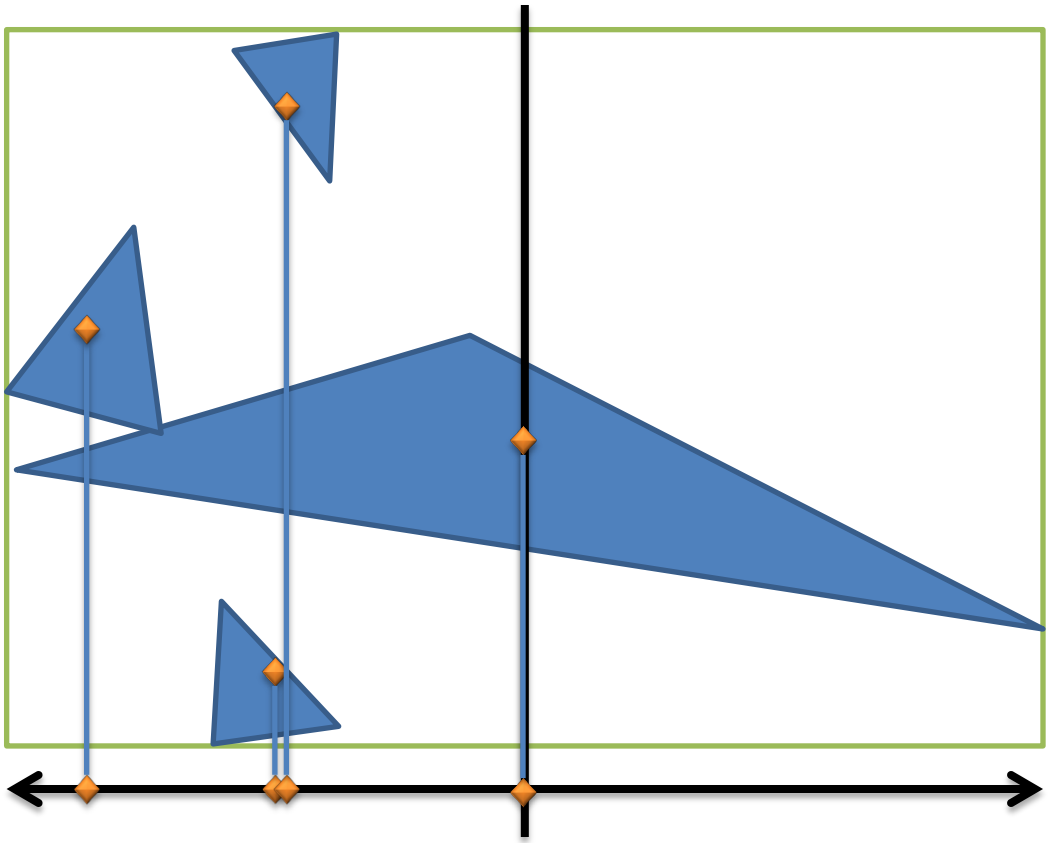
*Sort* the bounding box *centroids* in the largest axis direction. Then split into a *left* and a *right* side

# Construction



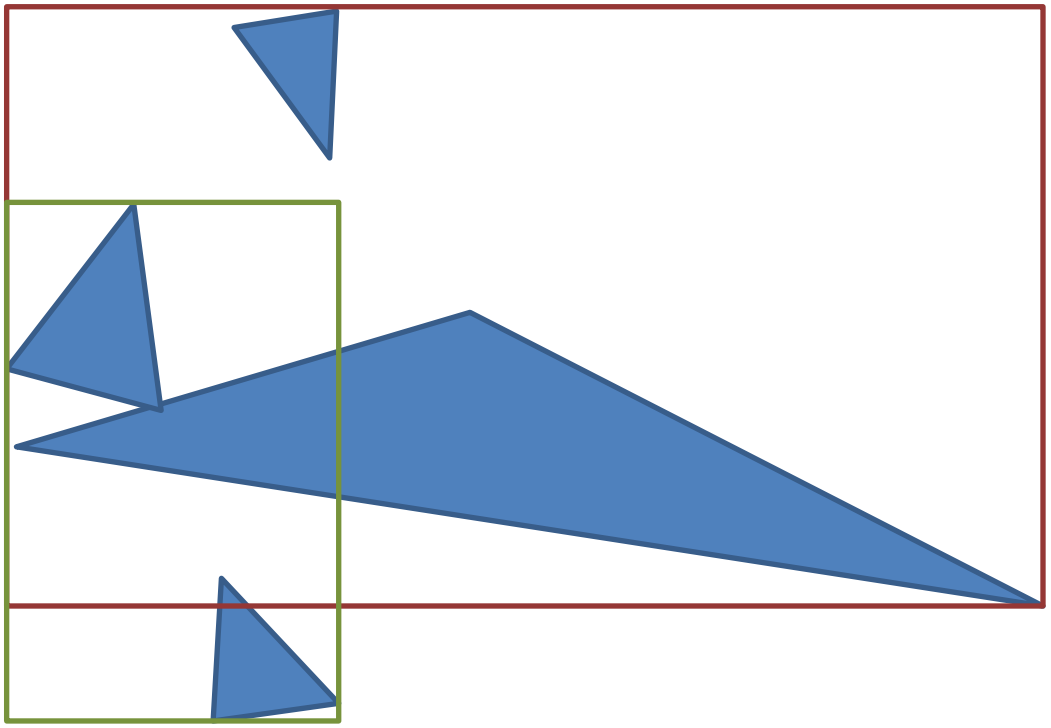
Lather, rinse and repeat. Terminate when a node contains few intersectables (I used 4, which worked well)

# Construction



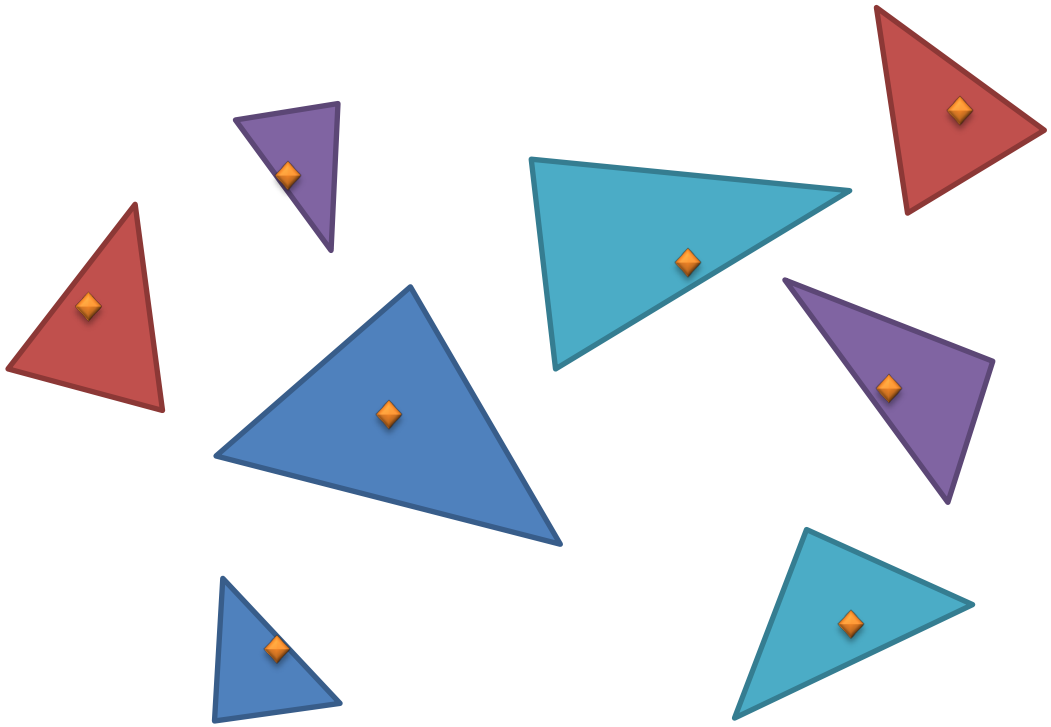
There is a hazard in getting all intersectables on one side – we could end up with empty nodes!

# 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.

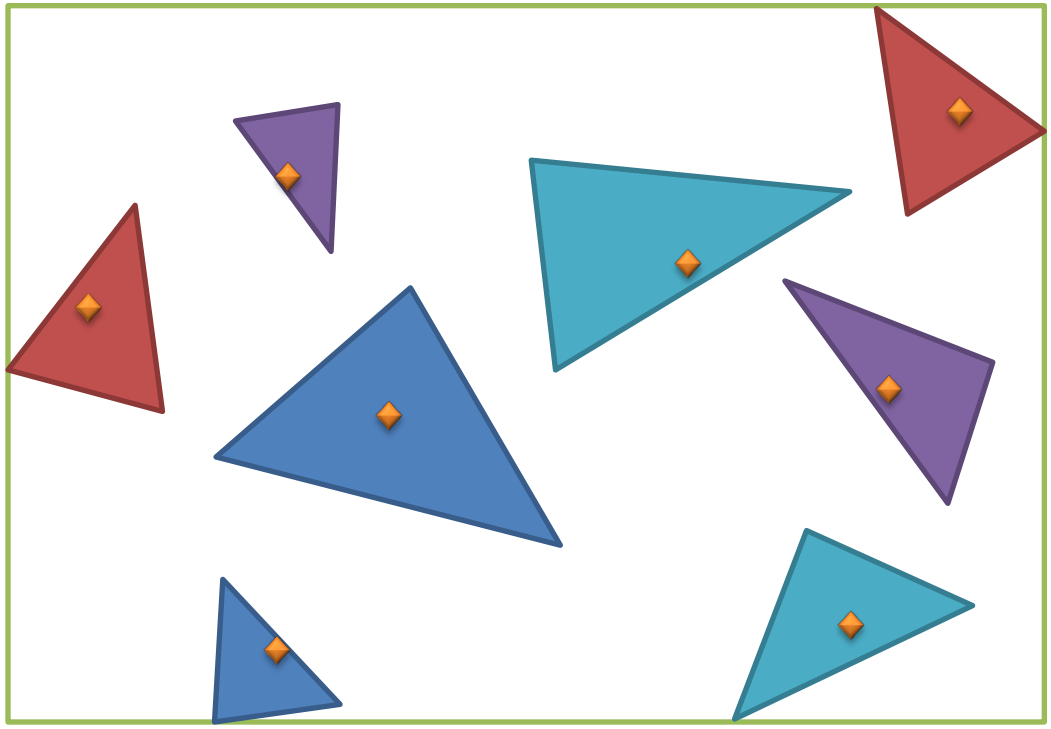
# Node class

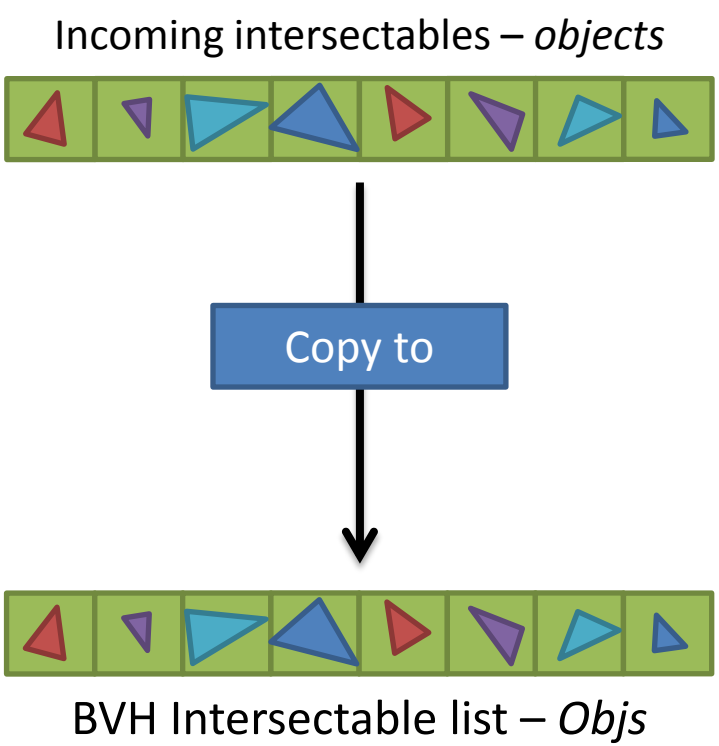
- BVH node class (inner class of *BVHAccelerator*)

```
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; };
};
```

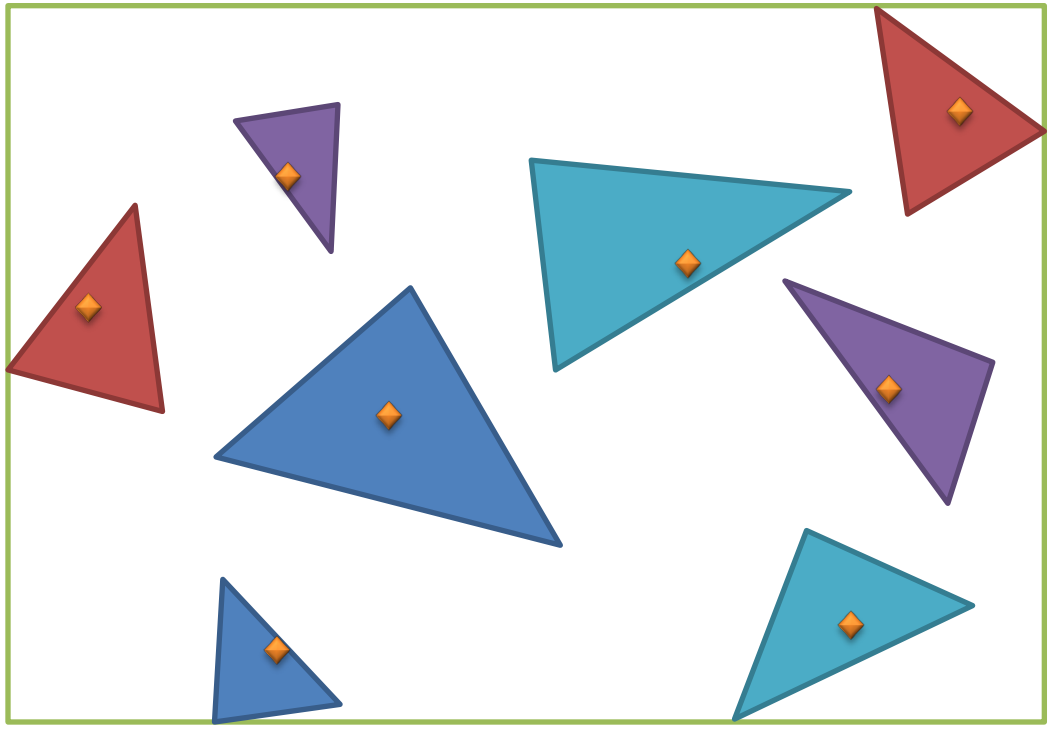
# Construction






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

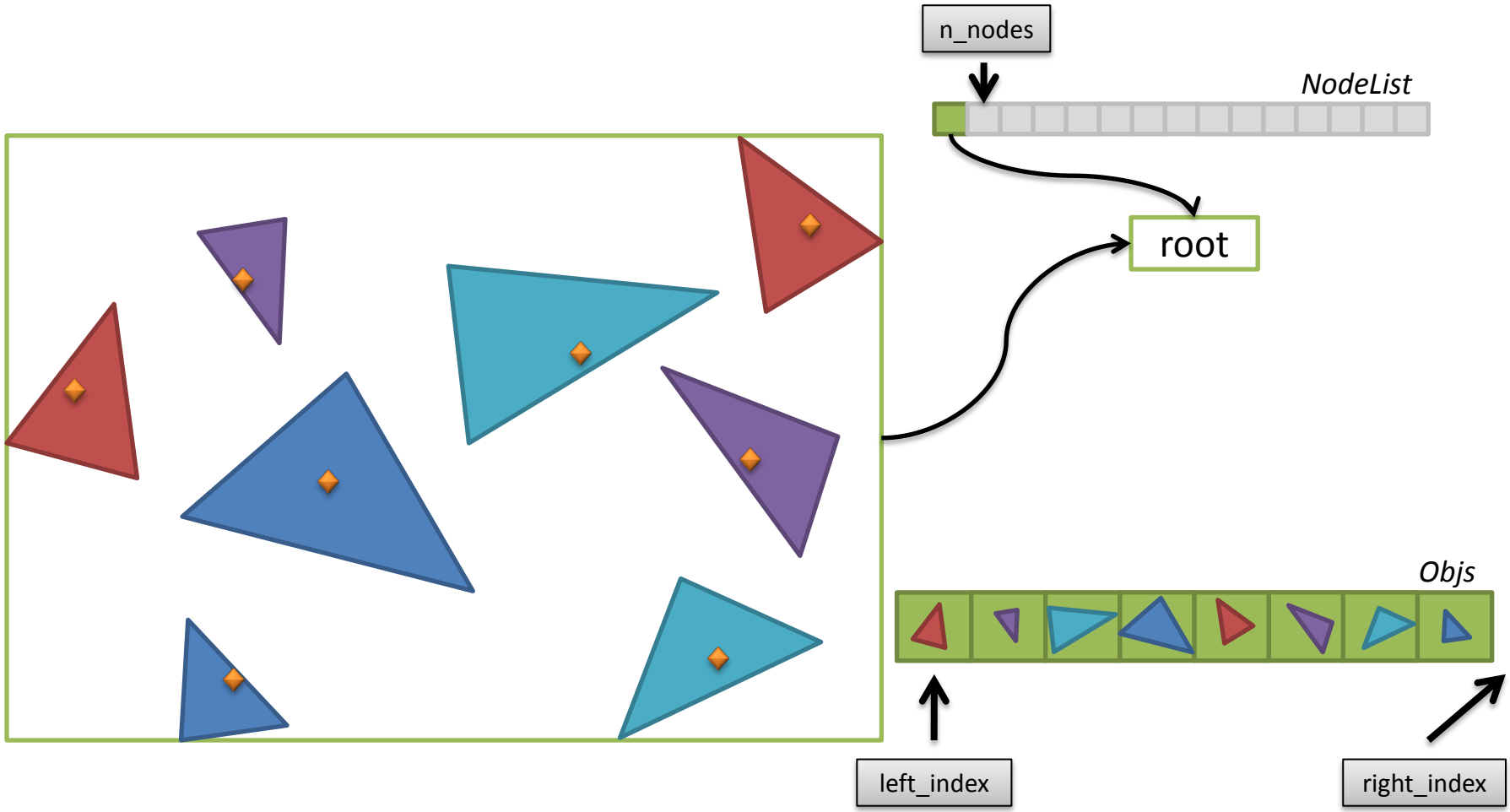


*NodeList*

BVH Intersectable list – *Objs*

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

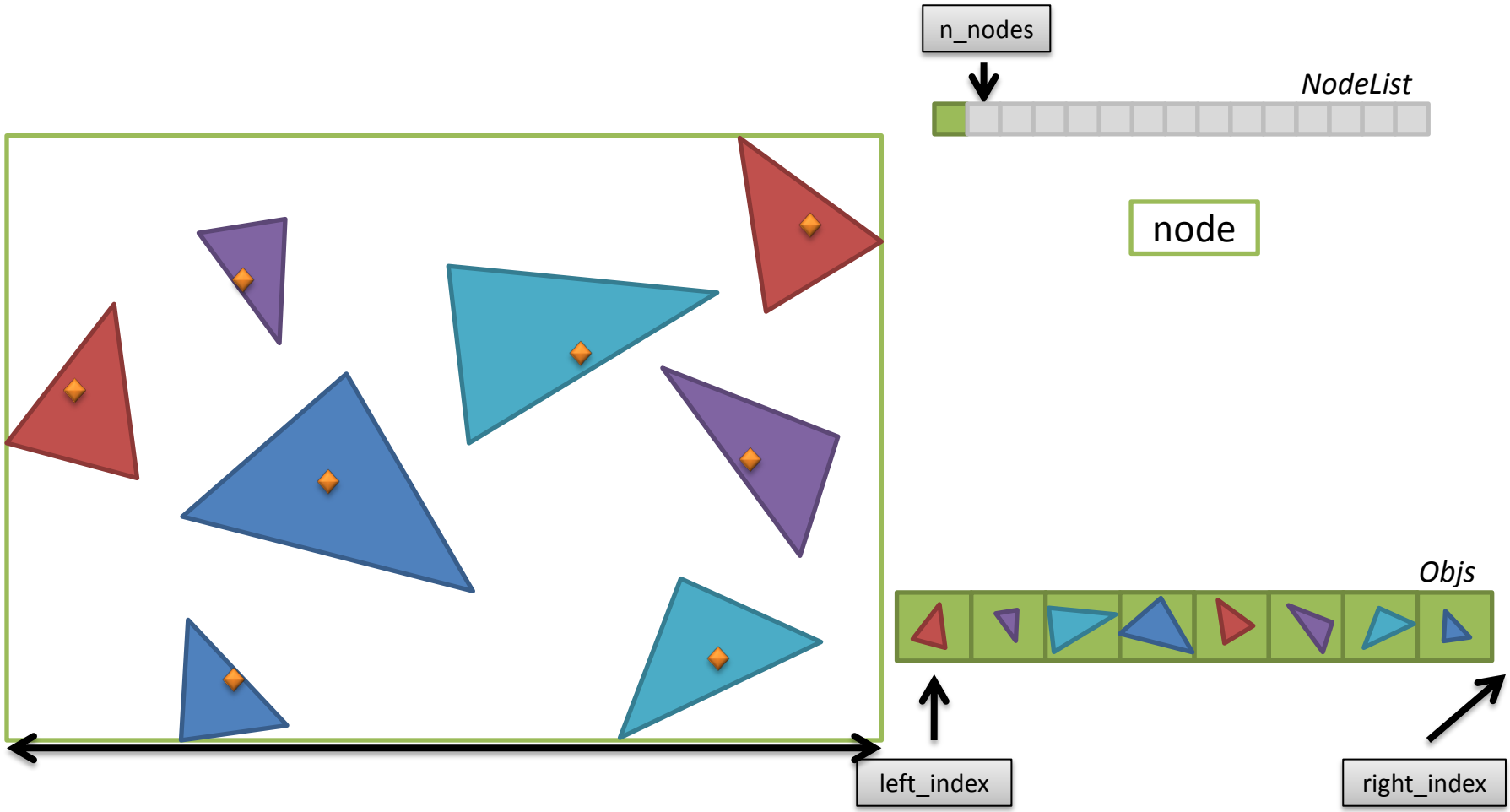



Set *left_index* = 0, *right_index* = Objs.size() and n_nodes = 1.

Set the world bounding box to the root node using BVHNode::setAABB(box).

