图形绘制技术（Rendering）
Chapter 3: Ray Tracing Basics

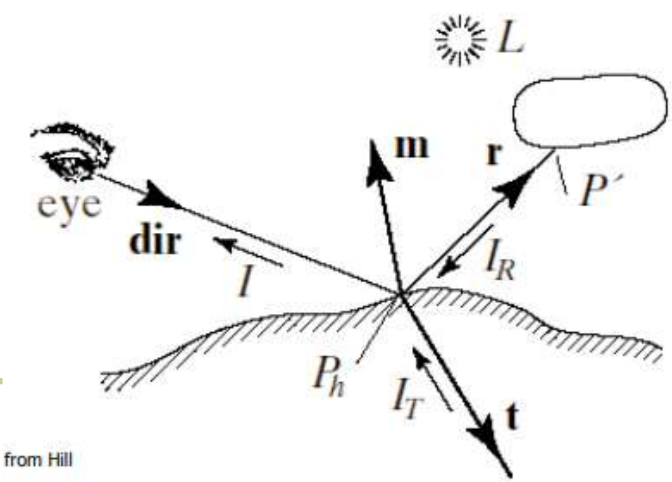
过 洁
南京大学计算机科学与技术系
guojie@nju.edu.cn

Ray Tracing: Introduction

- Ray tracing is one of the most popular methods used in 3D computer graphics to render an image
- Good at simulating specular effects
- Tracing the path taken by a ray of light through the scene
- Rays are reflected, refracted, or absorbed whenever intersect an object
- Can produce shadows

Ray Tracing: Model

- Perceived color at point **p** is an additive combination of local illumination (e.g., Phong), reflection, and refraction effects
- Compute reflection, refraction contributions by tracing respective rays back from **p** to surfaces they came from and evaluating local illumination at those locations
- Apply operation recursively to some maximum depth to get:
  - – Reflections of reflections of ...
  - – Refractions of refractions of ...
  - – And of course mixtures of the two

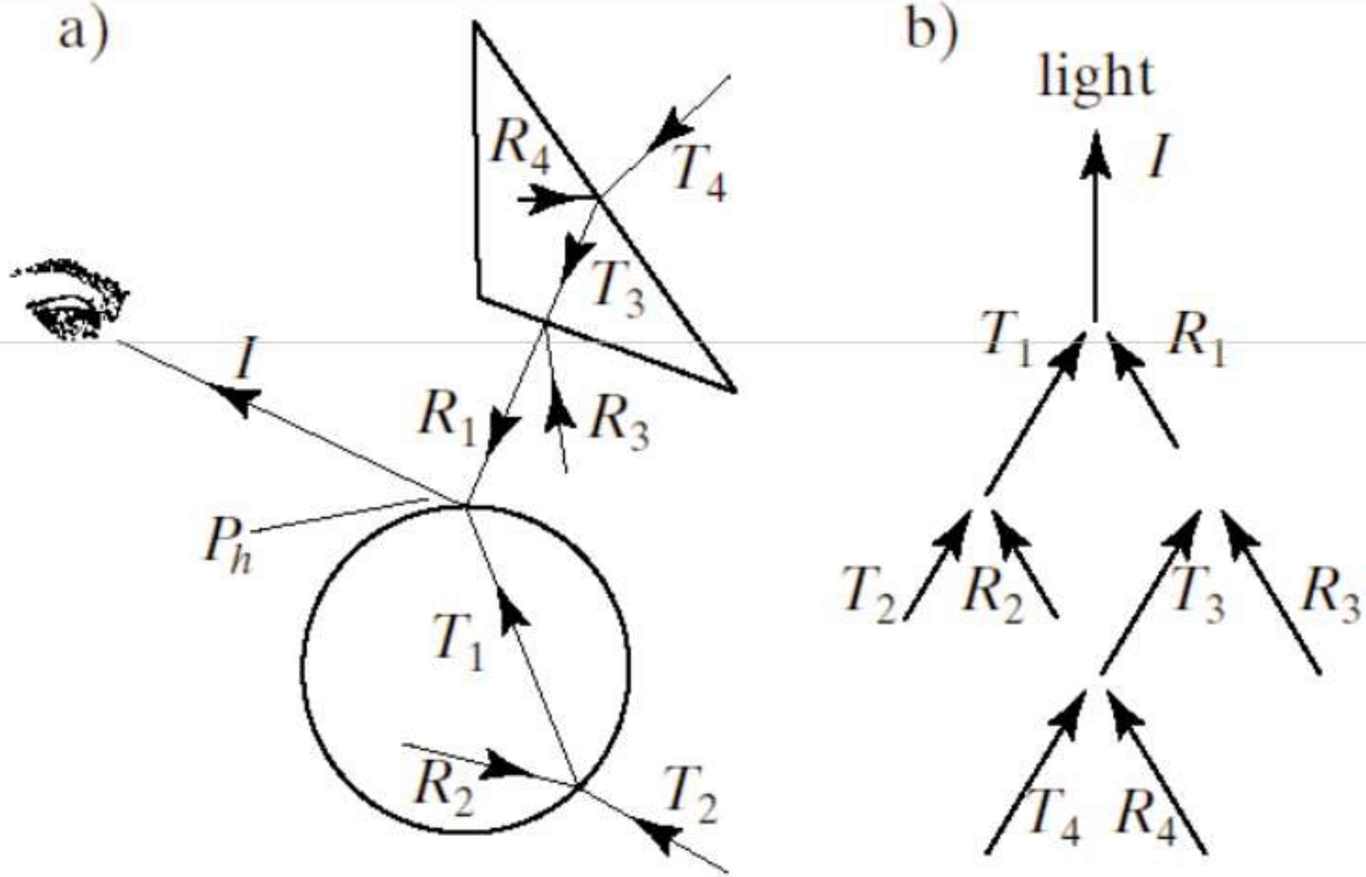


from Hill

Ray Tracing: History

- Appel 68
- Whitted 80 [recursive ray tracing]
  - Landmark in computer graphics
- Lots of work on various geometric primitives
- Lots of work on accelerations
- Current Research
  - Real-Time raytracing (historically, slow technique)
  - Ray tracing architecture

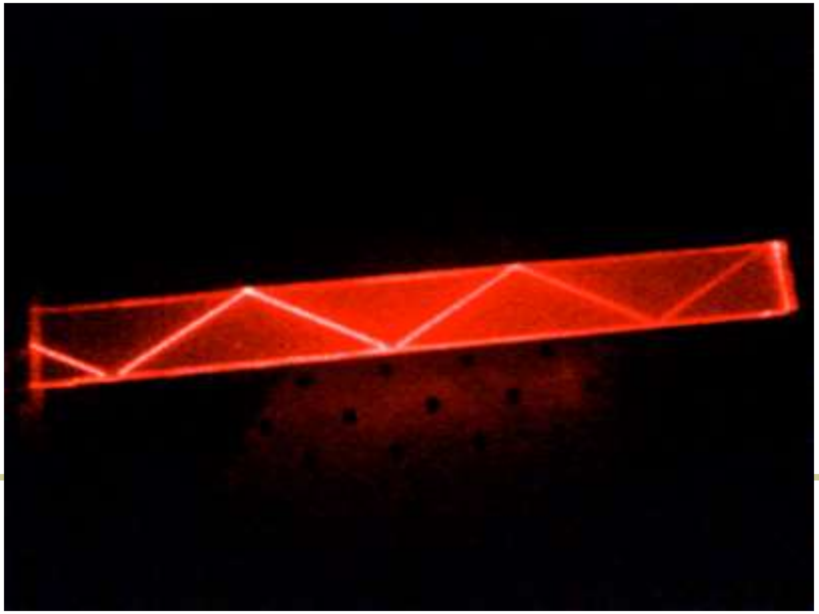
Ray Tracing: Recursion



from Hill

Problems with Recursion

- Reflection rays may be traced forever
- Generally, set maximum recursion depth
- Same for transmitted rays (take refraction into account)

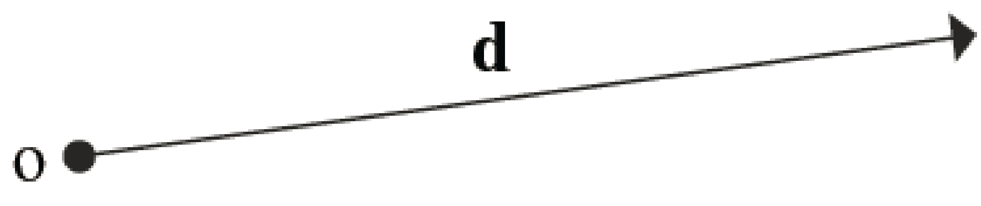


Ray/Object Intersections

- Heart of Ray Tracer
  - One of the main initial research areas
  - Optimized routines for wide variety of primitives
- Various types of info
  - Shadow rays: Intersection/No Intersection
  - Primary rays: Point of intersection, material, normals
  - Texture coordinates
- Work out examples
  - Triangle, sphere, polygon, general implicit surface

Rays

- A ray is a semi-infinite line define by its origin **o** and its direction vector **d**:

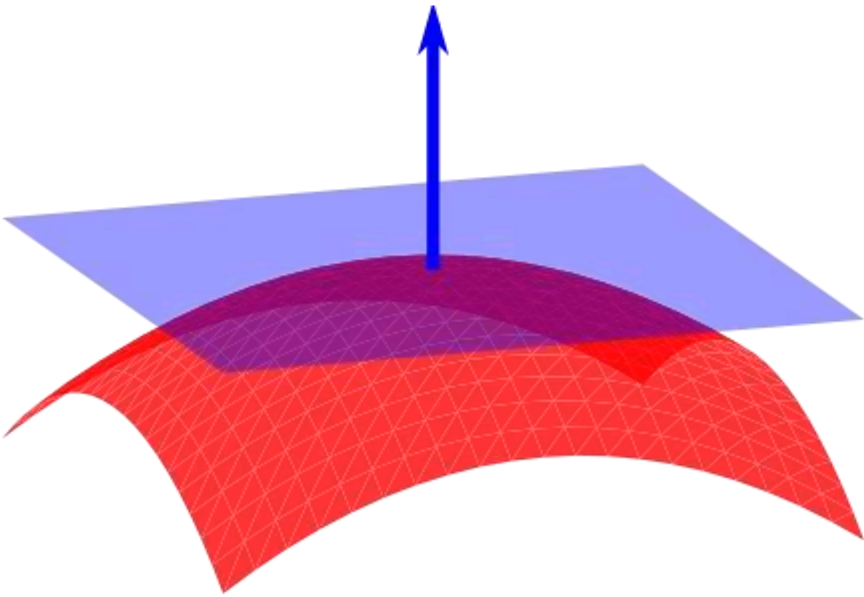


$$\mathrm{r}(t) = \mathrm{o} + t\mathbf{d} \qquad 0 \leq t \leq \infty$$

Normals

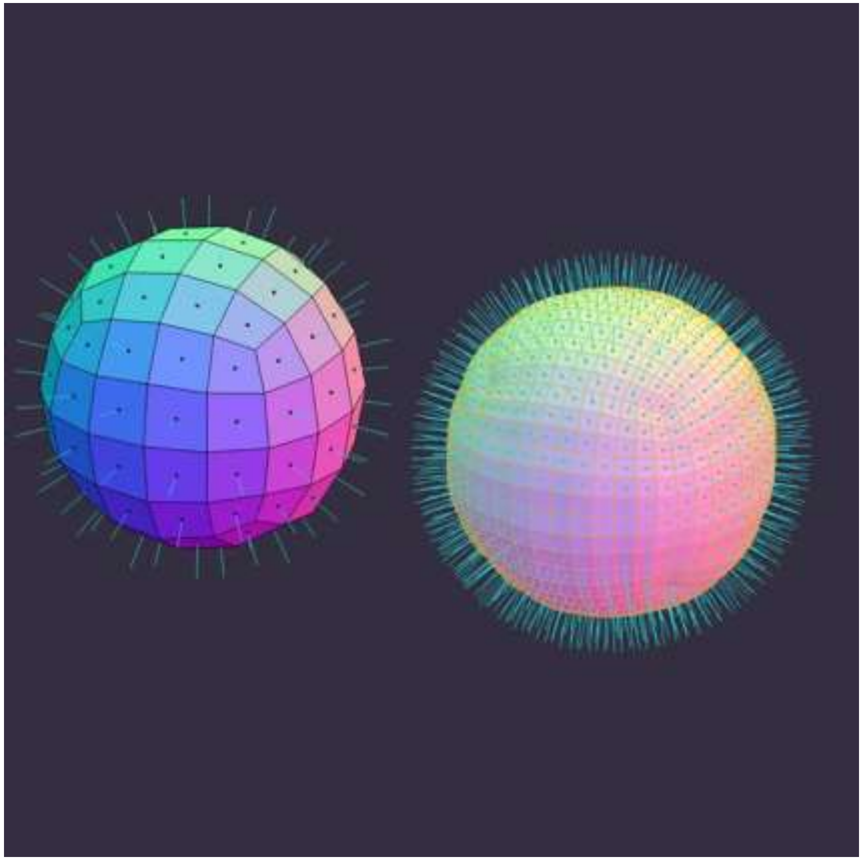
- A *surface normal* (or just *normal*) is a vector that is perpendicular to a surface at a particular position.

- Very important for surface shading.



Normals

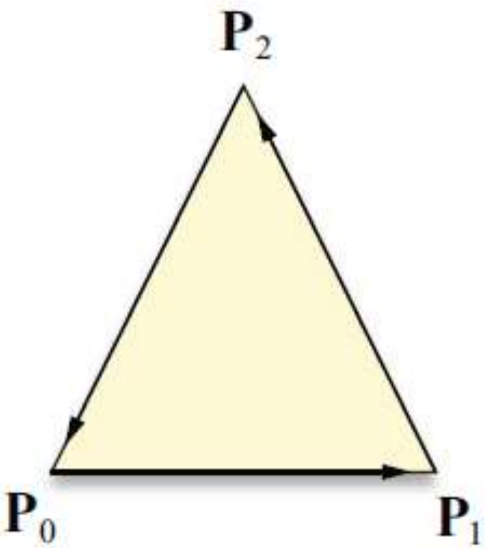
- Visualization of surface normals



Calculating Normal Vectors

- Calculate the normal vector for a single triangle by using the cross product.

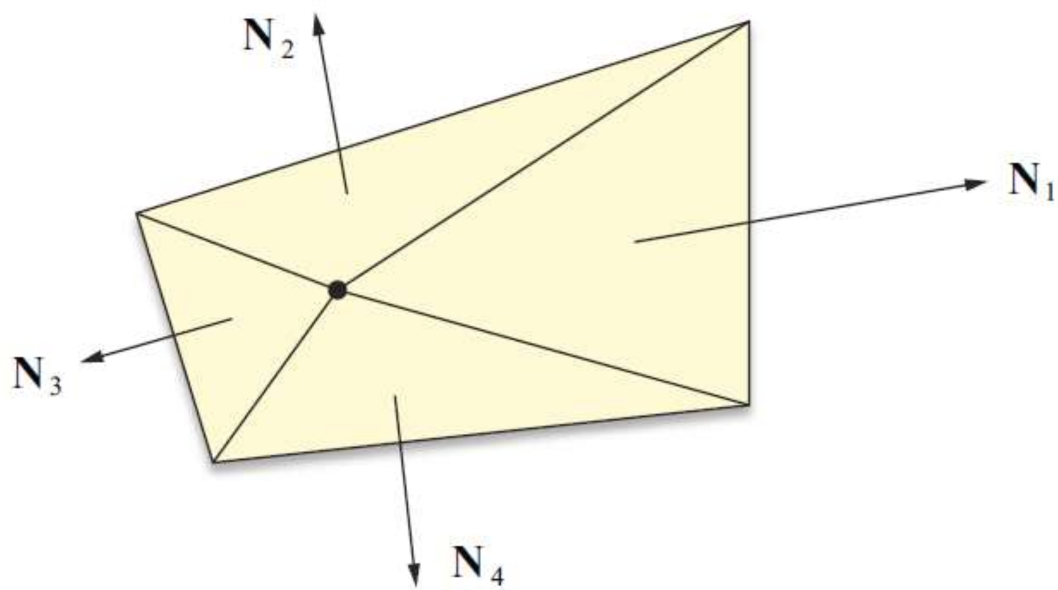
$$\mathbf{N} = \frac{(\mathbf{P}_1 - \mathbf{P}_0) \times (\mathbf{P}_2 - \mathbf{P}_0)}{\|(\mathbf{P}_1 - \mathbf{P}_0) \times (\mathbf{P}_2 - \mathbf{P}_0)\|}$$



Calculating Normal Vectors

- Calculate the normal vector for a vertex

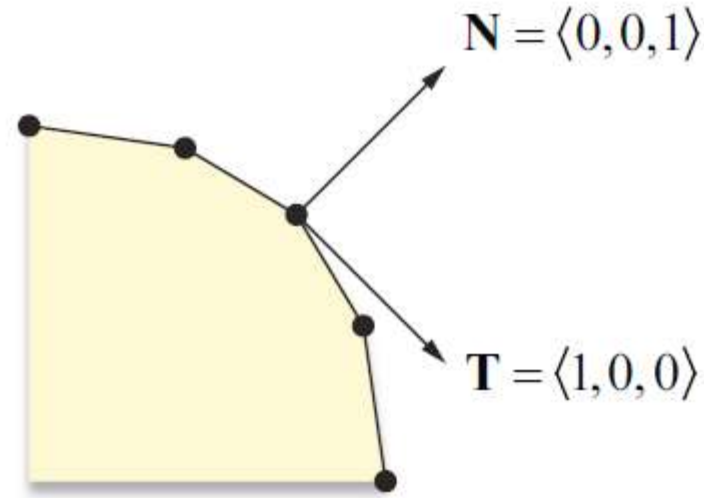
$$\mathbf{N}_{\text{vertex}} = \frac{\sum_{i=1}^{k} \mathbf{N}_i}{\left\| \sum_{i=1}^{k} \mathbf{N}_i \right\|}$$



Tangent Space

- Defined by three vectors: the normal vector **N**, the tangent vector **T**, and the bitangent vector **B**.
- Transform from tangent space into object space using the matrix:

$$\begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$



Sphere-Definition

- A sphere of radius r at the origin
- Implicit: $x^2+y^2+z^2-r^2=0$
- Parametric: $f(\theta,\phi)$

$$x=r\sin\theta\cos\phi$$

$$y=r\sin\theta\sin\phi$$

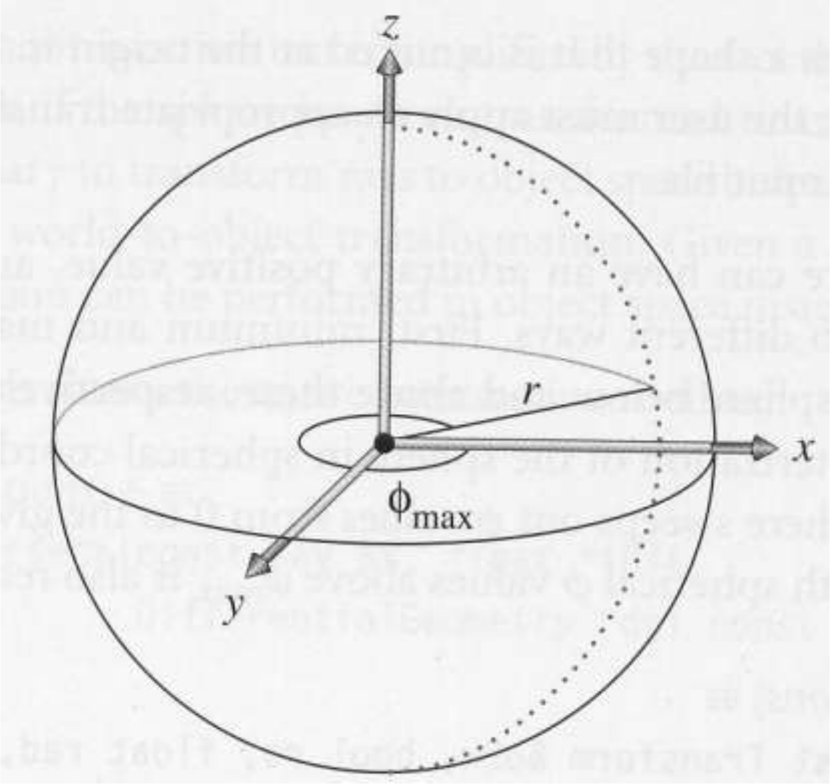
$$z=r\cos\theta$$

mapping $f(u,v)$ over $[0,1]^2$

$$\phi=u\phi_{max}$$

$$\theta=\theta_{min}+v(\theta_{max}-\theta_{min})$$

useful for texture mapping



Sphere-Intersection

$$x^2 + y^2 + z^2 = r^2$$

$$(o_x + td_x)^2 + (o_y + td_y)^2 + (o_z + td_z)^2 = r^2$$

$$At^2 + Bt + C = 0$$

Step 1

$$A = d_x^2 + d_y^2 + d_z^2$$

$$B = 2(d_x o_x + d_y o_y + d_z o_z)$$

$$C = o_x^2 + o_y^2 + o_z^2 - r^2$$

Sphere-Intersection

$$t_0 = \frac{-B - \sqrt{B^2 - 4AC}}{2A} \qquad t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

Step 2

If (B^2-4AC<0) then the ray misses the sphere

Step 3

Calculate t_0 and test if $t_0<0$ (actually mint, maxt)

