Acceleration

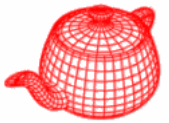
Digital Image Synthesis

Yung-Yu Chuang

10/5/2006

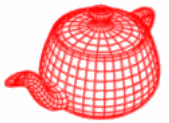
with slides by Mario Costa Sousa, Gordon Stoll and Pat Hanrahan

Classes



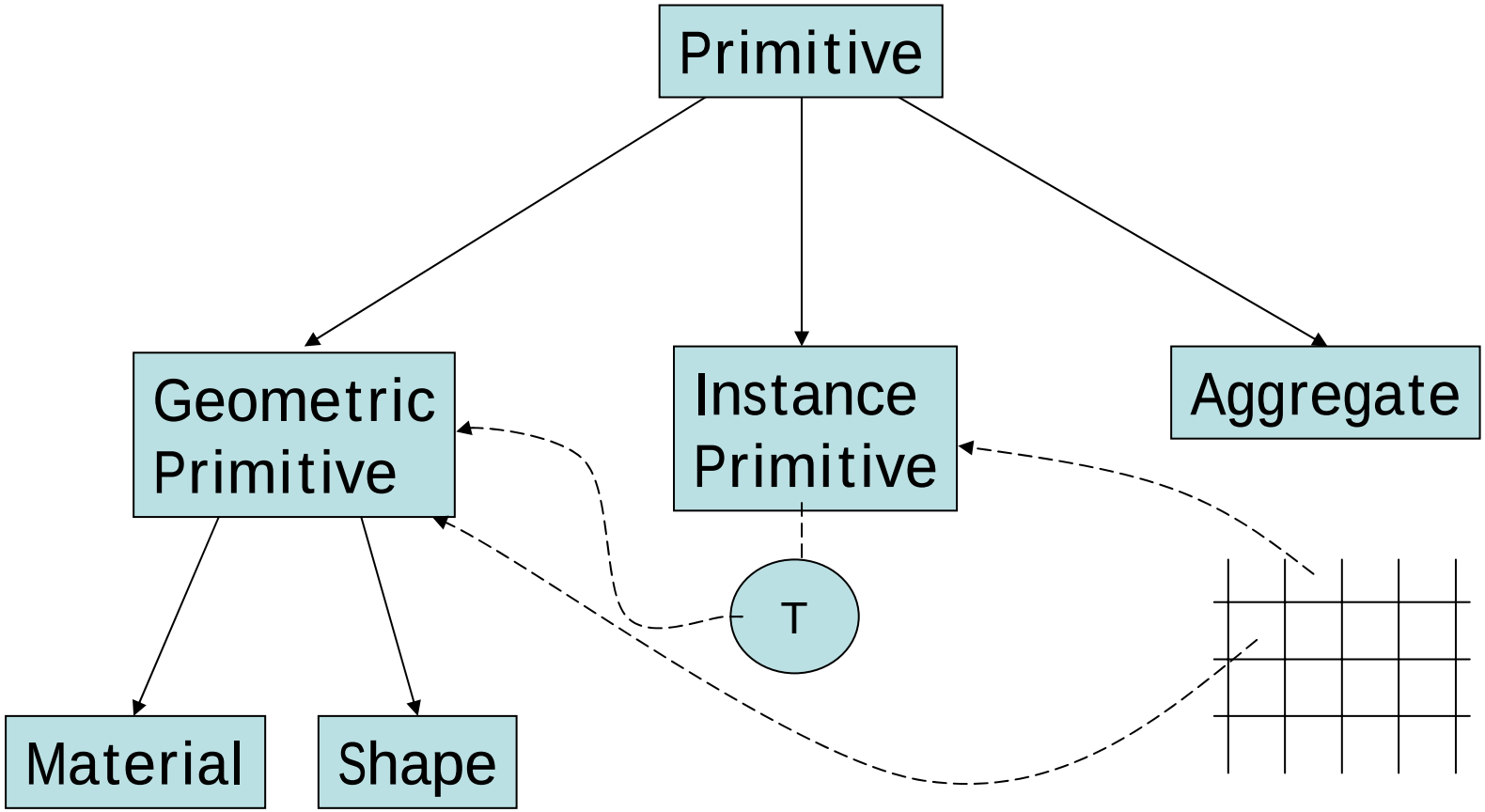
- **`Primitive`** (in core/primitive.*)
  - **`GeometricPrimitive`**
  - **`InstancePrimitive`**
  - **`Aggregate`**
- Two types of accelerators are provided (in accelerators/*.cpp)
  - **`GridAccel`**
  - **`KdTreeAccel`**