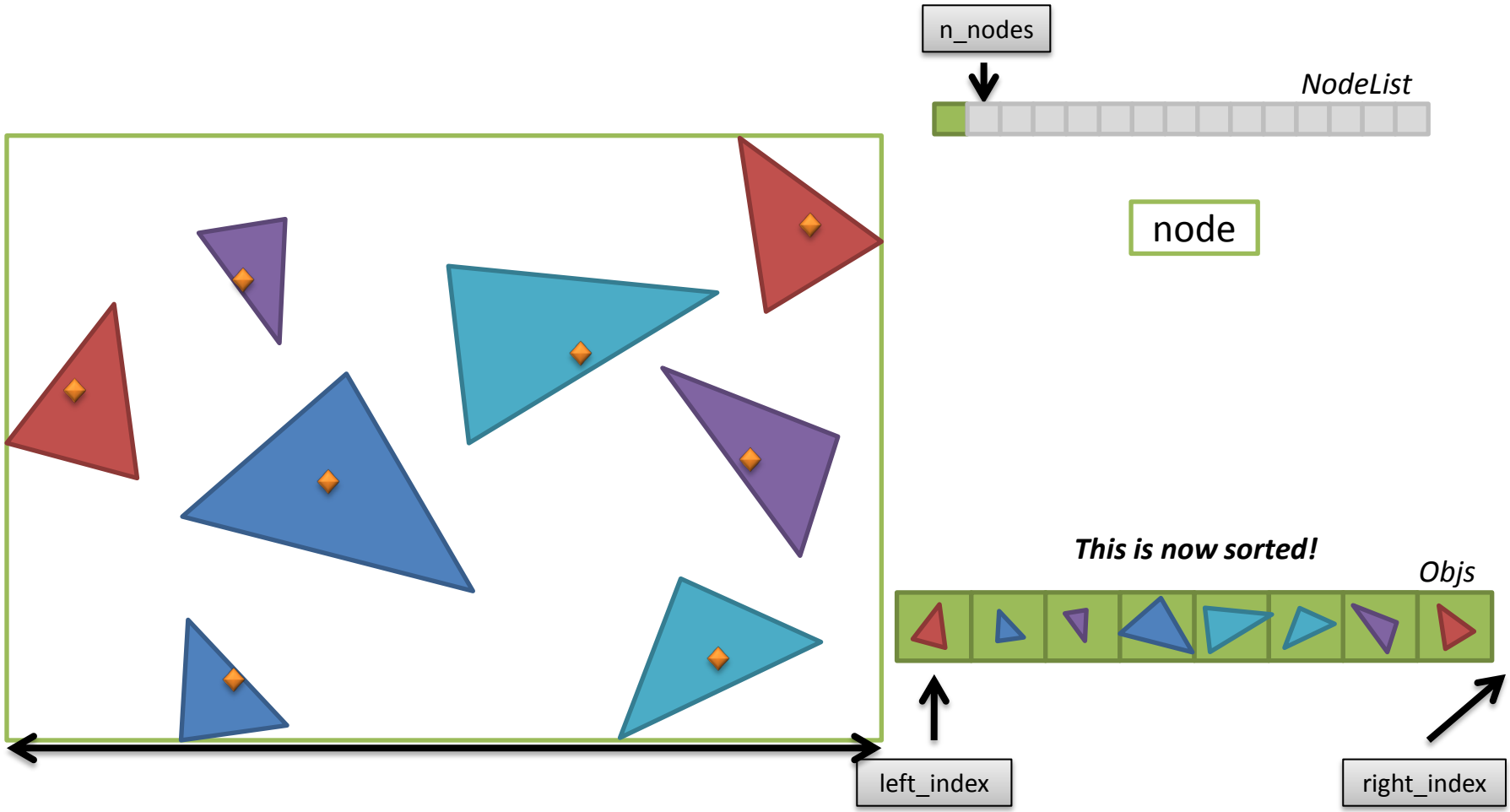
***Then start building recursively.***

# Construction



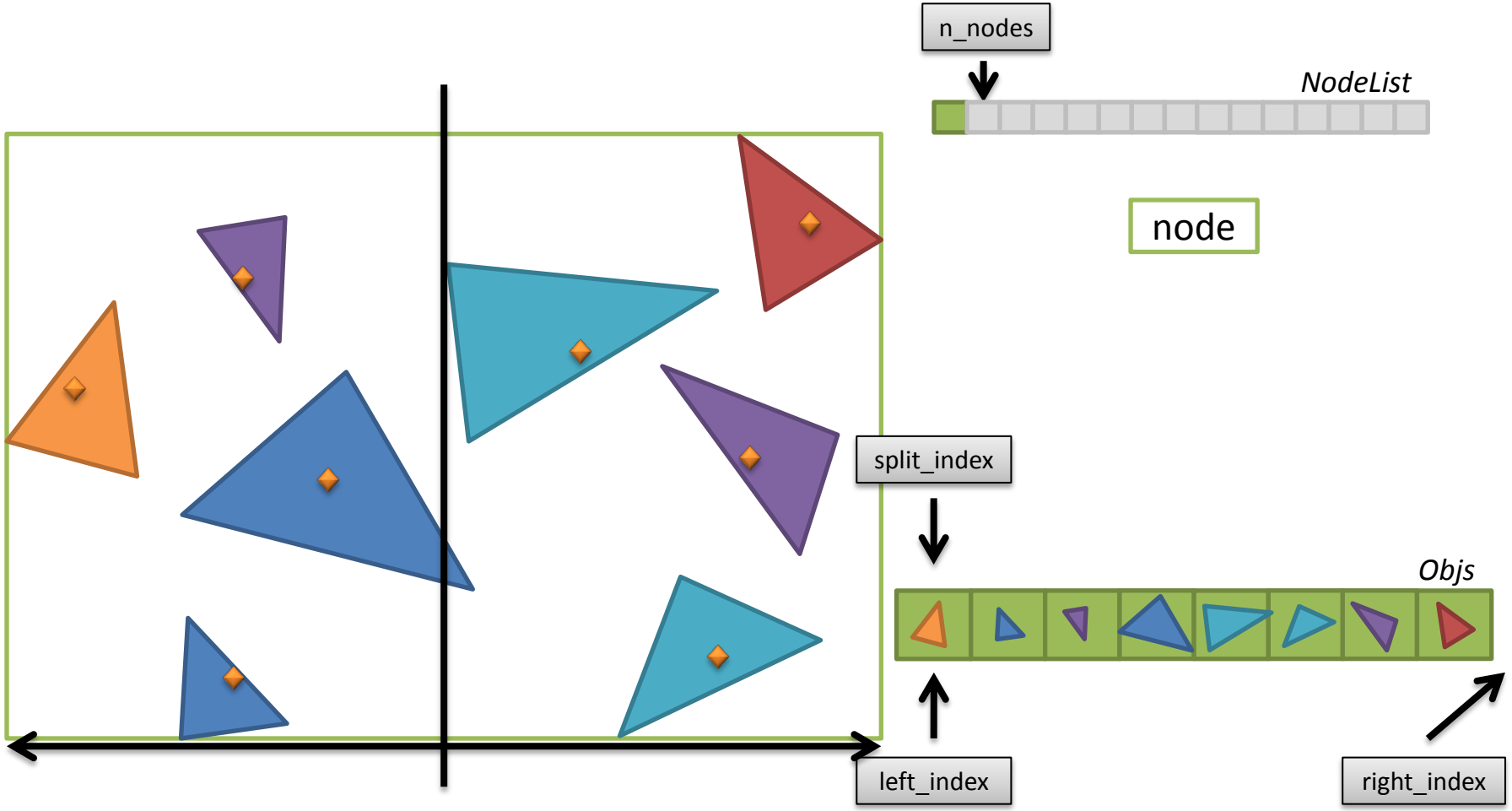

First, check if the number of intersectables is fewer than the threshold (let's say 2 in this case). It isn't.

# Construction



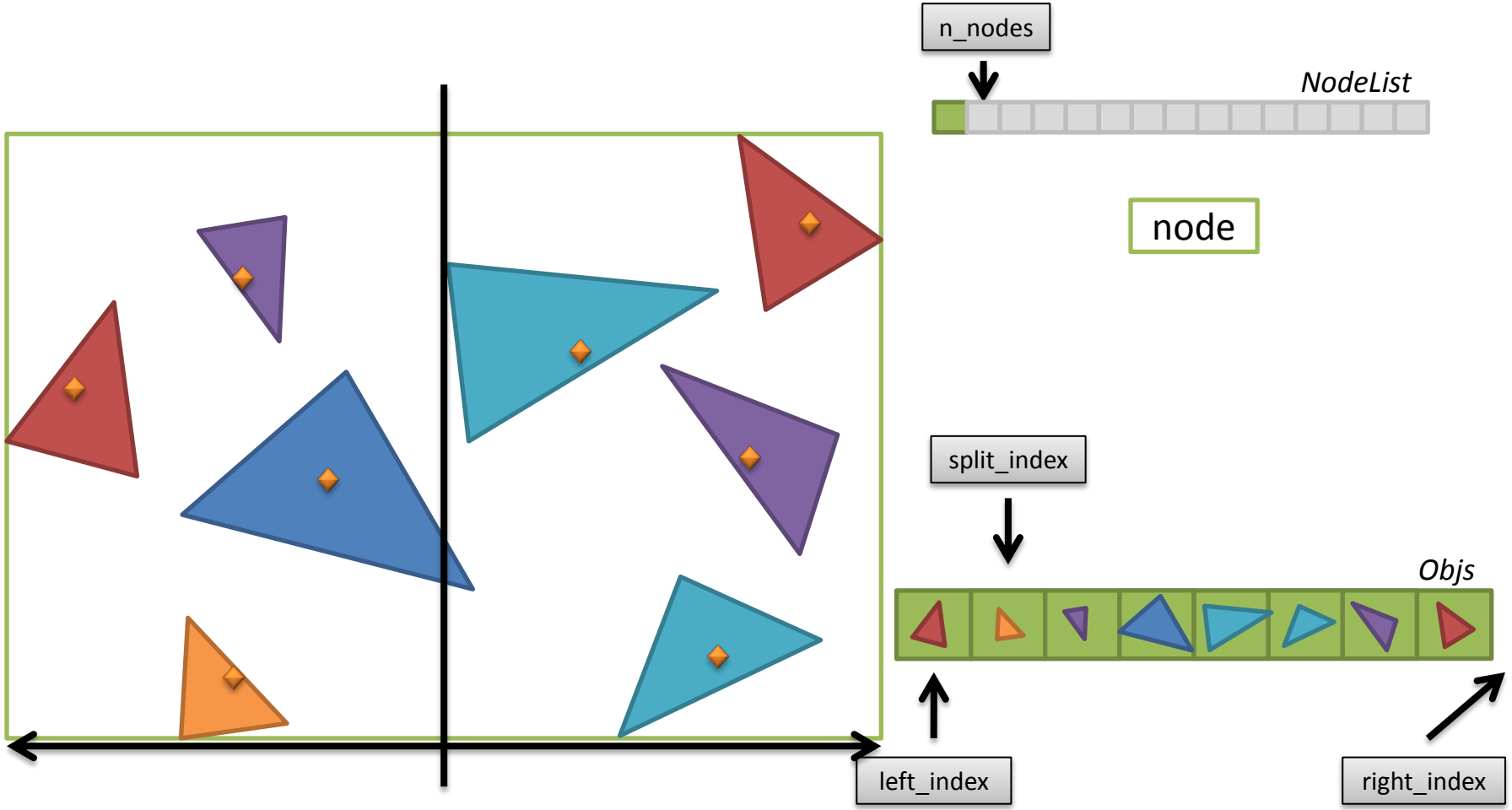

Find largest dimension *d* and sort the elements in that dimension.

# Construction



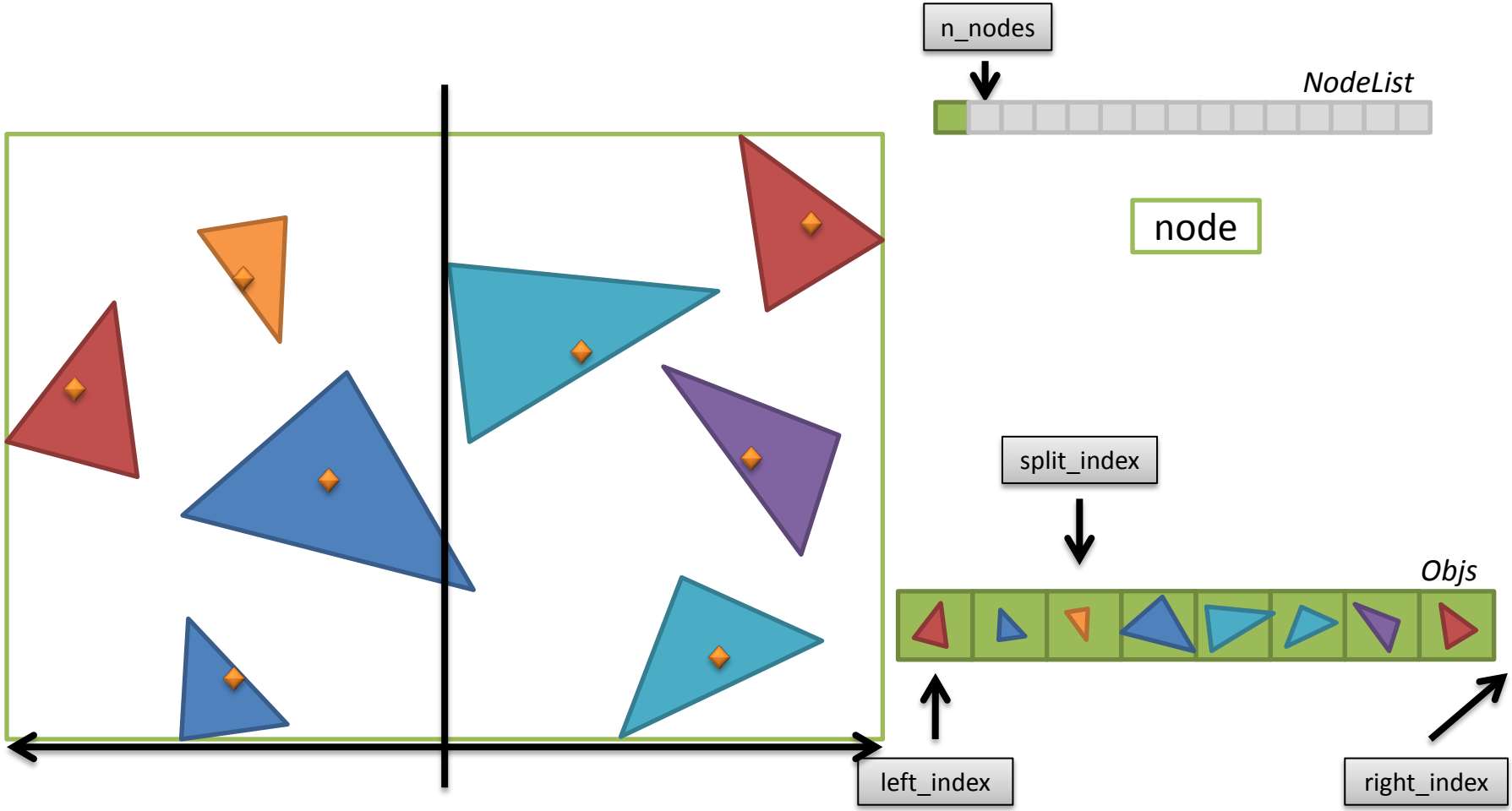

Find the *split_index,* where the mid point divides the primitives in a *left* and *right* side

# Construction



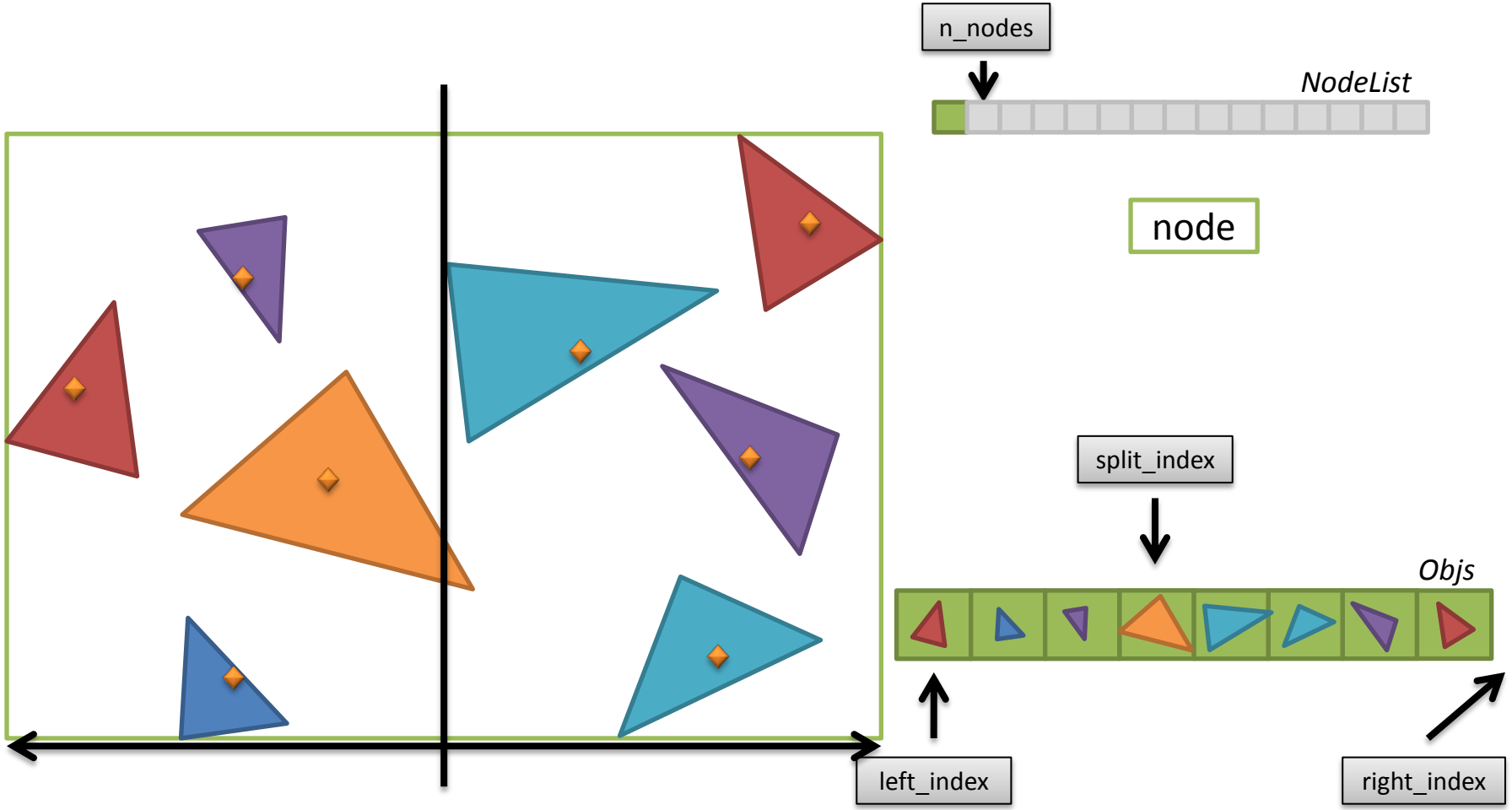

Find the *split_index,* where the mid point divides the primitives in a *left* and *right* side

# Construction



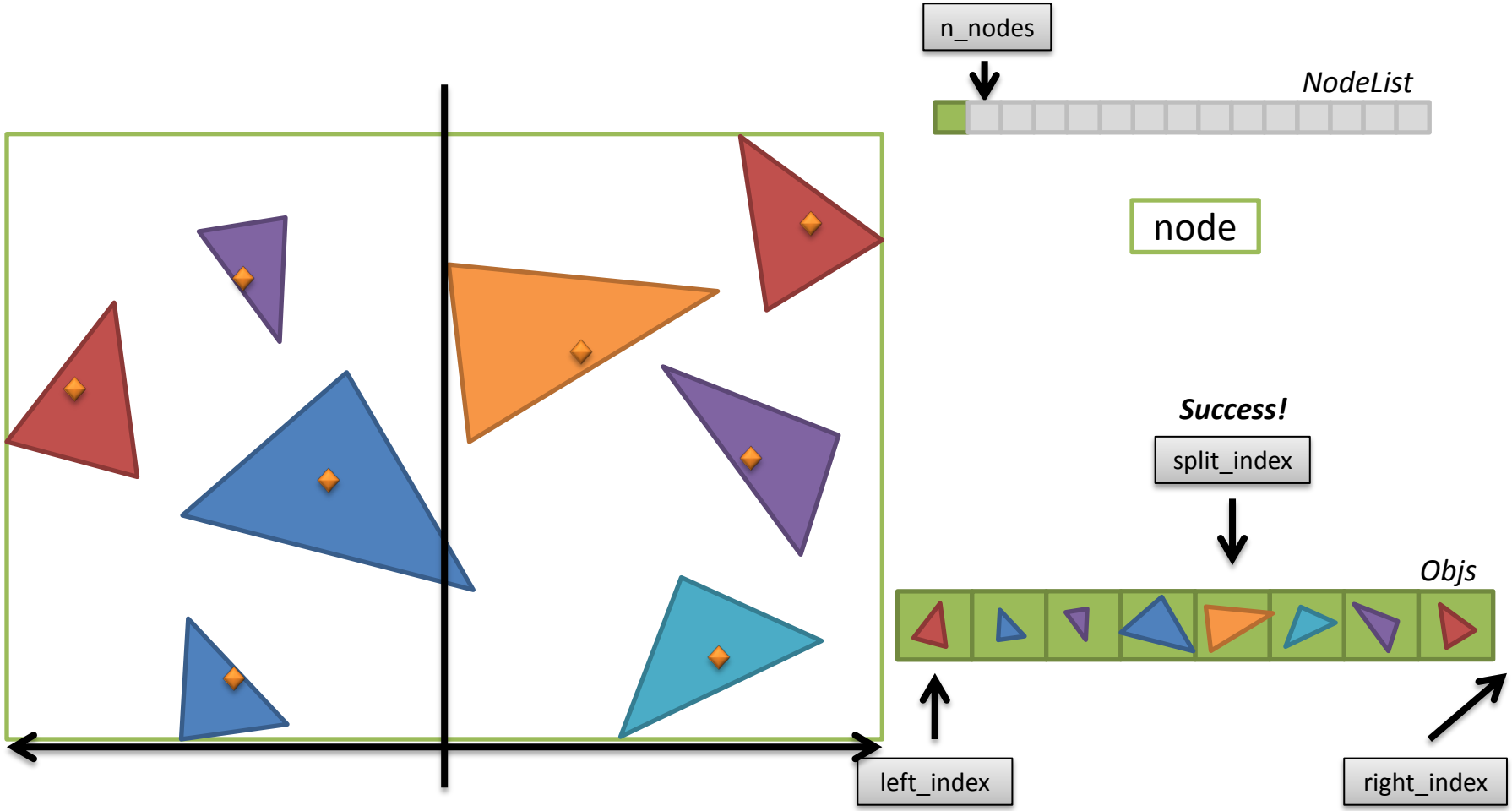

Find the *split_index,* where the mid point divides the primitives in a *left* and *right* side

# Construction



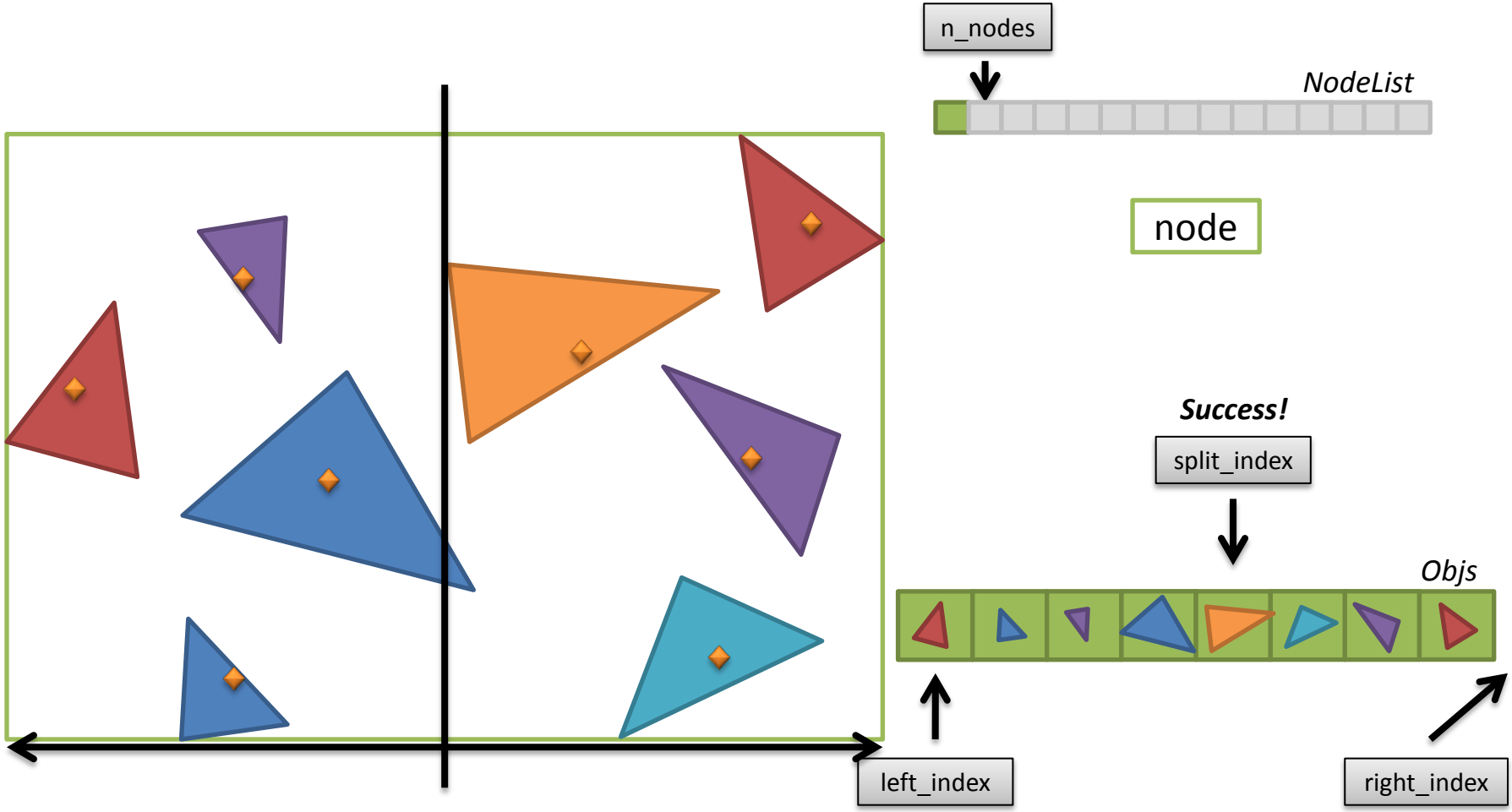

Find the *split_index,* where the mid point divides the primitives in a *left* and *right* side

# Construction



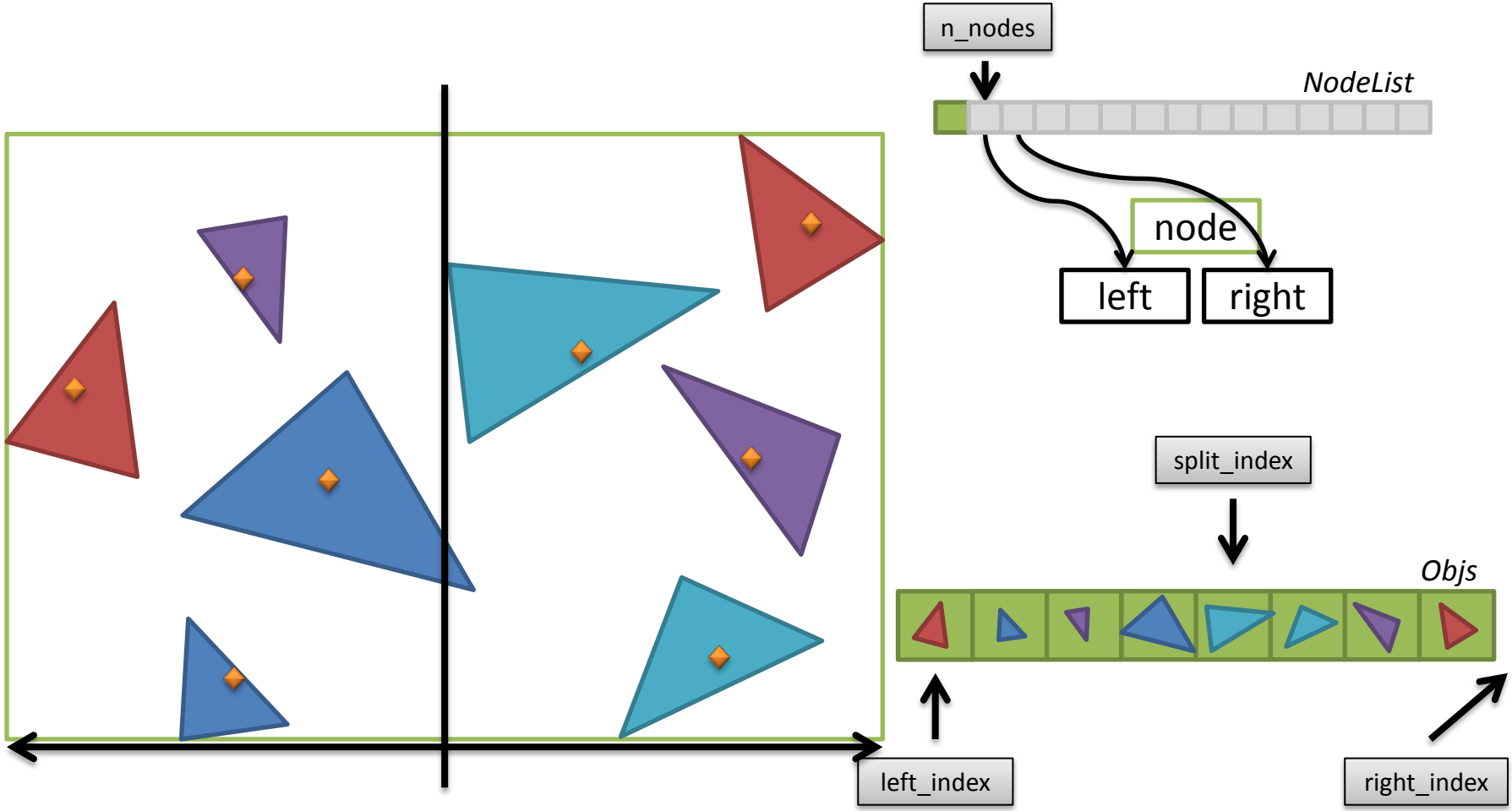

Find the *split_index,* where the mid point divides the primitives in a *left* and *right* side