Step 4

Calculate t_1 and test if $t_1<0$

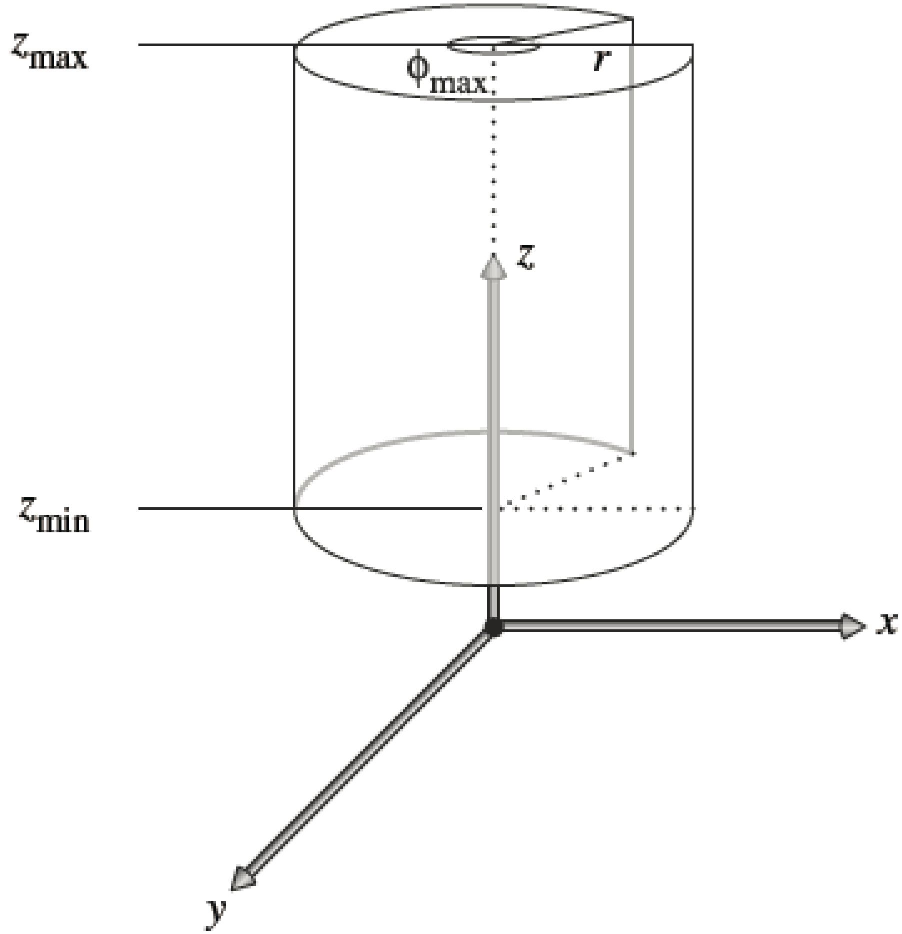
Cylinder-Definition

$$\phi = u\phi_{\max}$$

$$x = r\cos\phi$$

$$y = r\sin\phi$$

$$z = z_{\min} + v(z_{\max} - z_{\min})$$



Cylinder-Intersection

$$x^2 + y^2 = r^2$$

$$\left(o_x + td_x\right)^2 + \left(o_y + td_y\right)^2 = r^2$$

$$At^2 + Bt + C = 0$$

$$A = d_x^2 + d_y^2$$

$$B = 2(d_x o_x + d_y o_y)$$

$$C = o_x^2 + o_y^2 - r^2$$

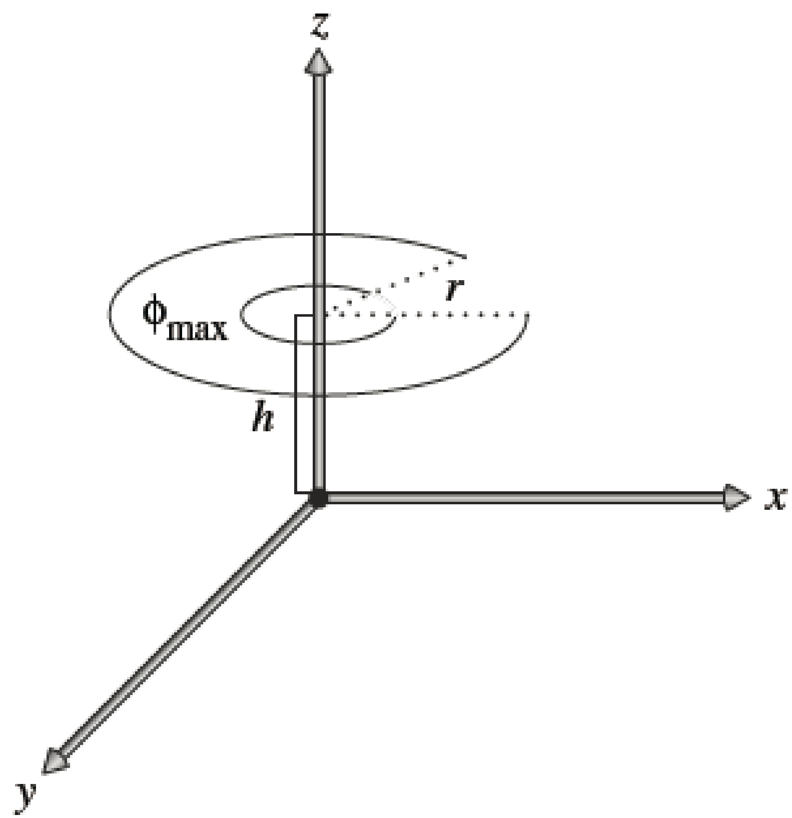
Disk-Definition

$$\phi = u\phi_{\max}$$

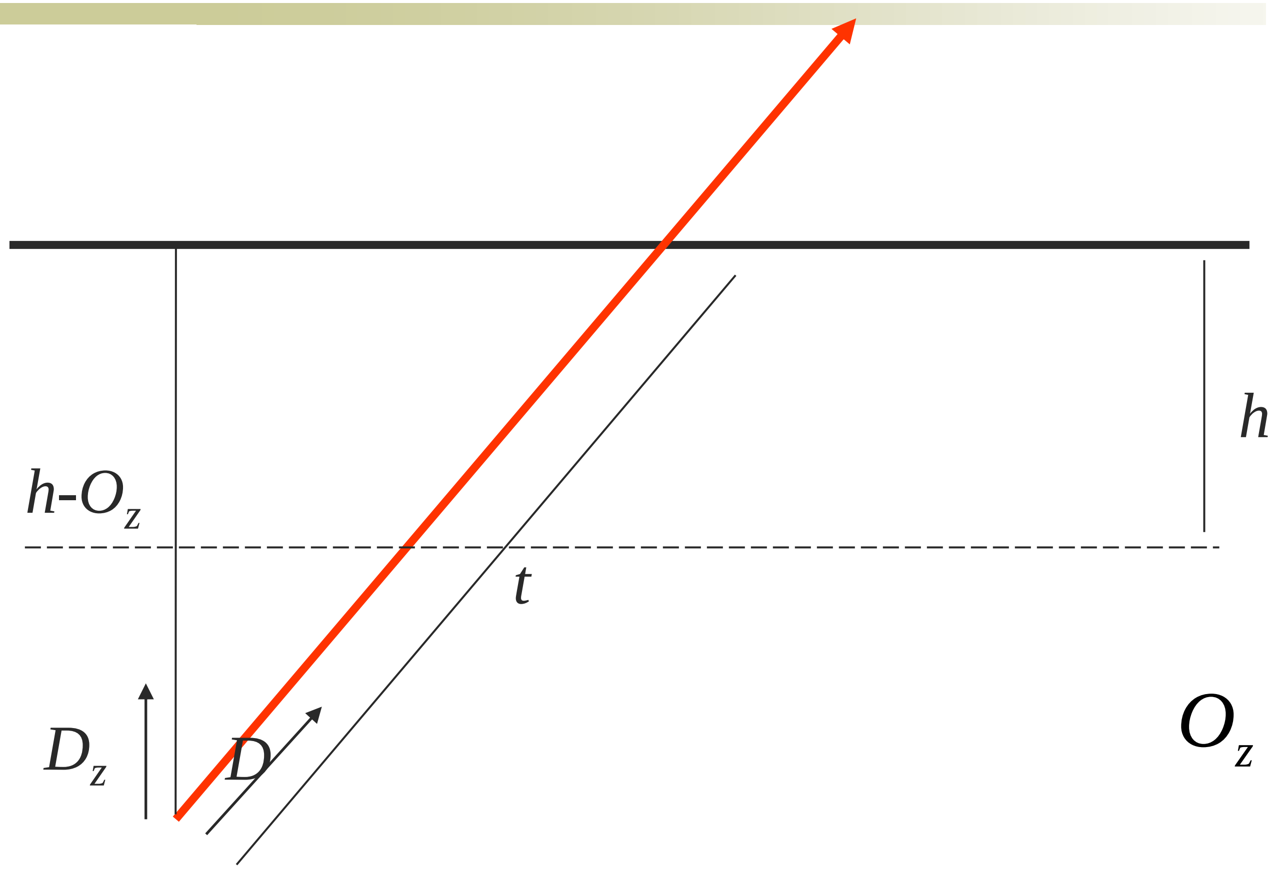
$$x = ((1-v)r_i + vr)\cos\phi$$

$$y = ((1-v)r_i + vr)\sin\phi$$

$$z = h$$



Disk-Intersection

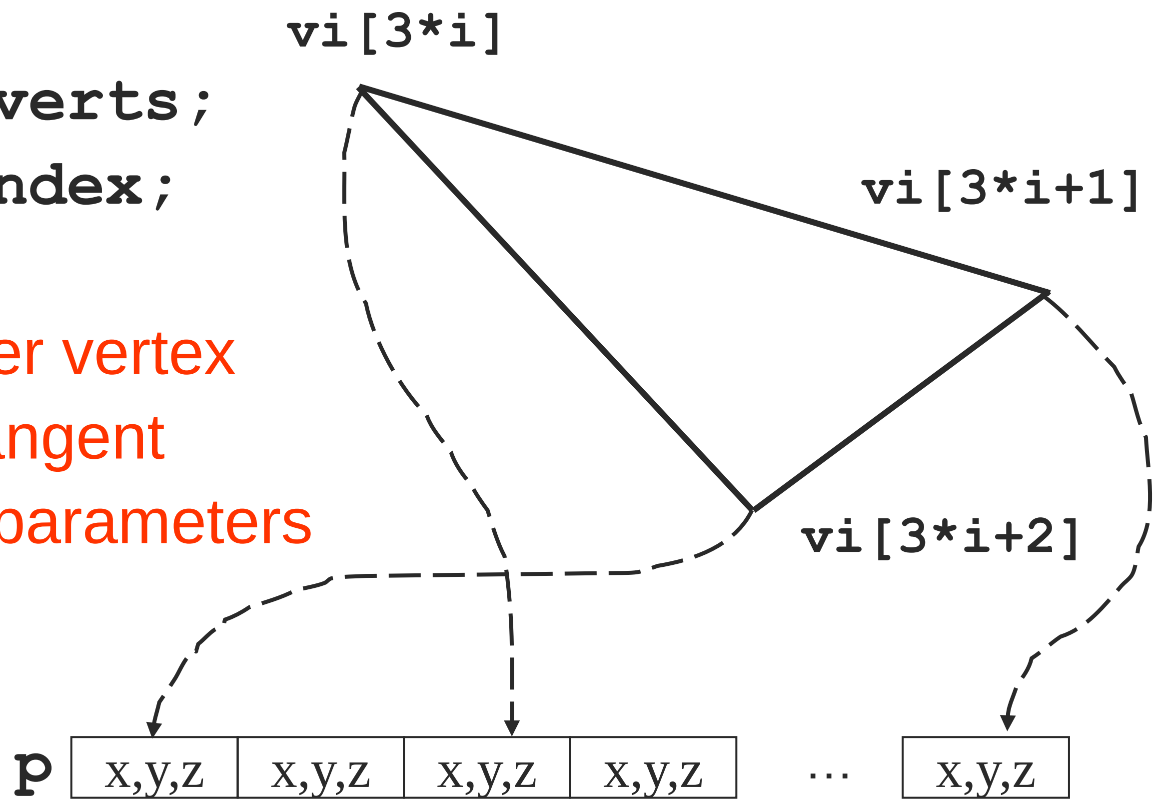


$$O_z + tD_z = h$$

$$t = \frac{h - O_z}{D_z}$$

Triangle Mesh

```
class TriangleMesh : public Shape {
…
  int ntris, nverts;
  int *vertexIndex;
  Point *p;
  Normal *n;  per vertex
  Vector *s;  tangent
  float *uvs;  parameters
}
```



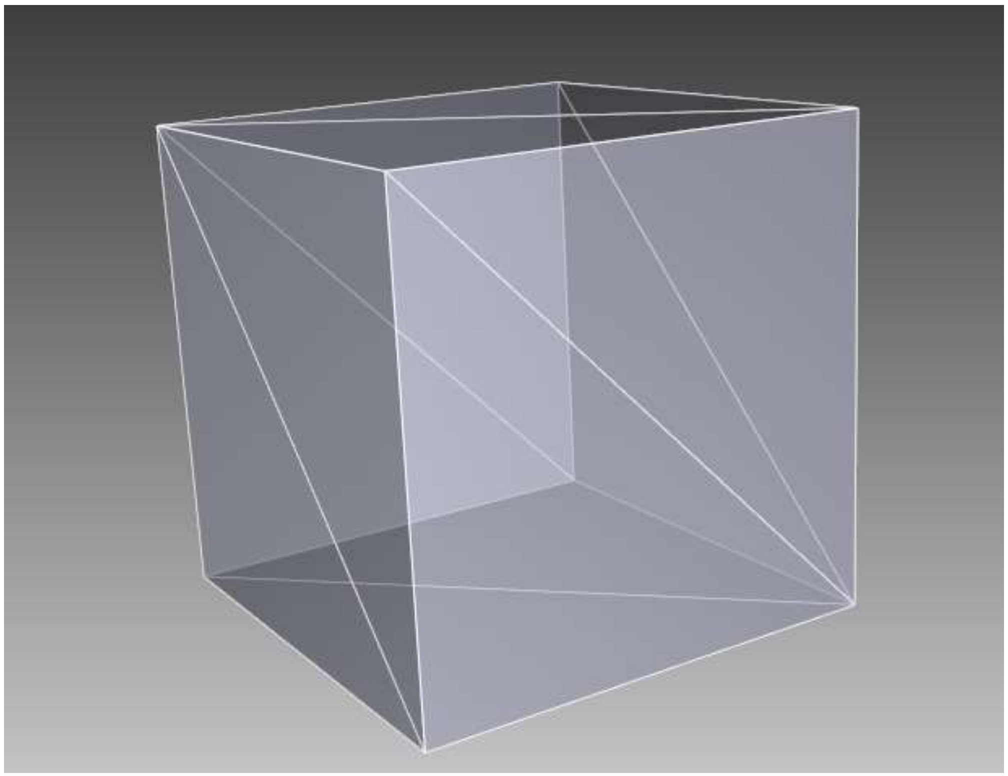
Triangle Mesh

- Wavefront .obj file

```
# List of geometric vertices, with (x,y,z[,w]) coordinates, w is optional and defaults to 1.0.
v 0.123 0.234 0.345 1.0
v ...
...
# List of texture coordinates, in (u, v [,w]) coordinates, these will vary between 0 and 1, w is optional and defaults to 0.
vt 0.500 1 [0]
vt ...
...
# List of vertex normals in (x,y,z) form; normals might not be unit vectors.
vn 0.707 0.000 0.707
vn ...
...
# Parameter space vertices in ( u [,v] [,w] ) form; free form geometry statement ( see below )
vp 0.310000 3.210000 2.100000
vp ...
...
# Polygonal face element (see below)
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f 7//1 8//2 9//3
f ...
...
```

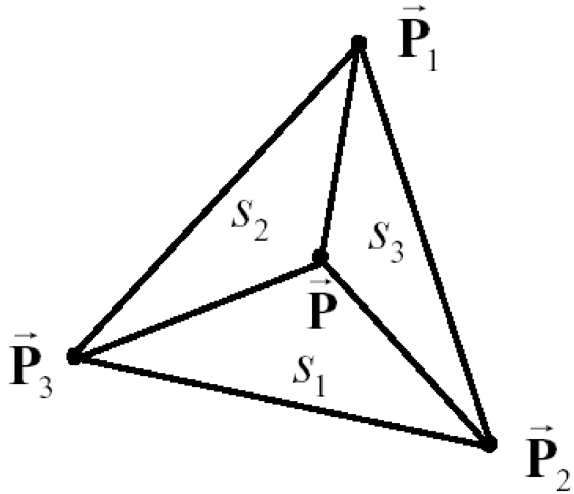
Triangle Mesh

- Wavefront .obj file



```
# cube.obj
#

g cube

v  0.0  0.0  0.0
v  0.0  0.0  1.0
v  0.0  1.0  0.0
v  0.0  1.0  1.0
v  1.0  0.0  0.0
v  1.0  0.0  1.0
v  1.0  1.0  0.0
v  1.0  1.0  1.0

vn  0.0  0.0  1.0
vn  0.0  0.0 -1.0
vn  0.0  1.0  0.0
vn  0.0 -1.0  0.0
vn  1.0  0.0  0.0
vn -1.0  0.0  0.0

f  1//2  7//2  5//2
f  1//2  3//2  7//2
f  1//6  4//6  3//6
f  1//6  2//6  4//6
f  3//3  8//3  7//3
f  3//3  4//3  8//3
f  5//5  7//5  8//5
f  5//5  8//5  6//5
f  1//4  5//4  6//4
f  1//4  6//4  2//4
f  2//1  6//1  8//1
f  2//1  8//1  4//1
```

Ray-Triangle Intersection



$$s_1 = \text{area}(\Delta \mathbf{P}\mathbf{P}_2\mathbf{P}_3)$$

$$s_2 = \text{area}(\Delta \mathbf{P}\mathbf{P}_3\mathbf{P}_1)$$

$$s_3 = \text{area}(\Delta \mathbf{P}\mathbf{P}_1\mathbf{P}_2)$$

Barycentric coordinates

$$\vec{\mathbf{P}} = s_1\vec{\mathbf{P}}_1 + s_2\vec{\mathbf{P}}_2 + s_3\vec{\mathbf{P}}_3$$

Inside criteria

$$0 \le s_1 \le 1$$

$$0 \le s_2 \le 1$$

$$0 \le s_3 \le 1$$

$$s_1 + s_2 + s_3 = 1$$

Ray-Triangle Intersection

a point on the ray

a point inside the triangle

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2$$

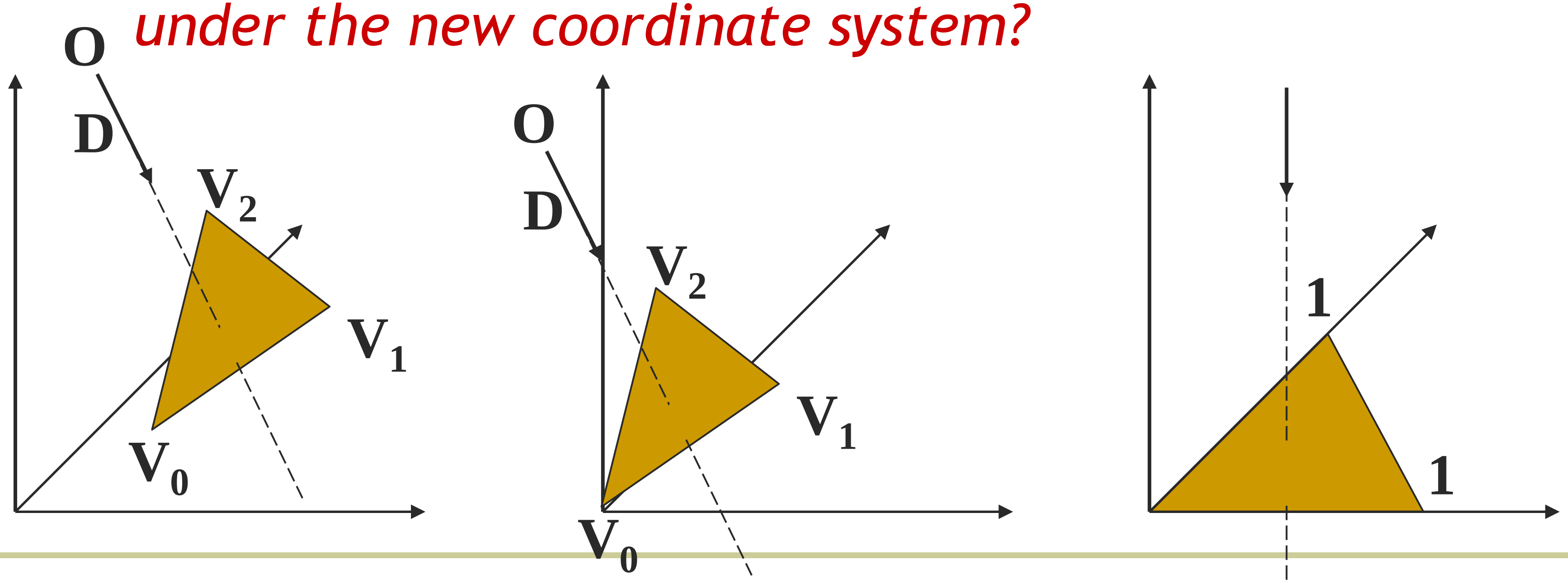
$$u, v \geq 0 \text{ and } u + v \leq 1$$

$$\begin{bmatrix} -D & V_1 - V_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0$$

Ray-Triangle Intersection

$$\begin{bmatrix} -D & V_1 - V_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0$$

Geometric interpretation: what is O's coordinate under the new coordinate system?



translation *rotation*

Ray-Triangle Intersection

$$\begin{bmatrix} -D & V_1 - V_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0$$

$$E_1 = V_1 - V_0 \qquad E_2 = V_2 - V_0 \qquad T = O - V_0$$

$$\begin{bmatrix} -D & E_1 & E_2 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = T$$

Ray-Triangle Intersection

$$\begin{bmatrix} -D & E_1 & E_2 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = T$$

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\left| -D, E_1, E_2 \right|} \begin{bmatrix} \left| T, E_1, E_2 \right| \\ \left| -D, T, E_2 \right| \\ \left| -D, E_1, T \right| \end{bmatrix}$$ Cramer's rule

$$\left| A, B, C \right| = -(A \times C) \cdot B = -(C \times B) \cdot A$$

Ray-Triangle Intersection

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\left| -D, E_1, E_2 \right|} \begin{bmatrix} \left| T, E_1, E_2 \right| \\ \left| -D, T, E_2 \right| \\ \left| -D, E_1, T \right| \end{bmatrix}$$

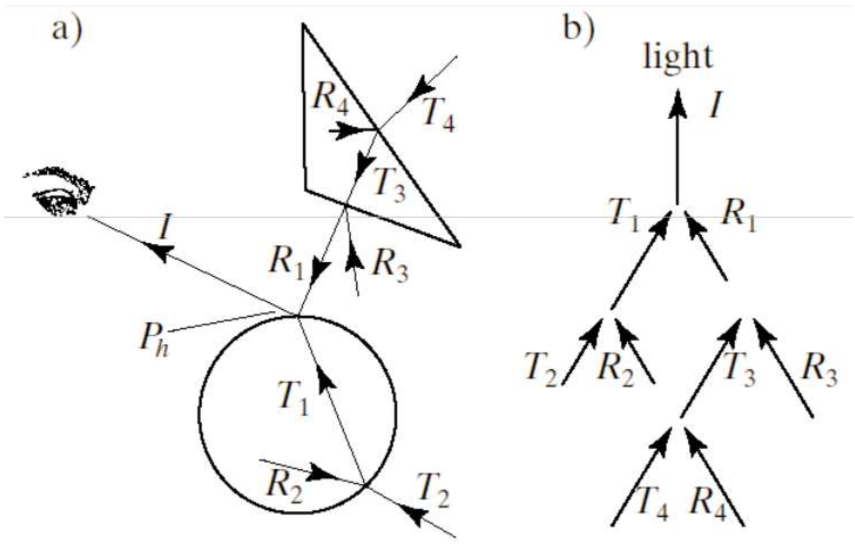
$$Q = T \times E_1 \qquad P = D \times E_2$$

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix} Q \cdot E_2 \\ P \cdot T \\ Q \cdot D \end{bmatrix}$$

1 division
27 multiplies
17 adds

The Cost of Ray Tracing

- Many Primitives
- Many Rays
- Expensive Intersections



from Hill

Acceleration

Testing each object for each ray is slow

- Fewer Rays

  Adaptive sampling, depth control

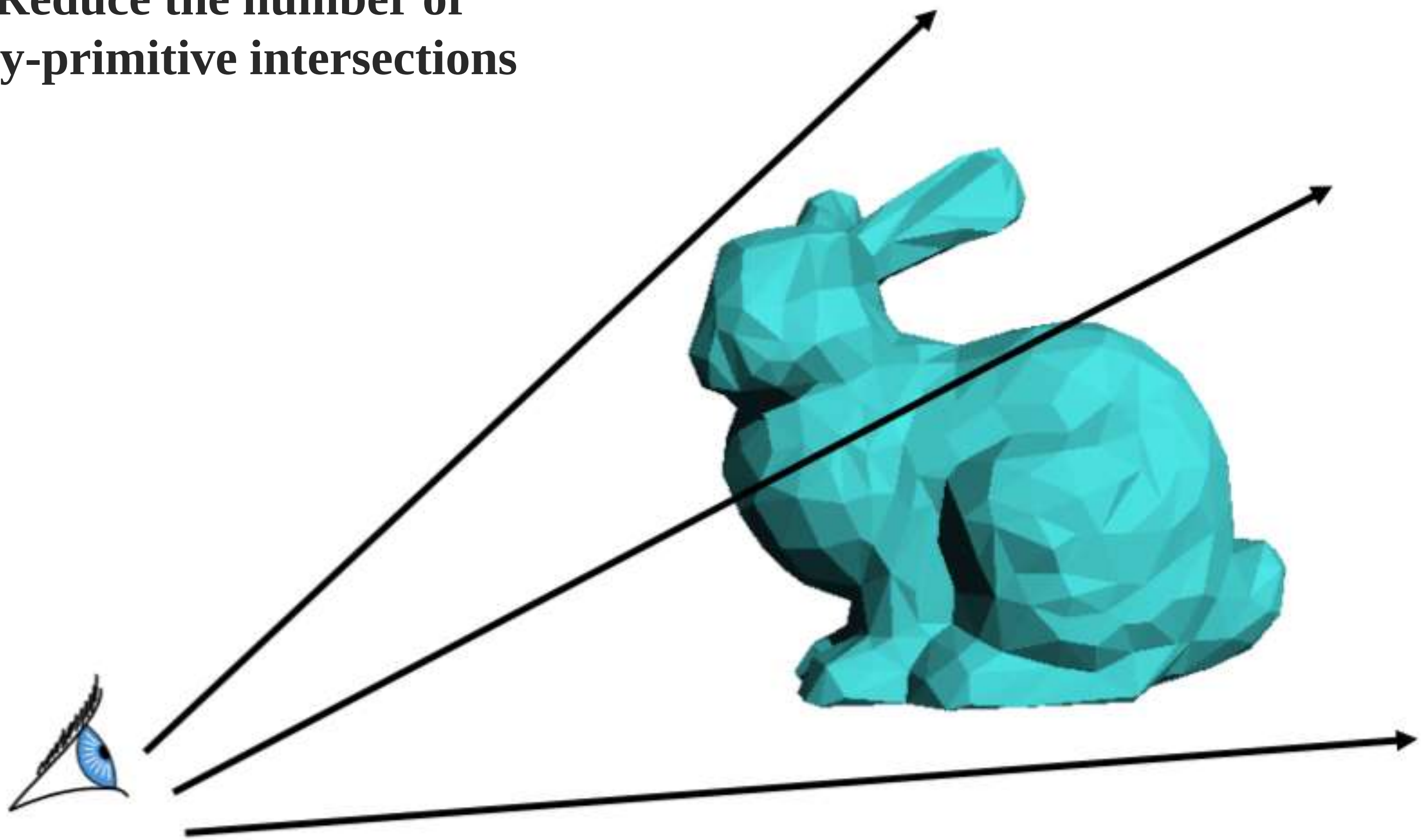
- Generalized Rays

  Beam tracing, cone tracing, pencil tracing etc.