Primitive

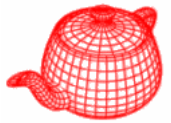


```
class Primitive : public ReferenceCounted {
  <Primitive interface>
}
```

Hierarchy

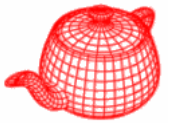

Primitive
Geometric Primitive
Instance Primitive
Aggregate
T
Material
Shape

Interface



```
BBox WorldBound();                                    geometry
bool CanIntersect();
bool Intersect(const Ray &r,          // update maxt
               Intersection *in);
bool IntersectP(const Ray &r);
void Refine(vector<Reference<Primitive>>
            &refined);
void FullyRefine(vector<Reference<Primitive>>
                 &refined);
-------------------------------------------------
AreaLight *GetAreaLight();                    material
BSDF *GetBSDF(const DifferentialGeometry &dg,
              const Transform &WorldToObject);
```

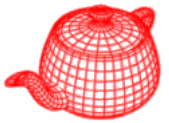
Intersection



- **`primitive`** stores the actual intersecting primitive, hence Primitive->GetAreaLight and GetBSDF can only be called for GeometricPrimitive

```
struct Intersection {
  <Intersection interface>
  DifferentialGeometry dg;
  const Primitive *primitive;
  Transform WorldToObject;
};
```

`GeometricPrimitive`



- represents a single shape
- holds a reference to a **`Shape`** and its **`Material`**, and a pointer to an **`AreaLight`**

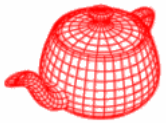
```
Reference<Shape> shape;
Reference<Material> material;
AreaLight *areaLight;
```

- Most operations are forwarded to shape

Object instancing

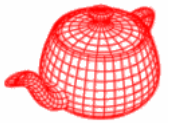
61 unique plant models, 1.1M triangles, 300MB
4000 individual plants, 19.5M triangles

InstancePrimitive



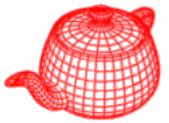
```
Reference<Primitive> instance;
Transform InstanceToWorld, WorldToInstance;



Ray ray = WorldToInstance(r);
if (!instance->Intersect(ray, isect))
  return false;
r.maxt = ray.maxt;
isect->WorldToObject = isect->WorldToObject
  *WorldToInstance;
```