# Construction



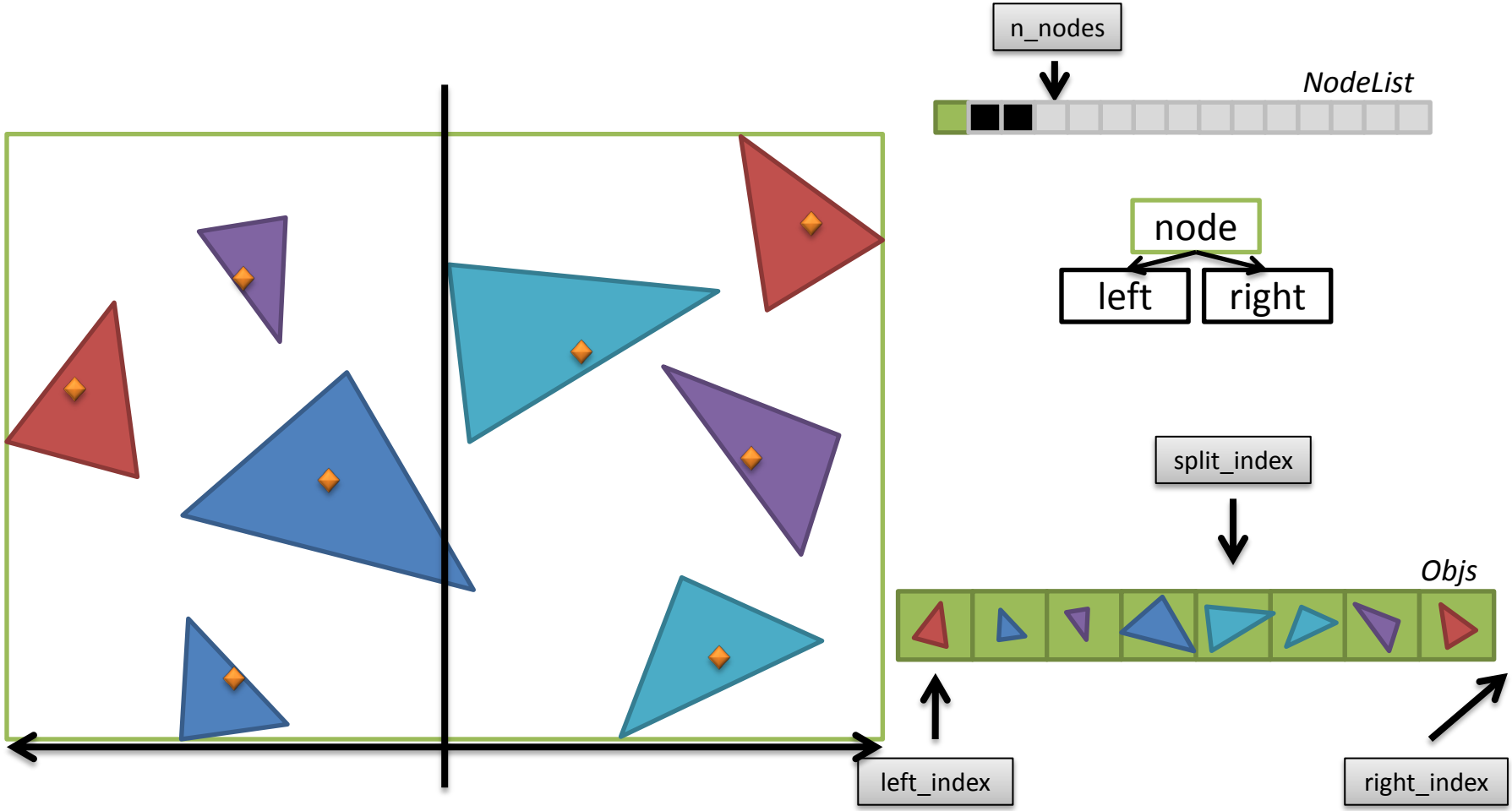

*(We could have used binary search)*

# 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***.

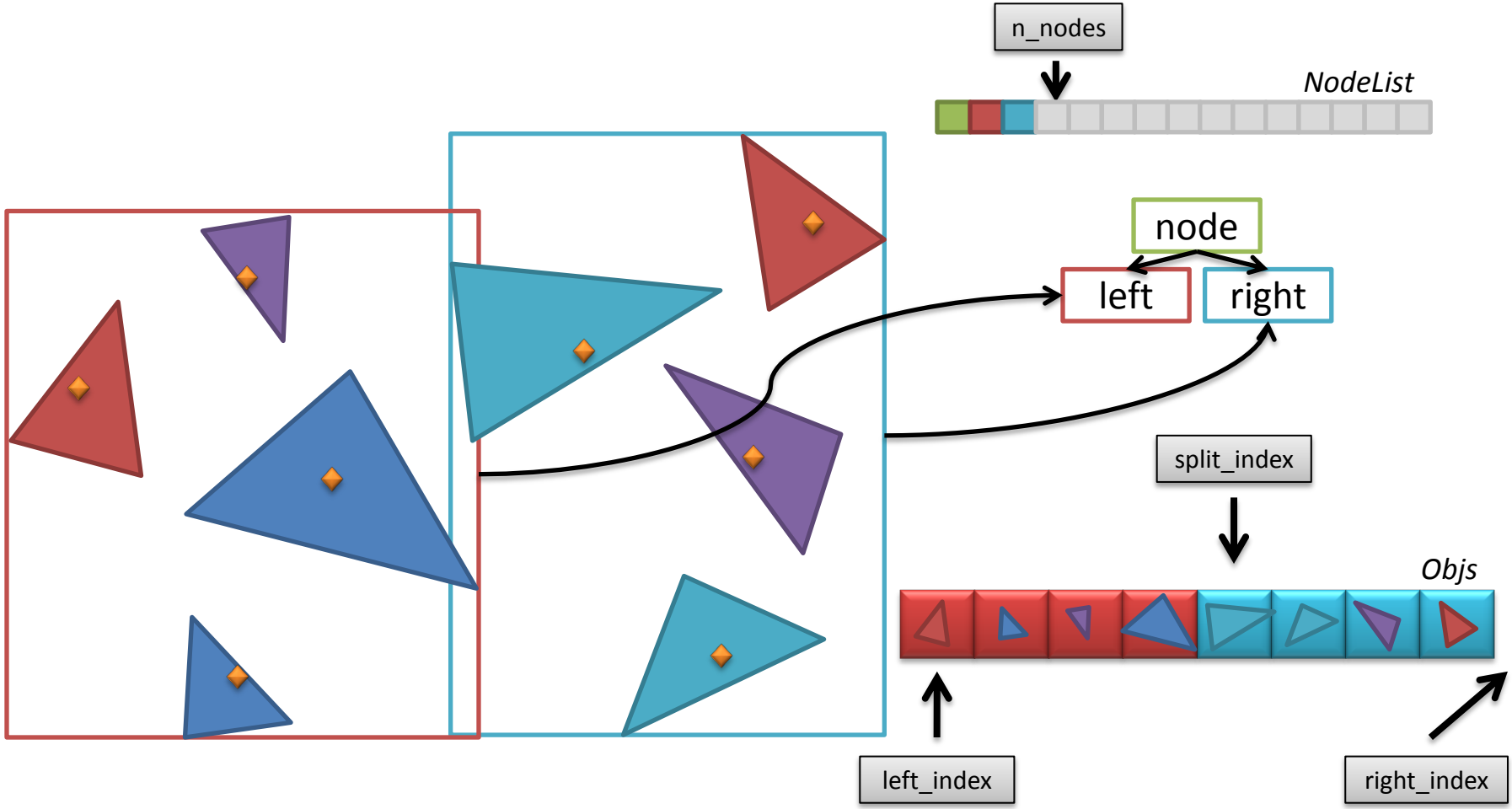
# Construction




Initiate **node** (which is currently the root node) with BVHNode::makeNode(*n_nodes*).

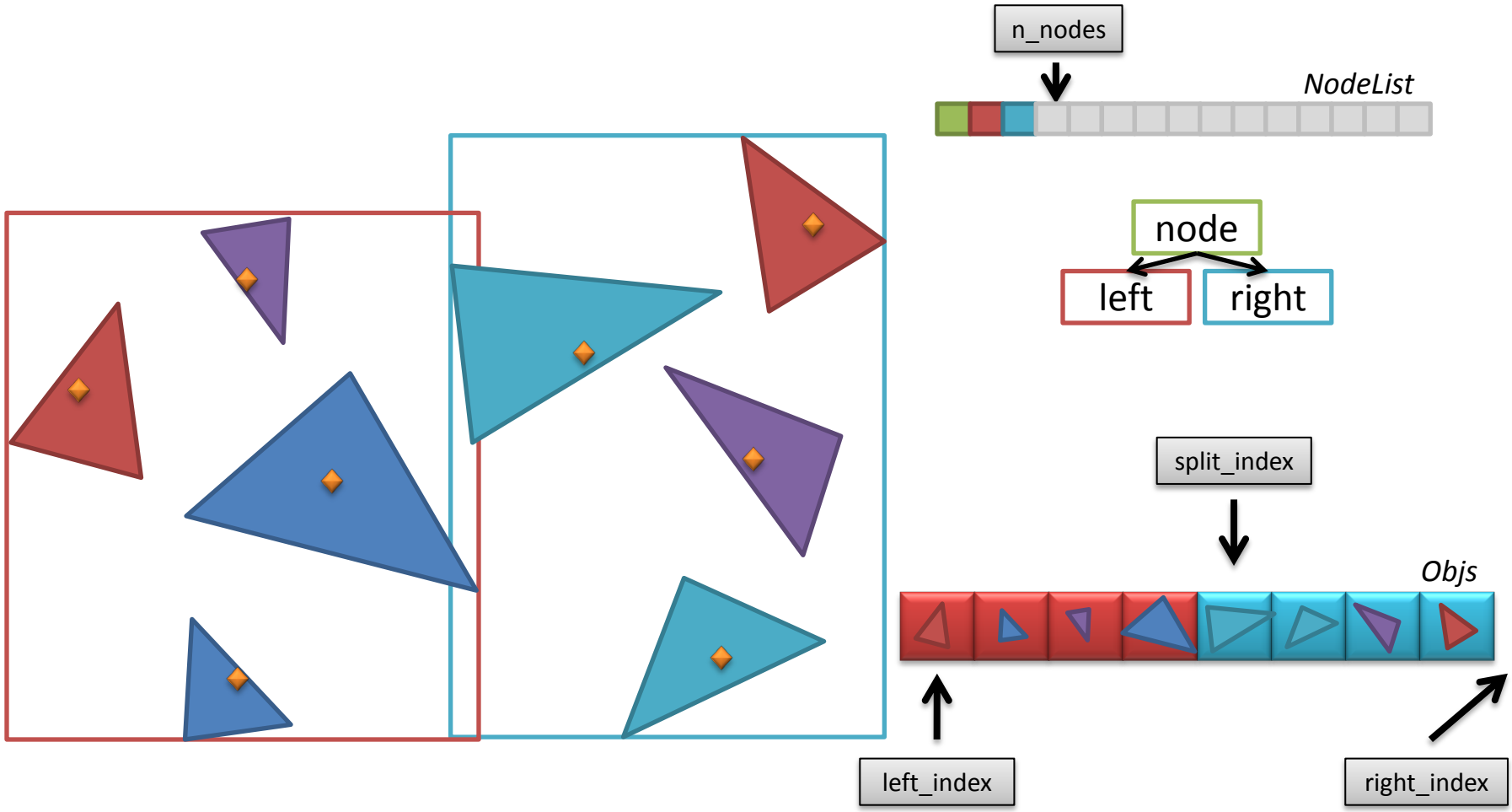
(The index to the right node is always n_nodes + 1)

# Construction



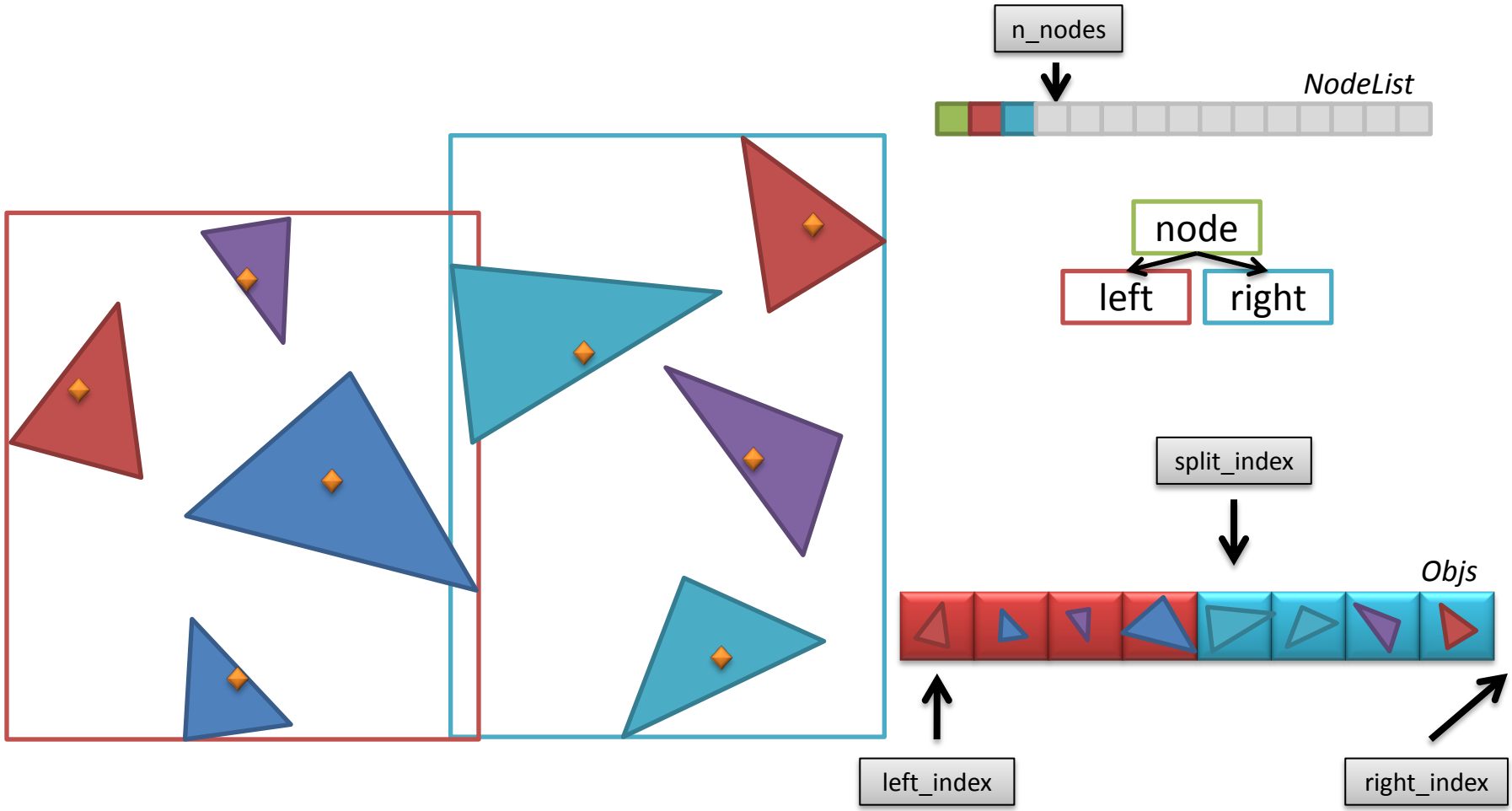

Calculate the bounding boxes for *left* and *right* and assign them to the two newly created nodes

# Construction



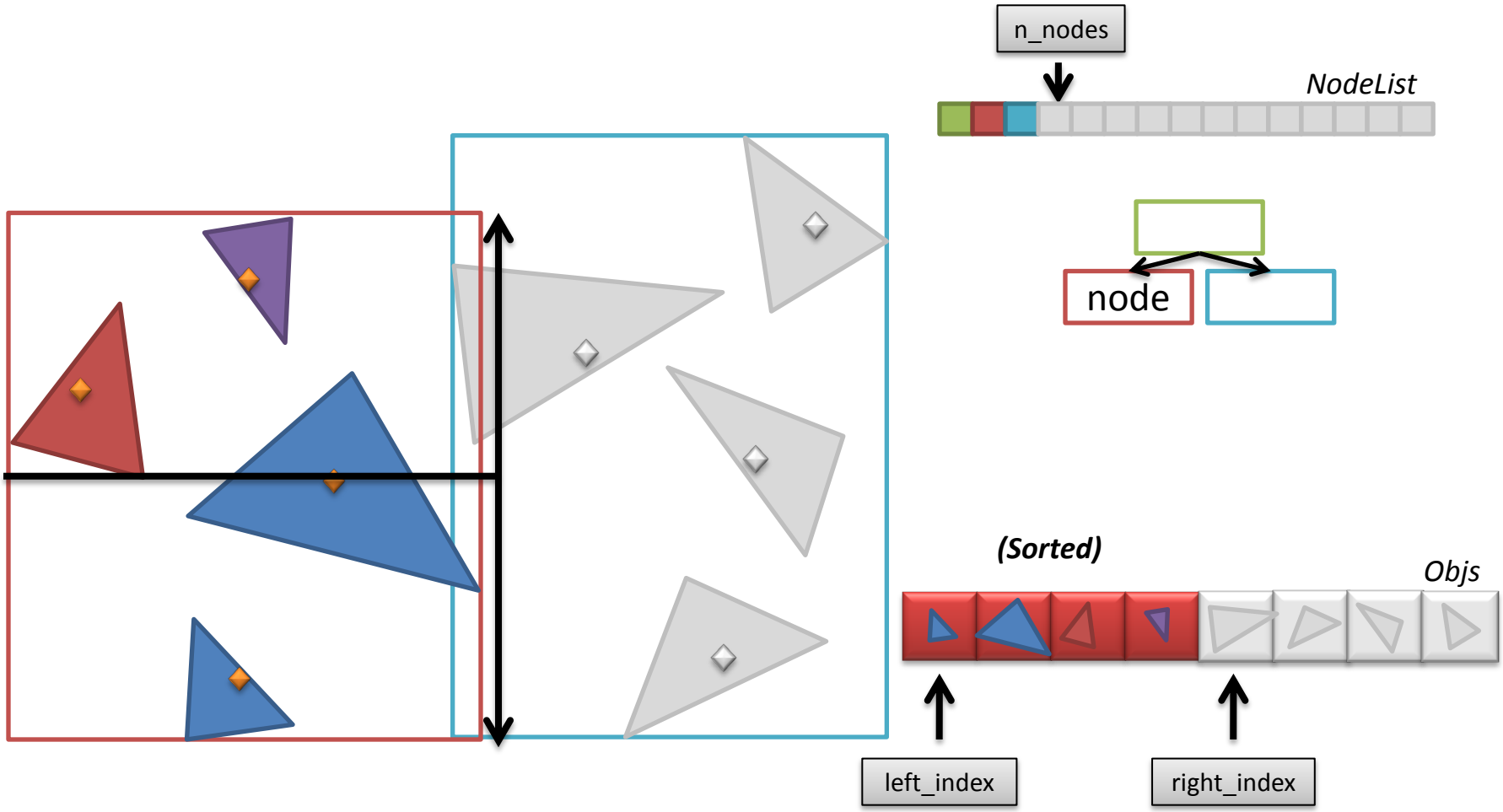

Call the **build_recursive**()-function for the *left* and then the *right* node.

# Construction



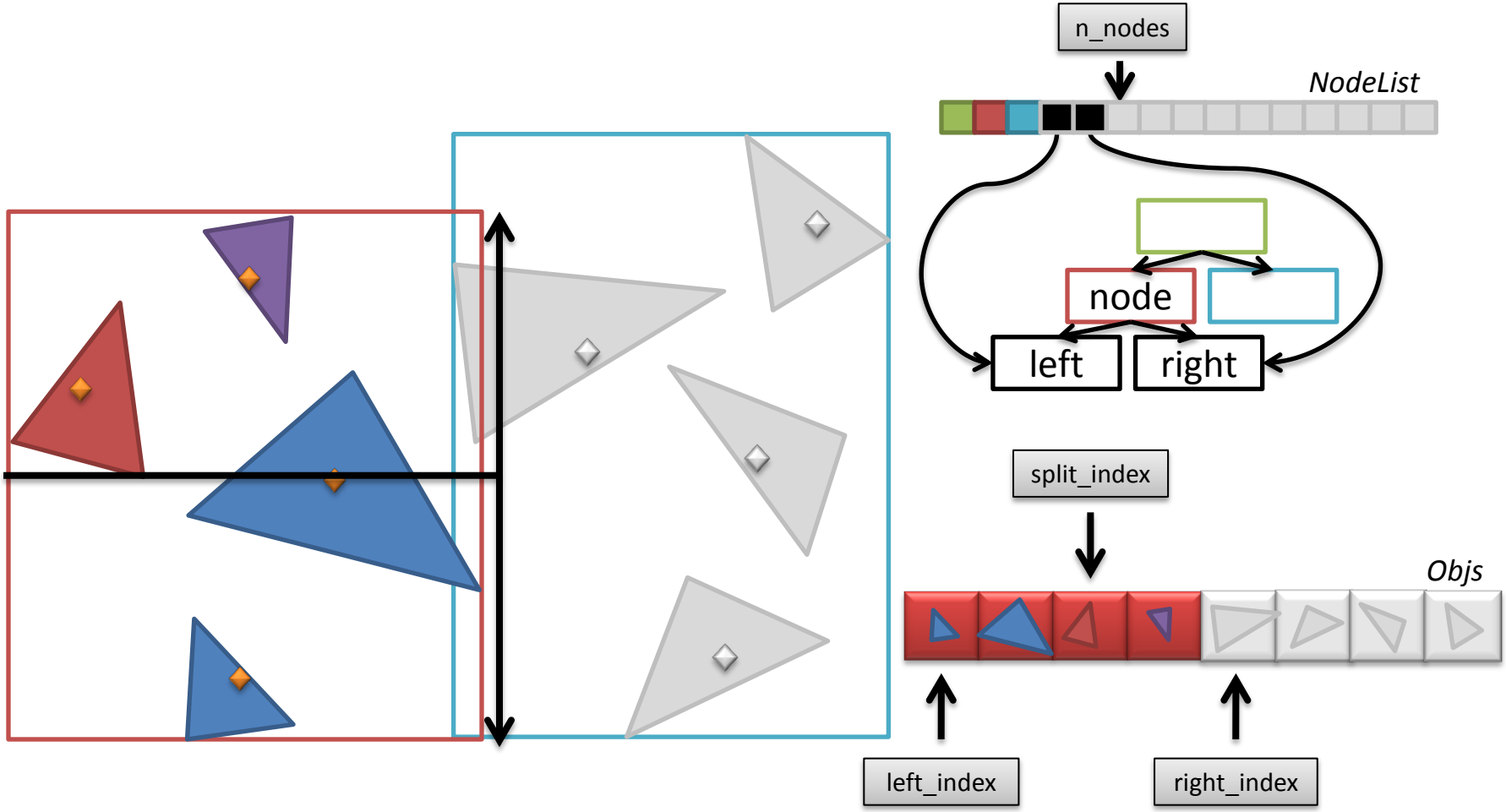

Be sure to pass the correct **Objs-vector indices** to the *left* and *right* nodes. The *left* node is now responsible for *[left_index, split_index)* and the *right* node for *[split_index, right_index)*.