- Faster Intersections
  - Optimized Ray-Object Intersections
  - ***Fewer Intersections***

Fewer Intersections

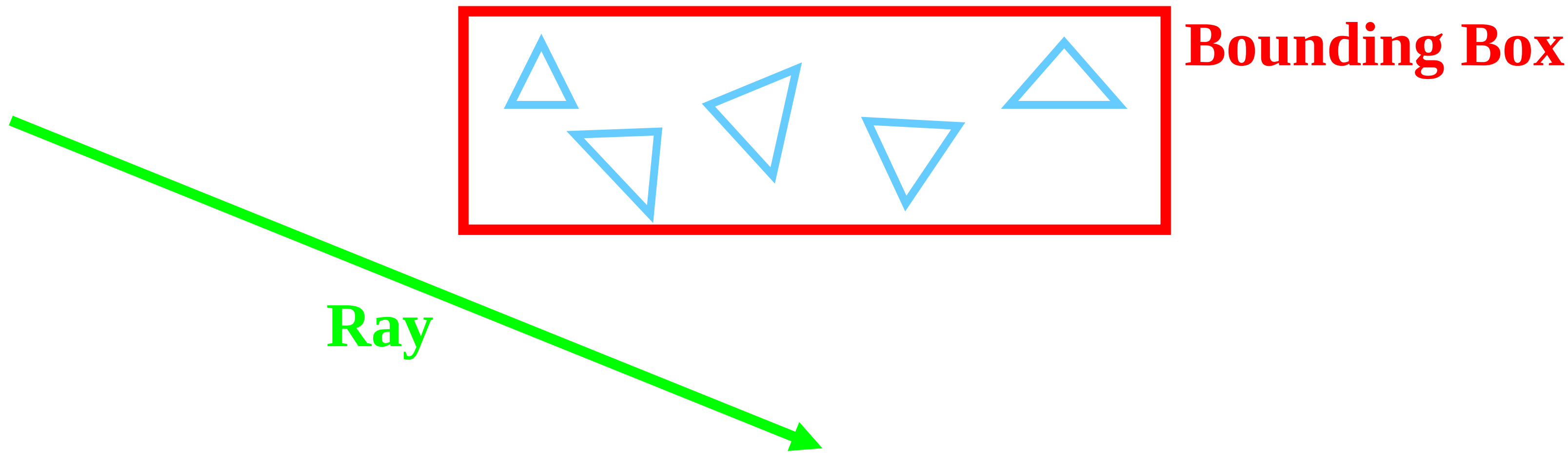
Reduce the number of ray-primitive intersections



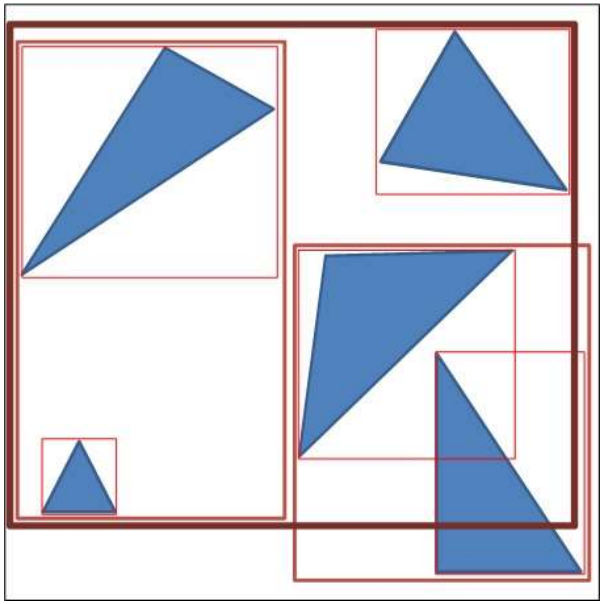
Acceleration Structures

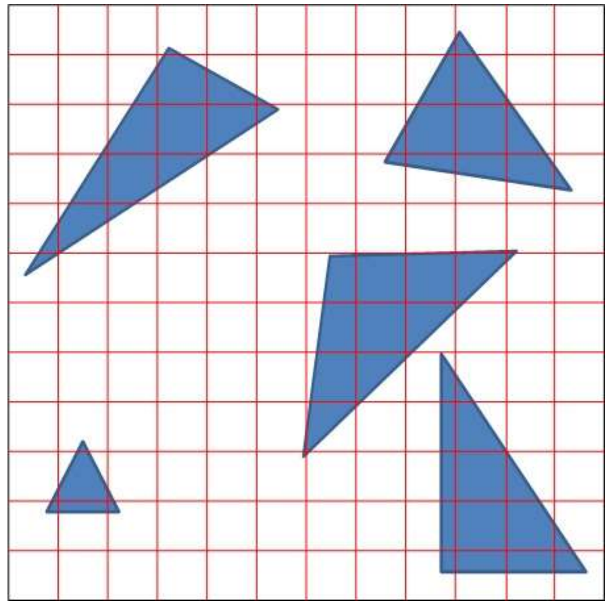
Bounding boxes (possibly hierarchical)

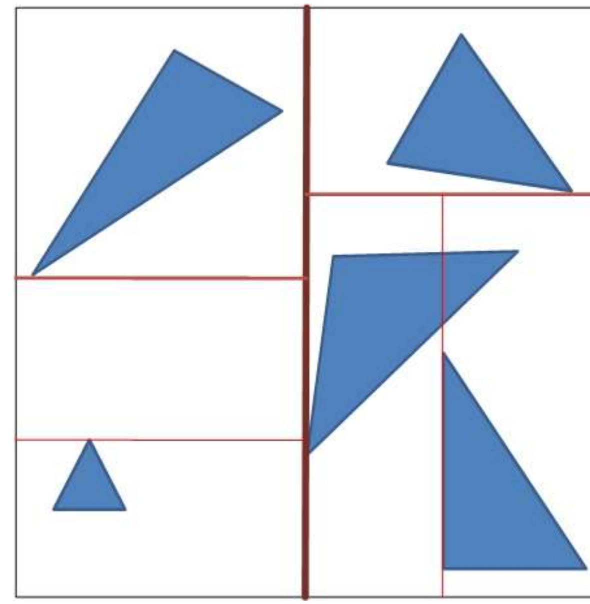
- If no intersection bounding box, needn't check objects