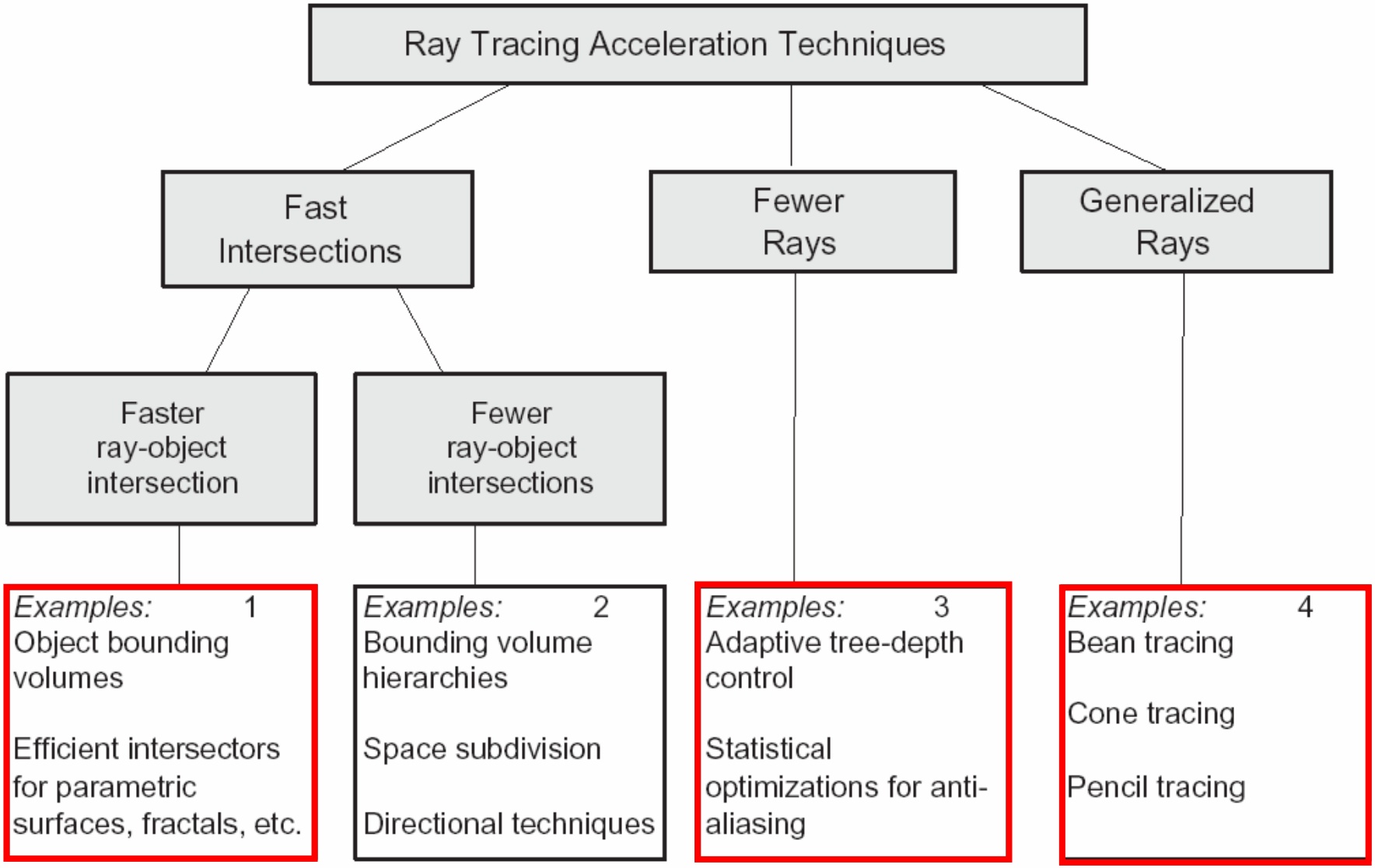
Aggregates



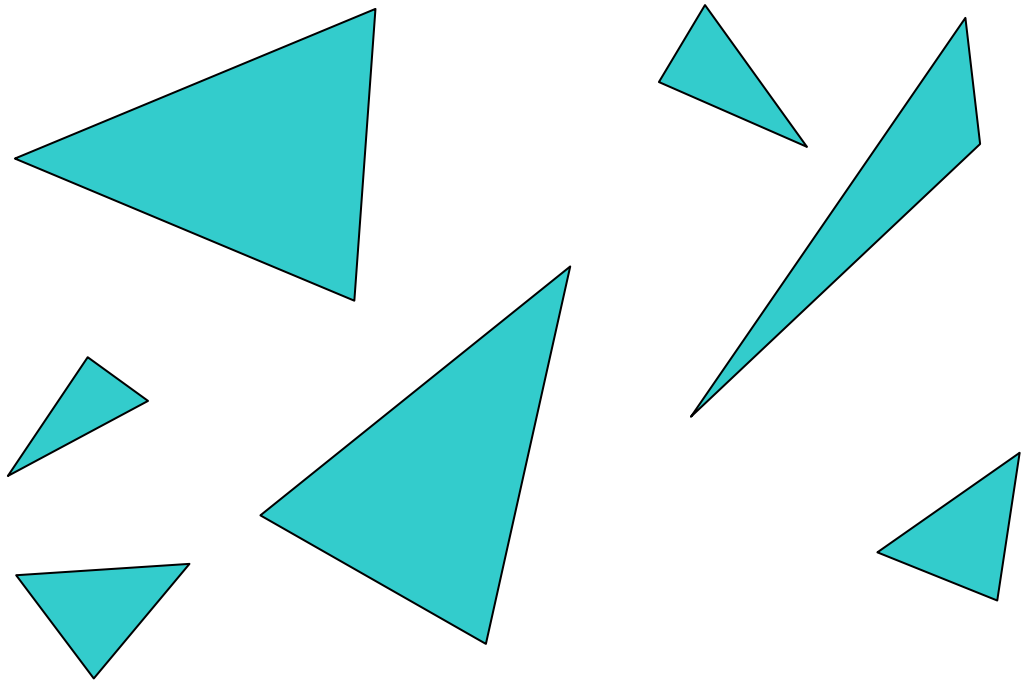
- Acceleration is a heart component of a ray tracer because ray/scene intersection accounts for the majority of execution time
- Goal: reduce the number of ray/primitive intersections by quick simultaneous rejection of groups of primitives and the fact