# Construction

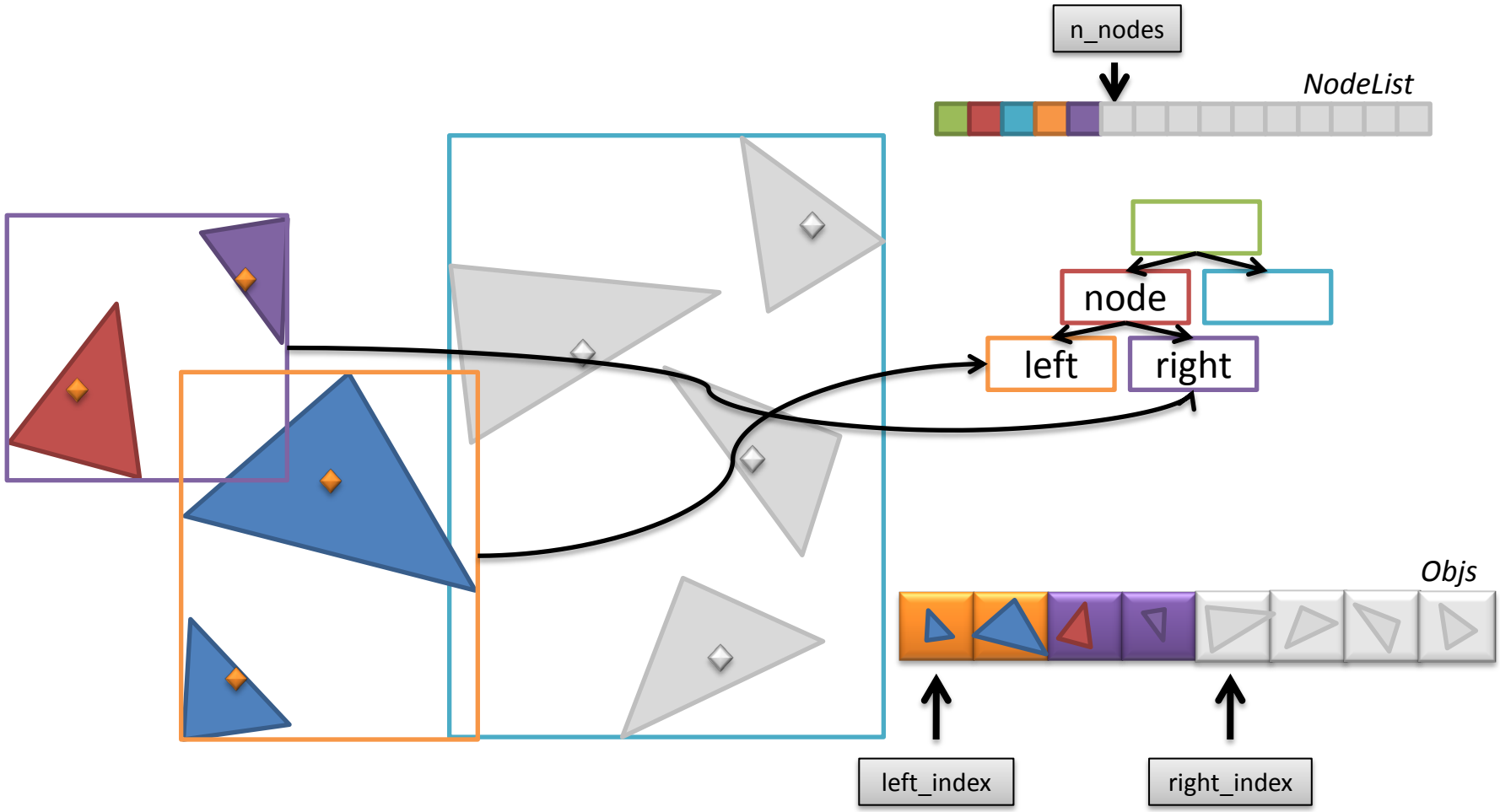



Processing of the left node yields the following result…

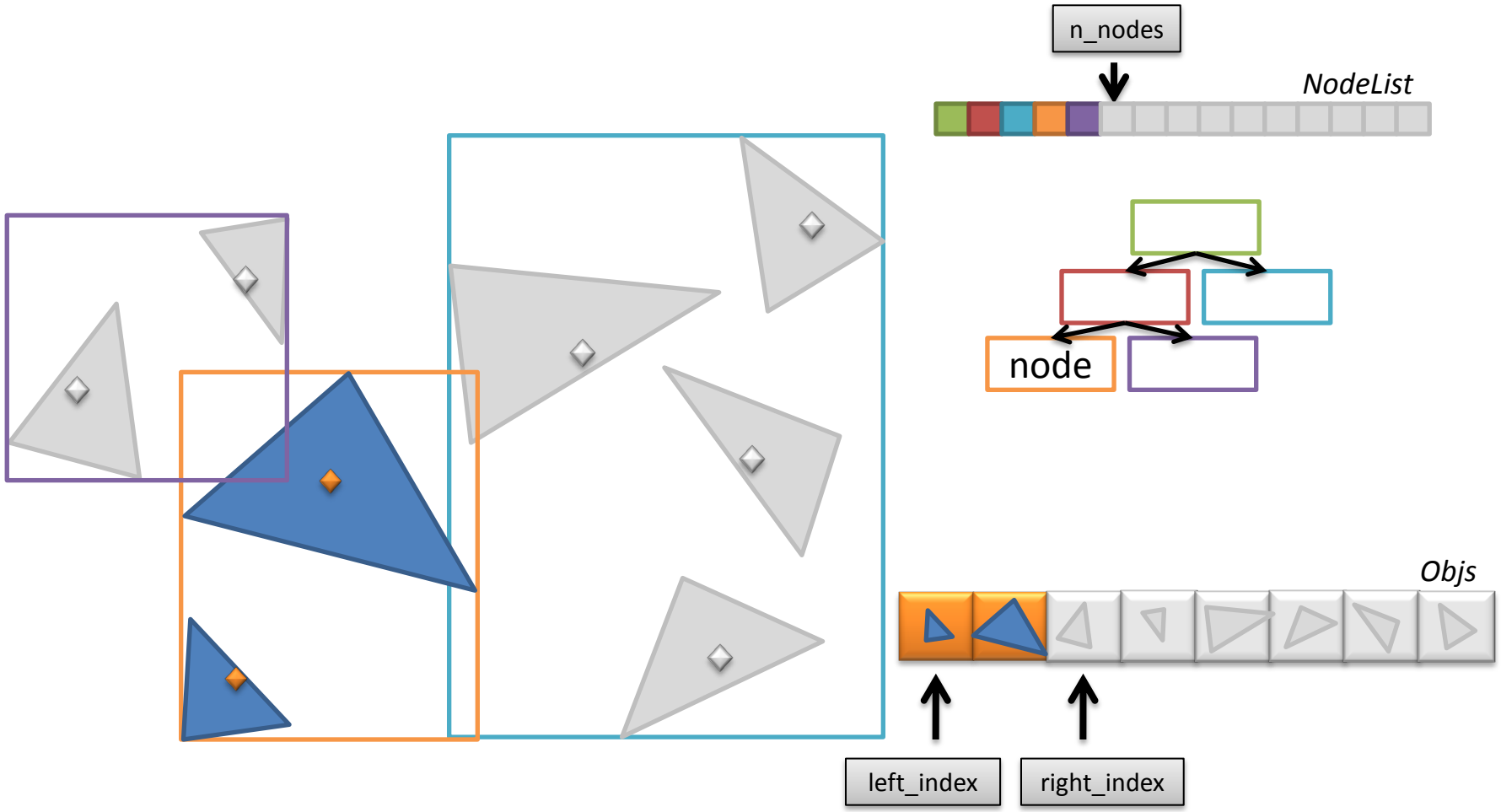
# Construction



# Construction



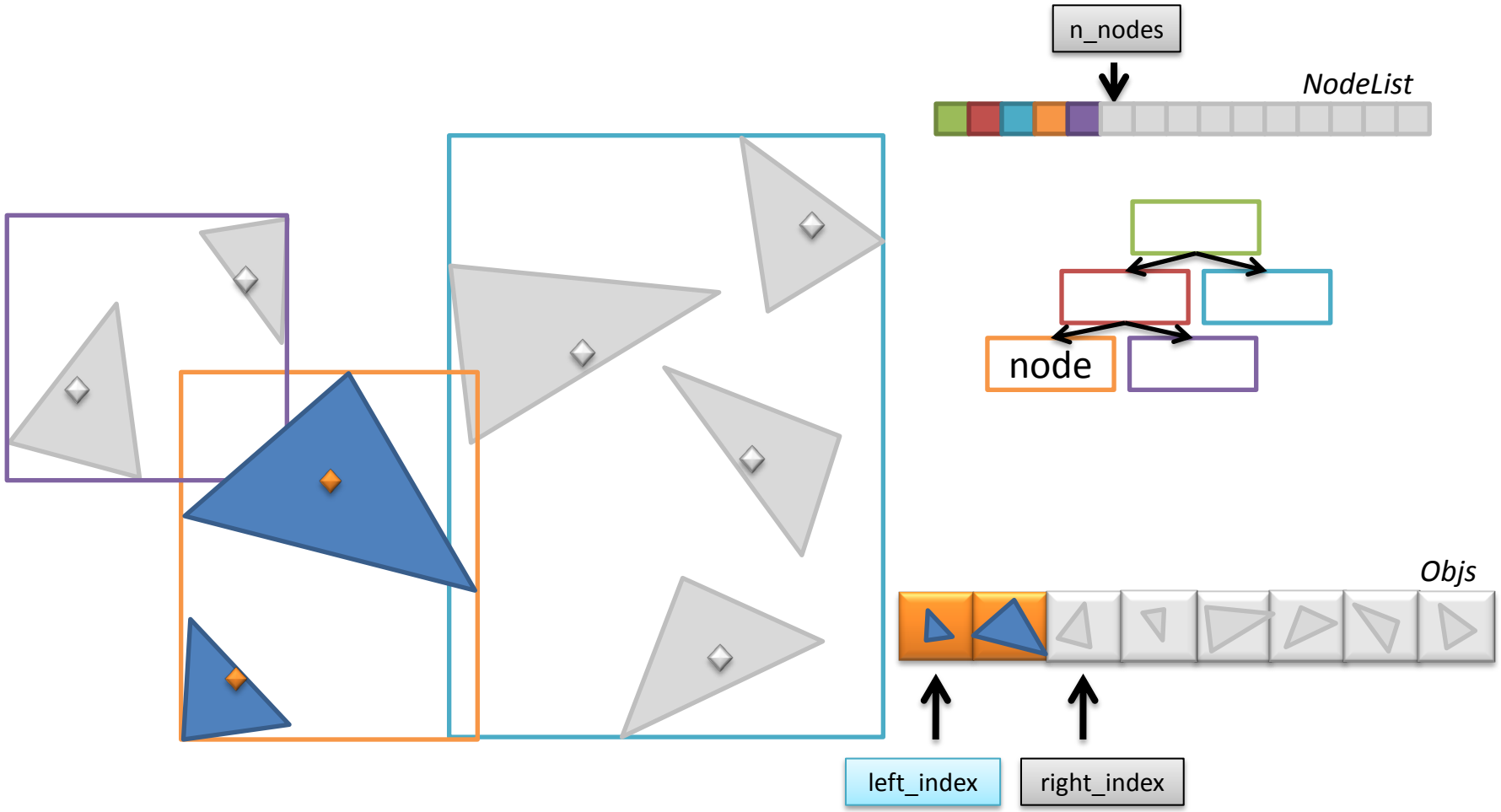
# Construction




Finally we have only 2 primitives in the node:

**(right_index – left_index <= 2)**

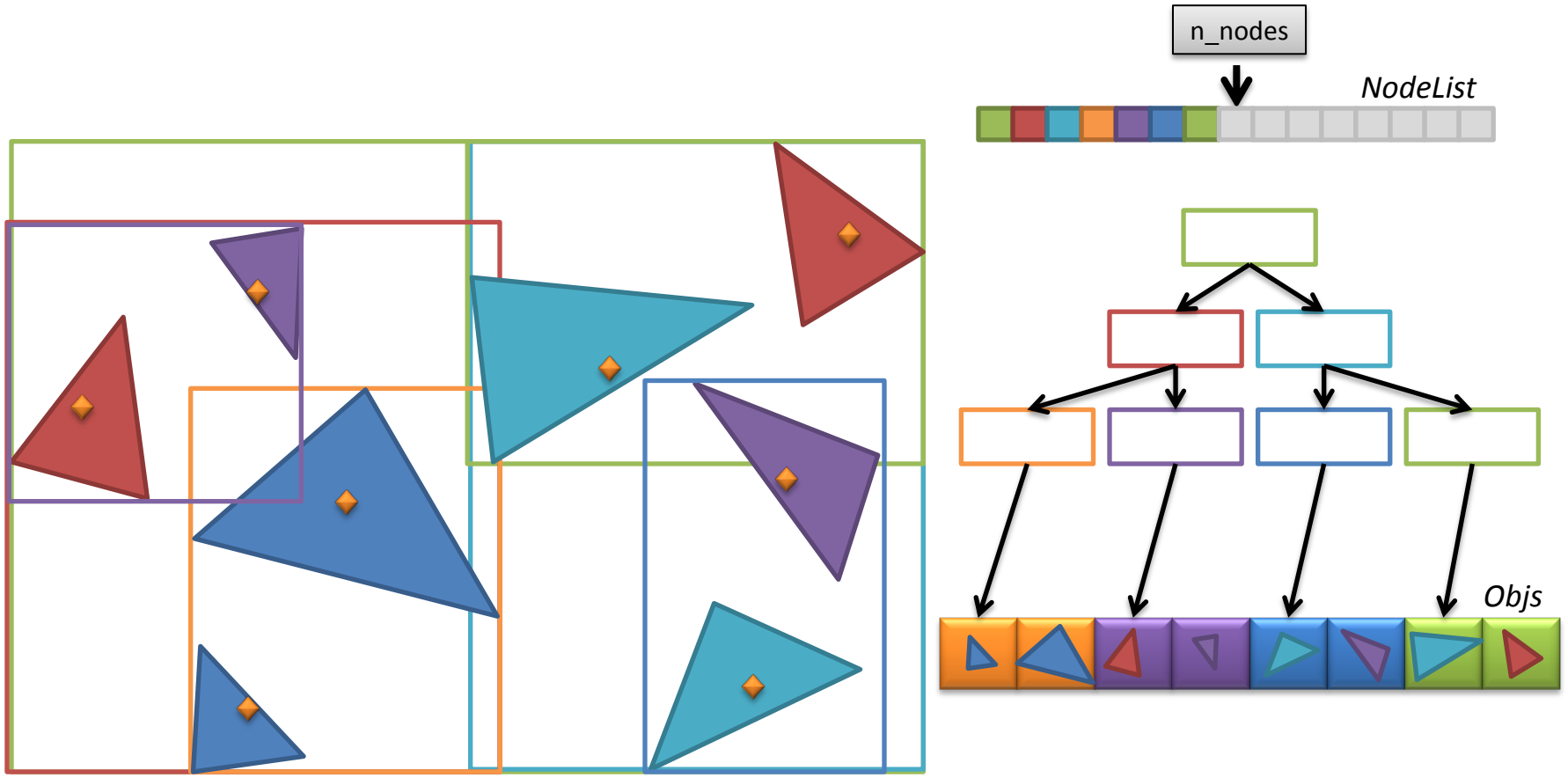
# Construction




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;
};
```

# BVH intersection

- Construction: offline process, usually CPU
- Intersection: rendering process, can be GPU

- Slides from Magnus Andersson (LTH)
- http://fileadmin.cs.lth.se/cs/Education/EDAN30/lectures/S2-bvh.pdf

# Intersection

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

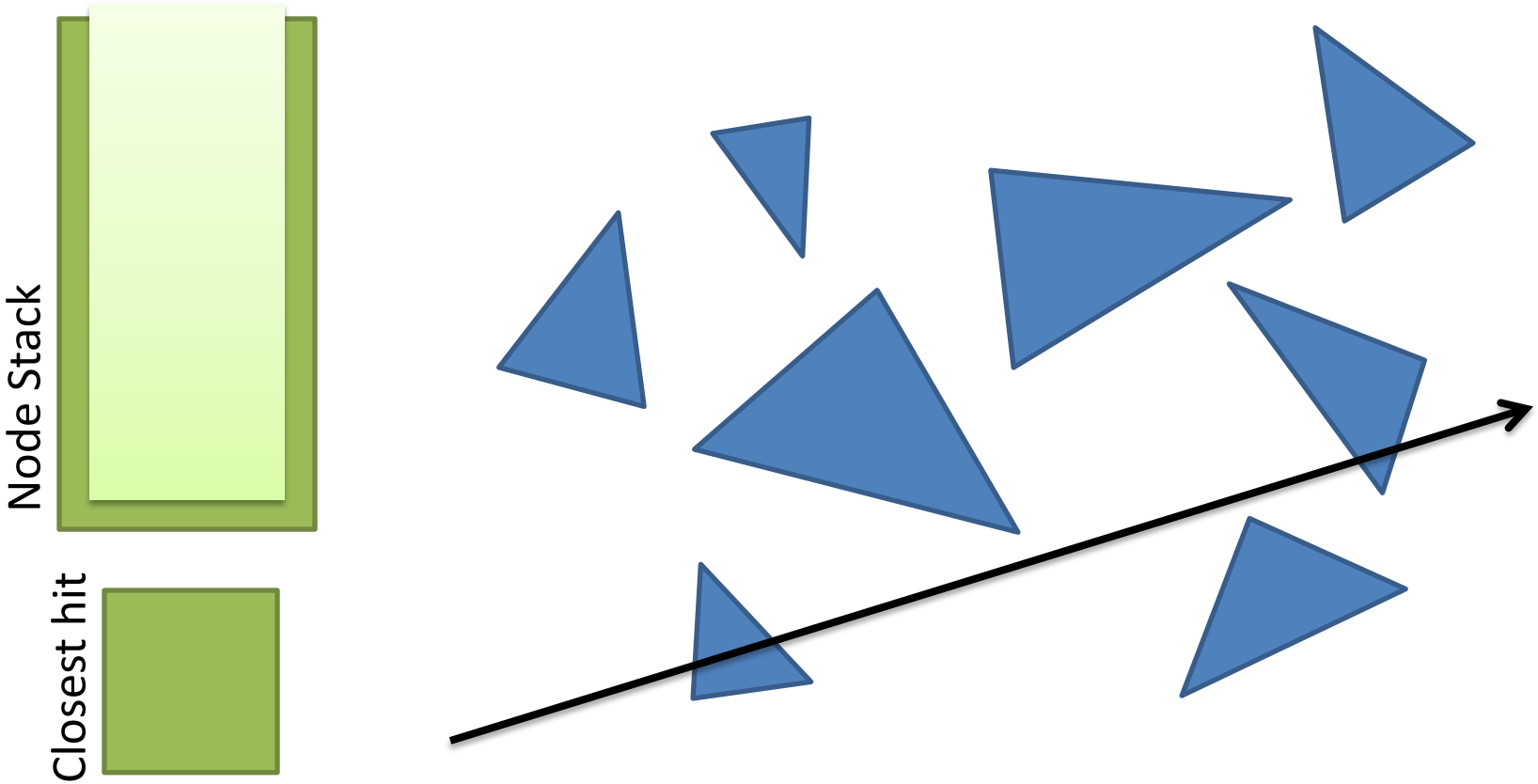
    ```
    bool BVHAccelerator::intersect(const Ray& ray);
    ```

  - *Closest hit*

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

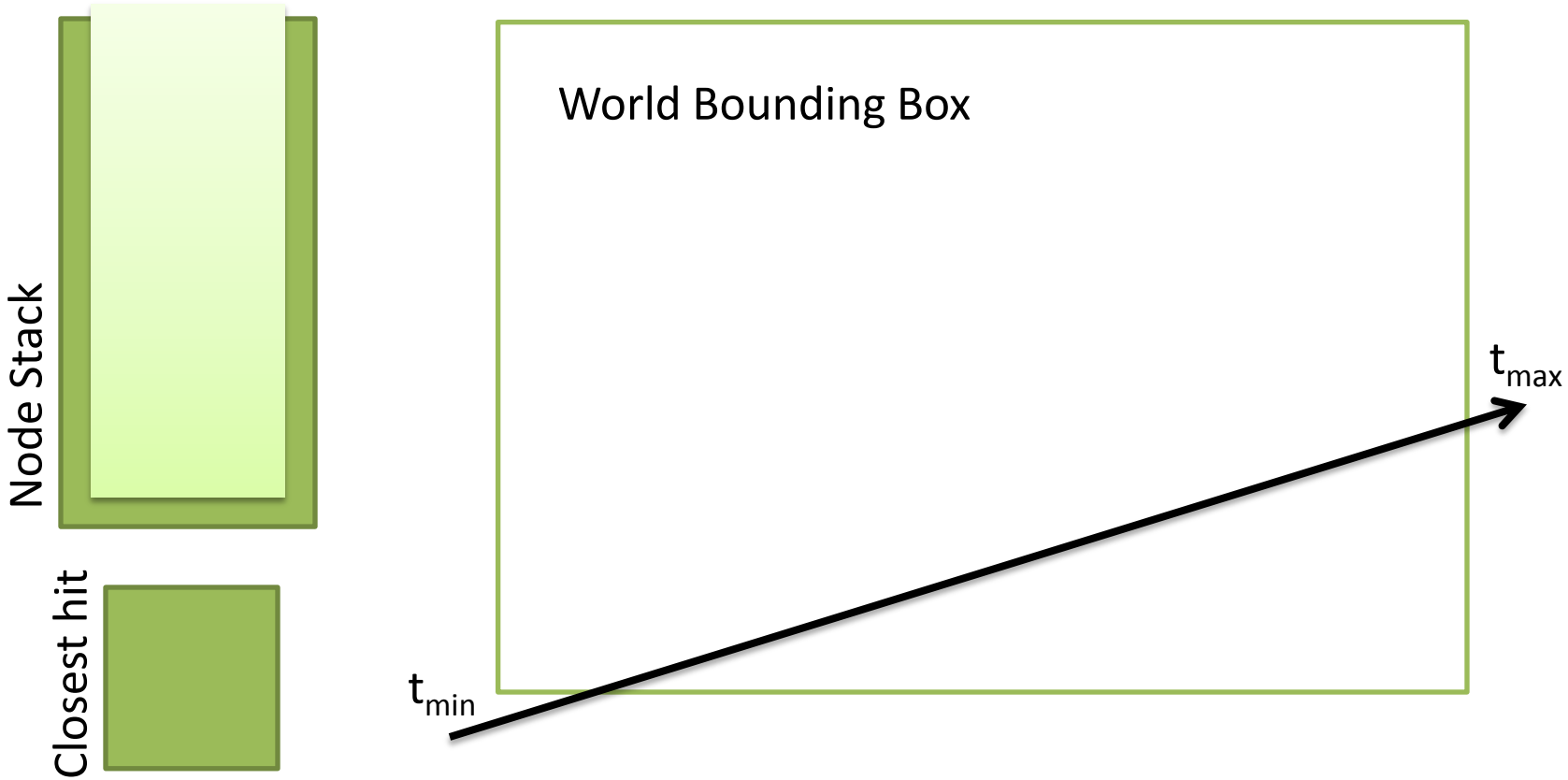
- The two functions are very similar. If you have one of them, you can easily implement the other.

# Closest-Hit Intersection




Find closest intersection point

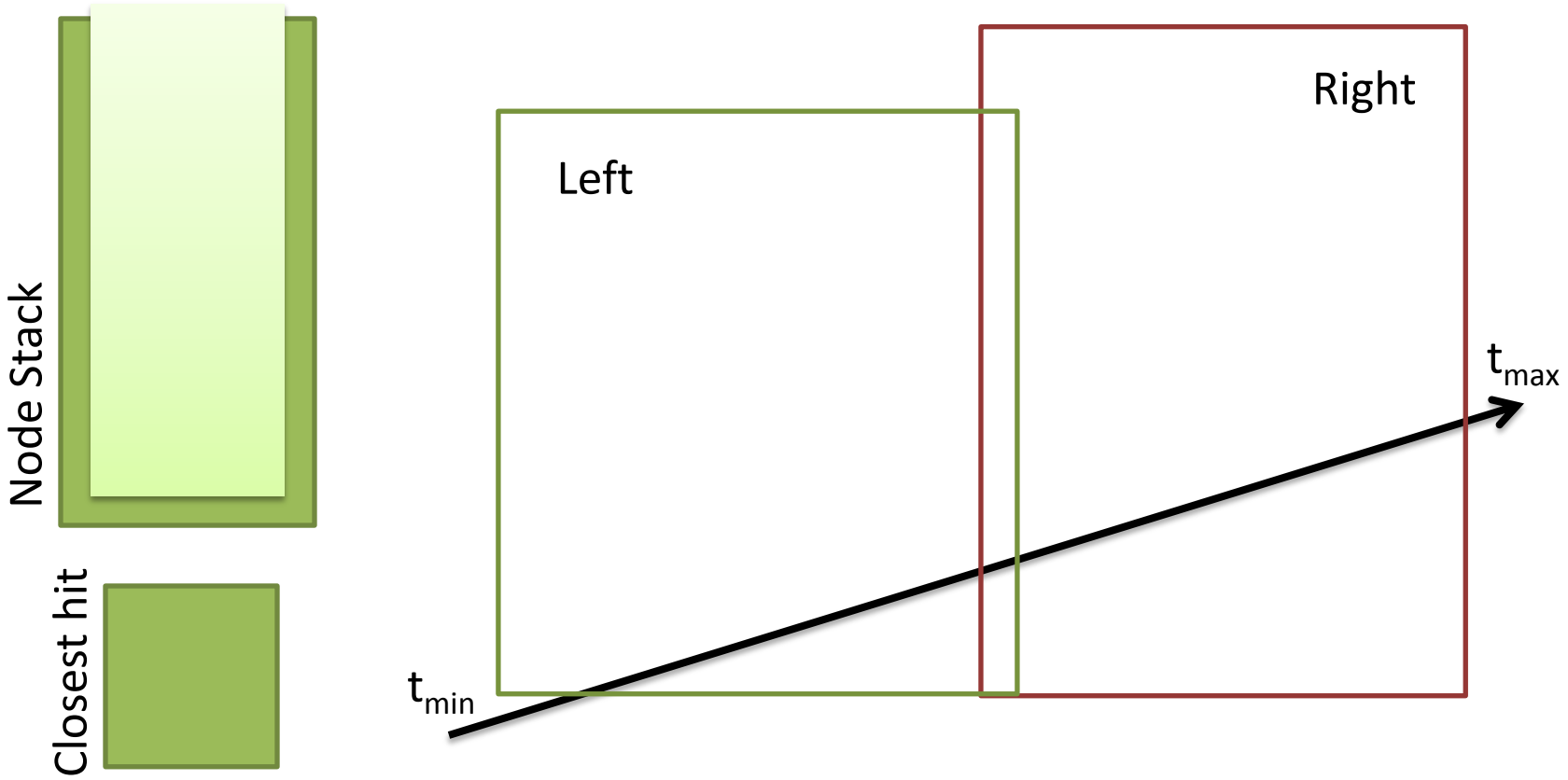
# Closest-Hit Intersection




First check if we even hit the world bounding box.

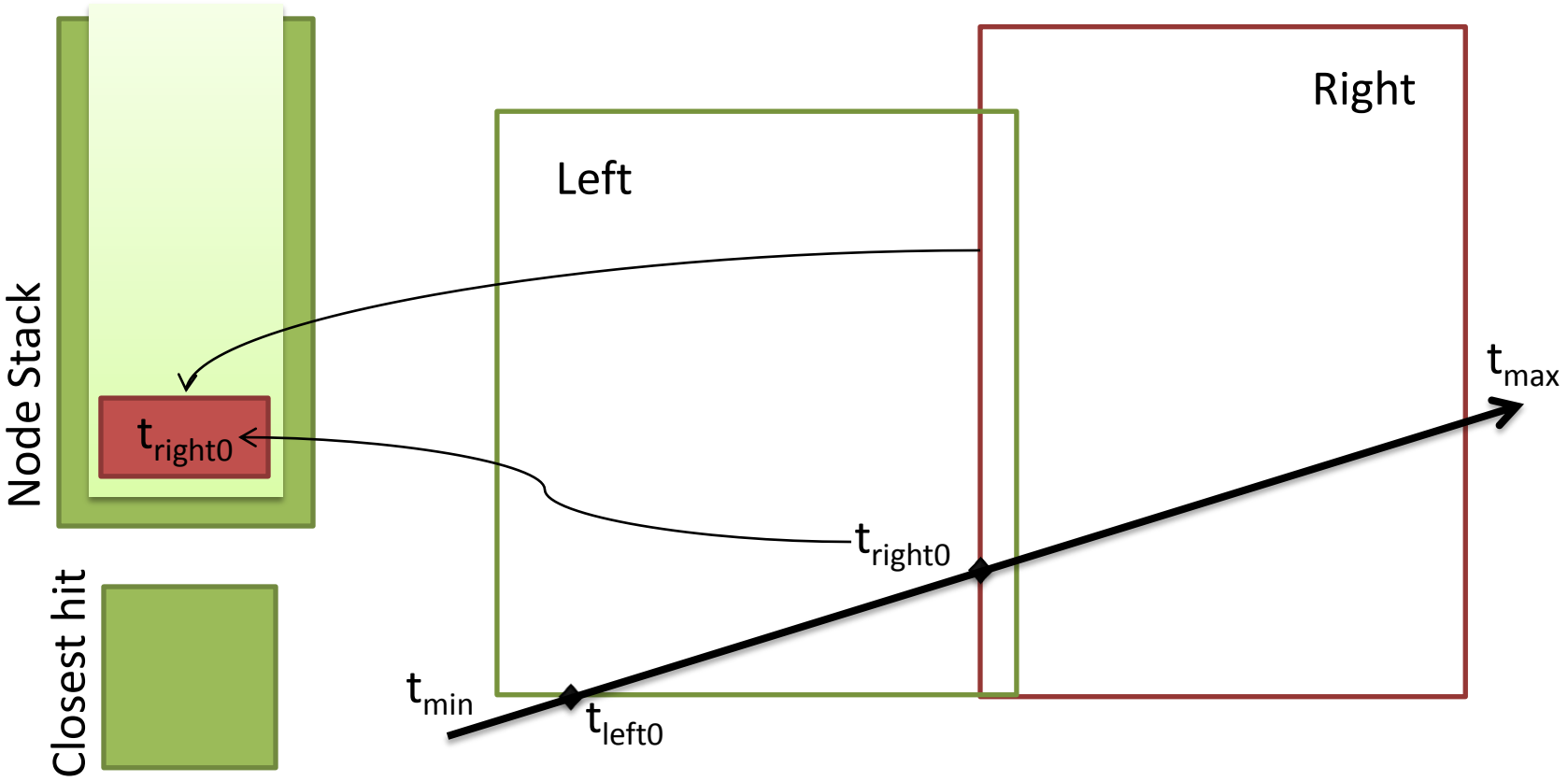
```
bool AABB::intersect(const Ray& r, float& tmin, float& tmax) const;
```

# Closest-Hit Intersection



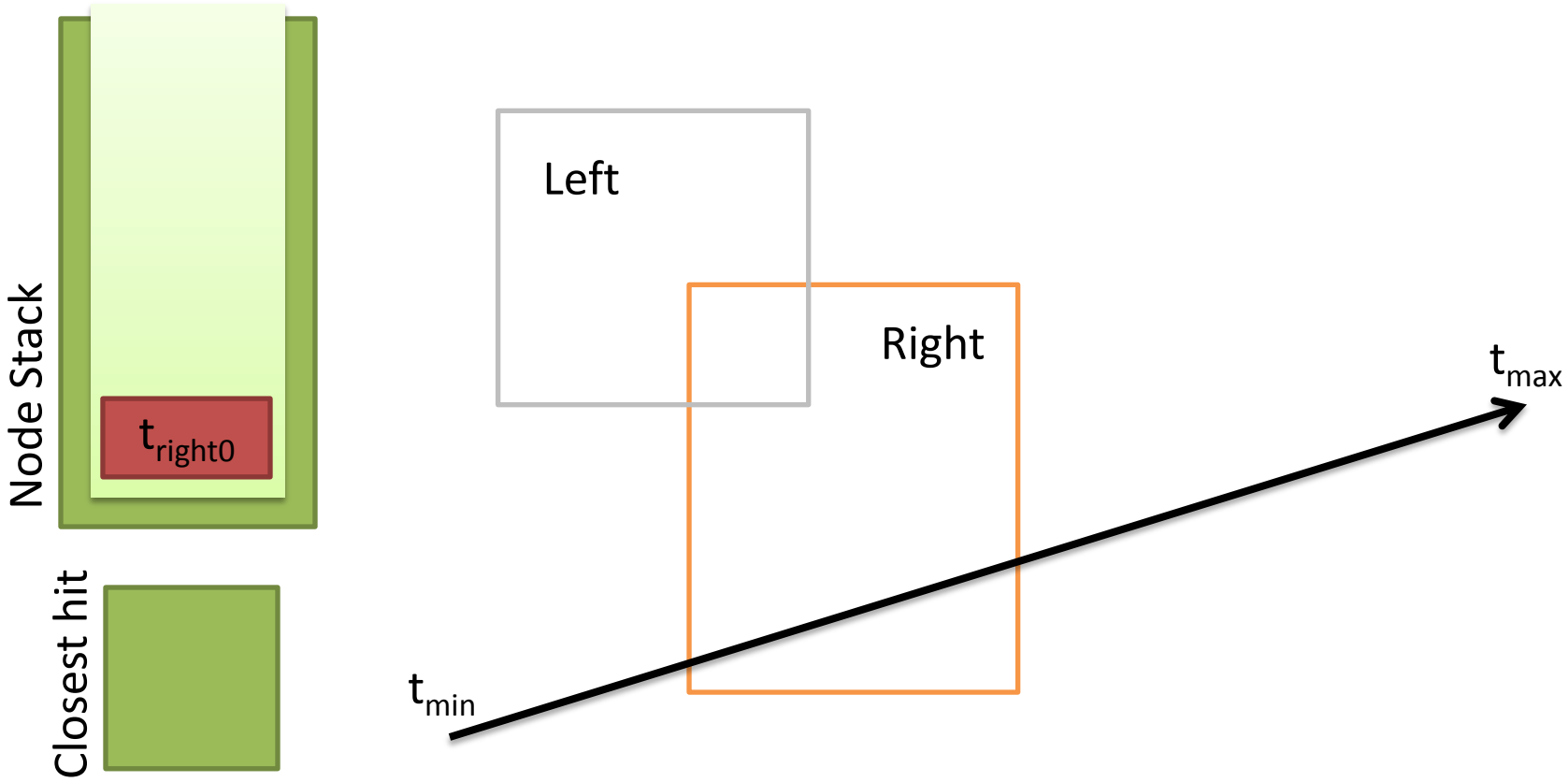

Check the two children for intersection (again using **AABB::intersect(…)**). In this case, both boxes were hit.

# Closest-Hit Intersection



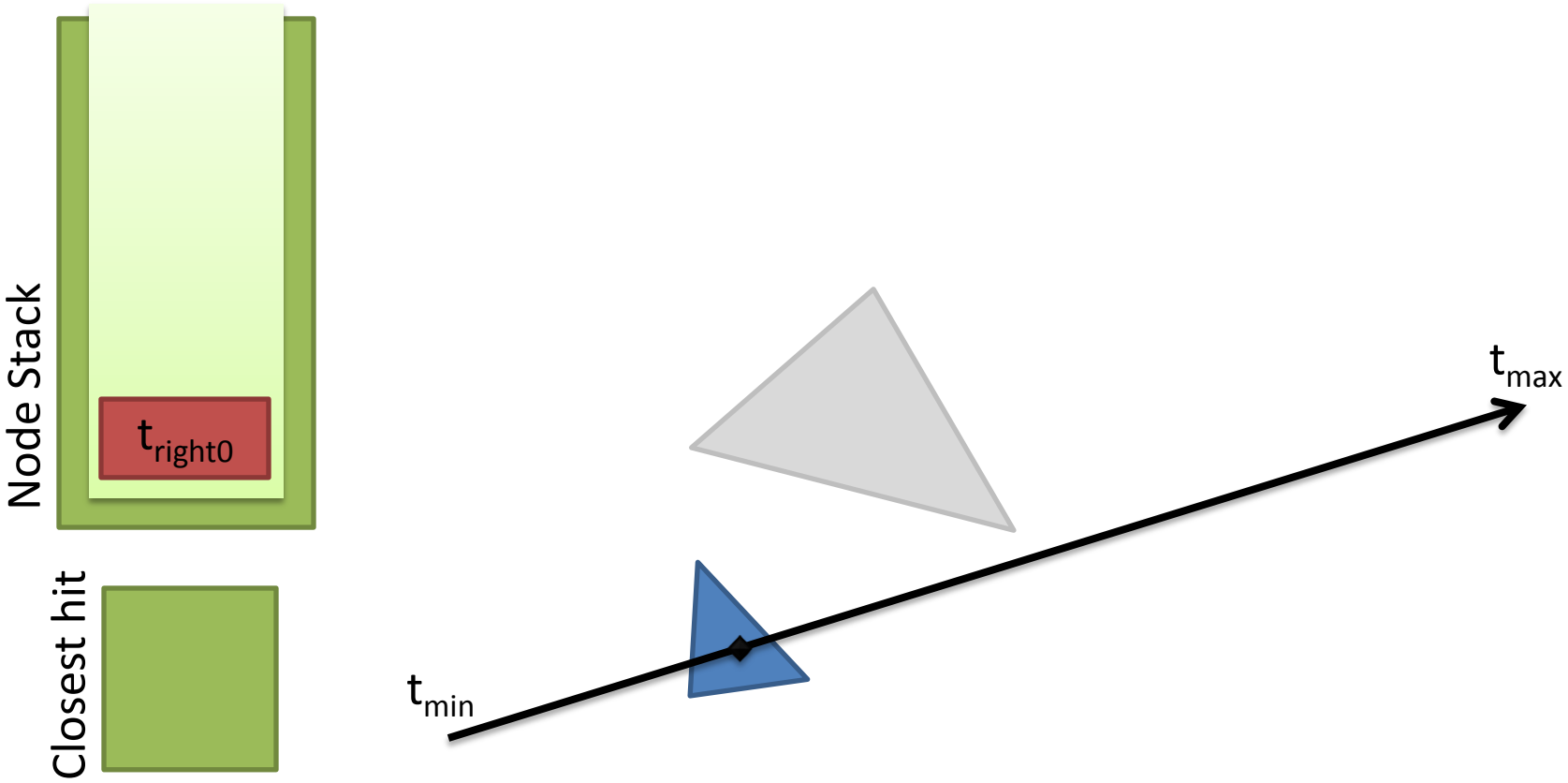

Put the node furthest away on the stack along with it's hit parameter *t*. Traverse the closest node

# Closest-Hit Intersection




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

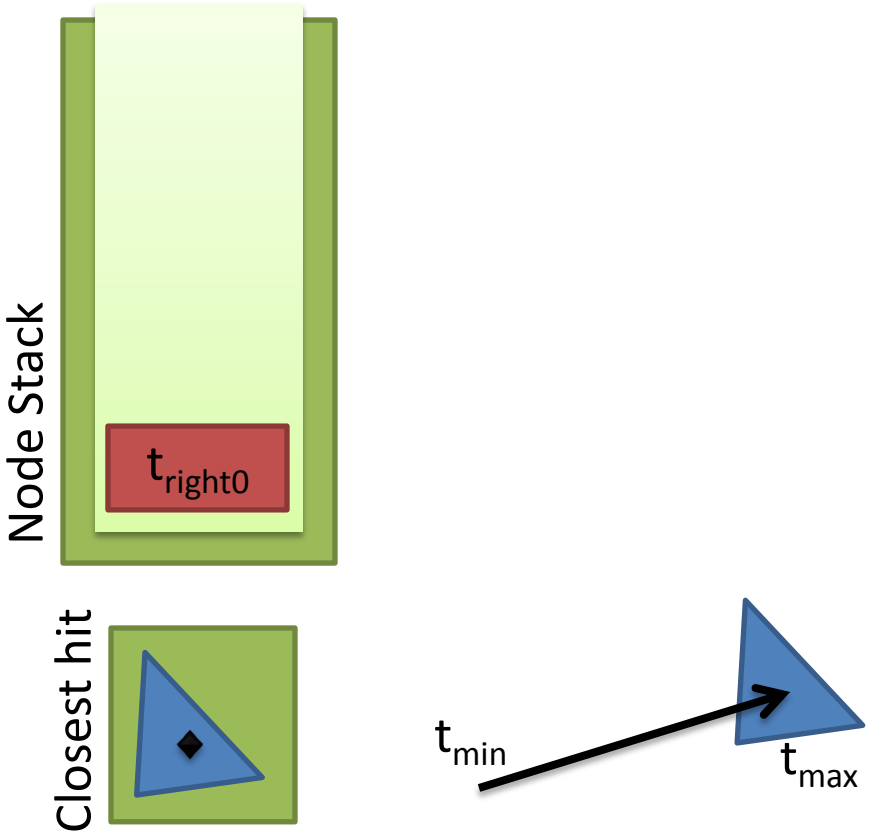
# Closest-Hit Intersection




Intersection test with each primitive in the leaf.

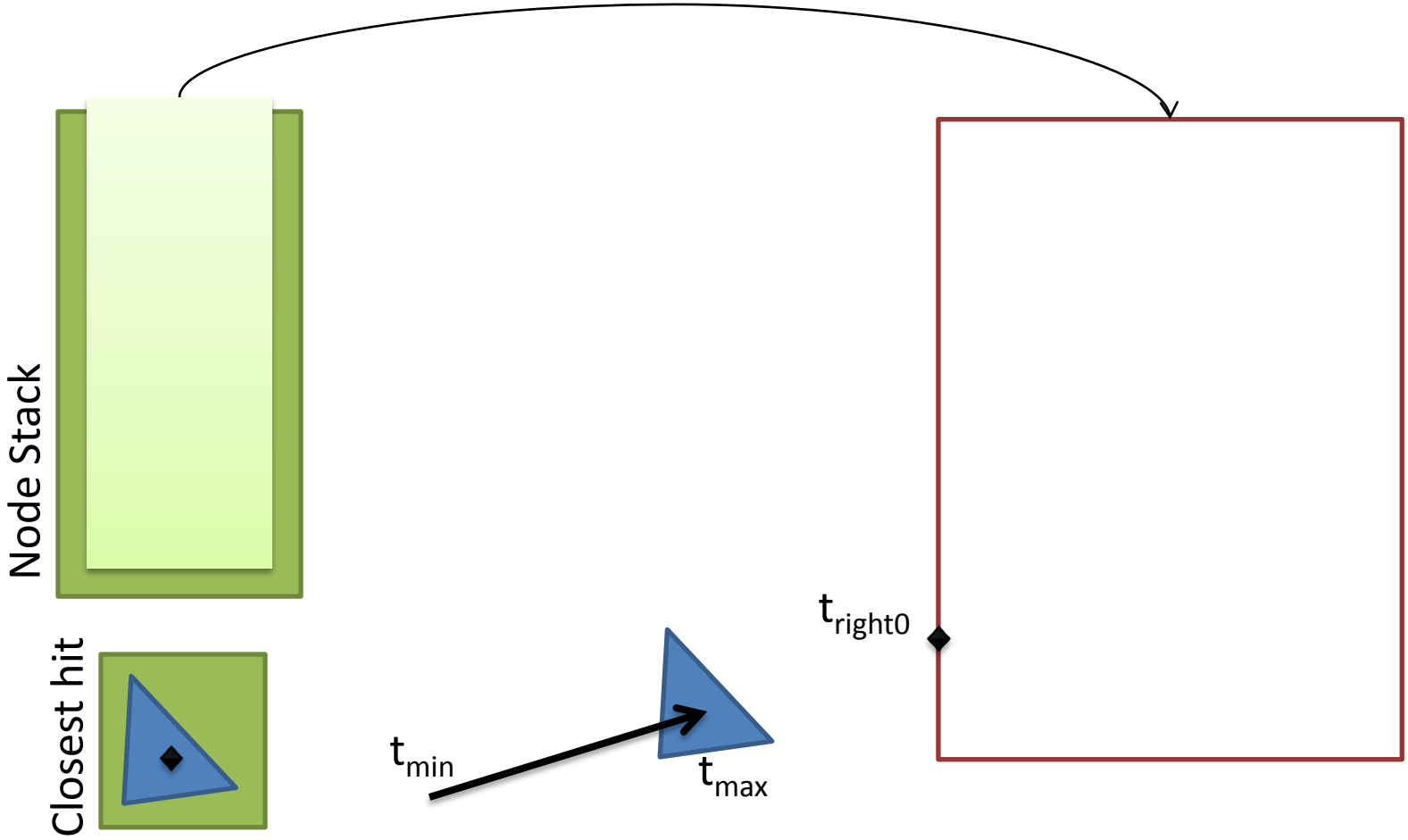
```
bool Intersectable::intersect(const Ray& ray, Intersection& is) const;
```

# Closest-Hit Intersection



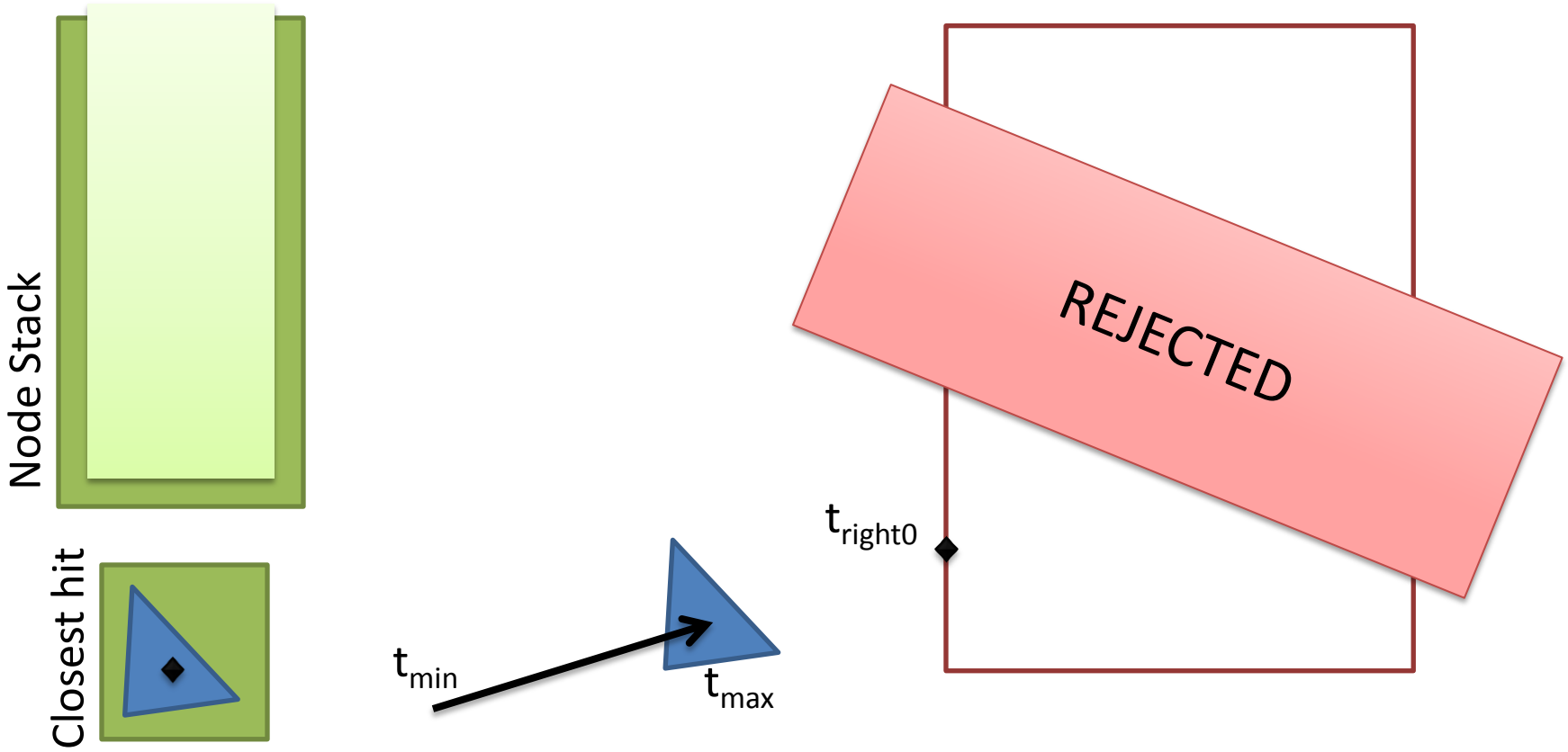