Acceleration Structures







Bounding volume hierarchy
BVH（层次包围盒）

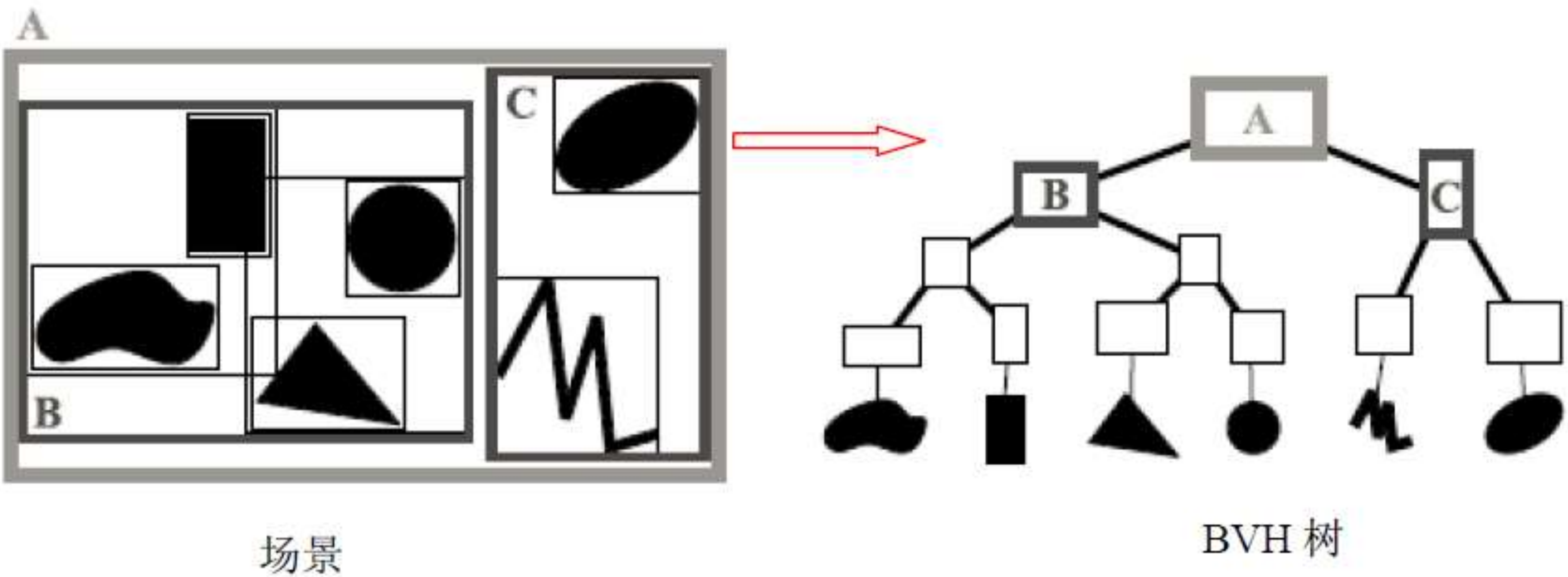
Uniform grid
均匀网格

Kd-tree

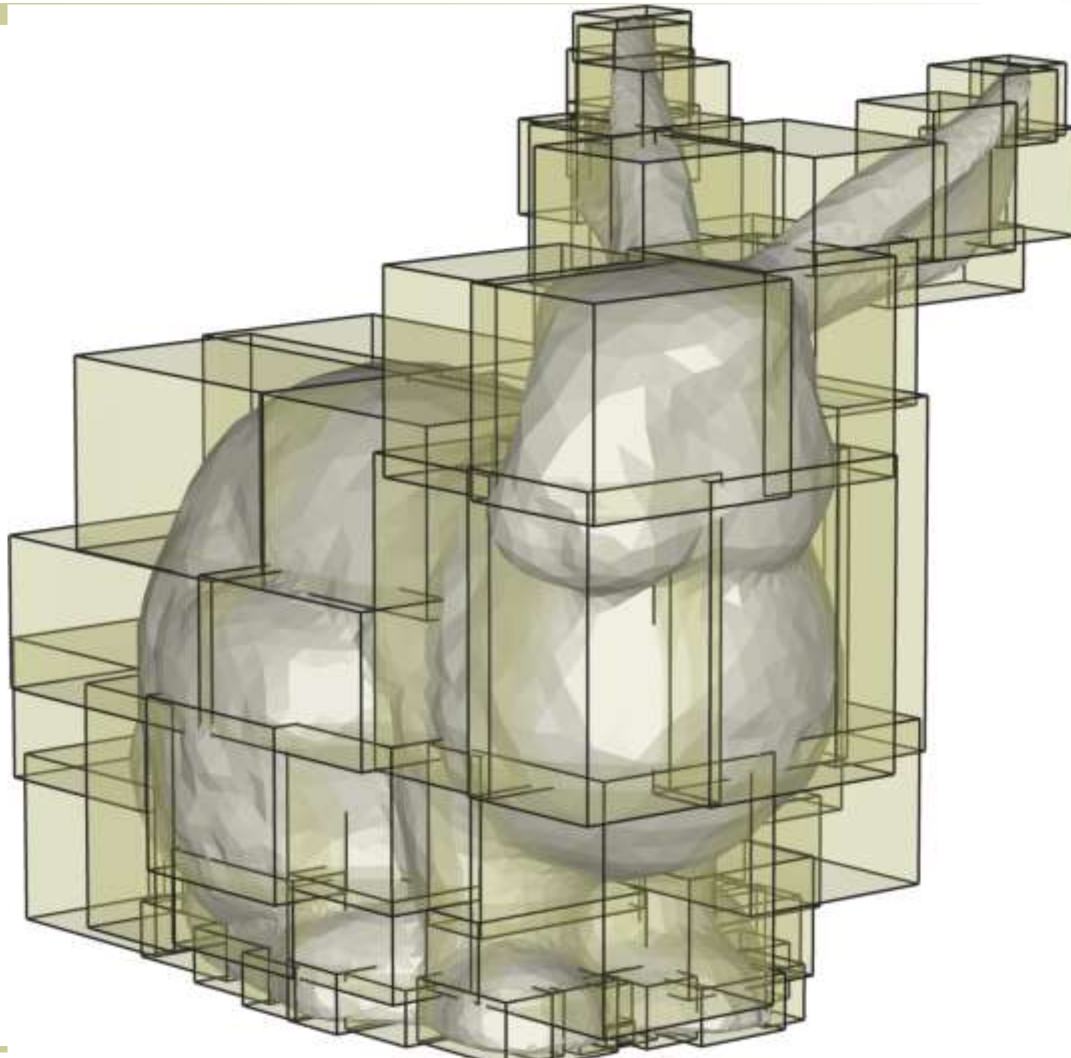
Object subdivision approaches

Space subdivision approaches

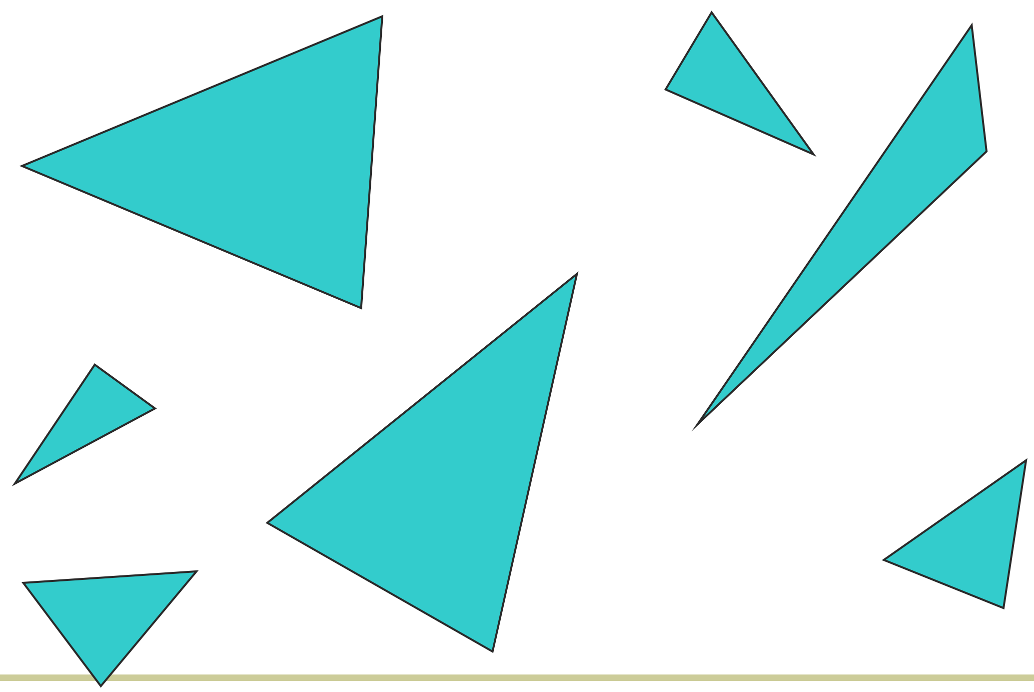
BVH in 2D



BVH in 3D

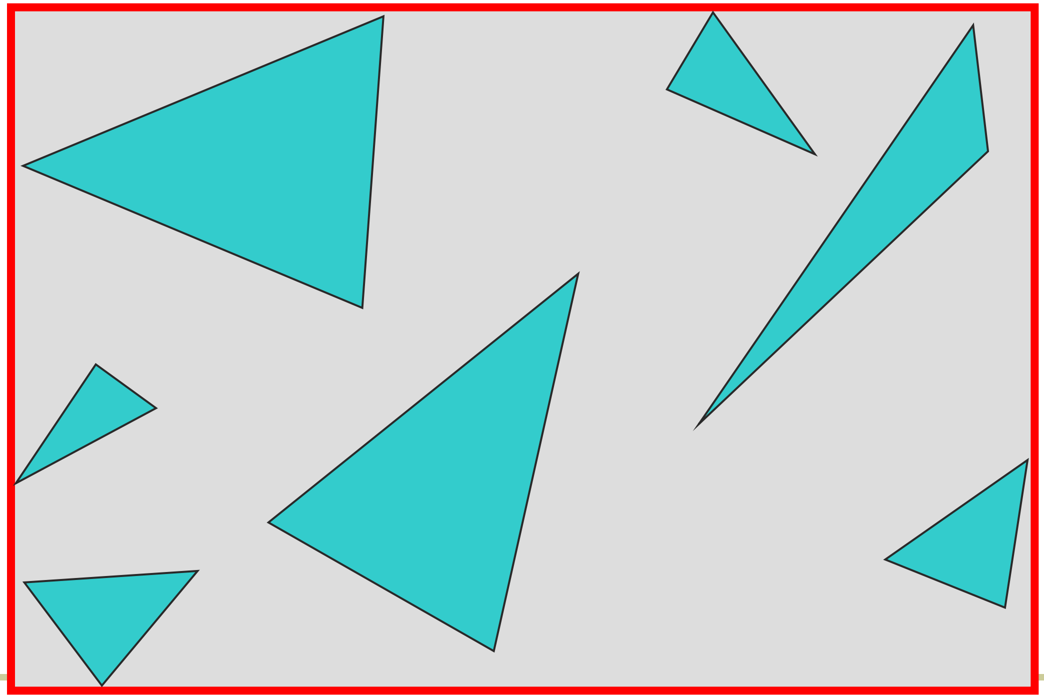


BVH Construction



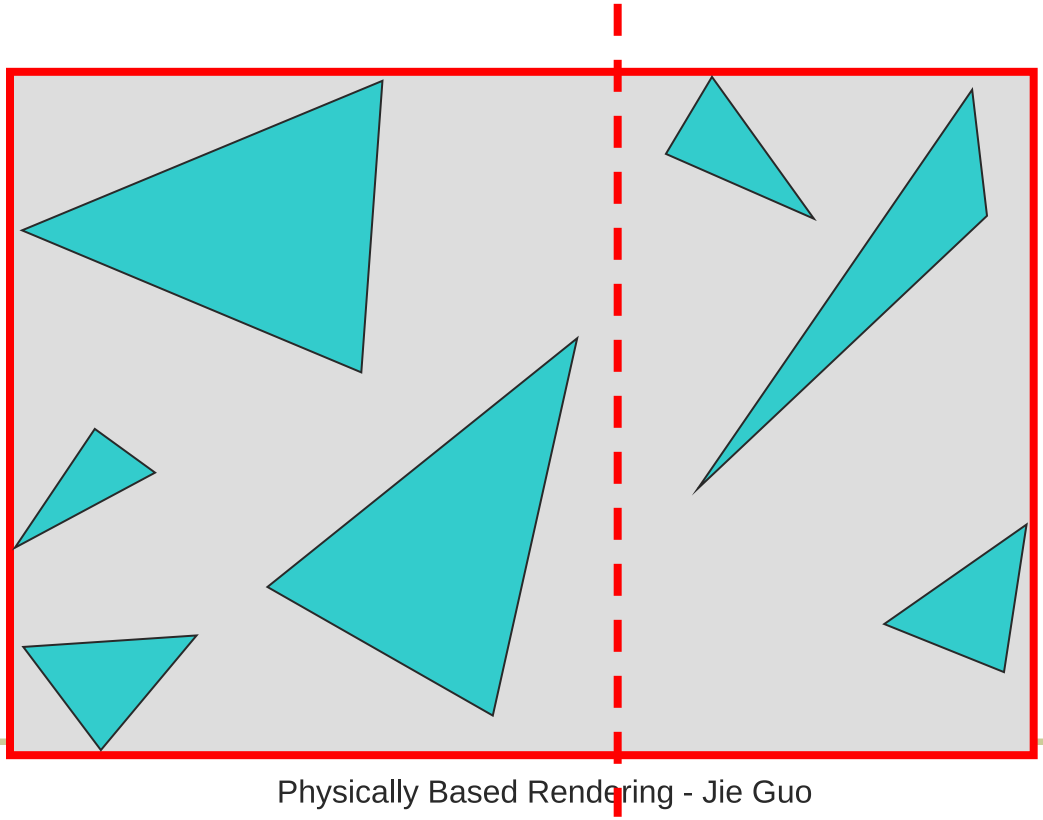
BVH Construction

- Find bounding box of objects



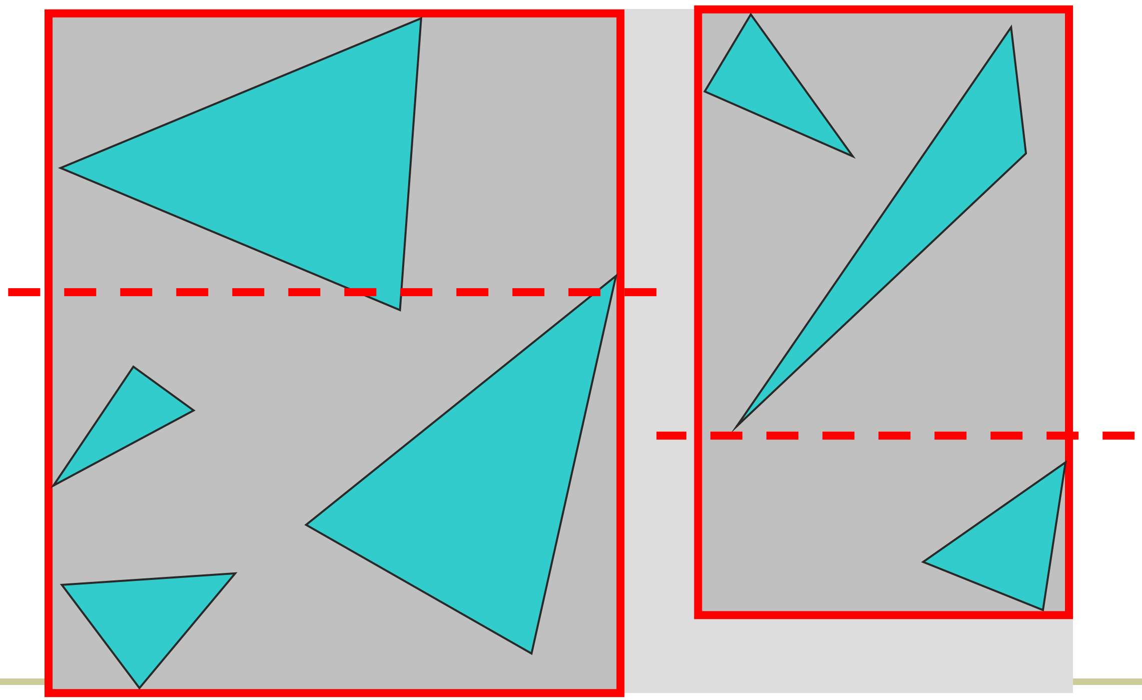
BVH Construction

- Find bounding box of objects
- Split objects into two groups



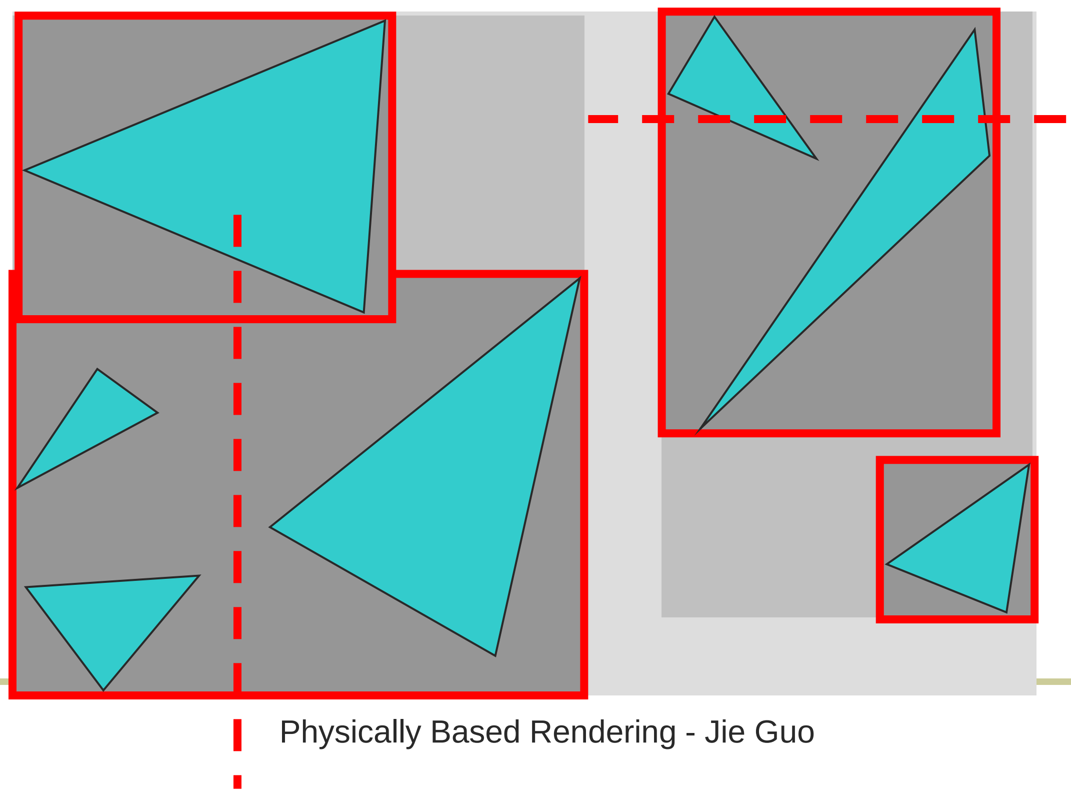
BVH Construction

- Find bounding box of objects
- Split objects into two groups
- Recurse



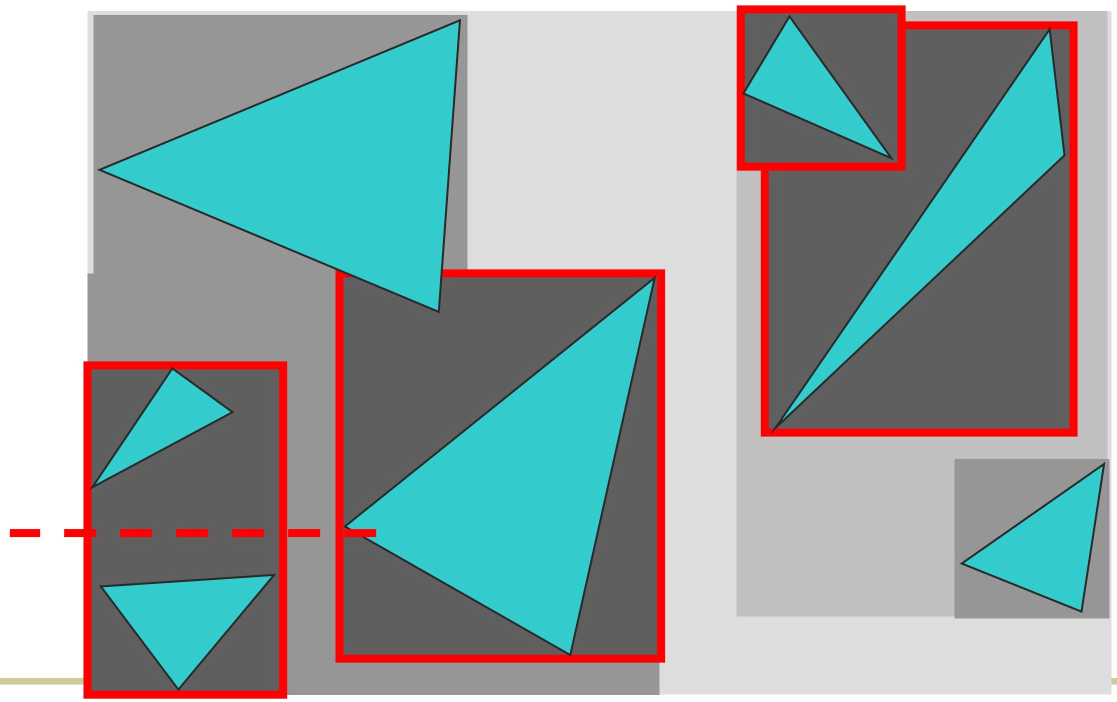
BVH Construction

- Find bounding box of objects
- Split objects into two groups
- Recurse



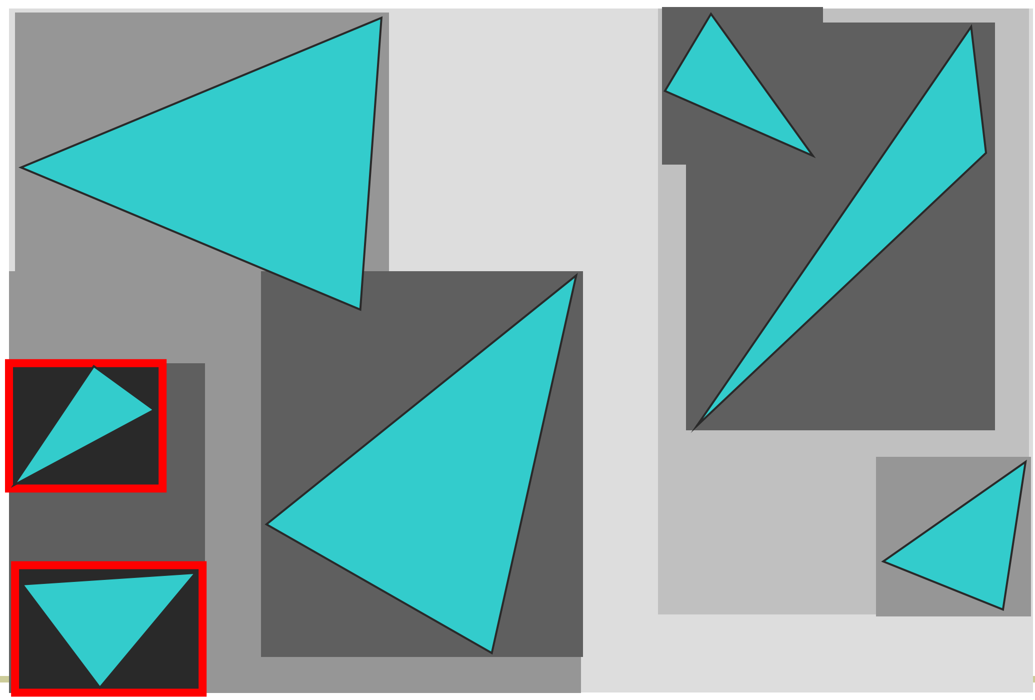
BVH Construction

- Find bounding box of objects
- Split objects into two groups
- Recurse



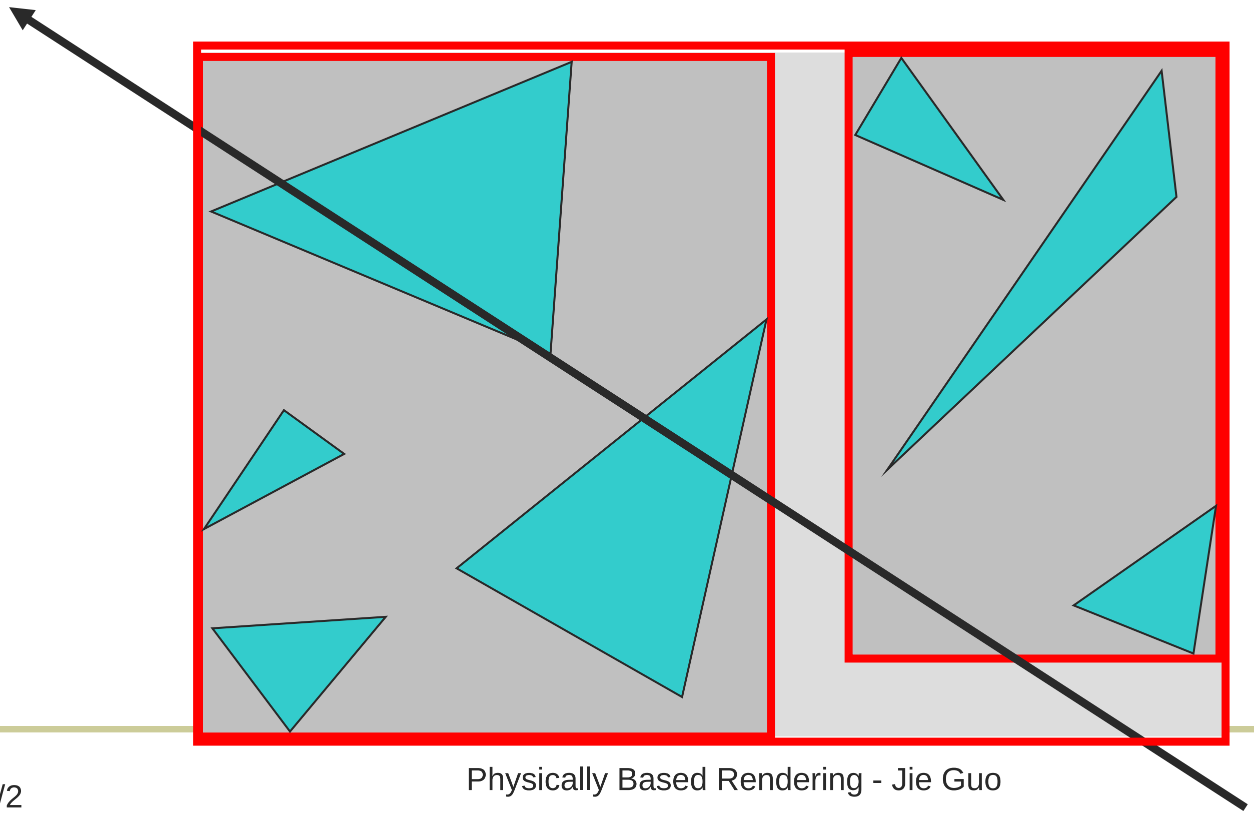
BVH Construction

- Find bounding box of objects
- Split objects into two groups
- Recurse



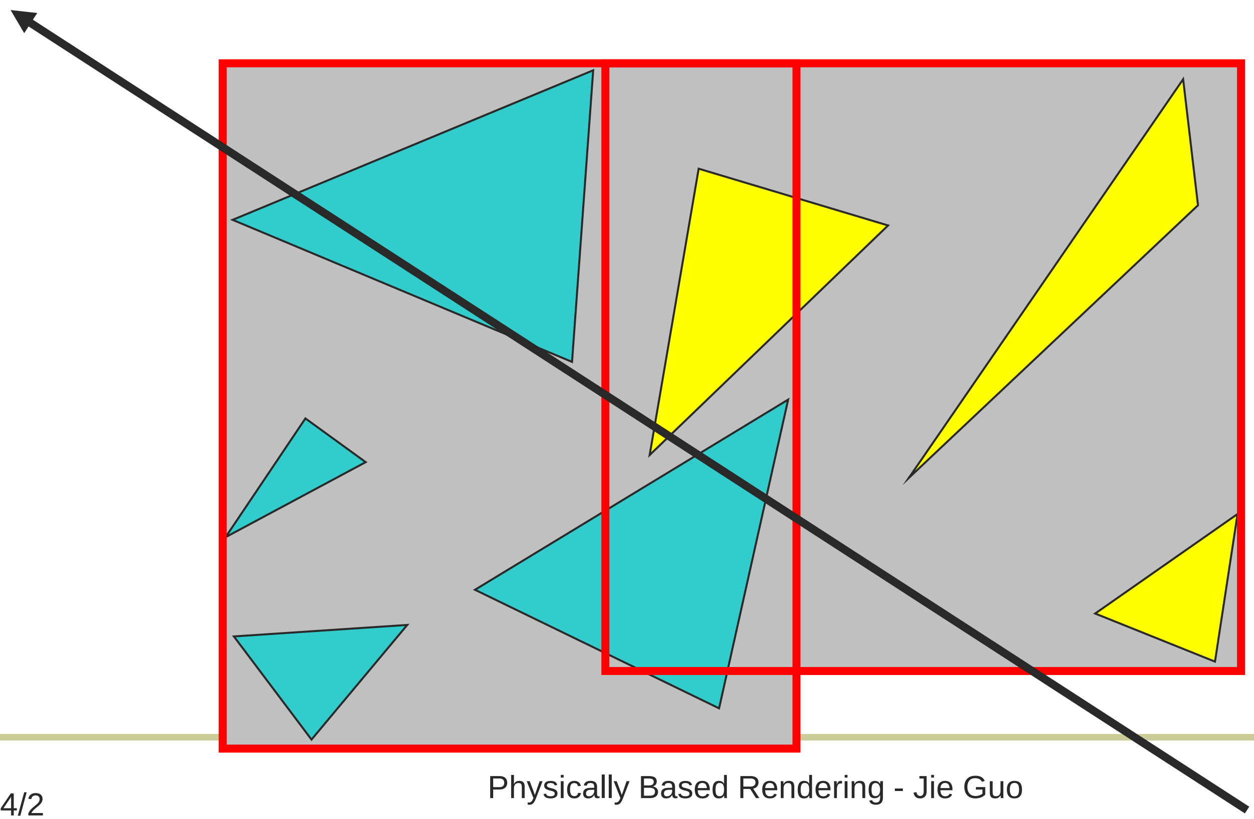
BVH Traversal

- If hit parent, then check all children



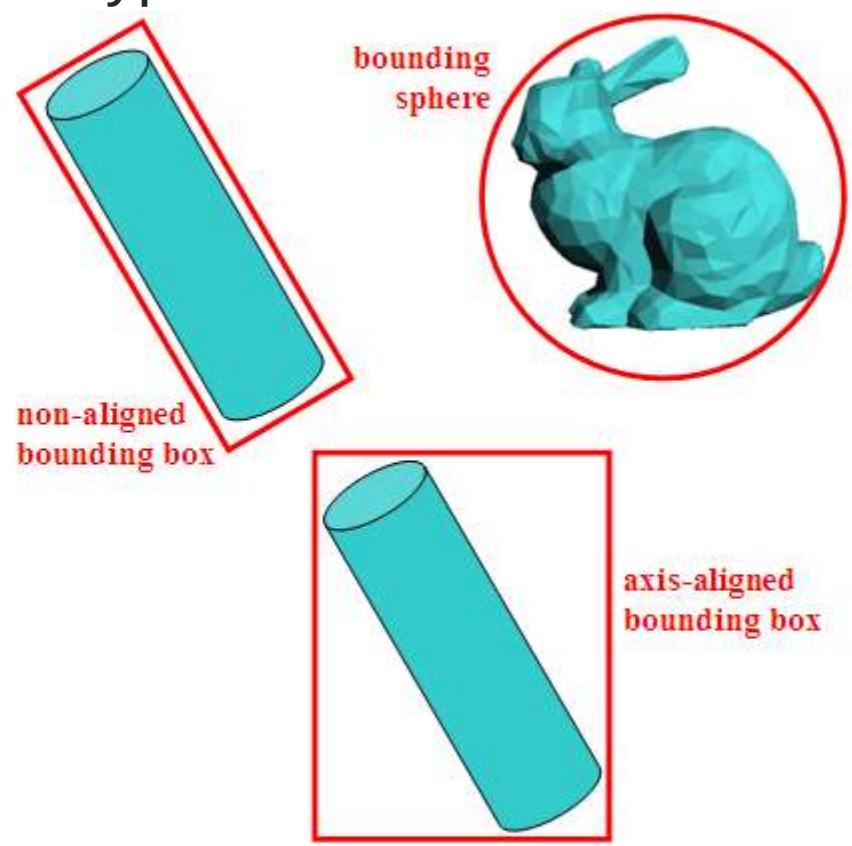
BVH Traversal

- Don't return intersection immediately because the other subvolumes may have a closer intersection



Bounding Volumes (Boxes)

- Wrap things that are hard to test for intersection in things that are easy to test
- Most common bounding volume types:
  - Non-aligned bounding box
  - bounding sphere
  - axis-aligned bounding box (AABB)

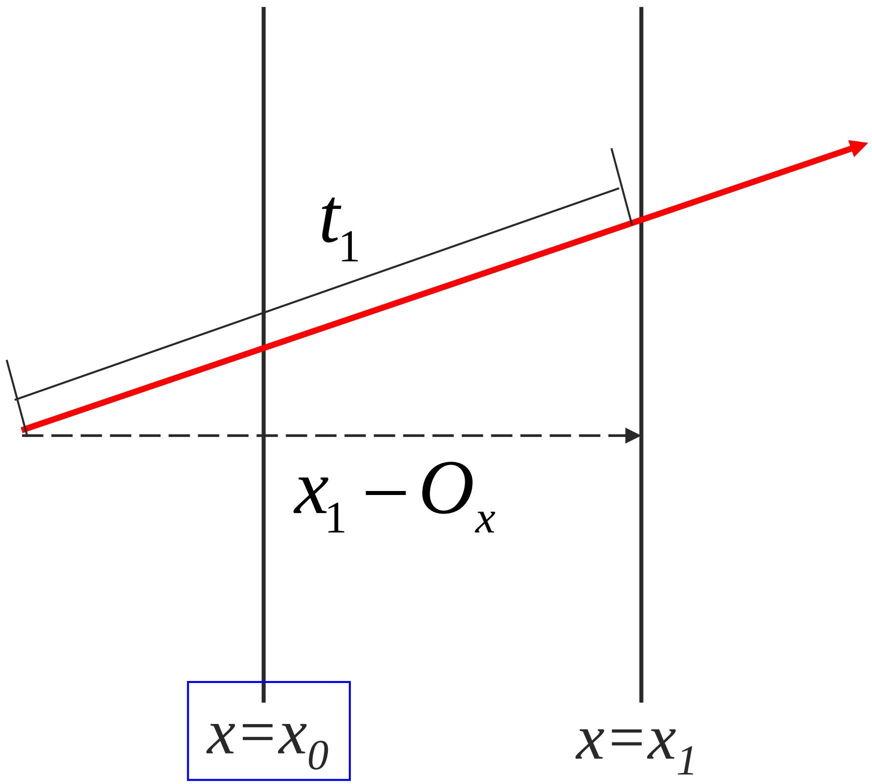


AABB

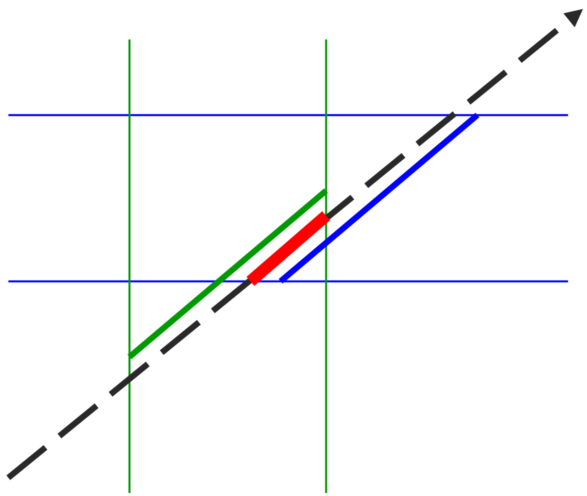
- Axis-aligned bounding box(AABB): the box is parallel to the x,y,z axis of the world coordinate system.
- Very easy to construct.
- Very efficient for intersection test.

AABB-Intersection

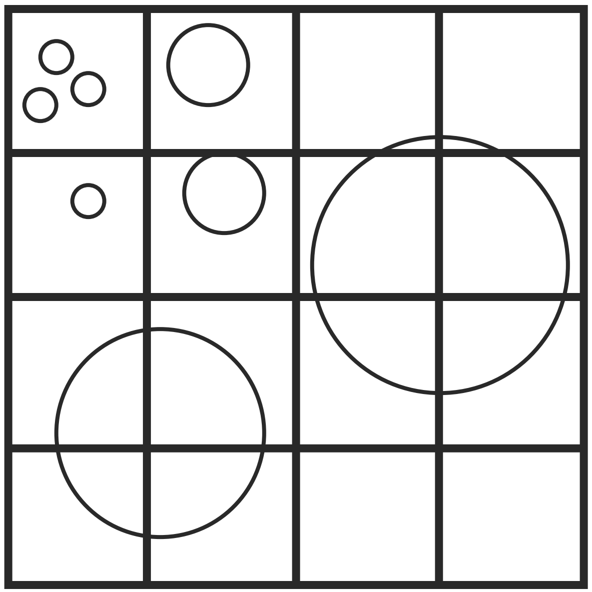
- Key idea: intersection of three slabs



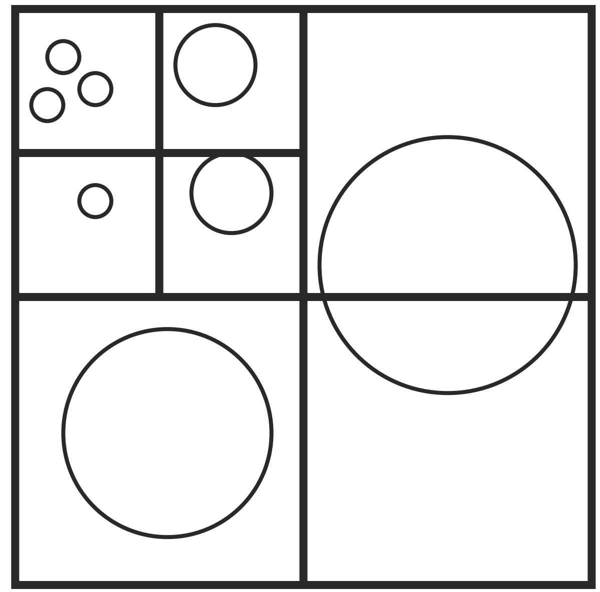
$$t_1 = \frac{x_1 - O_x}{D_x}$$



Space Subdivision

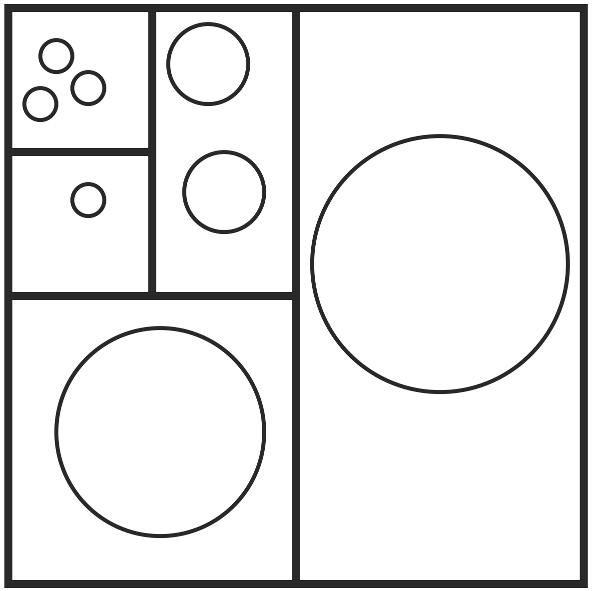


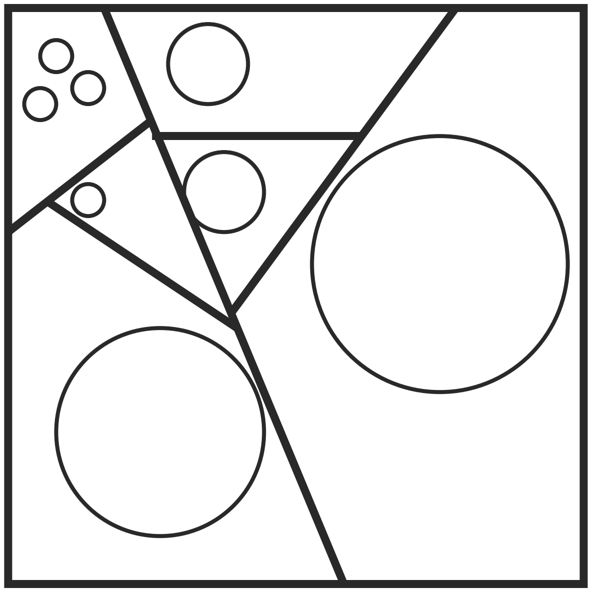
Unifrom grid



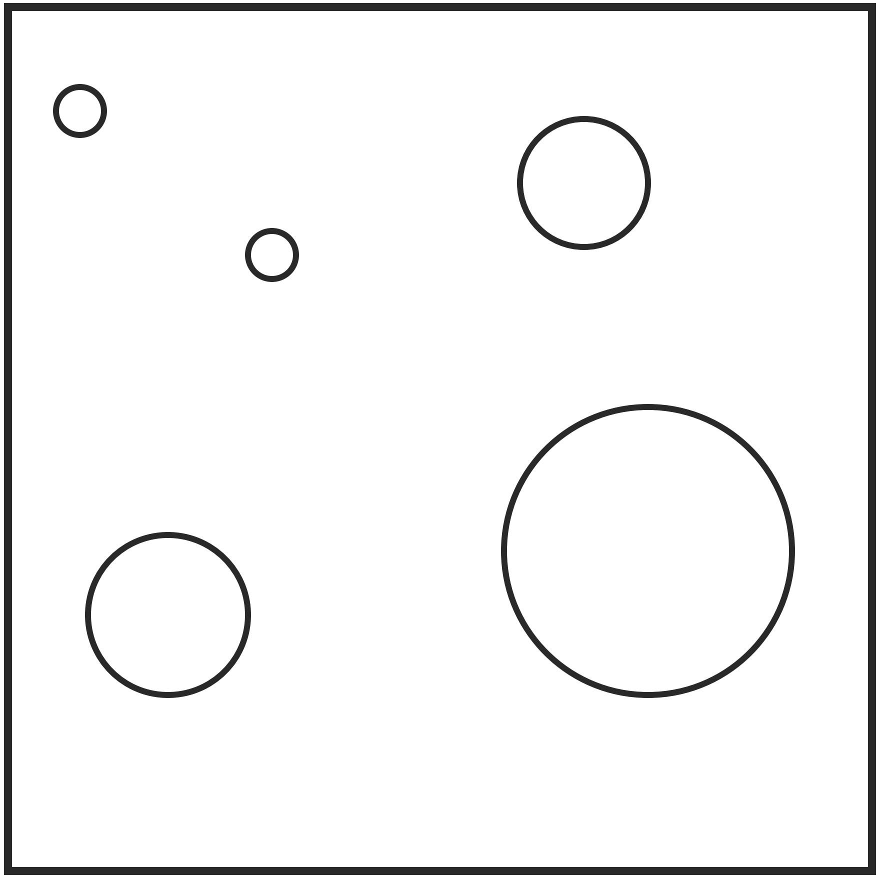
Quadtree (2D)
Octree (3D)

Space Subdivision


KD tree


BSP tree

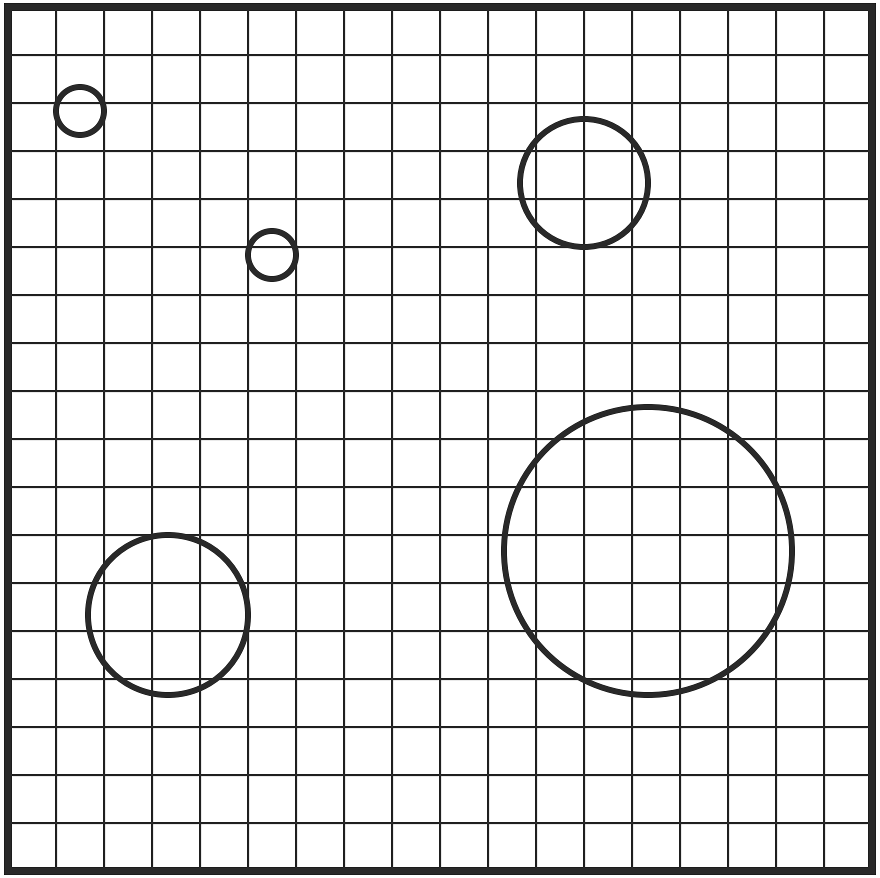
Uniform Grid



Preprocess scene

1. Find bounding box

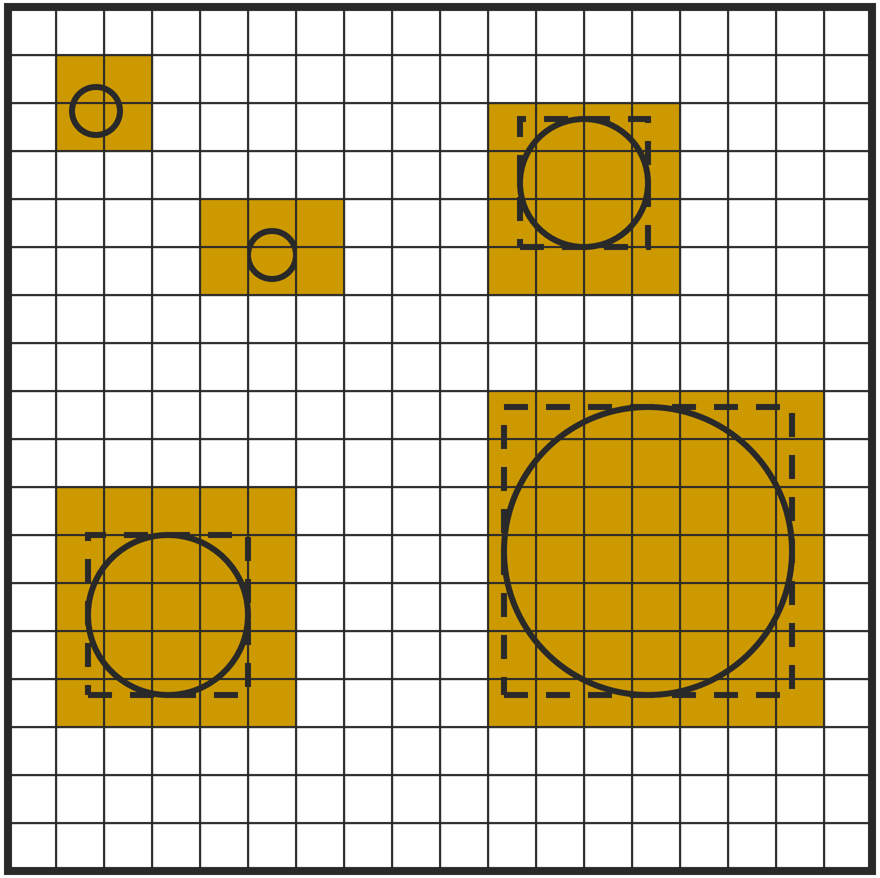
Uniform Grid



Preprocess scene

1. Find bounding box
2. Determine grid resolution

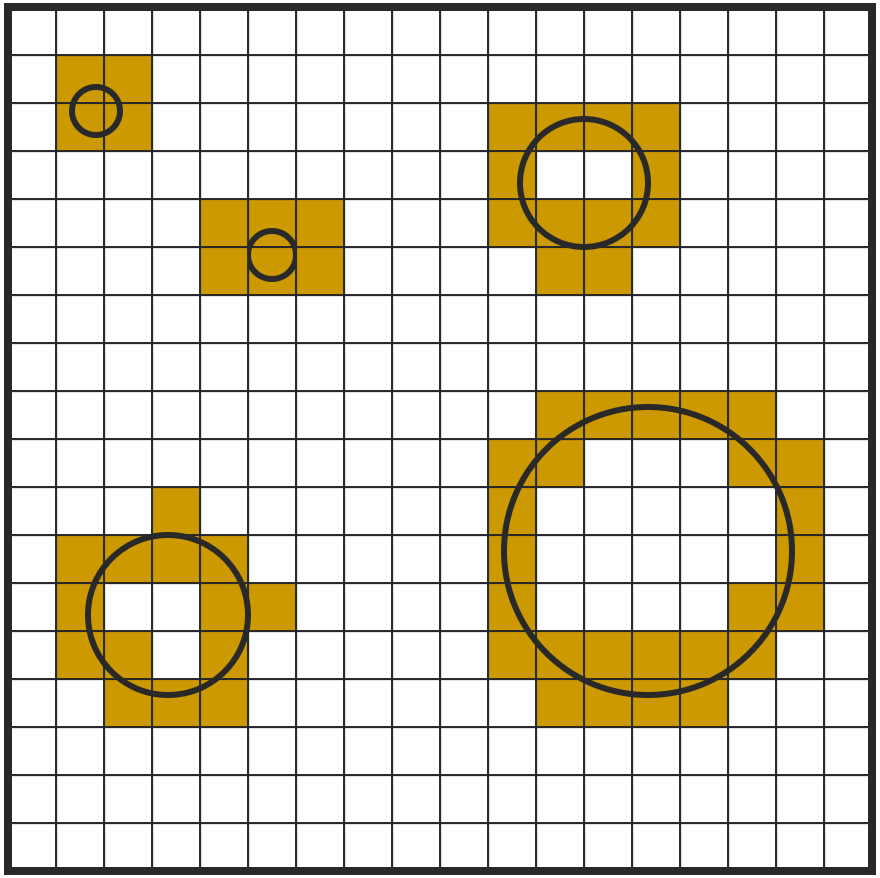
Uniform Grid



Preprocess scene

1. Find bounding box
2. Determine grid resolution
3. Place object in cell if its bounding box overlaps the cell

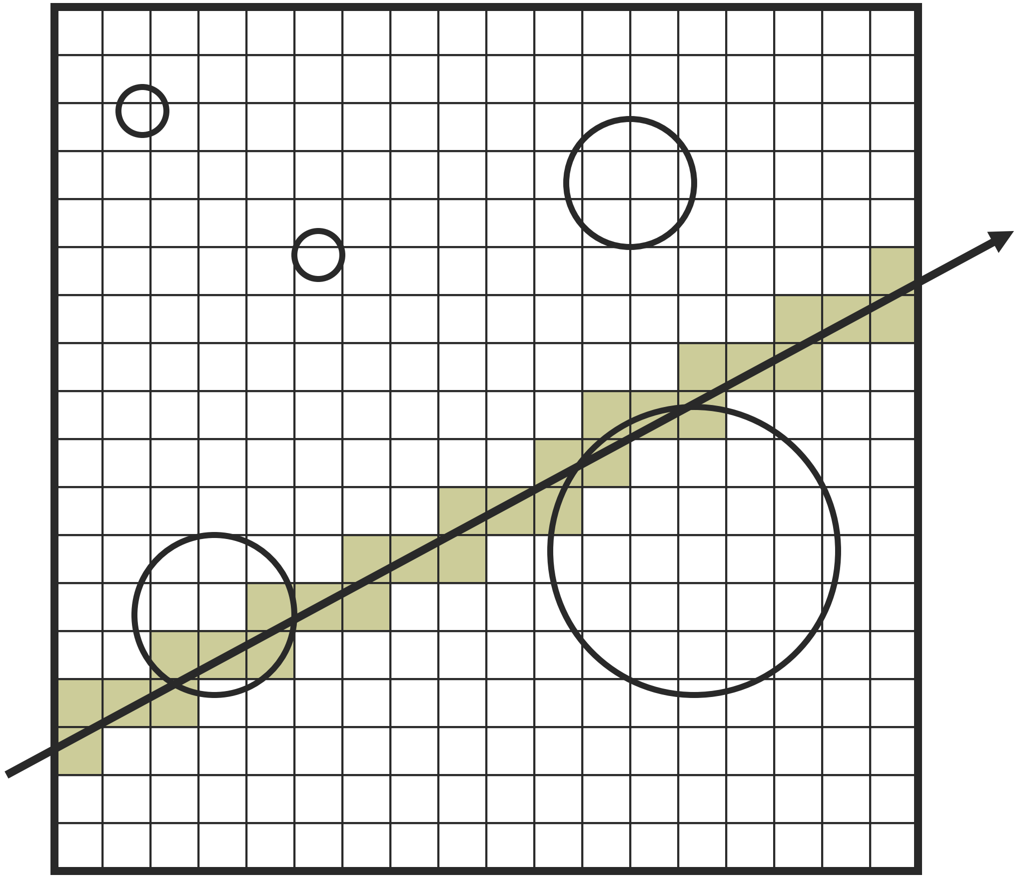
Uniform Grid



Preprocess scene

1. Find bounding box
2. Determine grid resolution
3. Place object in cell if its bounding box overlaps the cell
4. Check that object overlaps cell (expensive!)

Uniform Grid Traversal



Preprocess scene

Traverse grid

3D line = 3D-DDA (Digital Differential Analyzer)