Store intersection and shorten ray.

# Closest-Hit Intersection



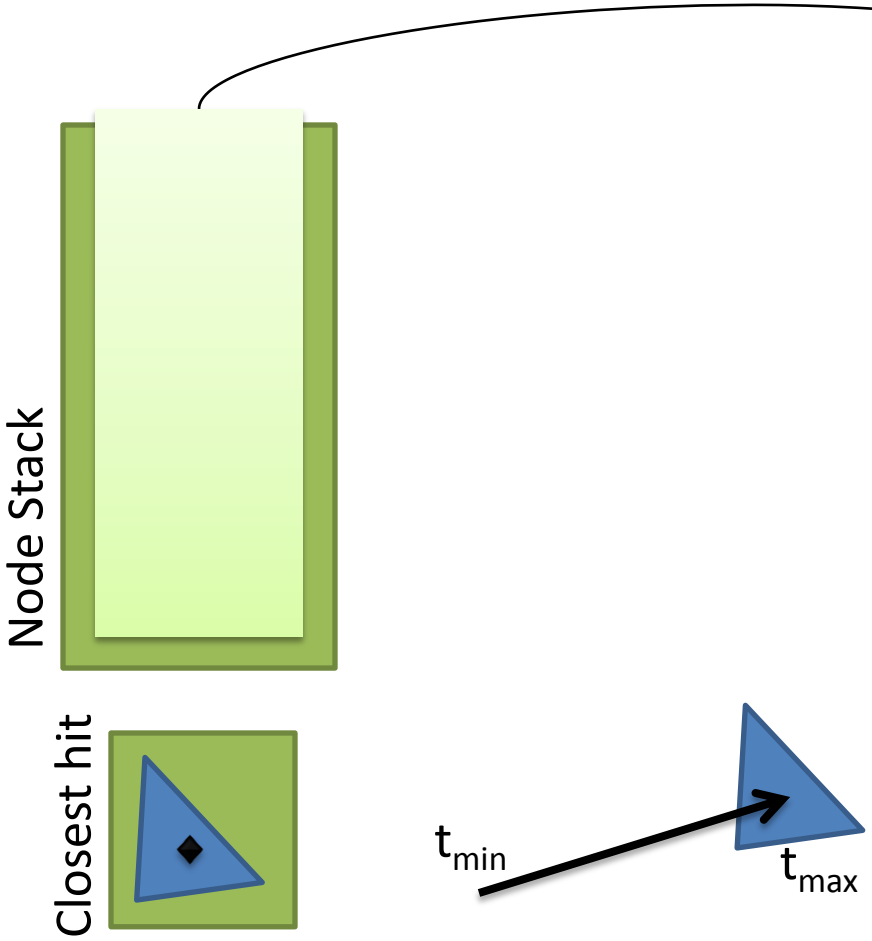

Pop the stack and recursively intersection test with the node.

# Closest-Hit Intersection



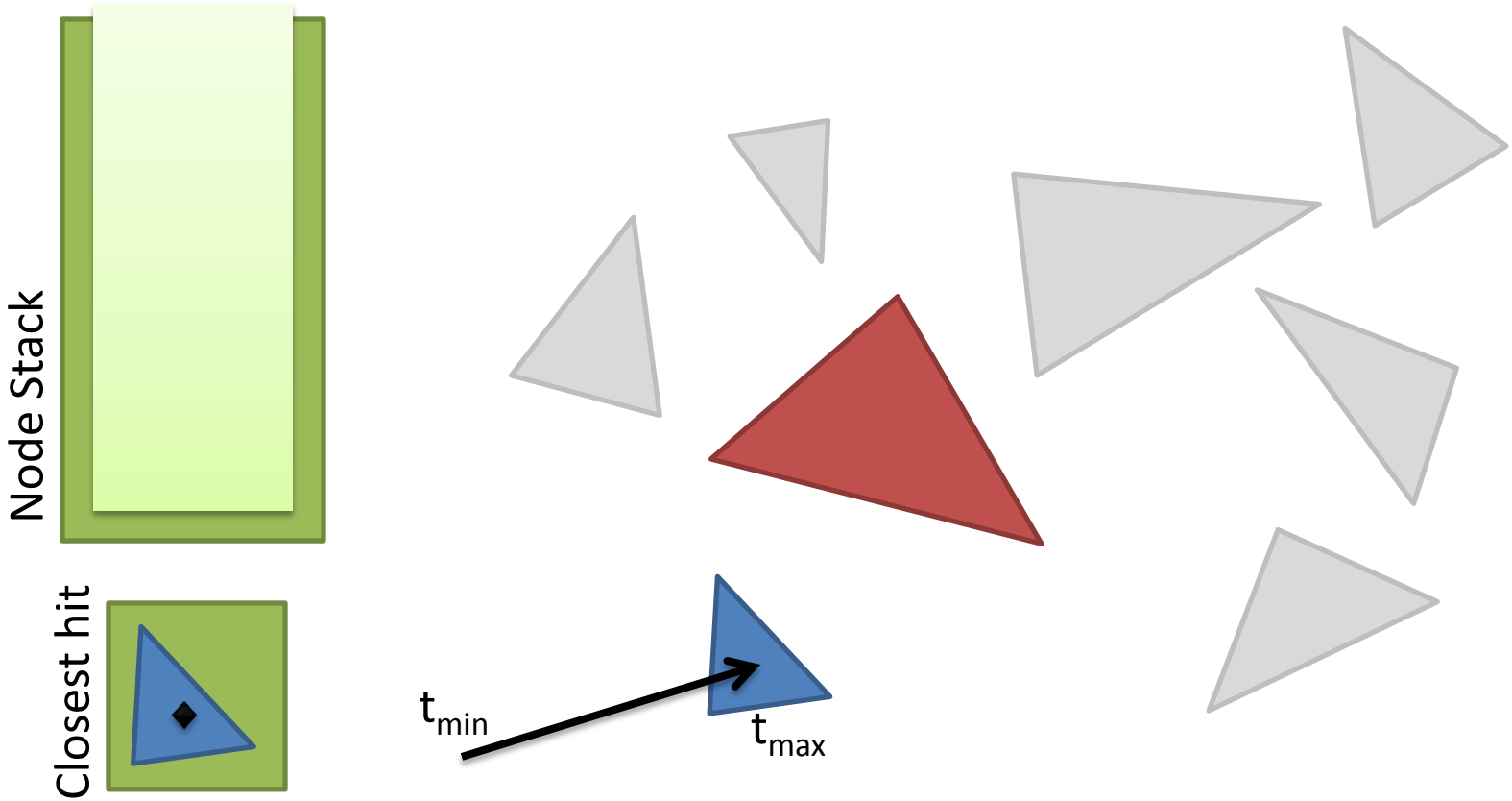

*Optimization* – We can trivially reject the pop'd node since its *t*-value is now further away than $t_{max}$ of the ray.

# Closest-Hit Intersection



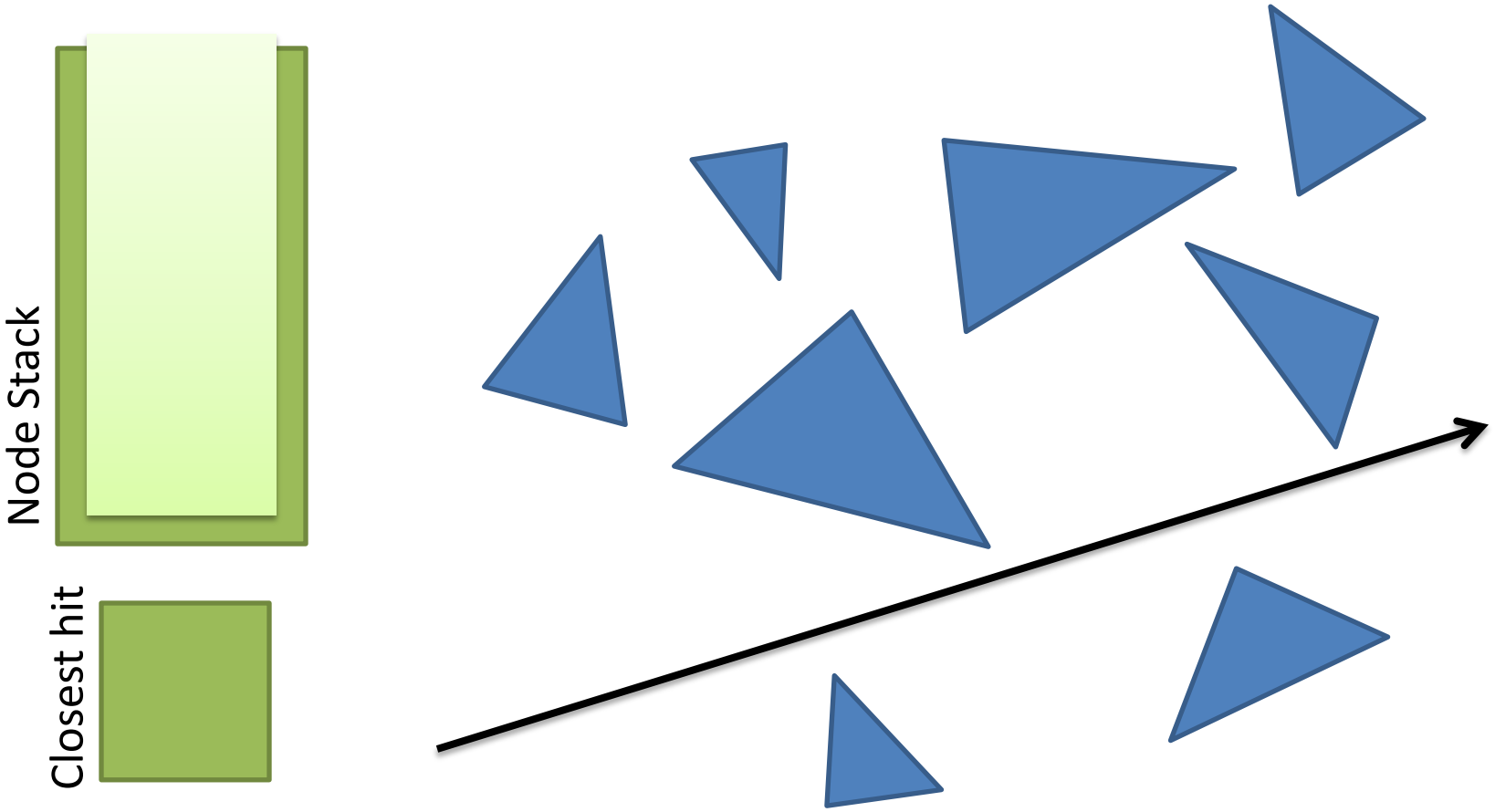

Try to pop the stack again to fetch the next node… but now it's empty, which means we're done!

# 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

# BVH traversal on the GPU

- No pointers!
  - Tree structure with no pointers
  - Tree = index into these arrays
  - Array of Structures / Structure of Arrays
- Efficient traversal: avoid recursion
  - Previous slide
- Precompute traversal: MTBVH
  - http://www.ci.i.u-tokyo.ac.jp/~hachisuka/tdf2015.pdf

# Ray-tracing on the GPU

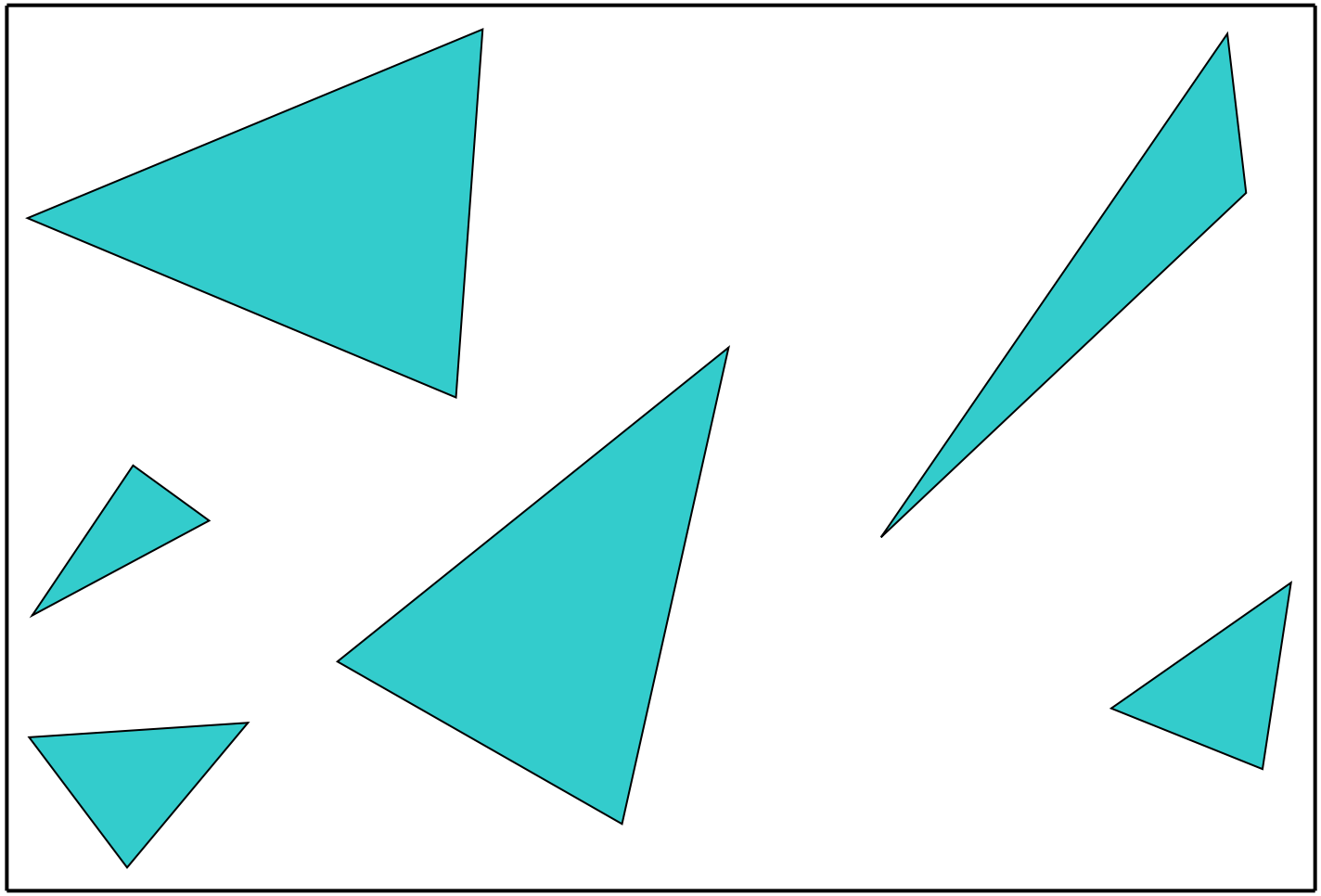
- Same problem: avoid recursive calls
- Naïve approach:
  - `out = direct_light + color * trace(reflected_ray)`
  - Problem: recursive calls to trace(ray)
- Better approach:
  - `for (s=0..max_depth)`
    - `Store reflected_ray[s](origin, direction)`
  - `for (s=max_depth..0)`
    - `out = direct_light(reflected ray[s]) + color * out`

# Other structures

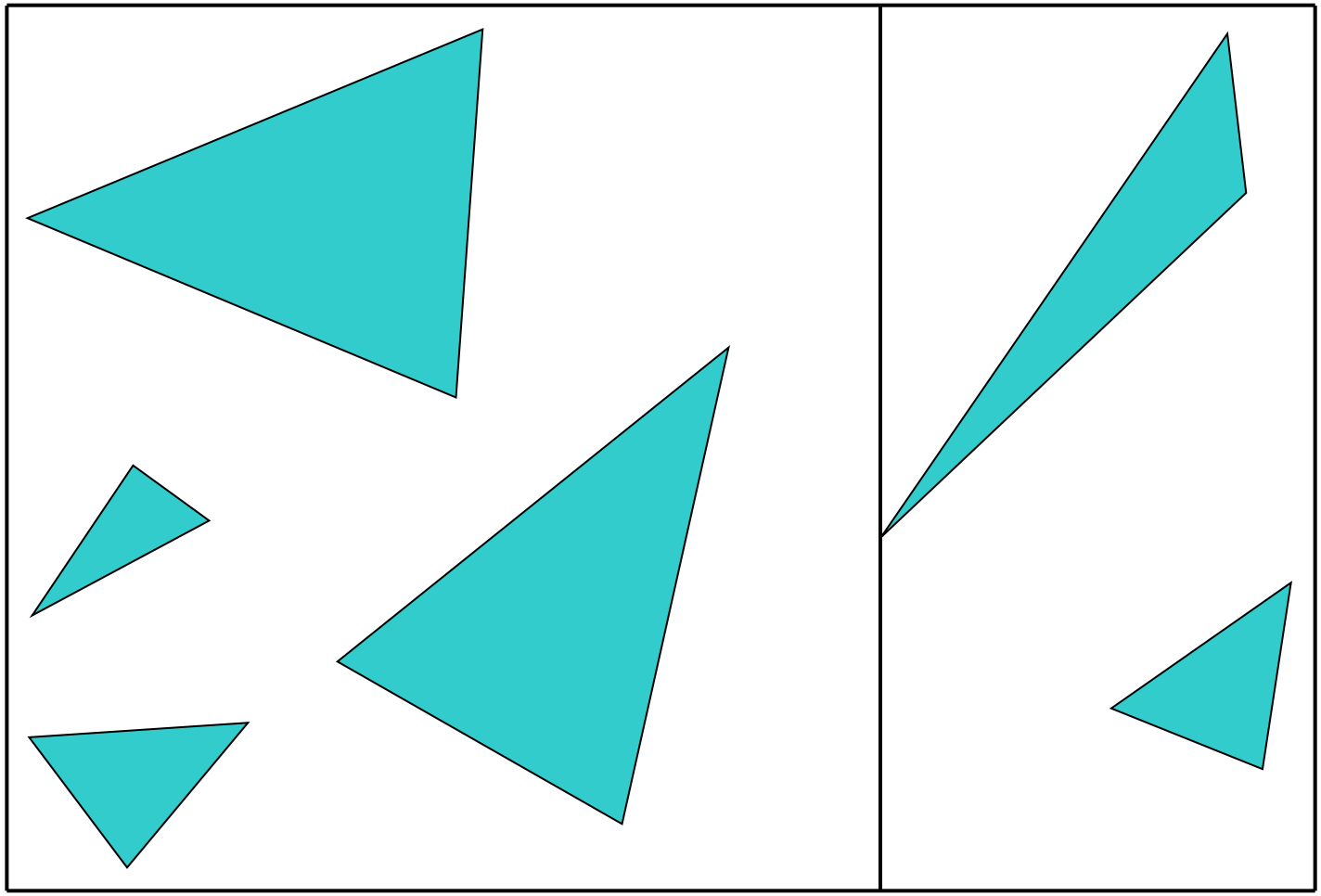
- Uniform grid
- Adaptive grid: octree
- k-D tree

- Keep nearest intersection found
- Check if cells are closer than this intersection

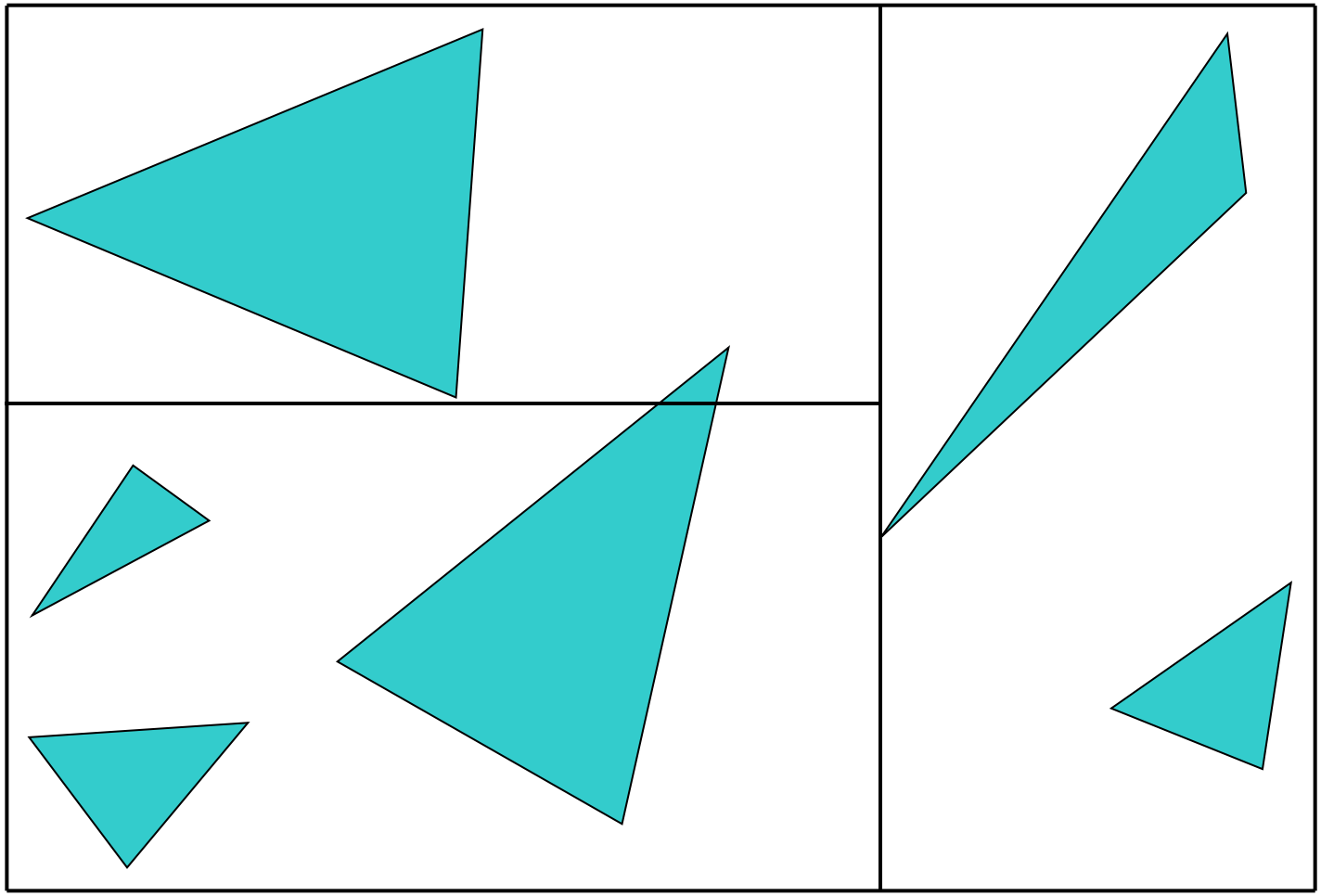
# k-D tree



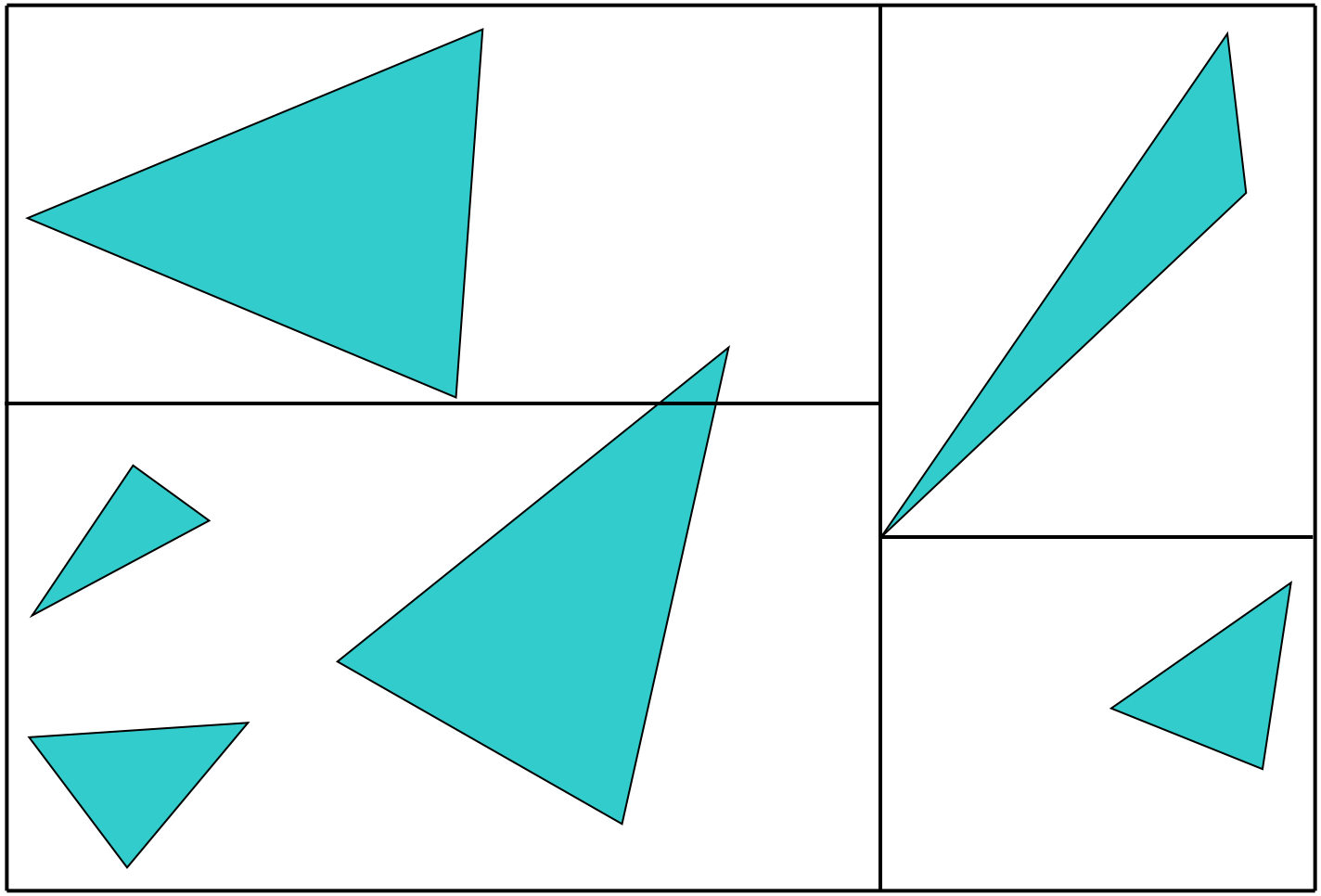
# k-D tree



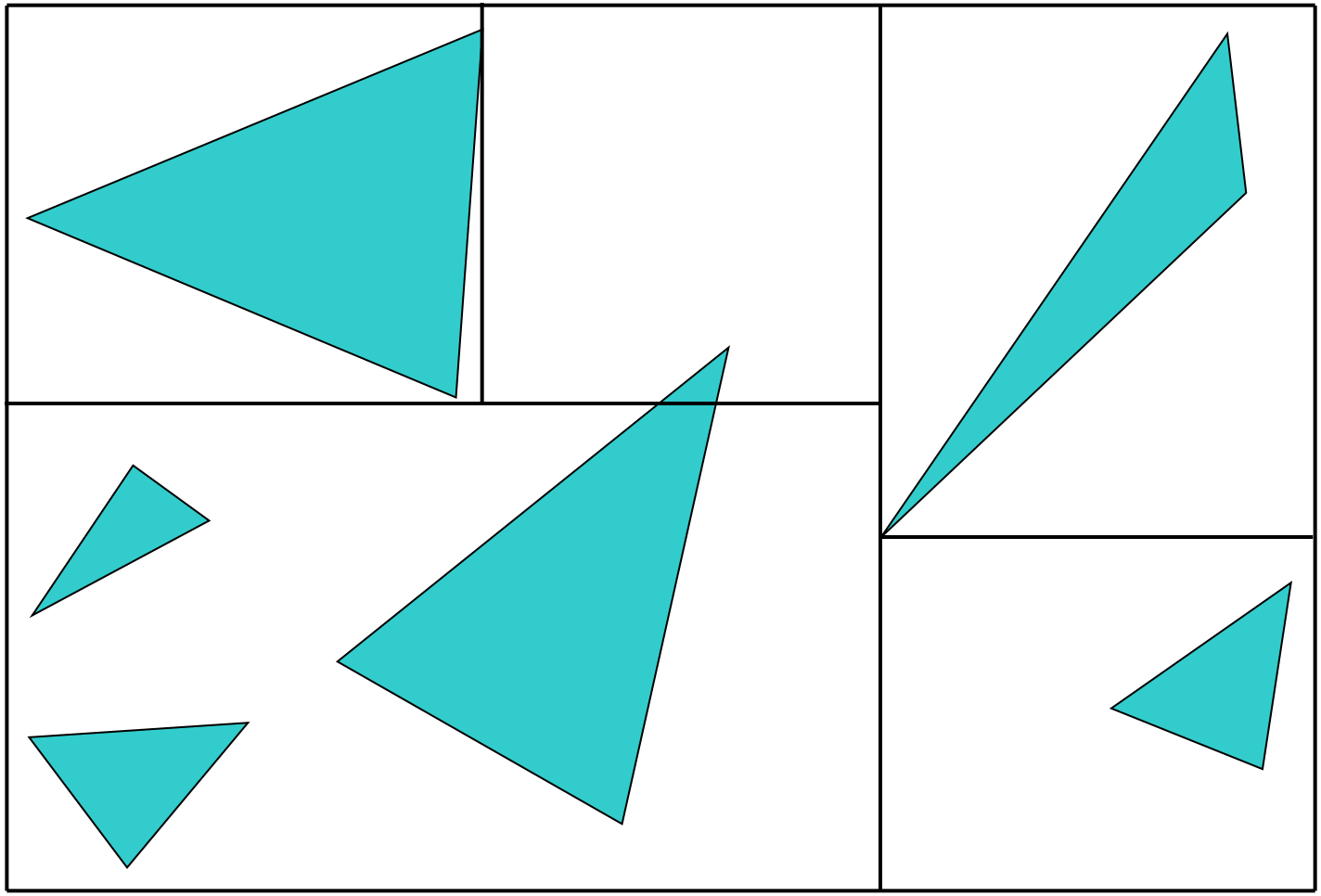
# k-D tree



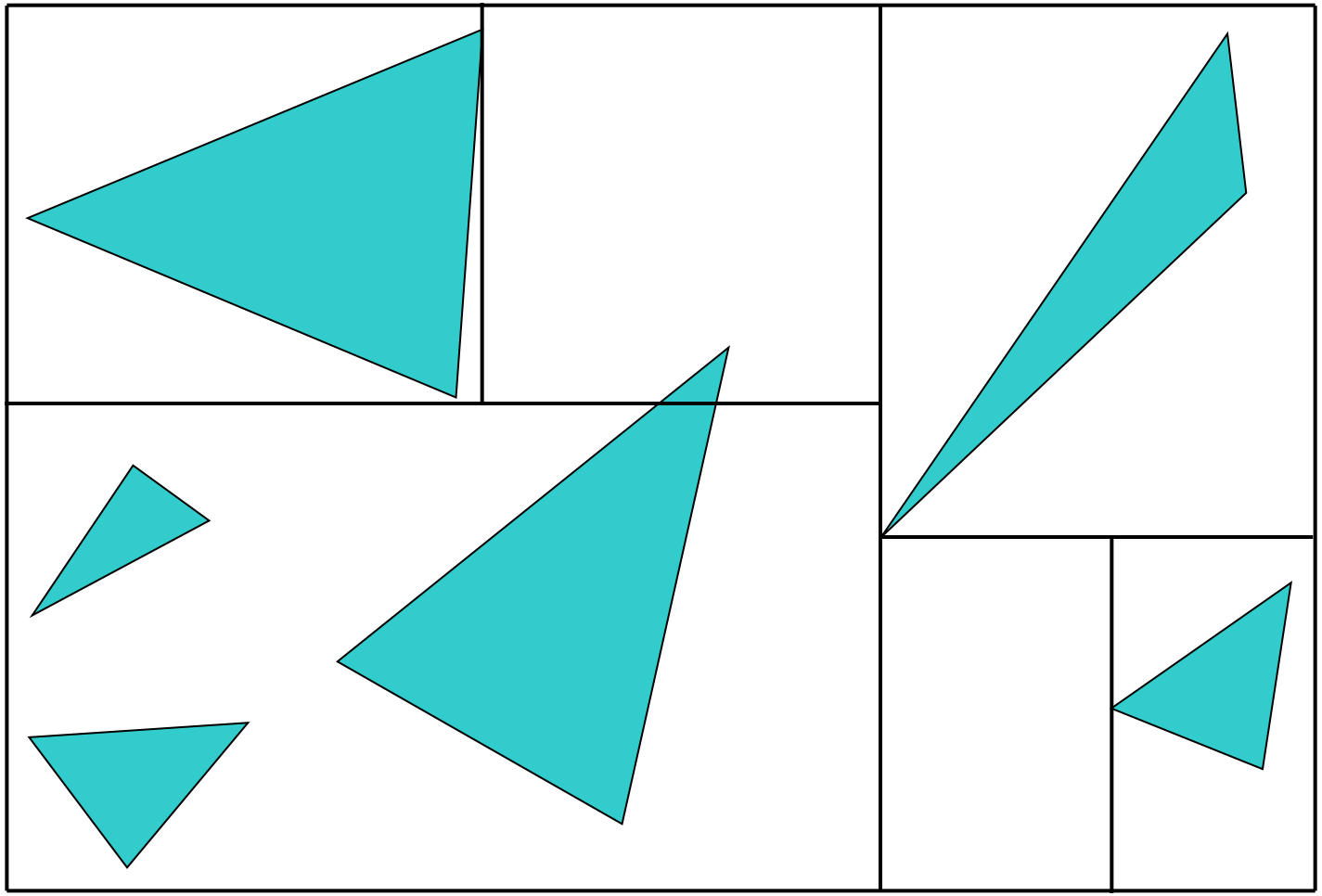
# k-D tree



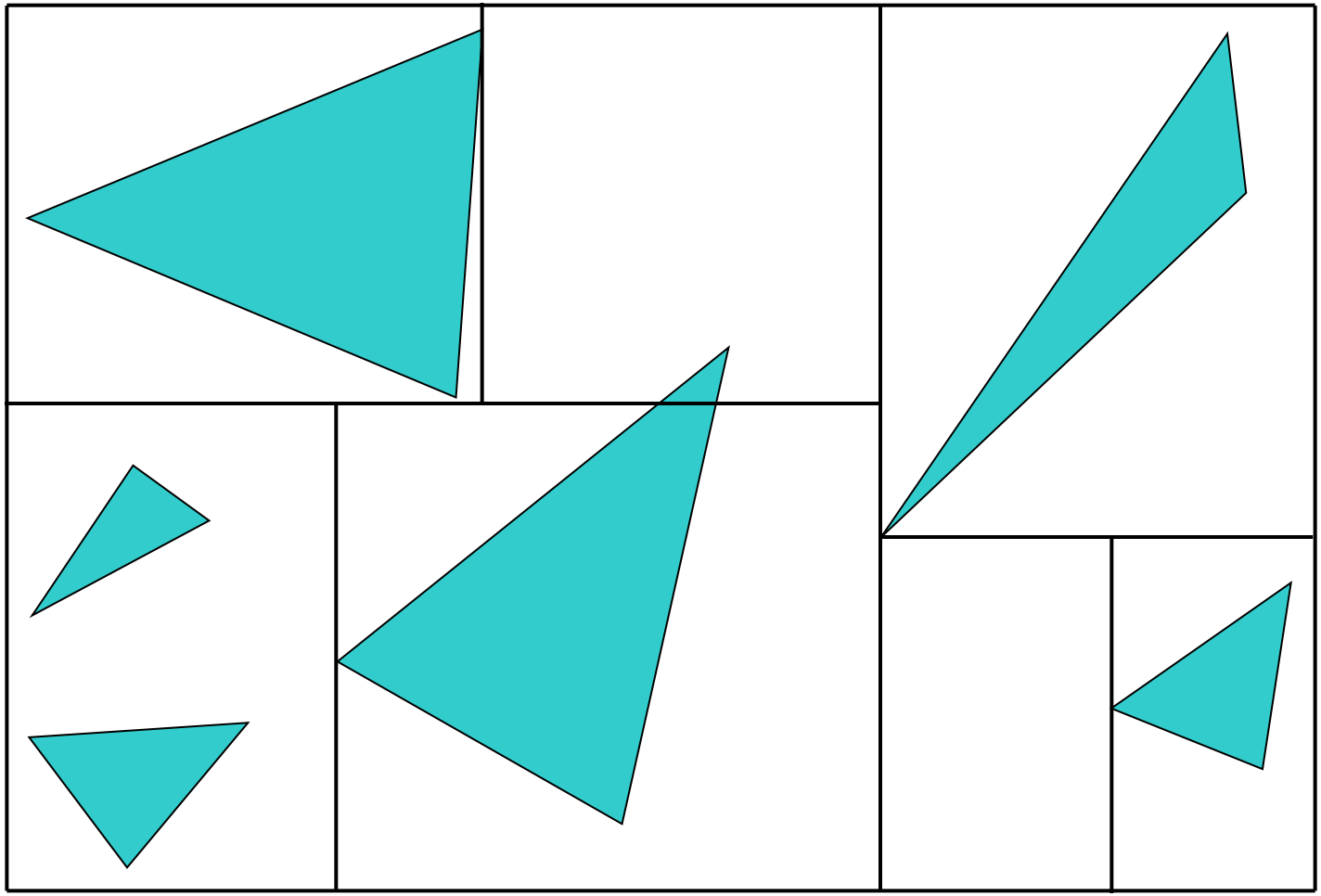