$$y = mx + b \qquad m = \frac{y_2 - y_1}{x_2 - x_1}$$

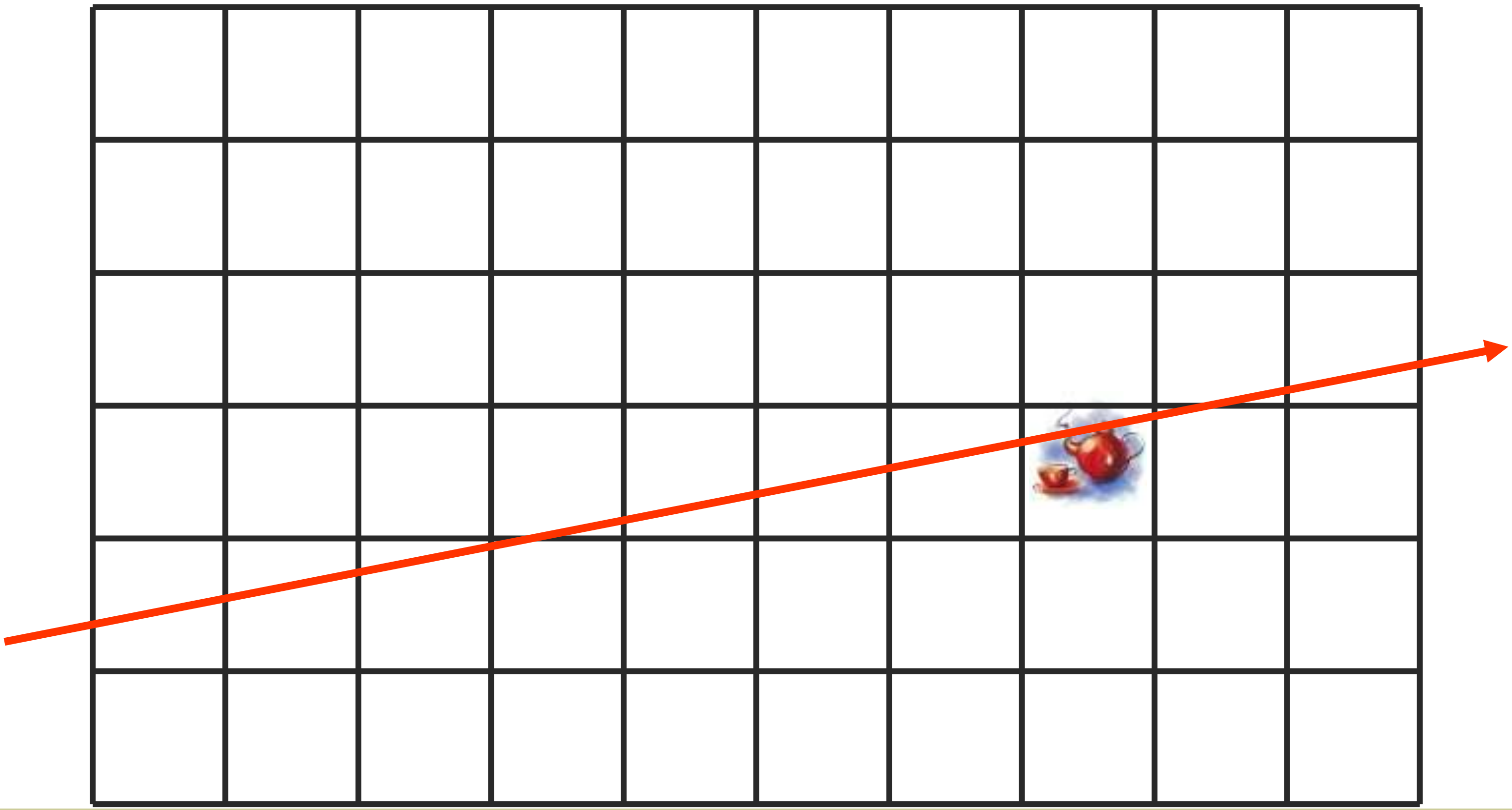
$$x_{i+1} = x_i + 1$$

$$y_{i+1} = mx_{i+1} + b$$ naive

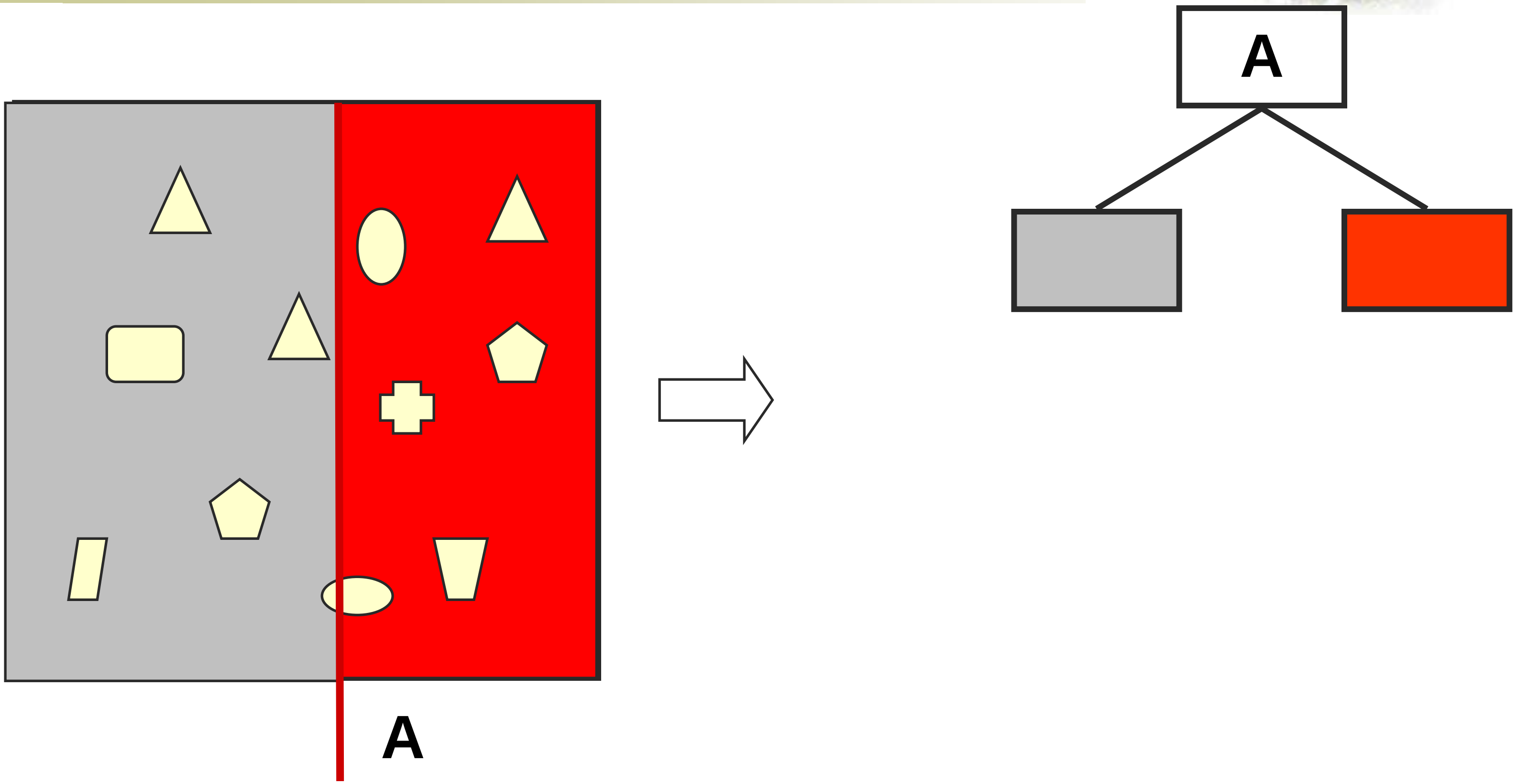
$$y_{i+1} = y_i + m$$ DDA

Teapot in a stadium problem

- Not adaptive to distribution of primitives

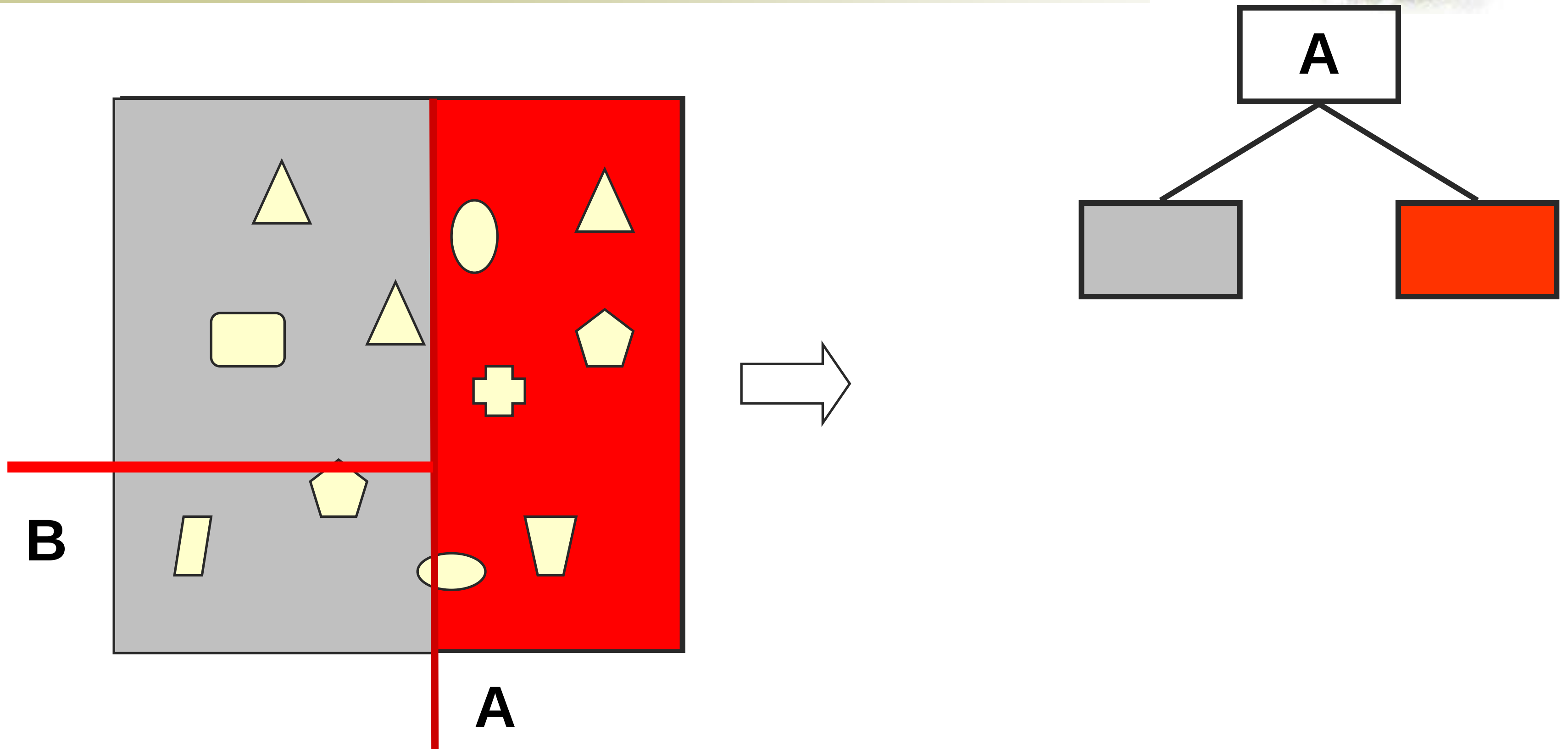


Kd-Tree

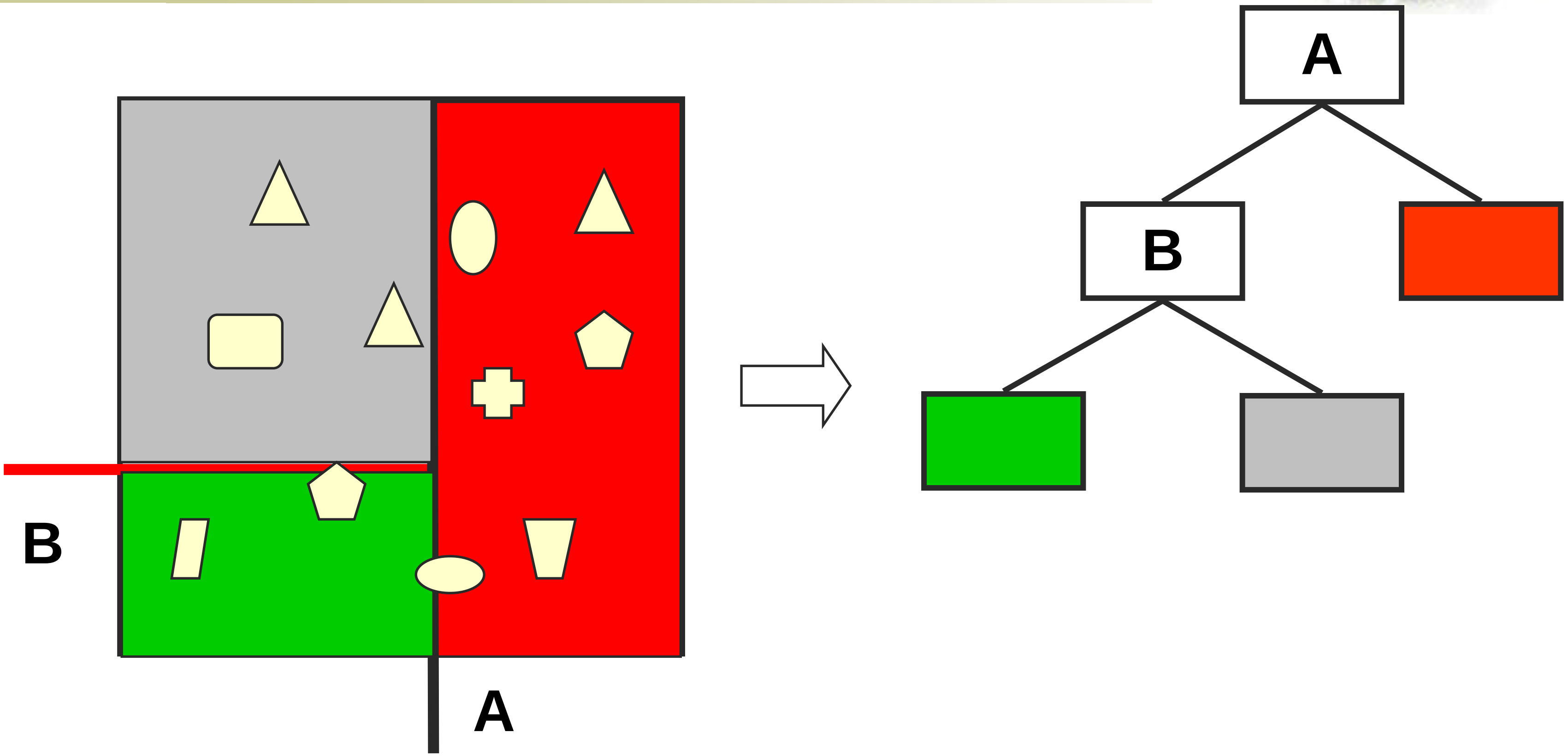


Leaf nodes correspond to unique regions in space

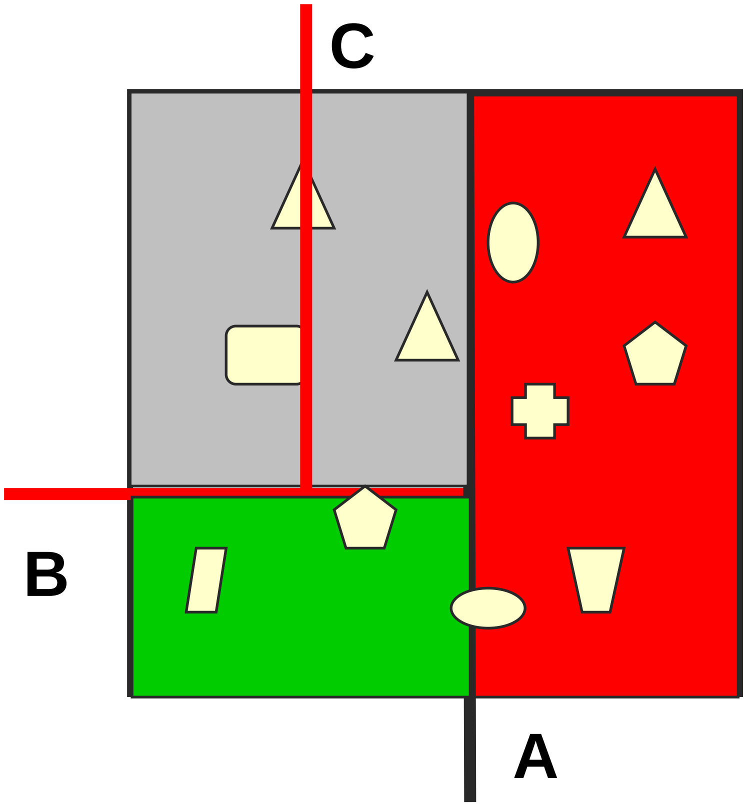
Kd-Tree

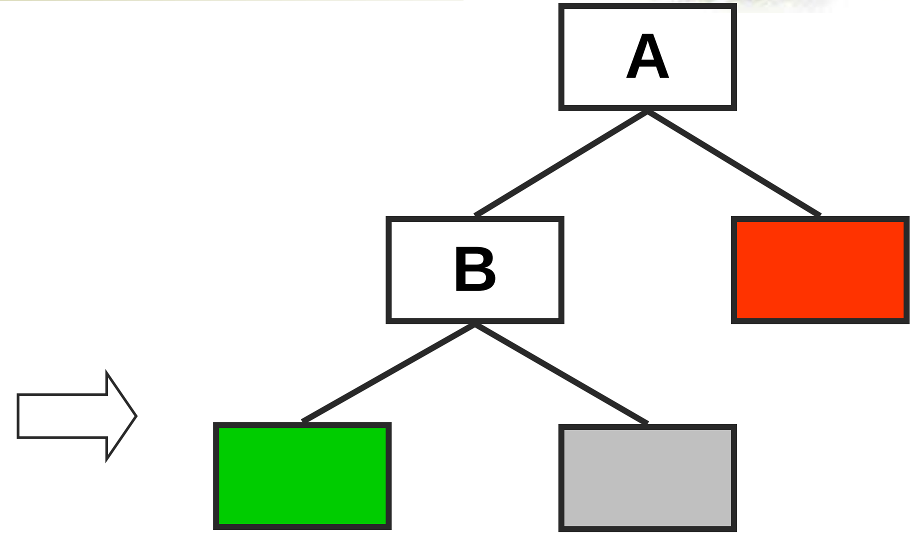


Kd-Tree

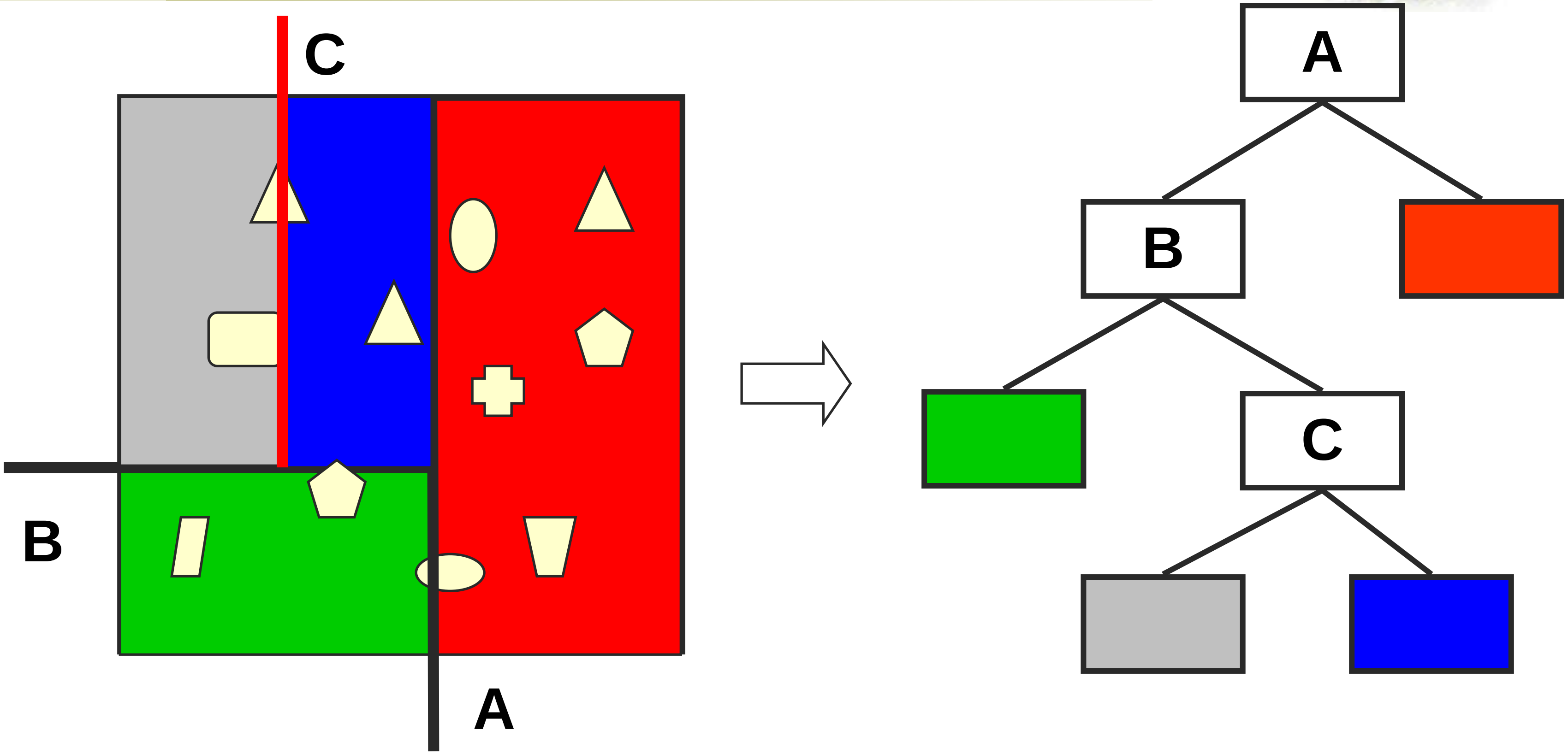


Kd-Tree

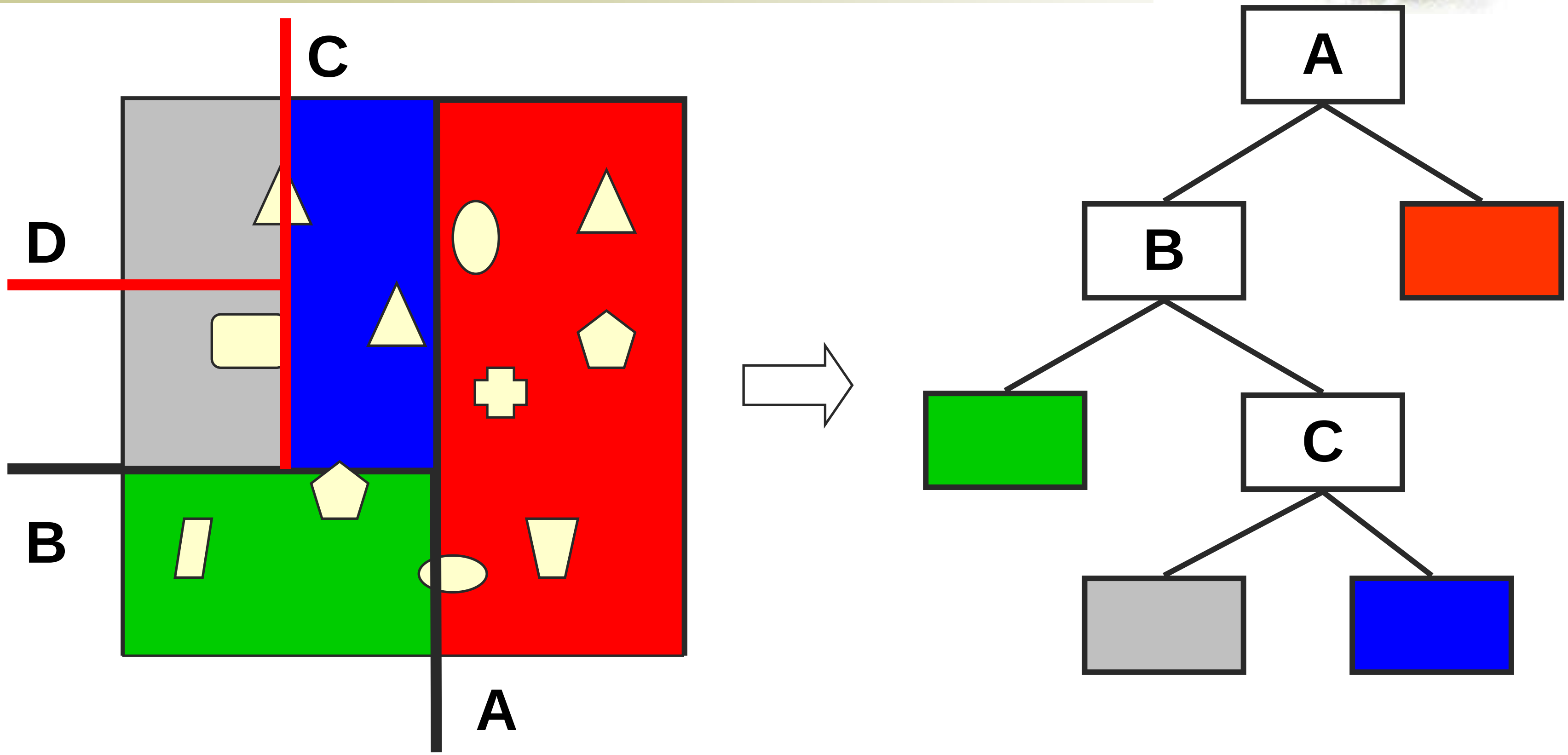




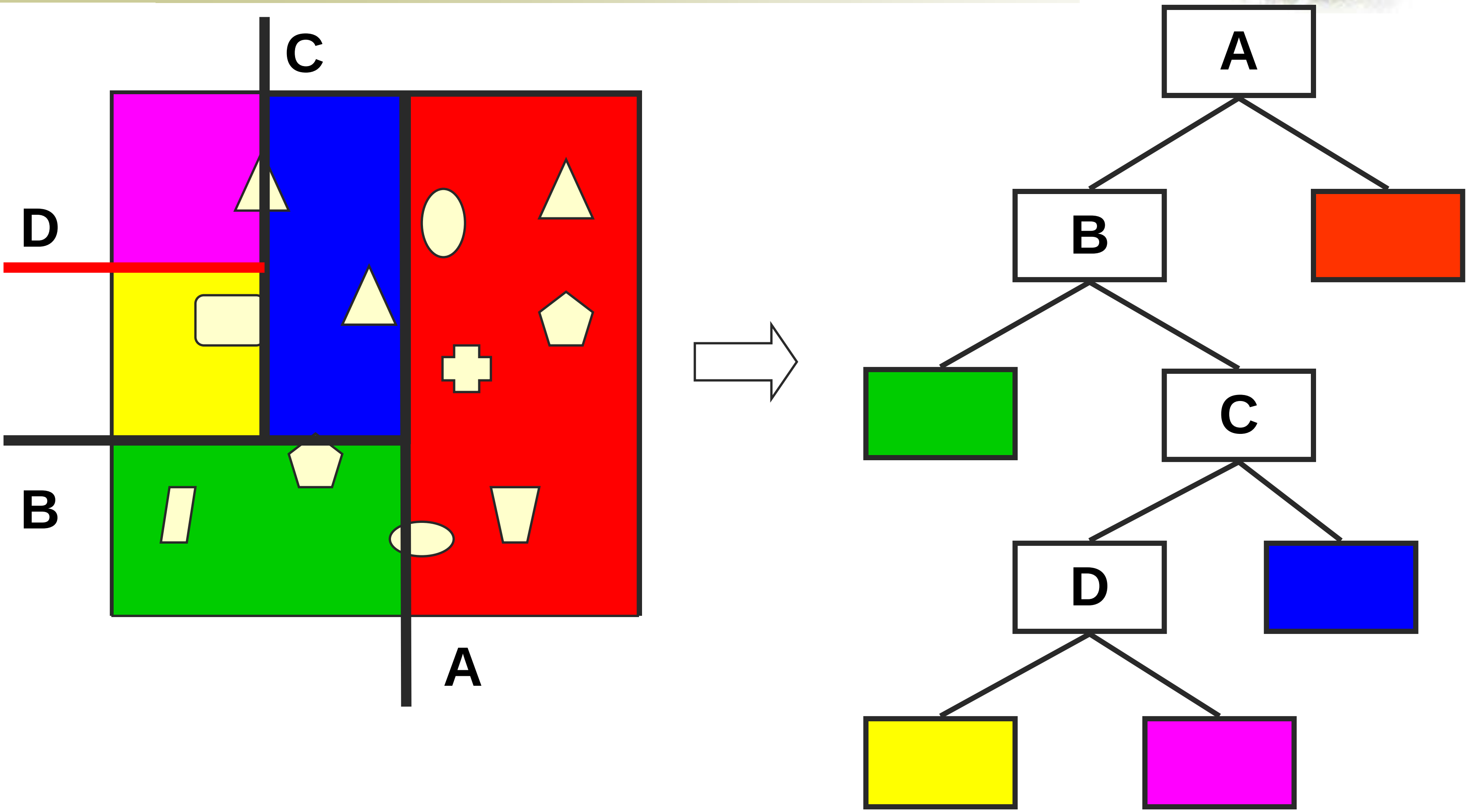
Kd-Tree



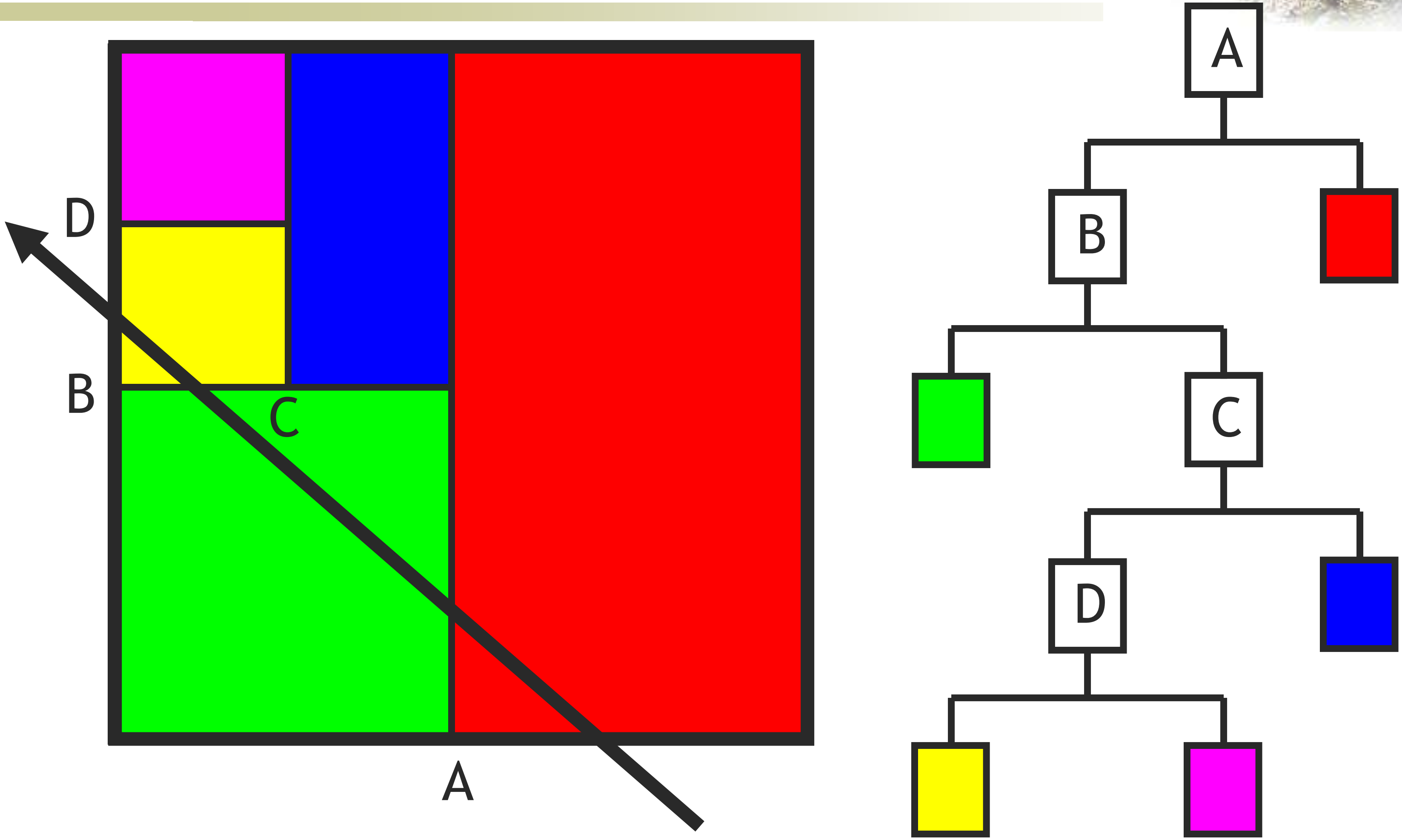
Kd-Tree



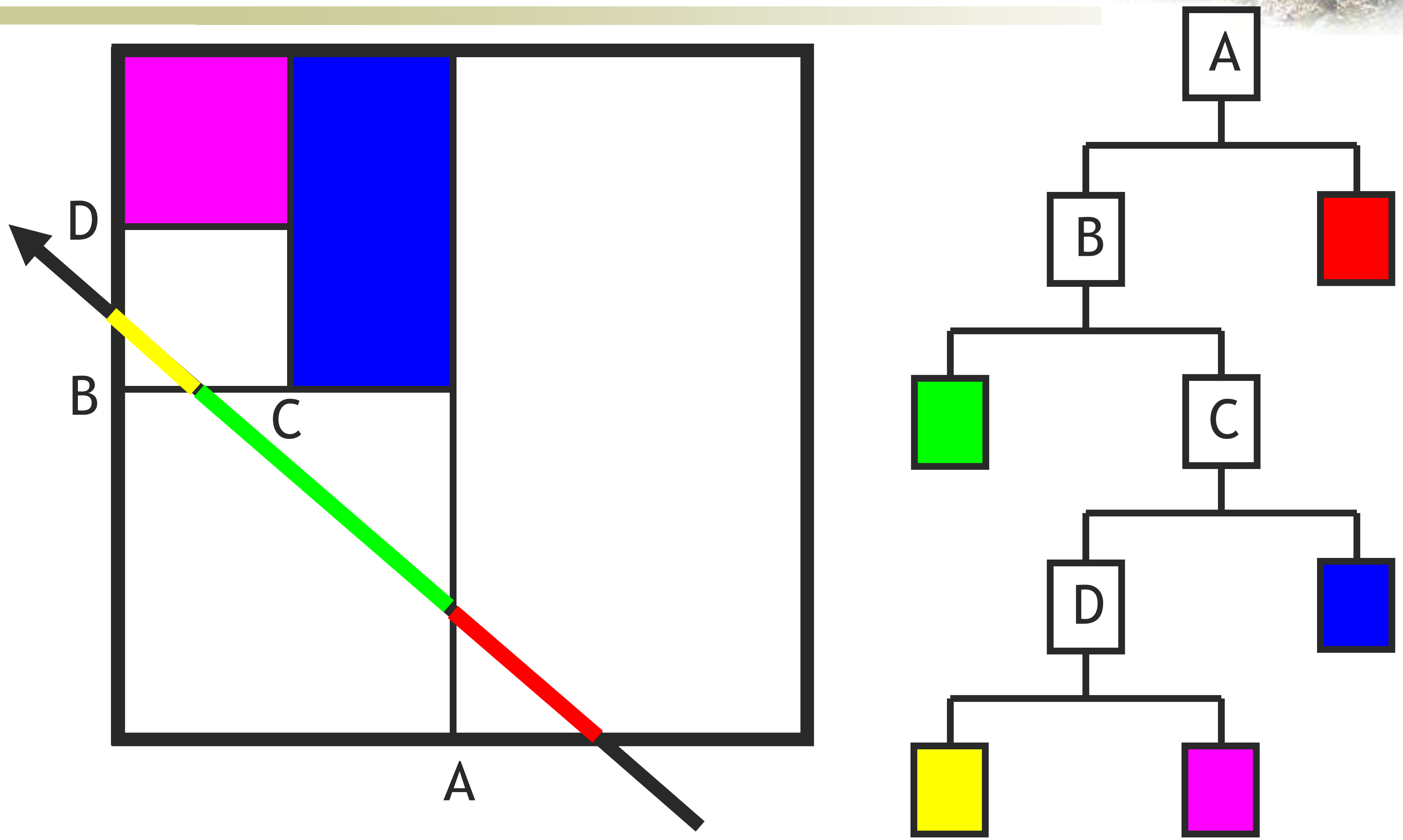
Kd-Tree



Kd-Tree Traversal



Kd-Tree Traversal



Kd-Tree Construction

1. Pick an axis, or optimize across all three
2. Build a set of candidate split planes (cost extrema must be at bbox vertices)
3. Sort or bin the triangles
4. Sweep to incrementally track L/R counts, cost
5. Output position of minimum cost split

Running time: $T(N) = N \log N + 2T(N/2)$

$$T(N) = N \log^2 N$$

- Characteristics of highly optimized tree
  - very deep, very small leaves, big empty cells

Kd-Tree Construction

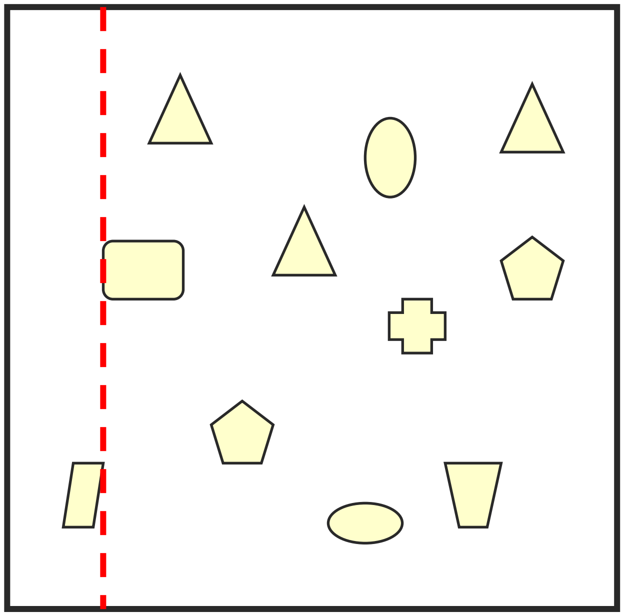
- Three key issues:
  - Choosing split axis
  - Determining split location
  - Termination

Kd-Tree Construction

- Choosing splitting axis:
  - Round-robin
  - Largest extent

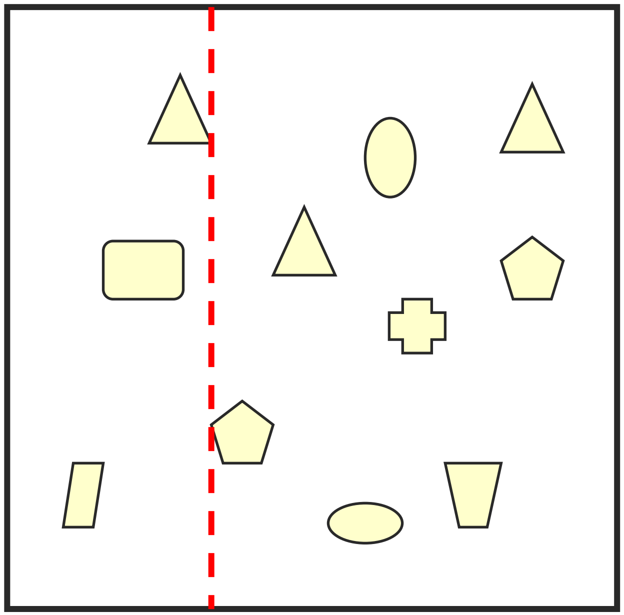
Kd-Tree Construction

- where to put the splitting planes?



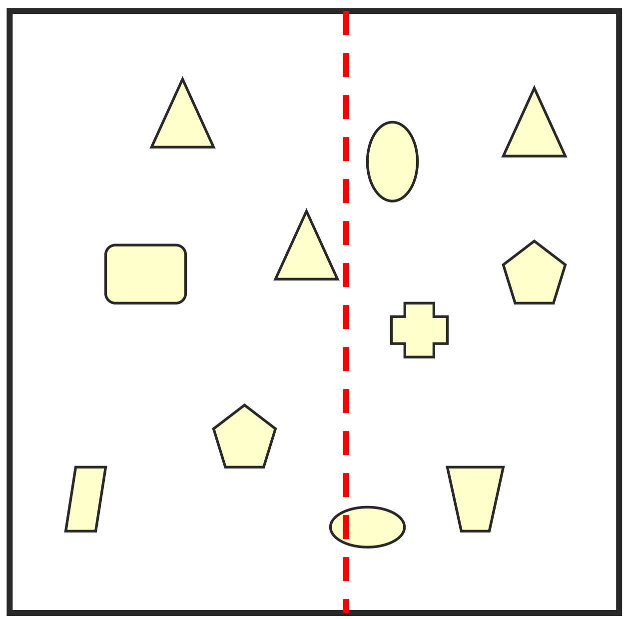
Kd-Tree Construction

- where to put the splitting planes?



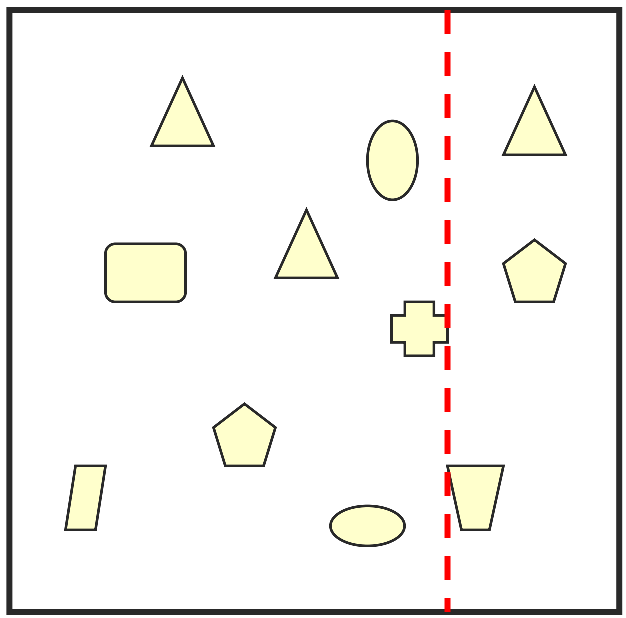
Kd-Tree Construction

- where to put the splitting planes?



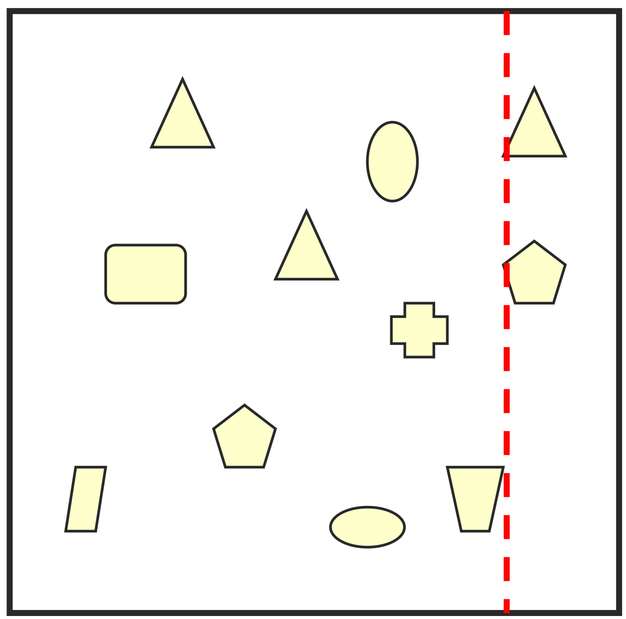
Kd-Tree Construction

- where to put the splitting planes?



Kd-Tree Construction

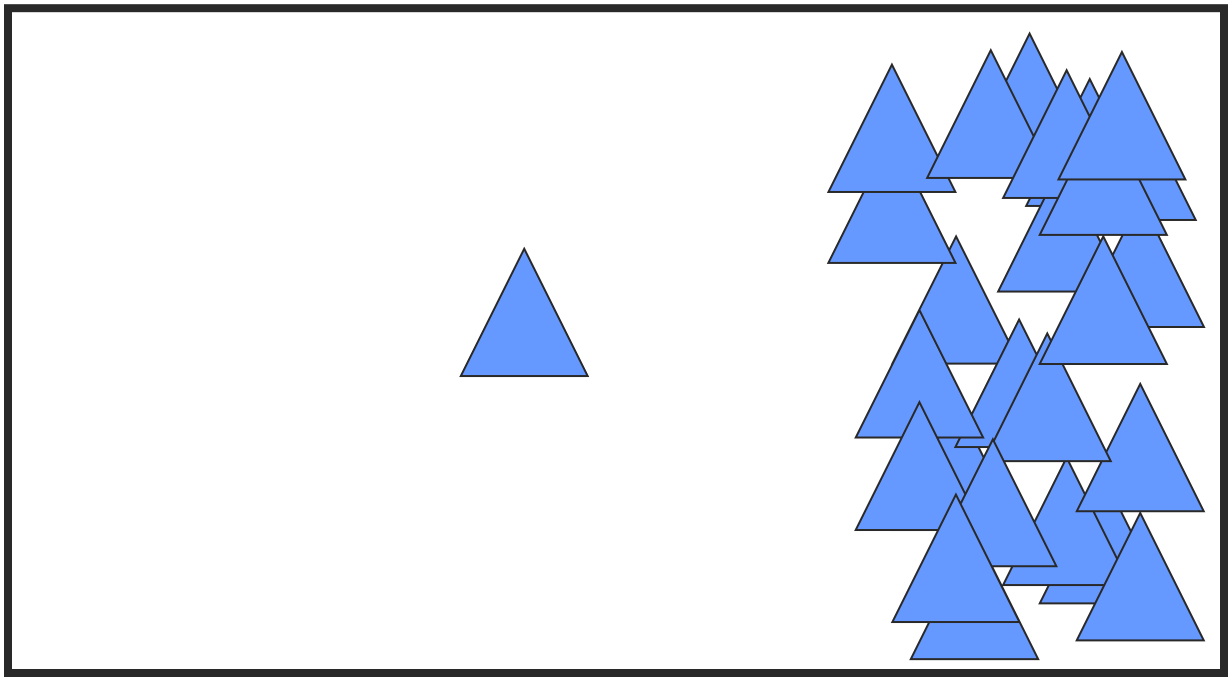
- where to put the splitting planes?



Kd-Tree Construction

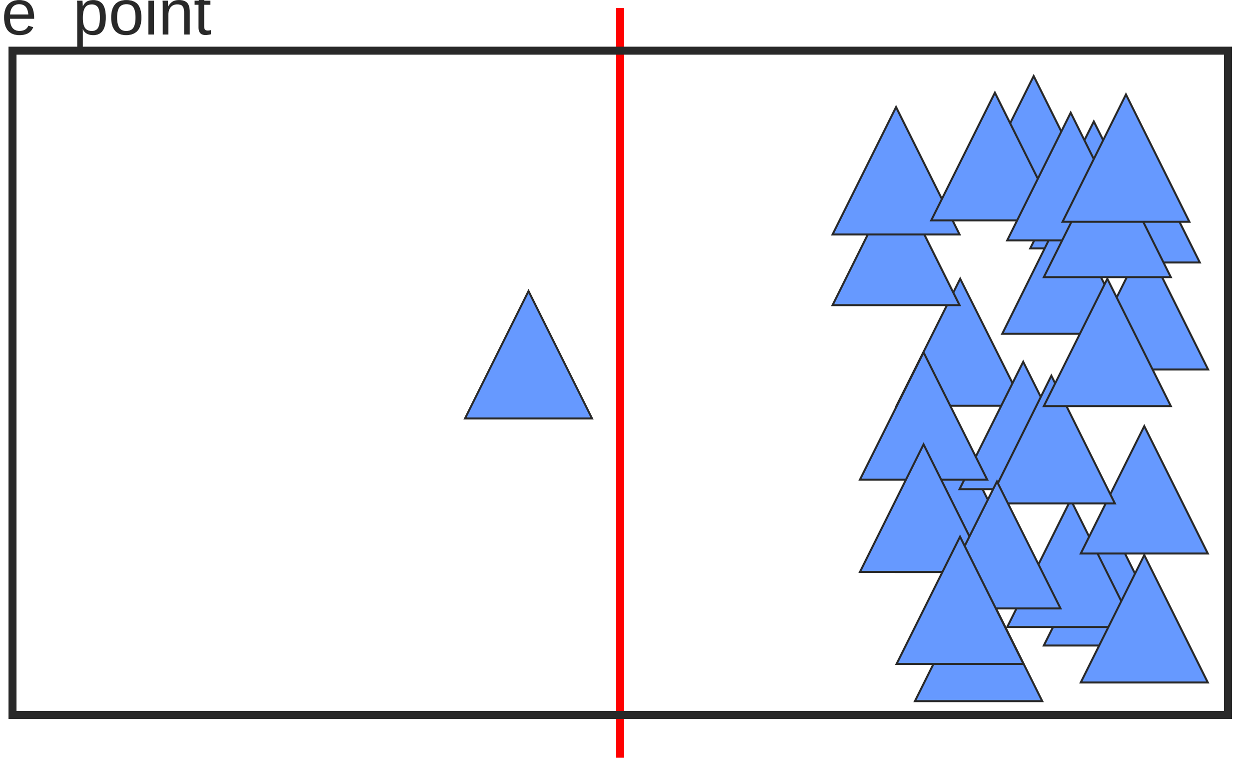
- Middle point
- Medium
- Surface area heuristic

Kd-Tree Construction



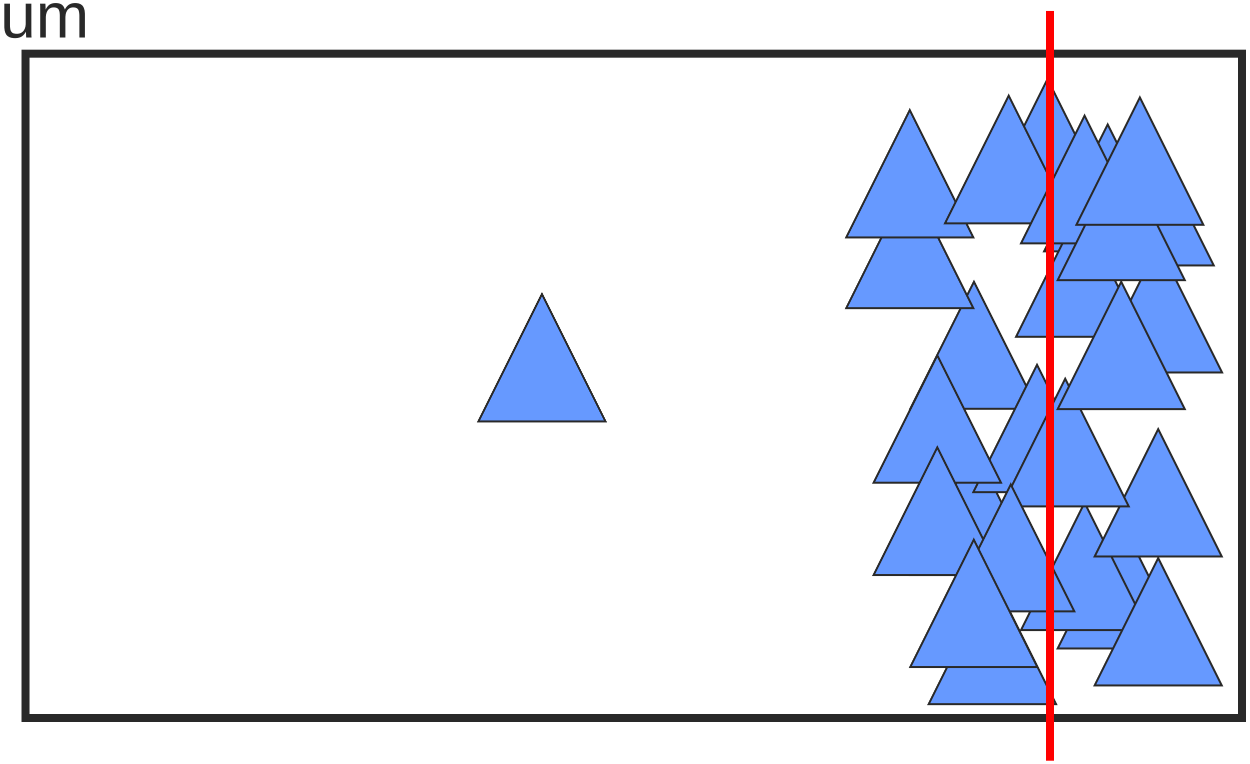
Kd-Tree Construction

- Middle point



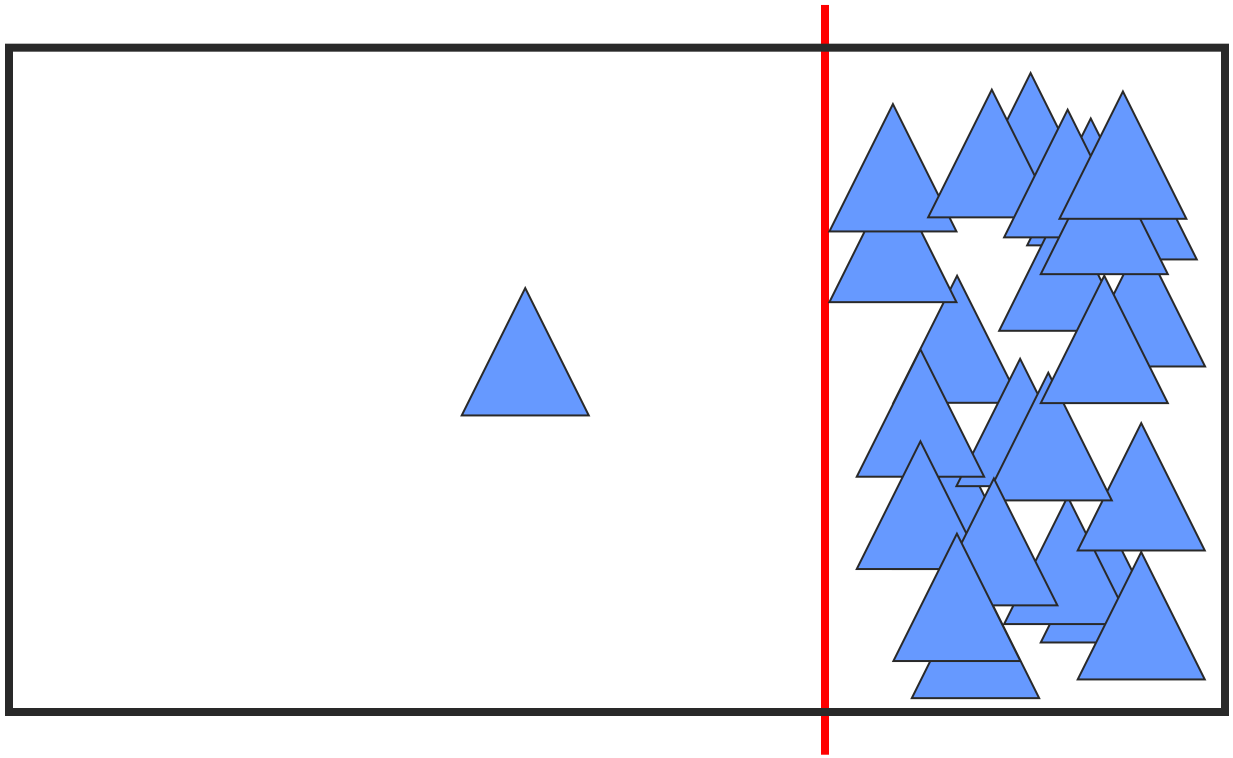
Kd-Tree Construction

- Medium



Kd-Tree Construction

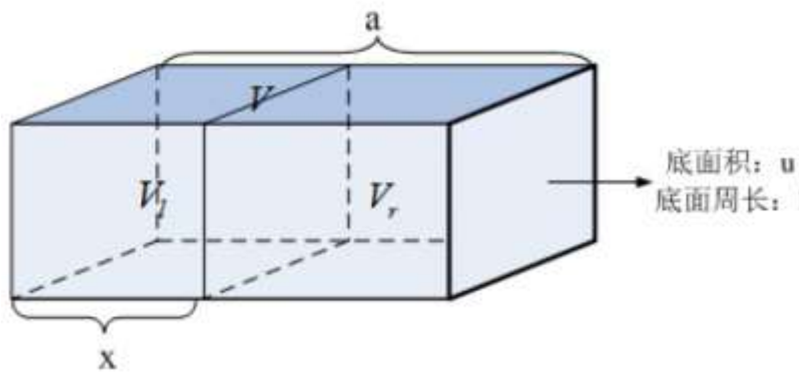
- SAH



Surface Area Heuristic

- Surface Area Heuristic：使用最为广泛的评估候选分割平面代价的评估函数
- 基于几何概率理论
- 三项假设条件：
  - #1：场景中的光线是随机均匀分布的直线；
  - #2：节点的遍历代价Kt和直线与三角形的相交测试代价Ki已知；
  - #3：与N个三角形相交的代价为NKi，即叶节点的相交测试代价与其包含的三角形数目成正比。

Surface Area Heuristic



击中左右子节点的条件概率分别为：

$$P_l = \frac{SA(V_l)}{SA(V)}$$

$$P_r = \frac{SA(V_r)}{SA(V)}$$

根据#2，某个节点的总的代价为：

$$C(V) = K_t + P_l \cdot C(V_l) + P_r \cdot C(V_r)$$

Surface Area Heuristic

$$C(V) = K_t + P_l \cdot C(V_l) + P_r \cdot C(V_r)$$

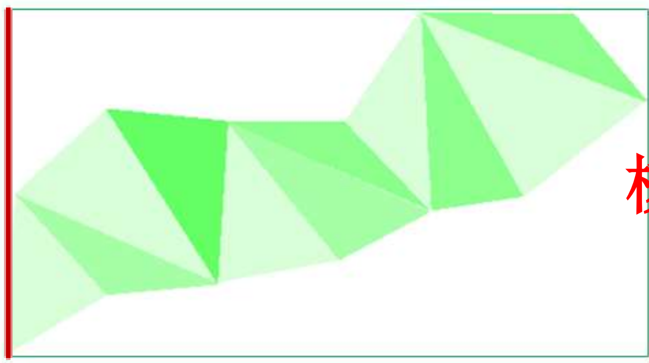
展开上式可得根节点的总代价为:

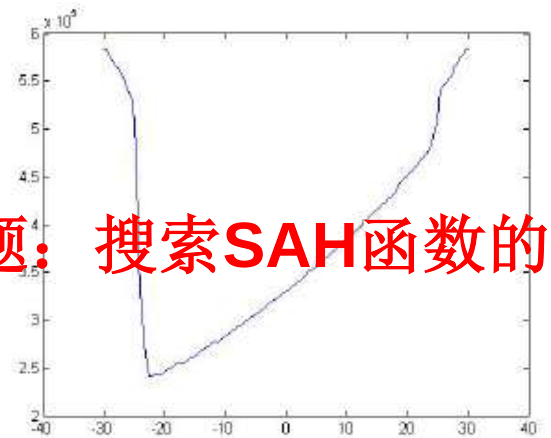
$$C(V_{root}) = \sum_{n \in internodes} \frac{SA(V_n)}{SA(V_{root})} K_t + \sum_{l \in leafnodes} \frac{SA(V_l)}{SA(V_{root})} K_i$$

使用局部贪心算法简化:

$$\begin{aligned}\tilde{C}(V) &\approx K_t + P_l \cdot |T_l| \cdot K_i + P_r \cdot |T_r| \cdot K_i \\ &= \boxed{K_t + K_i \left(\frac{SA(V_l)}{SA(V)} |T_l| + \frac{SA(V_r)}{SA(V)} |T_r| \right)}\end{aligned}$$

Surface Area Heuristic

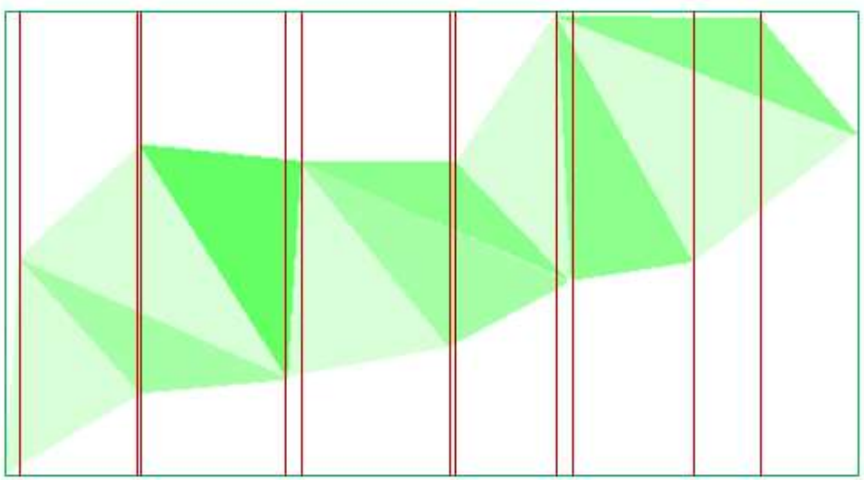


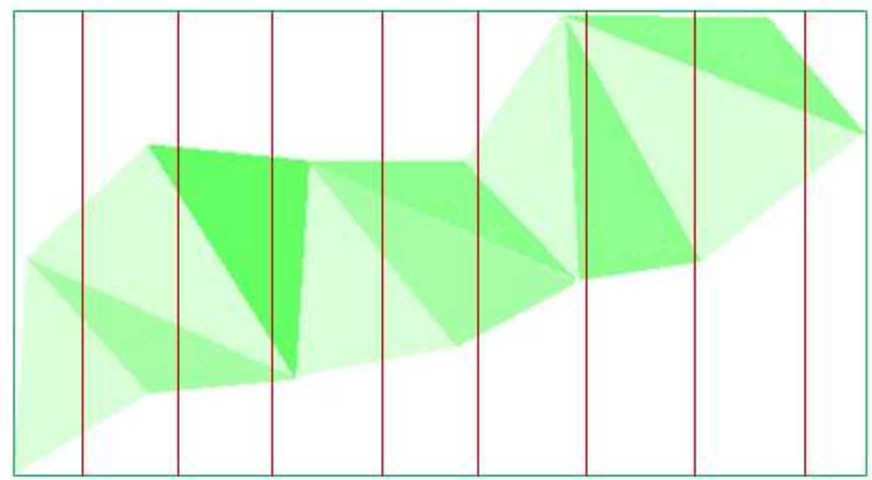


核心问题：搜索SAH函数的最小值

Kd-Tree Construction

- 基于三角形排序的算法：[*Wald et al. 2006*], ···
- 基于BIN的算法：[*Hurley et al. 2002*], ···
- 并行算法：[*Shevtsov et al. 2007*], [*Hunt et al. 2006*], [*Zhou et al. 2008*], [*Choi et al. 2010*], ···



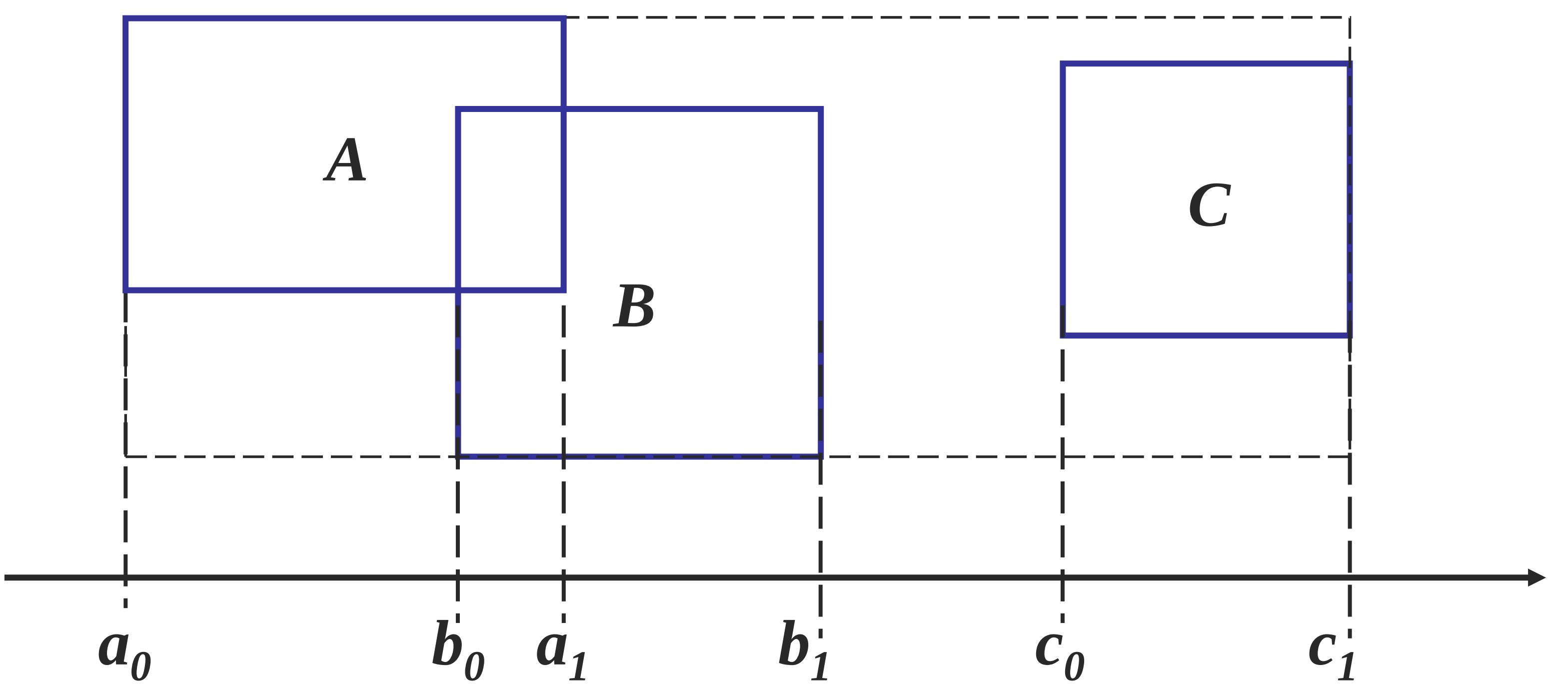


Kd-Tree Construction

- 其中并行化构造算法是趋势，包括multi-core CPU和many-core GPU。
- 对于BVH，情况类似。

Choose Split Planes By Sorting

Start from the axis with maximum extent, sort all edge events and process them in order



Kd-Tree Construction

- What about time complexity?