# k-D tree



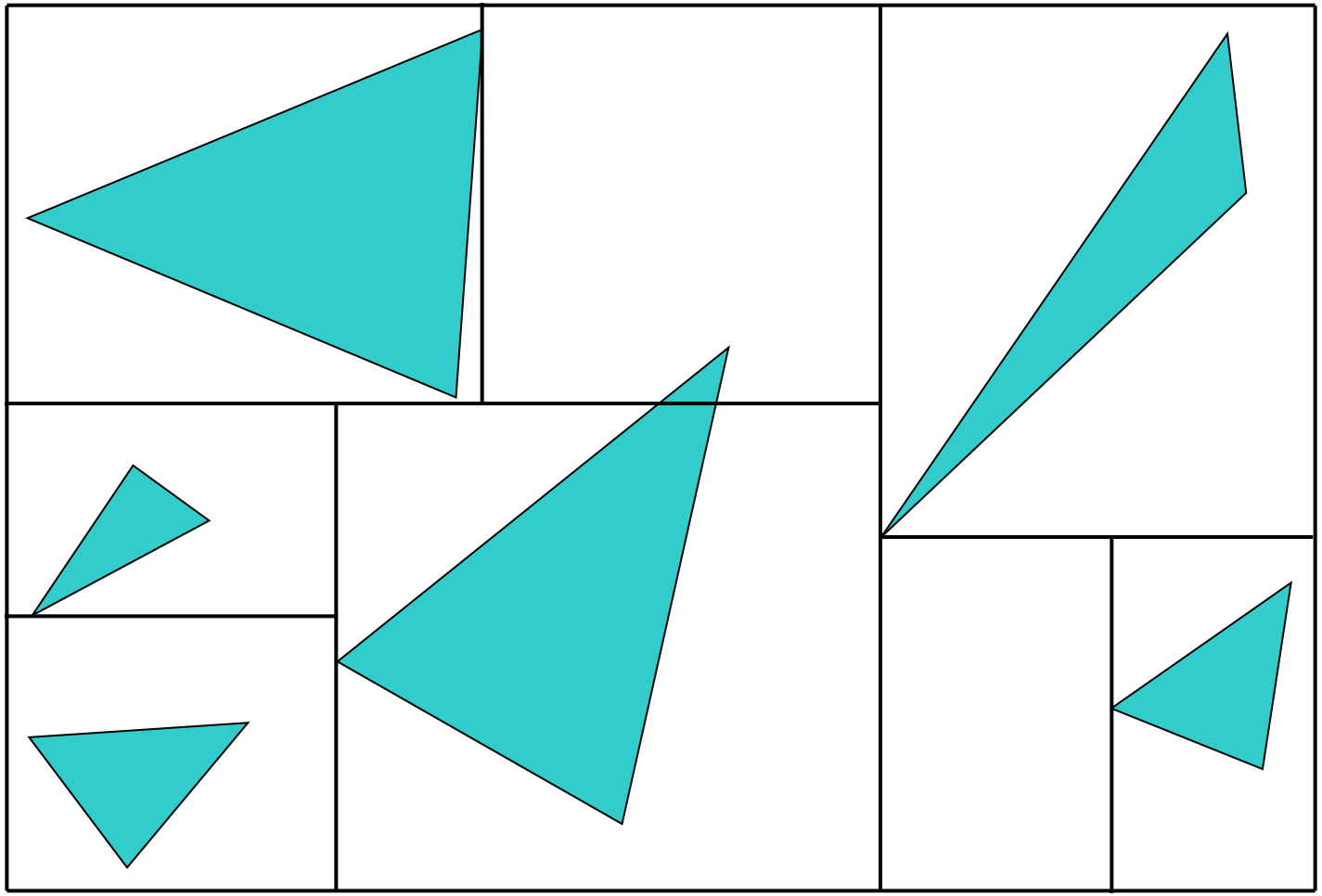
# k-D tree



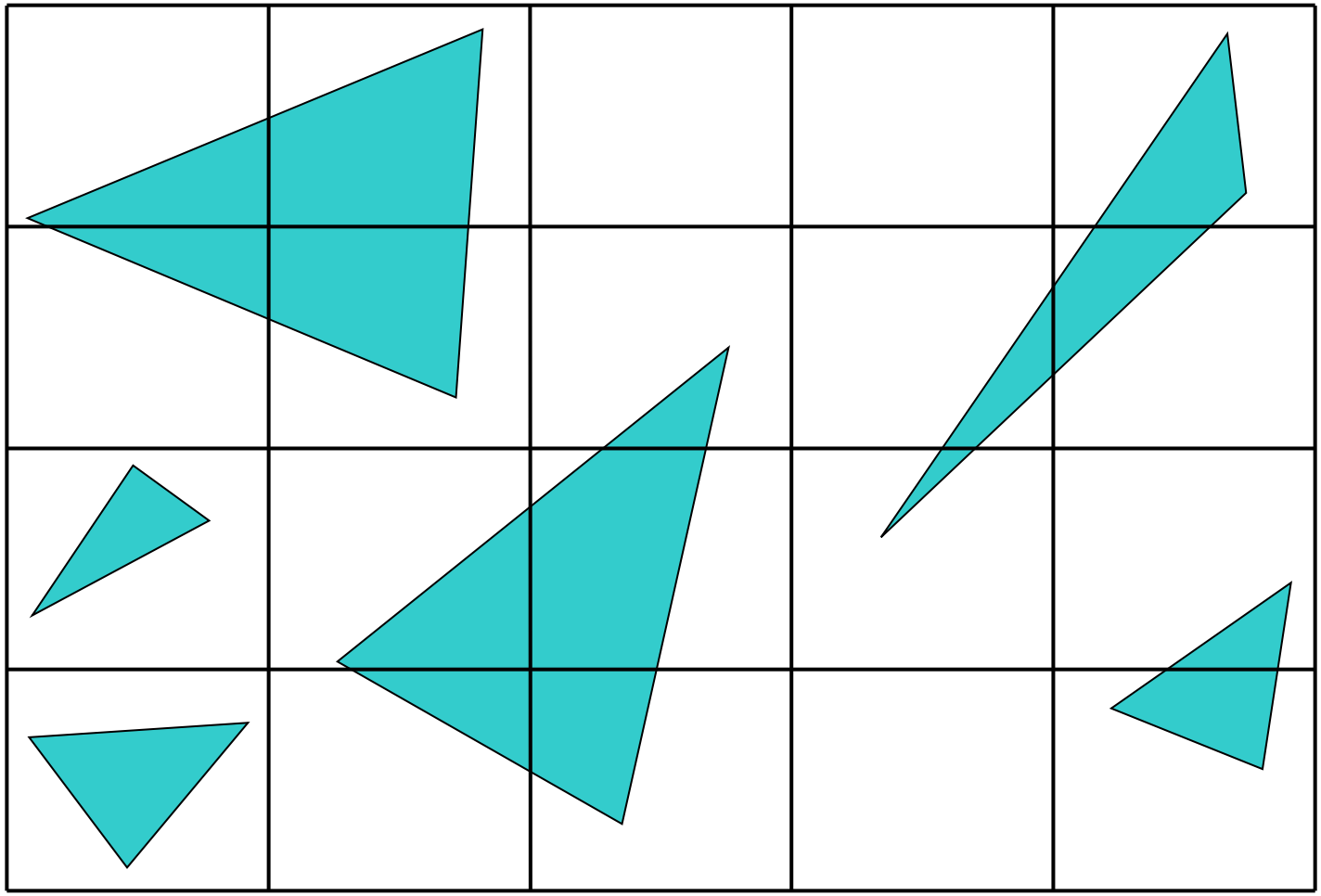
# k-D tree



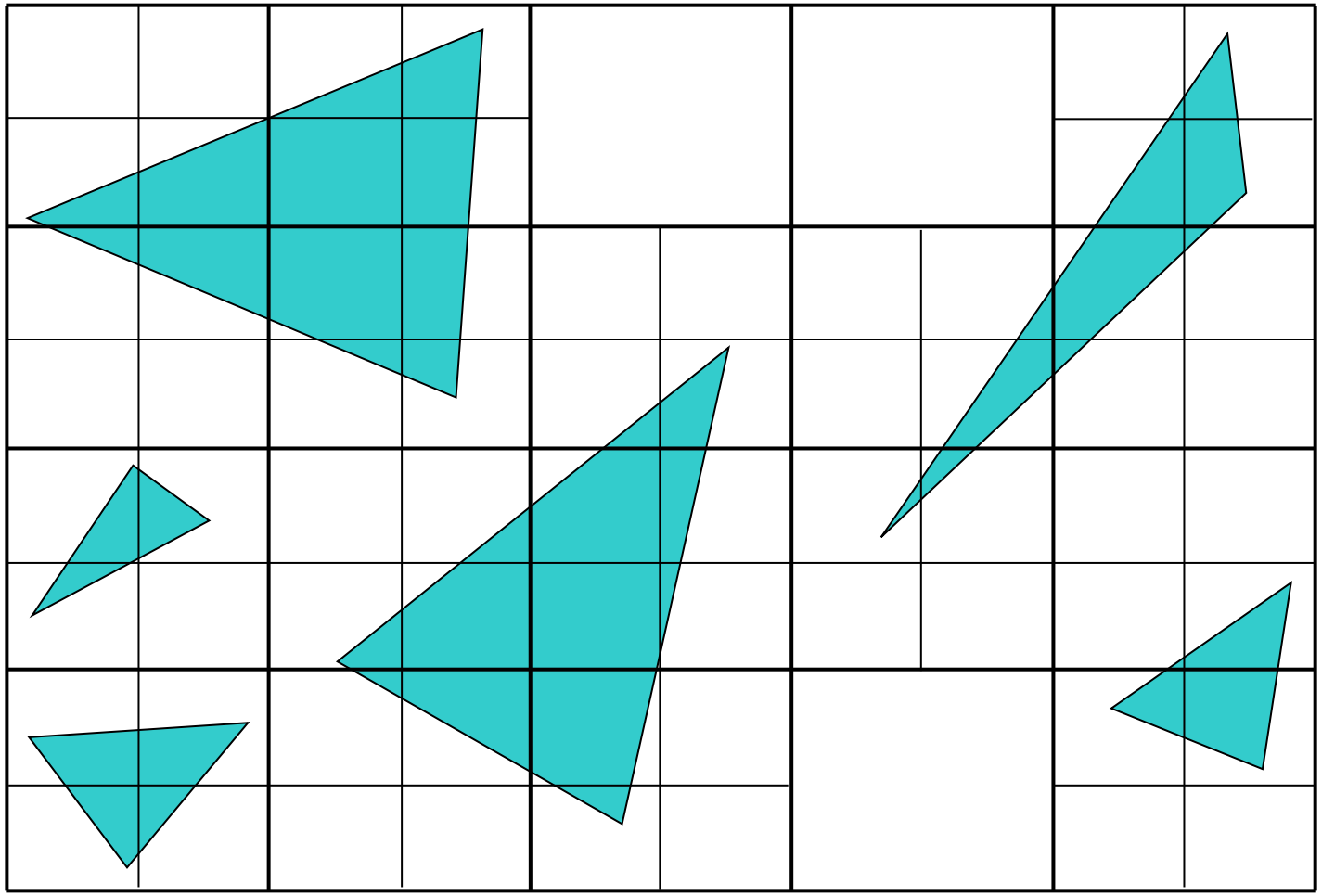
# k-D tree



# Uniform grid



# Adaptive grid: Octree



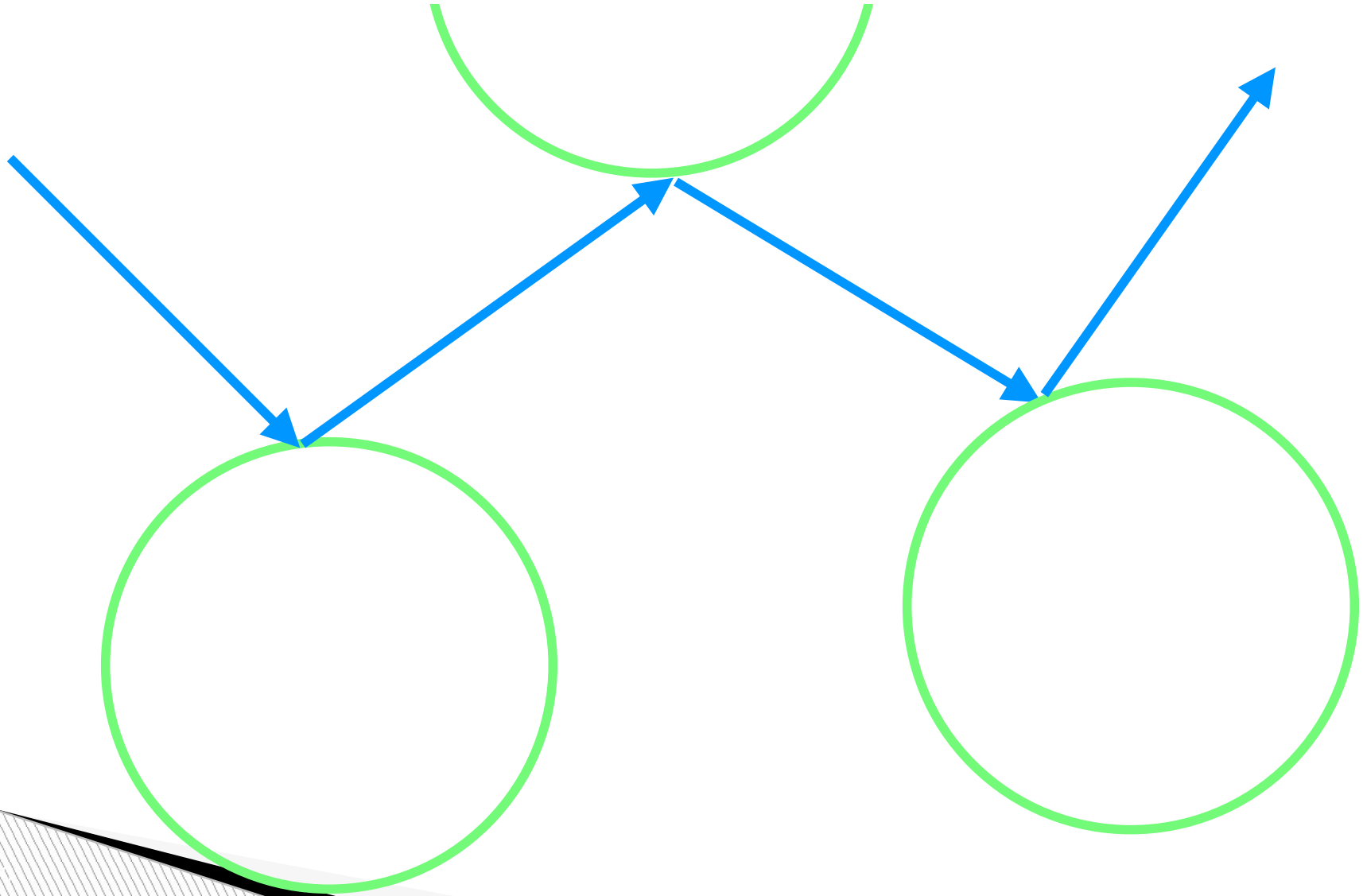
# GPU ray-tracing (fragment shader)

- Ray-tracing: embarrassingly parallel
  - Each ray independent from the others
- GPU: massively parallel computer
  - Fragment shader: each fragment, independent
- Use GPU for ray-tracing
  - It works
  - Issues: data input, memory replication, cache misses, synchronicity

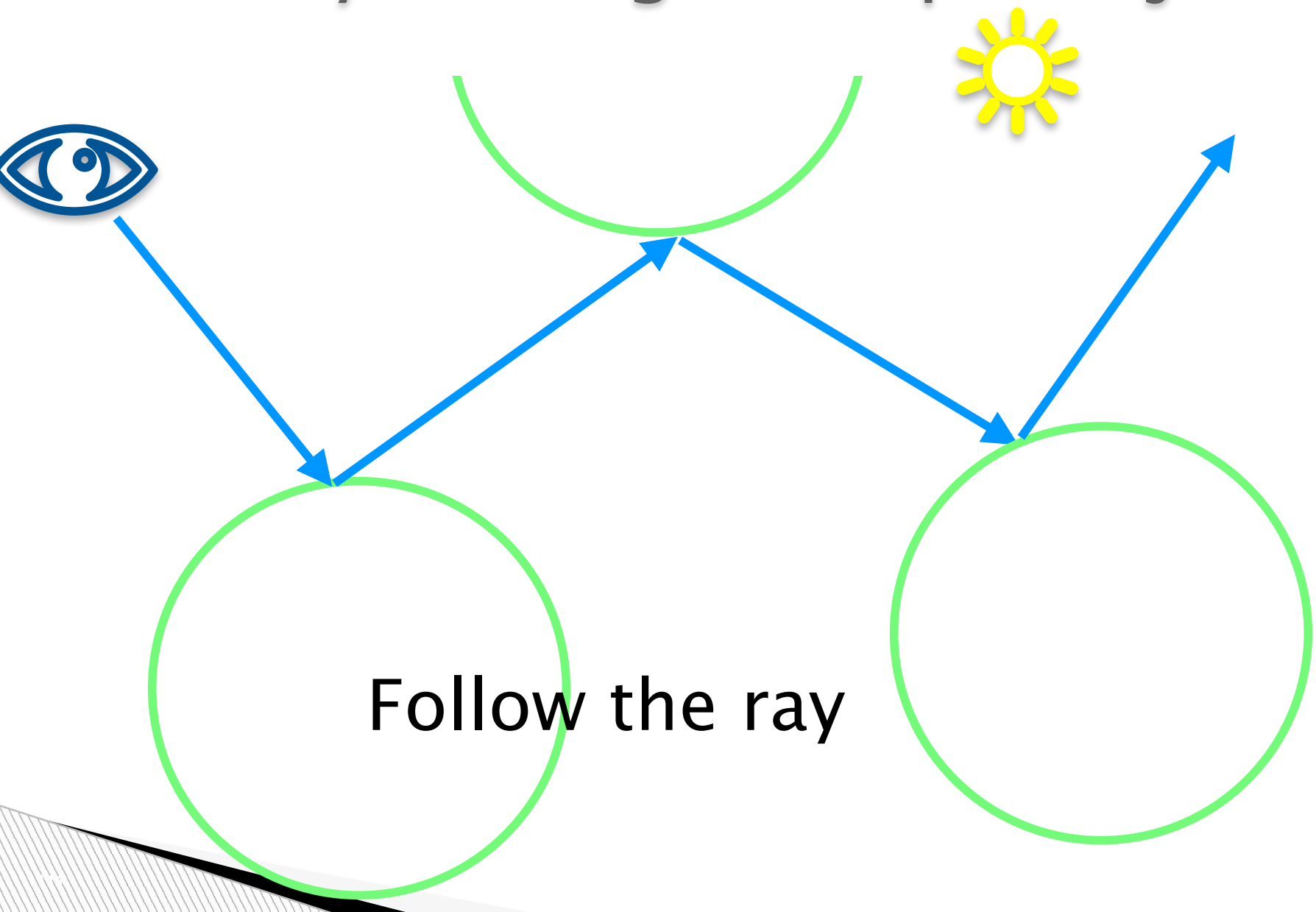
# GPU ray-tracing (fragment shader)

- How?
- 1 fragment shader call = 1 ray
- Fragment shader needs:
  - Pixel position
  - Ray: starting point $P$, direction $\boldsymbol{u}$
  - These are connected (use transform matrix)
- *How* do we cover all pixels?
  - Render a single quad, covering the screen
  - Add a parameter for screen position
  - Fragment shader gets interpolated value: pixel pos.

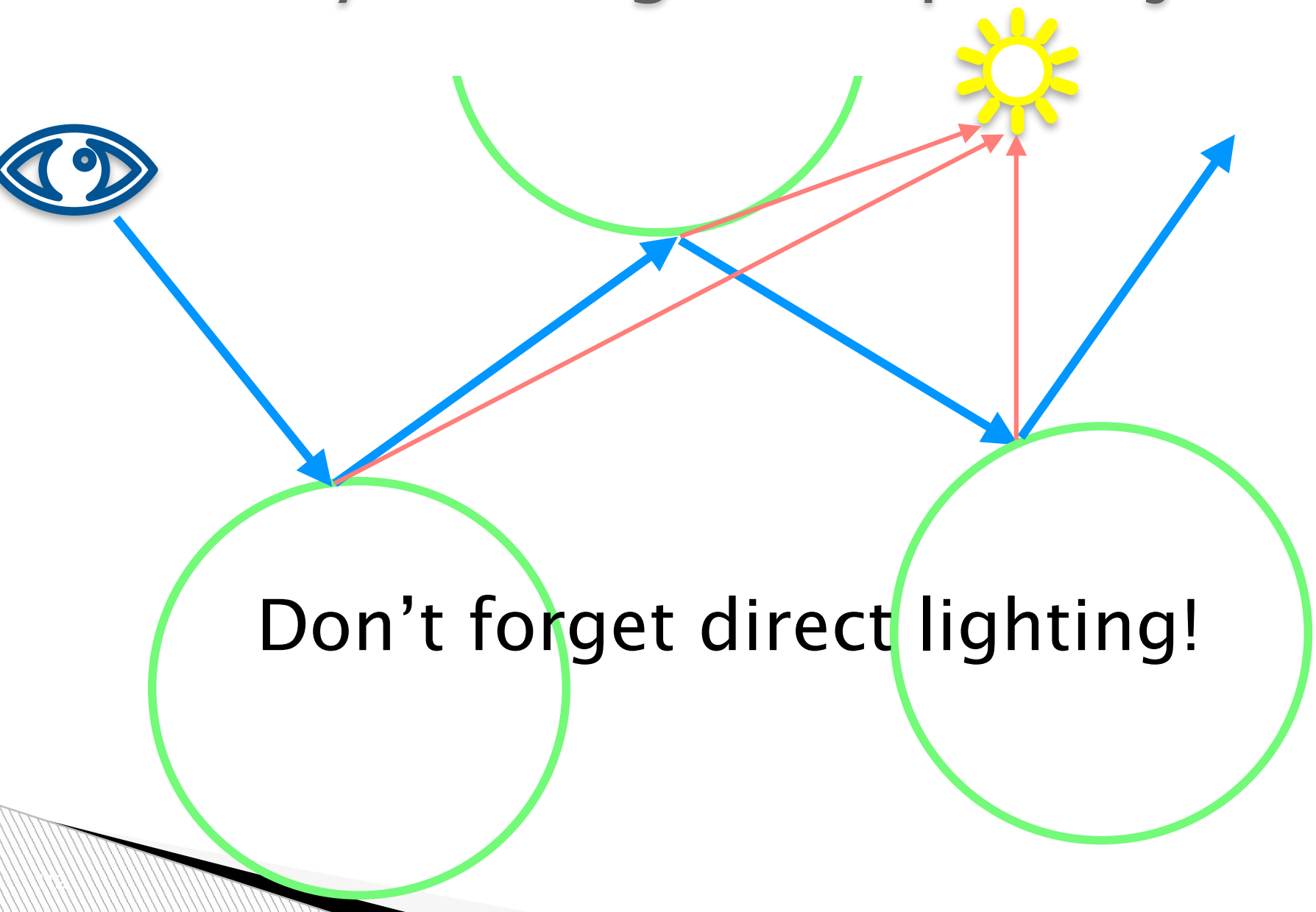
# GPU ray-tracing: multiple objects



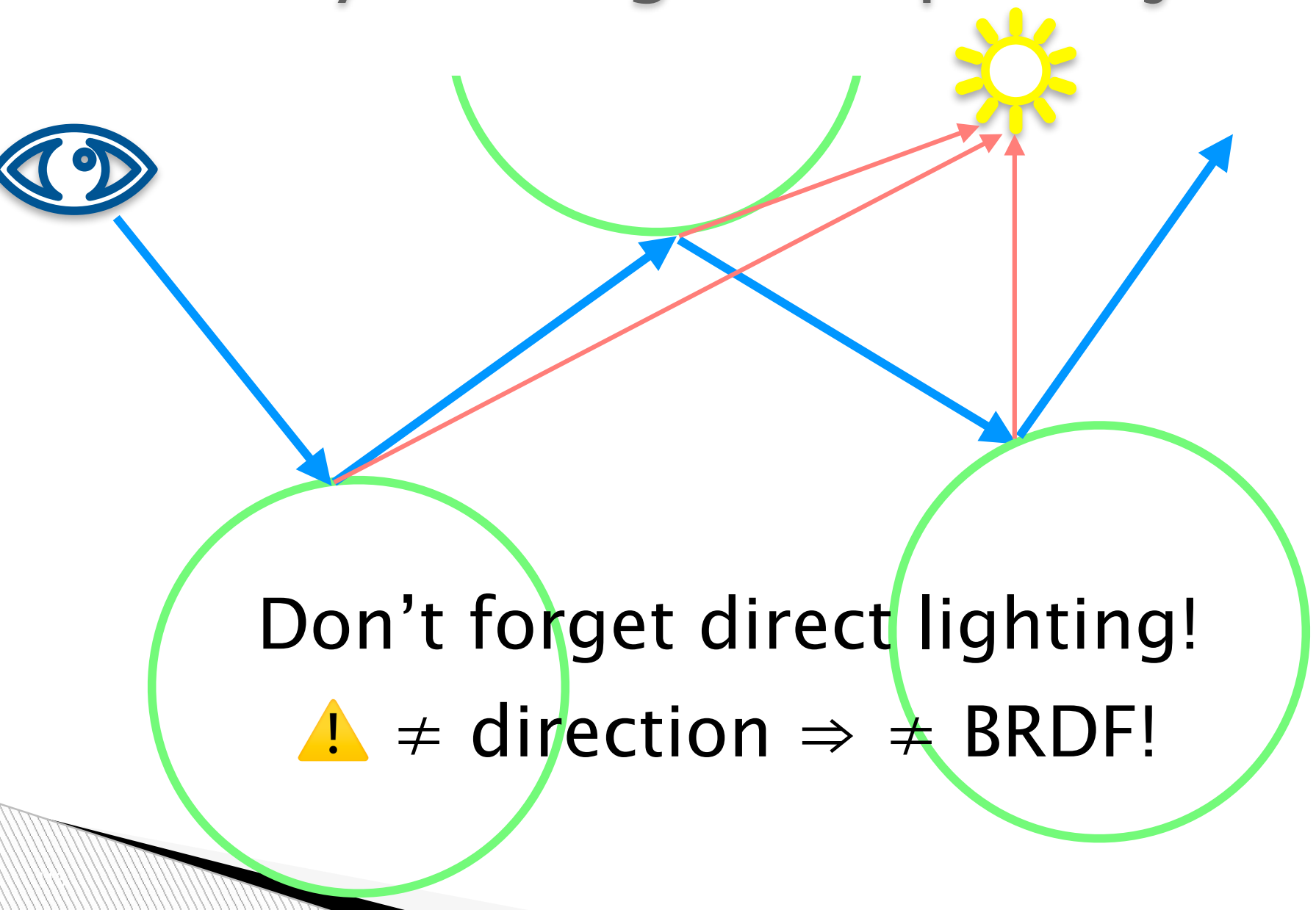
# GPU ray-tracing: multiple objects



# GPU ray-tracing: multiple objects



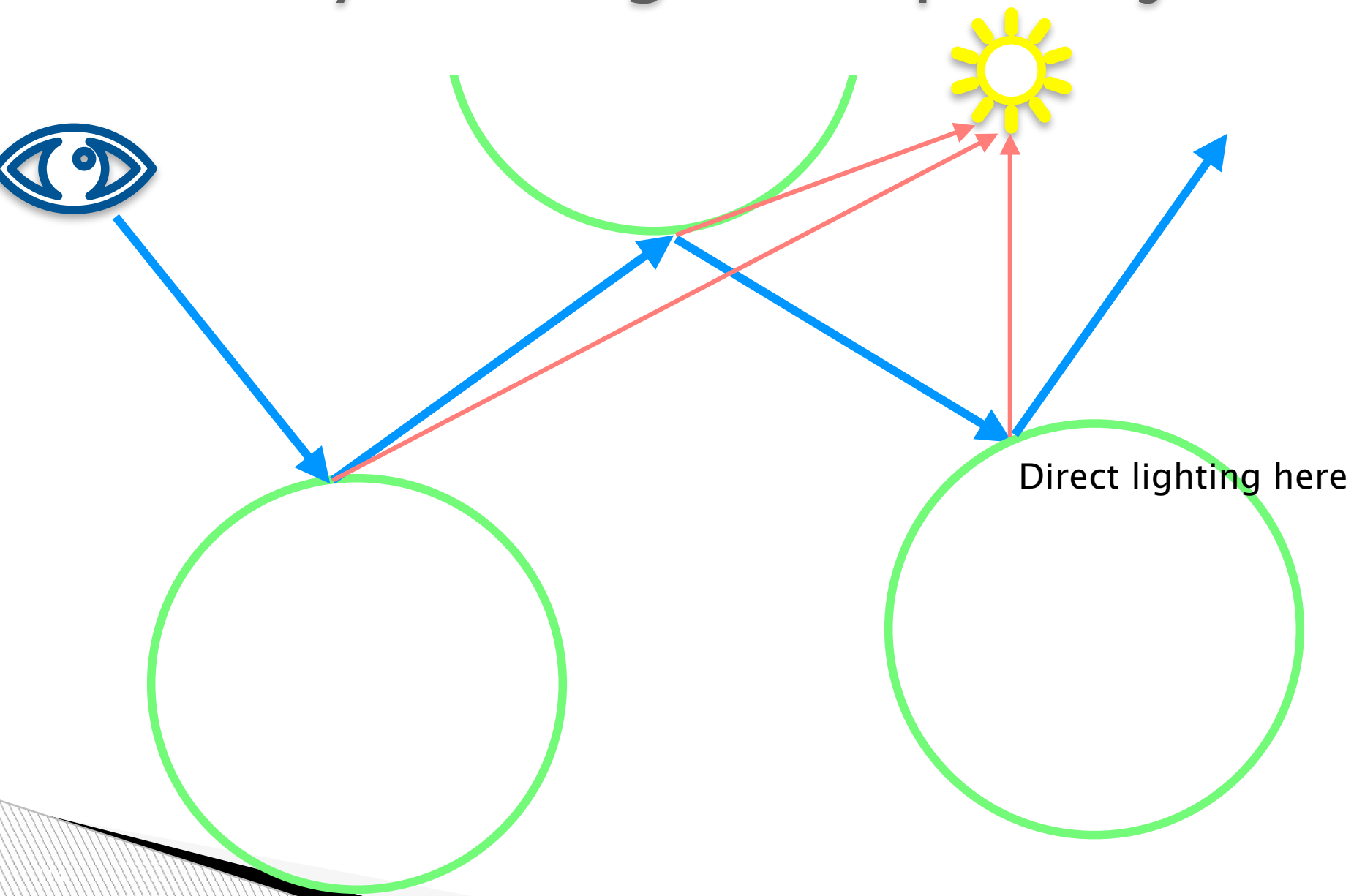
# GPU ray-tracing: multiple objects



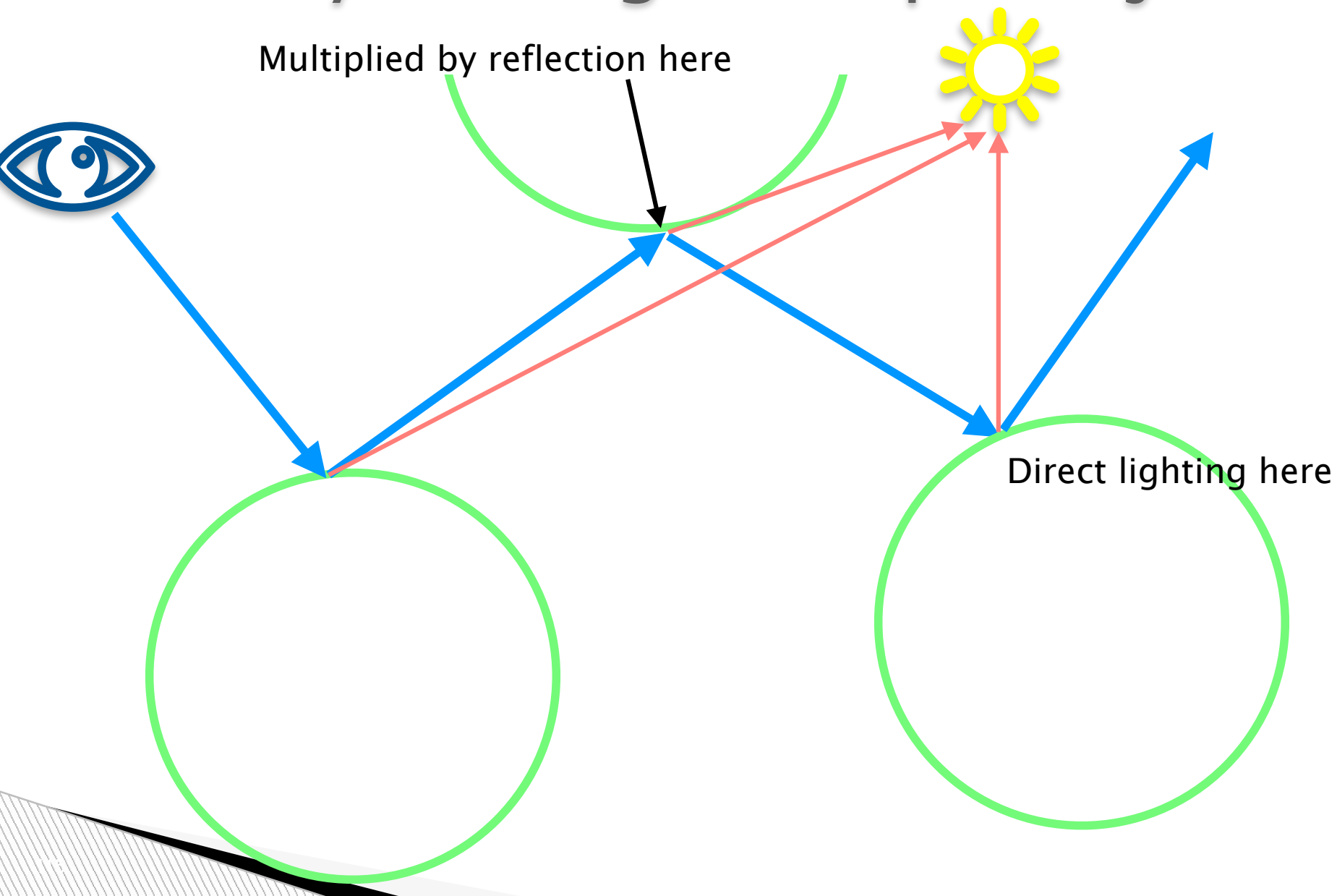
Don’t forget direct lighting!

⚠ ≠ direction ⇒ ≠ BRDF!

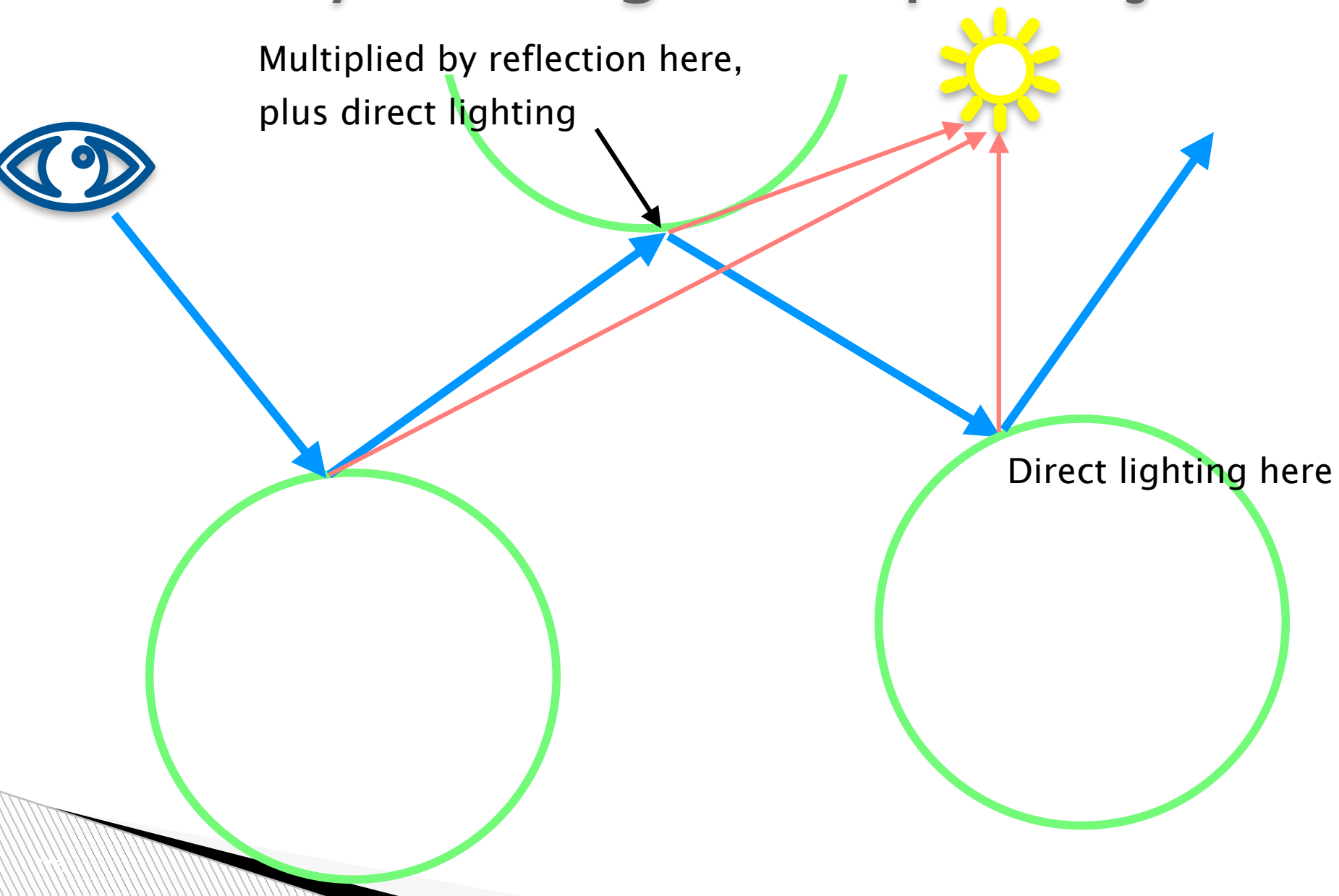
# GPU ray-tracing: multiple objects



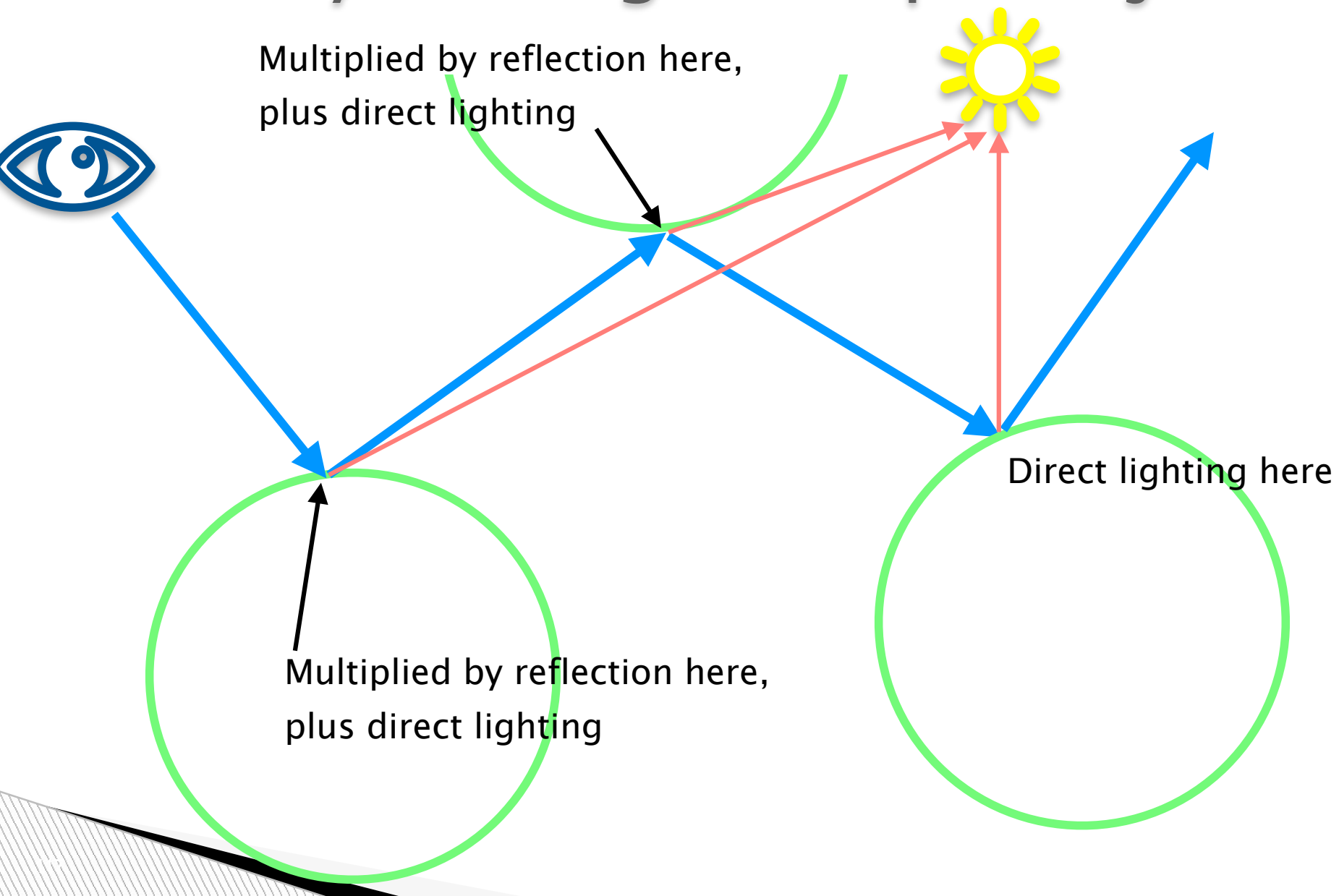
# GPU ray-tracing: multiple objects



# GPU ray-tracing: multiple objects



# GPU ray-tracing: multiple objects



# Computing pixel color:

- Pixel color:

$$C_p = d_0 + \rho_0 \left( d_1 + \rho_1 \left( d_2 + \rho_2 \ldots \rho_{n-2} \left( d_{n-1} + \rho_{n-1} d_n \right) \right) \right)$$

- How can we compute that?