Termination Criteria

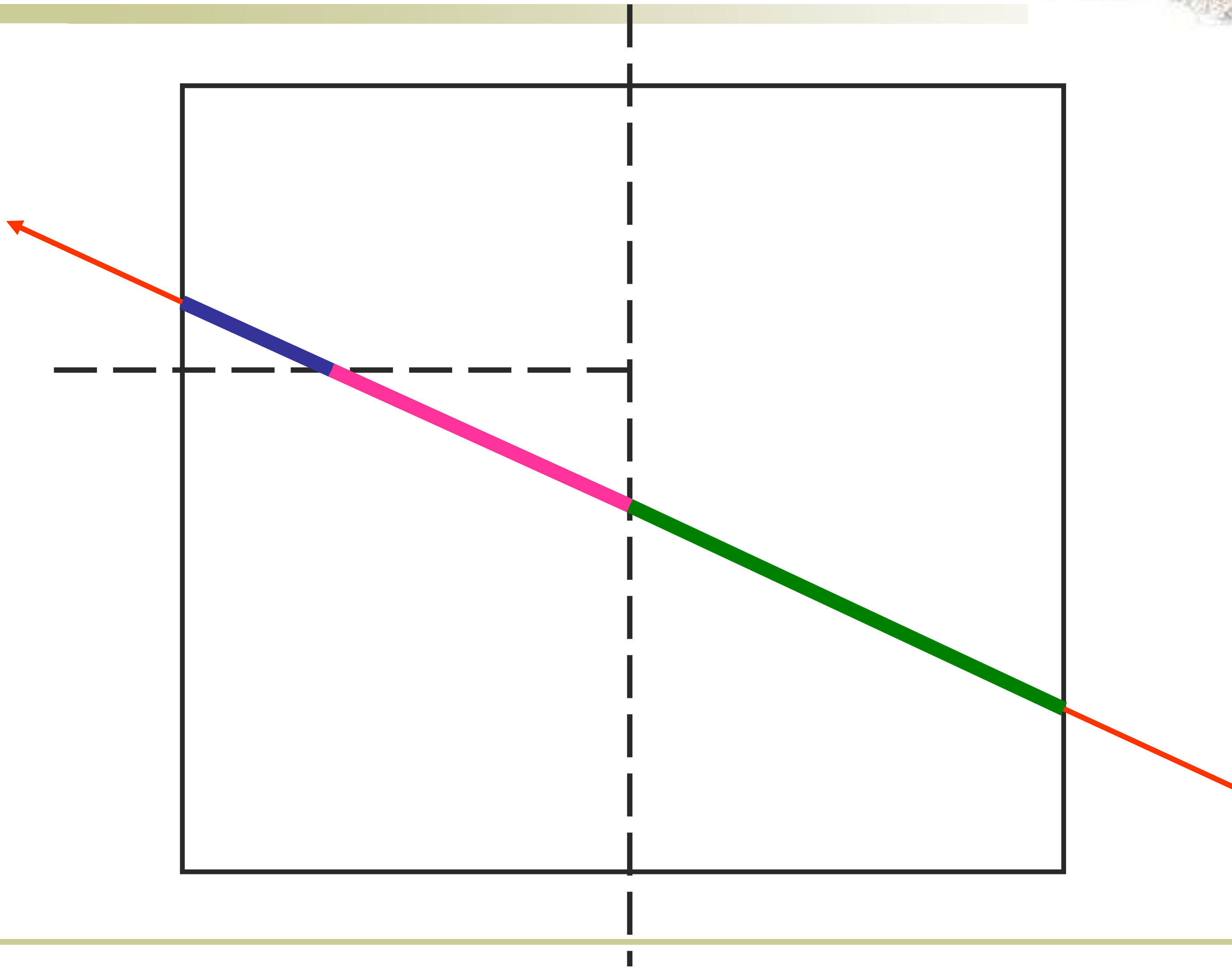
- When should we stop splitting?
  - Bad: depth limit, number of triangles
  - Good: when split does not help any more.
- Threshold of cost improvement
  - Stretch over multiple levels
  - For example, if cost does not go down after three splits in a row, terminate
- Threshold of cell size
  - Absolute probability SA(node)/SA(scene) small

Tree Representation

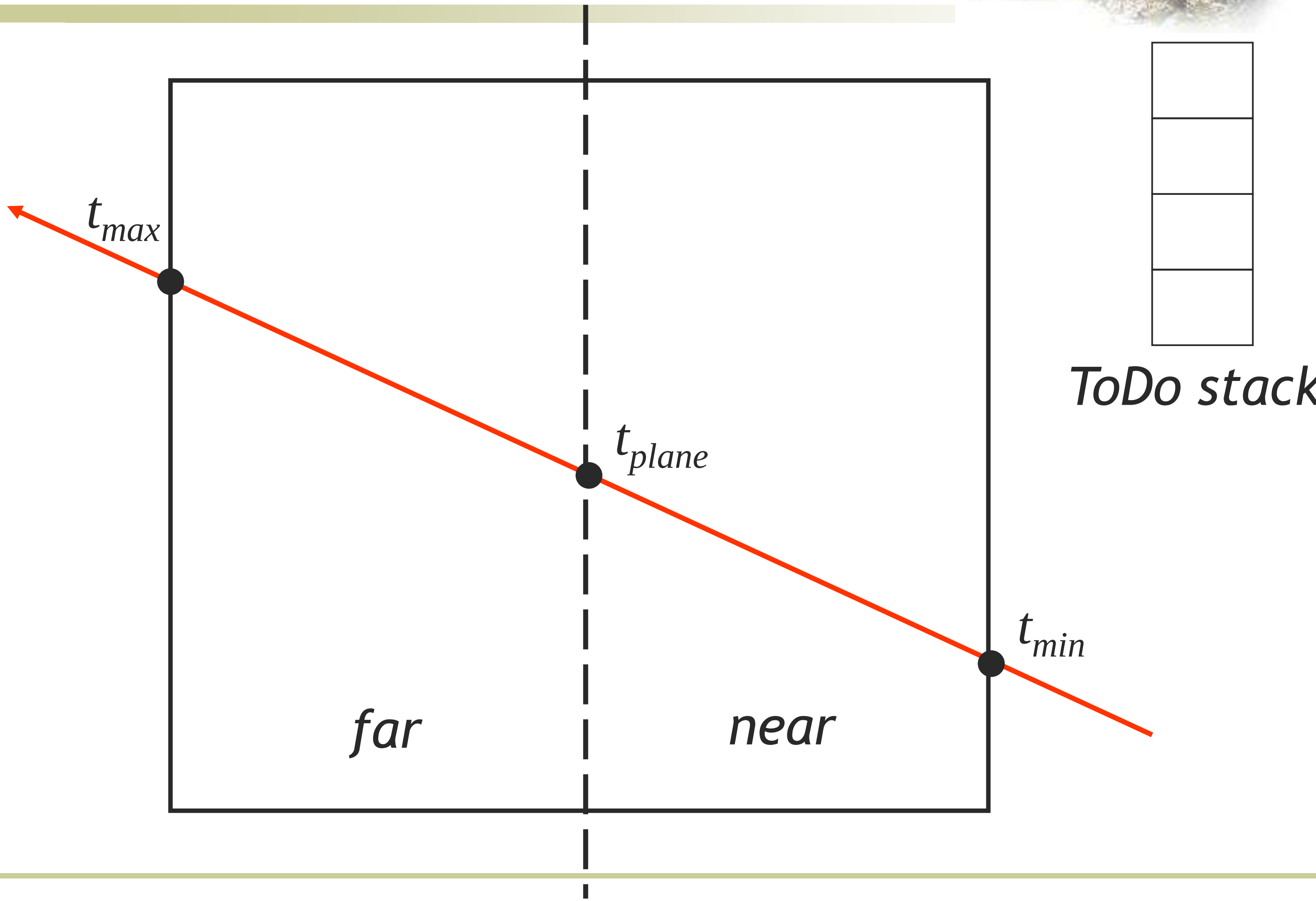
8-byte (reduced from 16-byte, 20% gain)

```
struct KdAccelNode {
  ...
  union {
    float split;           // Interior
    int onePrimitive;      // Leaf
    int *primitives;       // Leaf
  };
  union {
    int flags;              // Both
    int nPrims;             // Leaf
    int aboveChild;         // Interior
  };
}
```

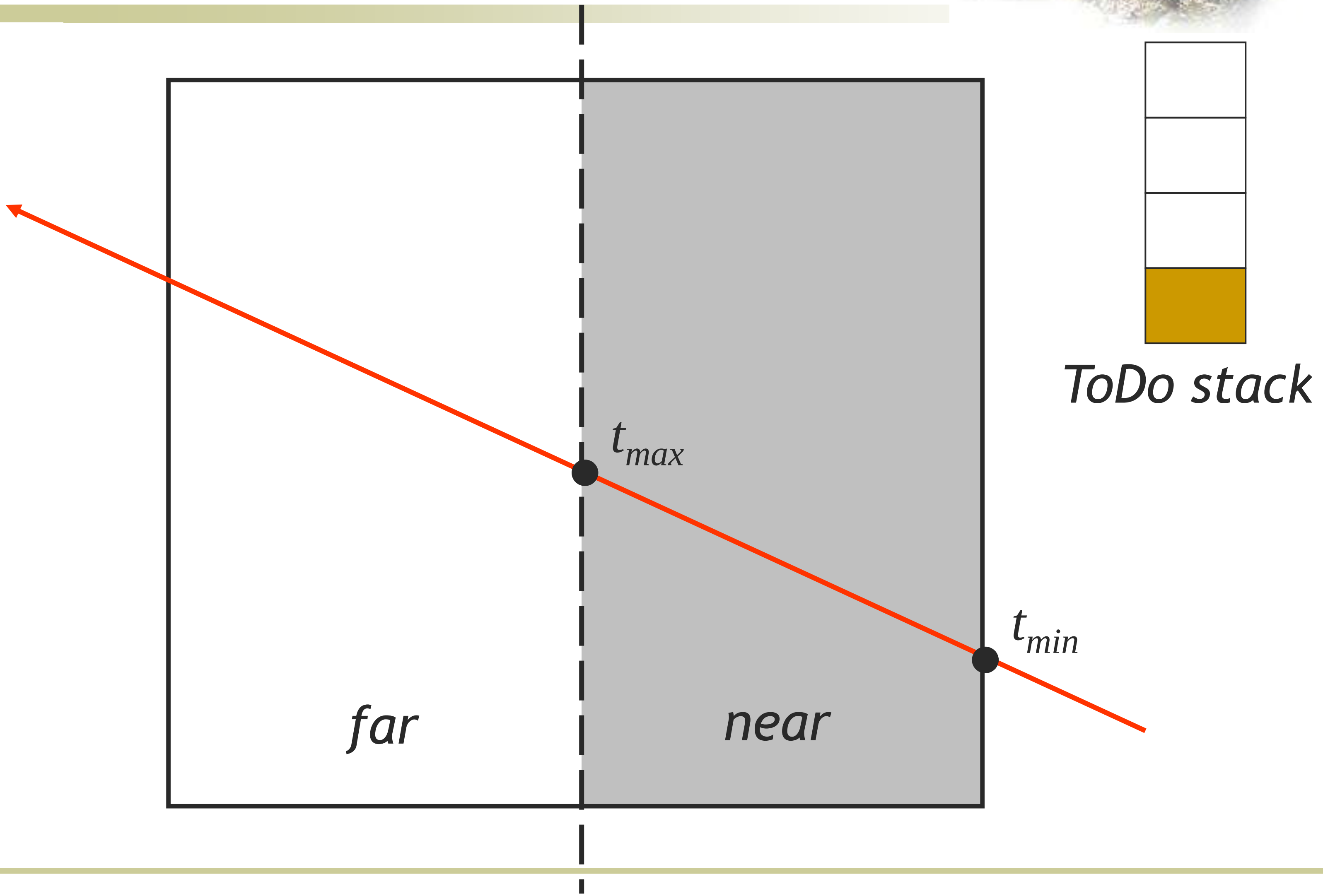
Kd-Tree Traversal



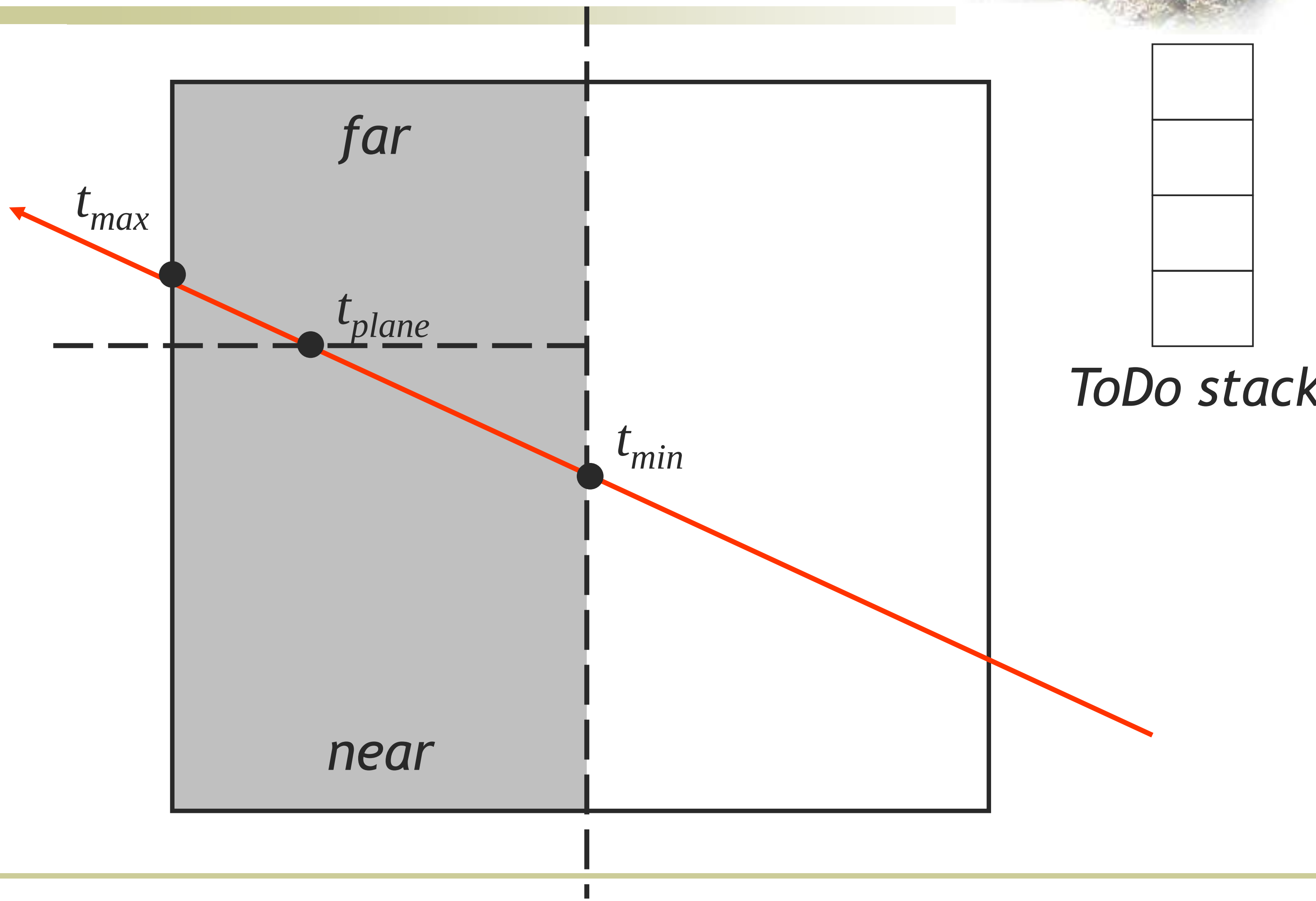
Kd-Tree Traversal



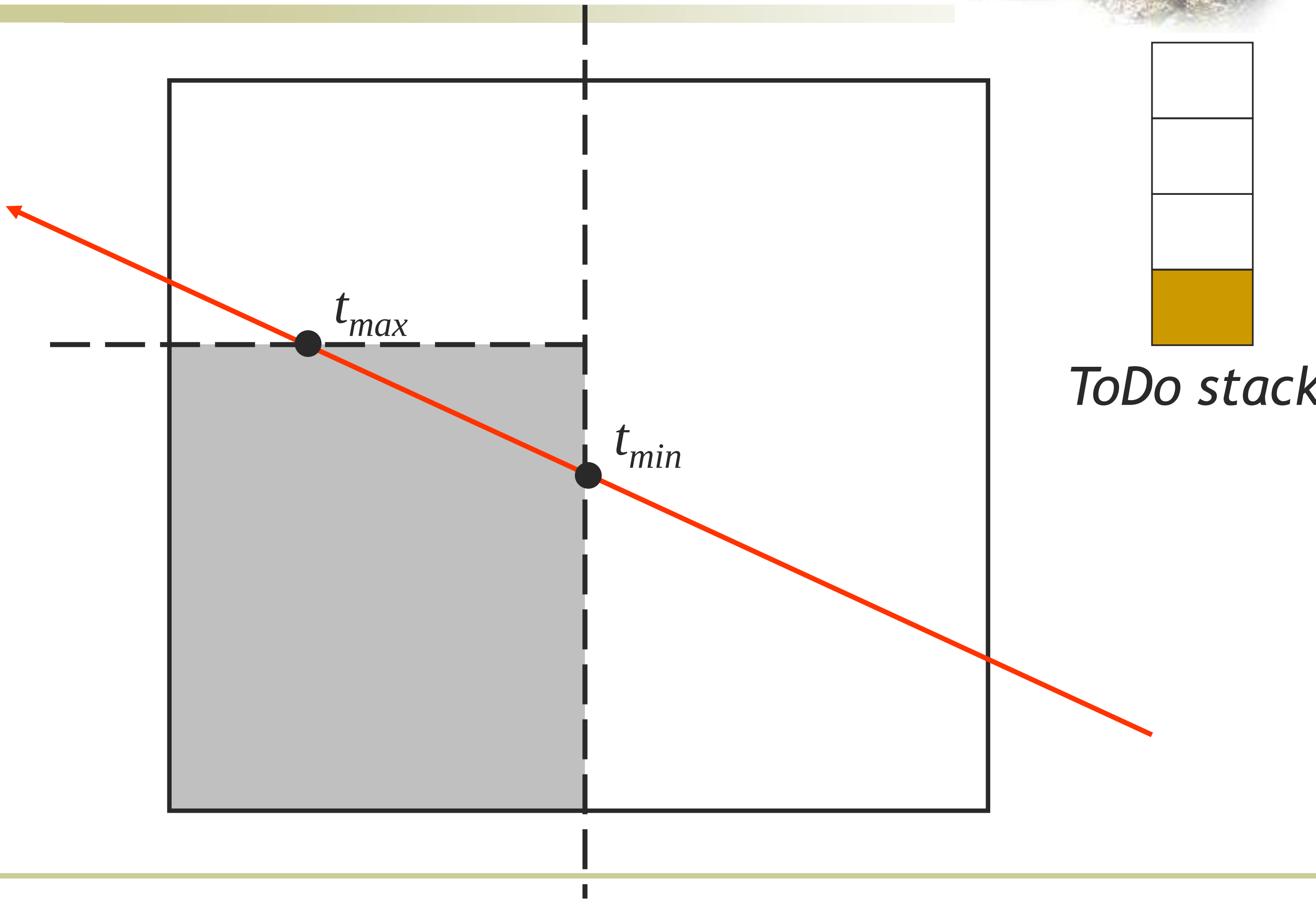
Kd-Tree Traversal



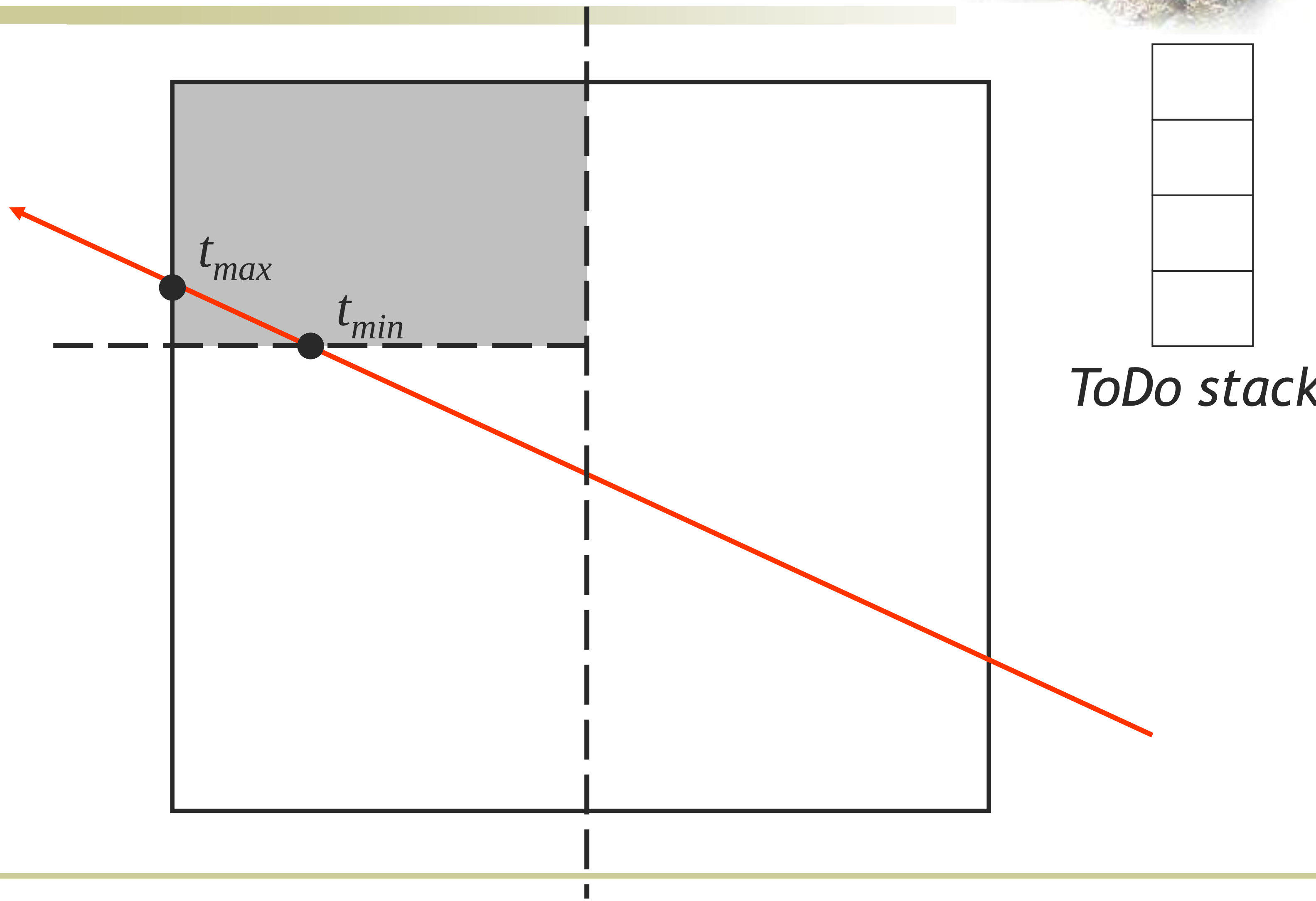
Kd-Tree Traversal



Kd-Tree Traversal



Kd-Tree Traversal

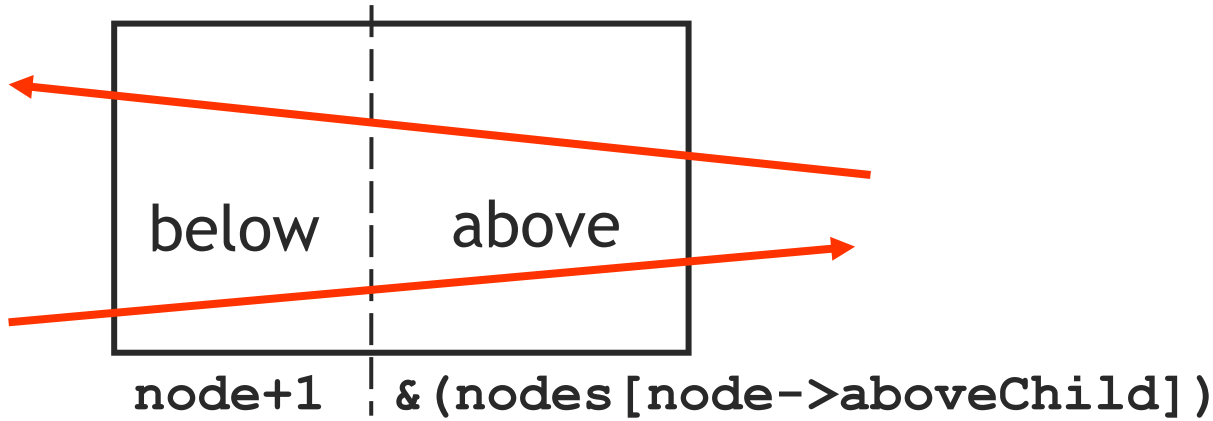


Kd-Tree Traversal-Leaf node

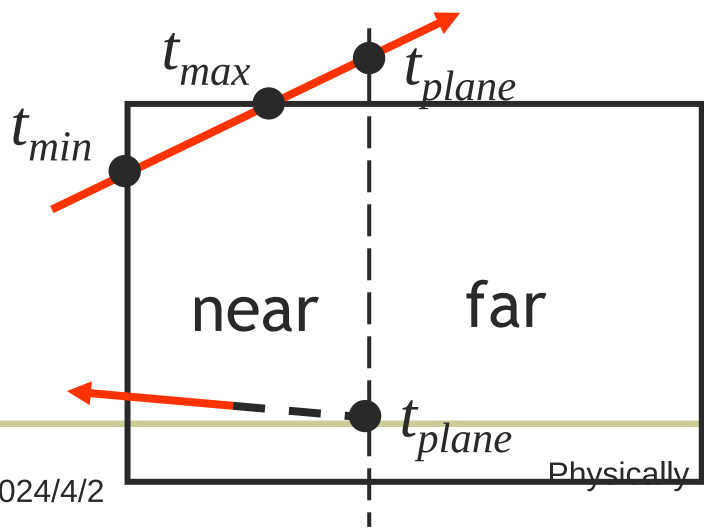
1. Check whether ray intersects primitive(s) inside the node; update ray's **`maxt`**
2. Grab next node from ToDo queue

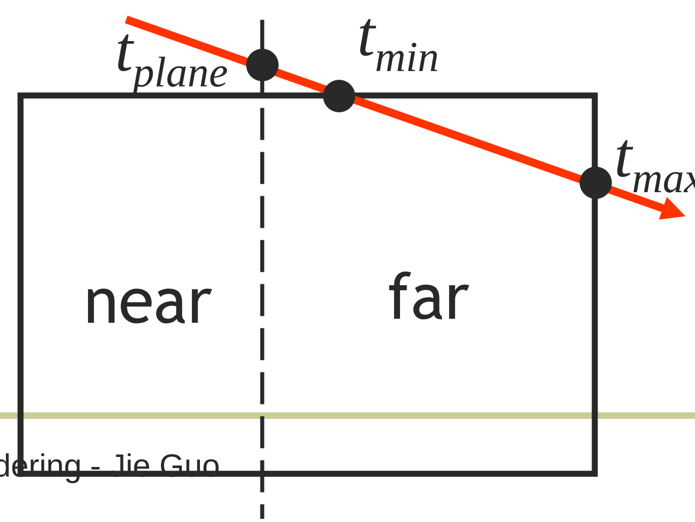
Kd-Tree Traversal-Interior node

1. Determine near and far (by testing which side O is)



2. Determine whether we can skip a node





Grid vs. KD-Tree

- Uniform grid acceleration structure
  - Regular structure = efficient traversal
  - Regular structure = poor partitioning

- KD-Tree
  - Adapt to scene complexity
  - Compact storage, efficient traversal
  - “Best” for CPU ray tracing

BVH vs. KD-Tree

- Building time: BVH < KD-Tree
- Ray intersection test: BVH > KD-Tree

How to choose?

- Type of rendering applications (e.g. offline, interactive)
- Type of scenes (e.g. static, deformable, dynamic)
- Hardware architecture: single core CPU, multi-core CPU, many-core GPU.
- Tradeoff: quality or building time?

How to choose?

- Hybrid methods: combines two structures in different levels.
- E.g., Irregular grid: build a coarse uniform grid for the top level and then subdivided each cell independently and adaptively.