  - Incrementally, from the end

# Computing pixel color

- Need to work from the end of the ray
- Ray is created from the start
- Two-step process:
  - Follow the ray from the camera
  - Store intersection points in a queue
  - Follow the ray from the end
  - Compute illumination at each point
  - ≠ BRDF for direct and indirect lighting

# Chromatic aberration

- Refractive index depends on wavelength
- Cauchy rule (dielectrics only):

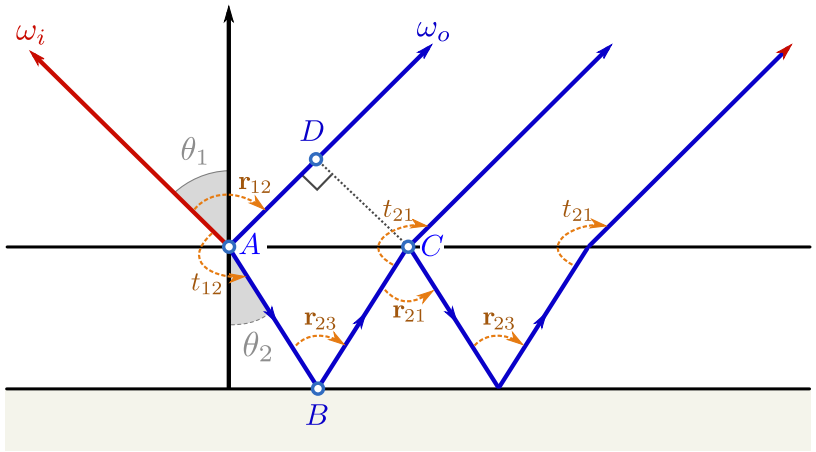
$$\eta = B + \frac{C}{\lambda^2}$$

- 3 *different* rays (R, G, B)
  - x3 computation time
  - Subtle effect

https://en.wikipedia.org/wiki/Cauchy%27s_equation

# Diffraction

- Soap bubbles = thin layer ≅ wavelength
- Diffraction caused by the layer + reflection
- Can be encoded in an RGB function




https://belcour.github.io/blog/research/2017/05/01/brdf-thin-film.html

# Ray-tracing: advantages

- Slow, but no extra charge for:
  - hidden surface removal,
  - shadows,
  - transparency,
  - texture-mapping (including procedural).
- Inter-reflexions between objects,
- Any graphics primitives,
- Global  illumination model.

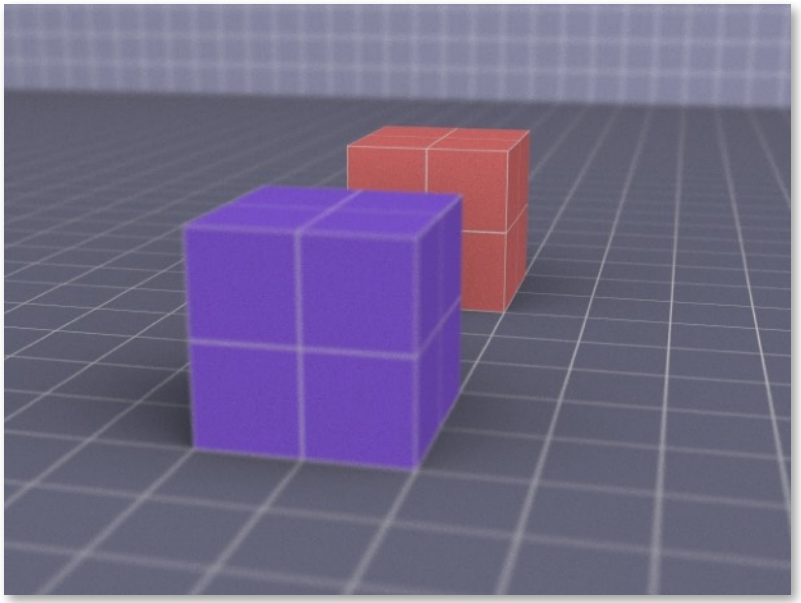
# Ray-tracing: issues

- Limited to Snell-Descartes:
  - all objects appear metallic.
- Ray tree limited to a certain depth:
  - complex objects may be a problem (diamonds, cristal glass)

# Even more rays

« Distributed Ray Tracing » Cook et al. (1984)

- Soft shadows
  - Several rays for each extended light source
- Anti-aliasing
  - Several rays for each pixel
- Glossy Reflection
  - Several rays are reflected
- Motion blur
  - Several rays during time
- Depth of field
  - Several rays per pixel, focusing with a lens



# Further readings

- Morgan McGuire: Ray-tracing tips
- Peter Shirley: Ray-tracing in a week-end
- Eric Haines: Ray-tracing resources