that nearby intersections are likely to be found first
- Two main approaches: spatial subdivision, object subdivision
- No clear winner

Acceleration techniques

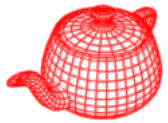




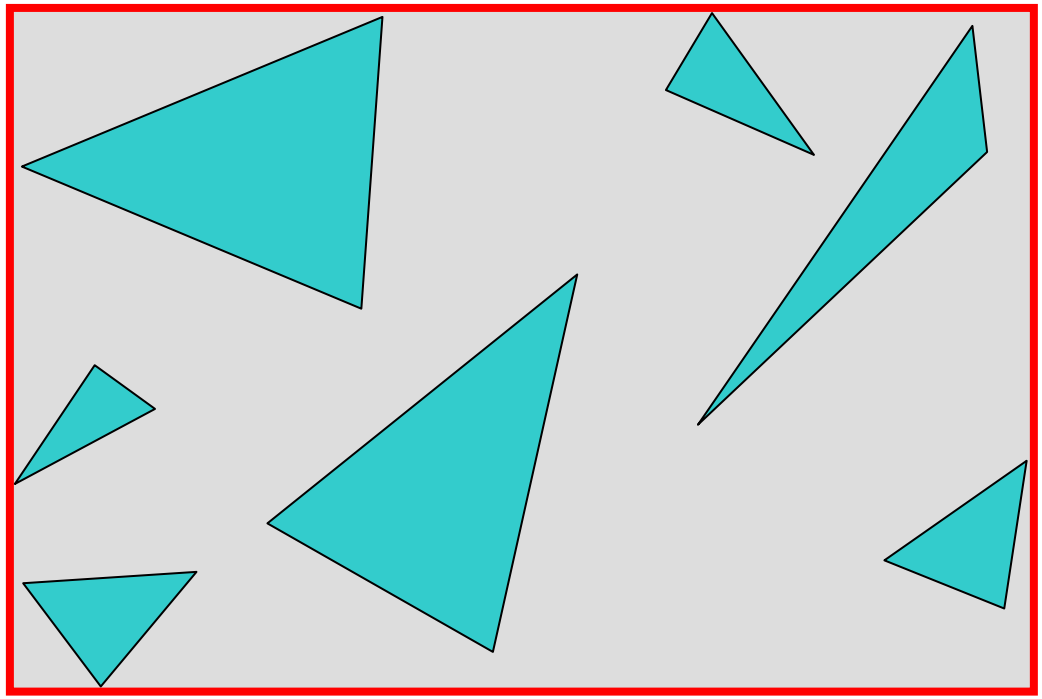
Bounding volume hierarchy



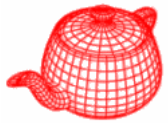
Bounding volume hierarchy



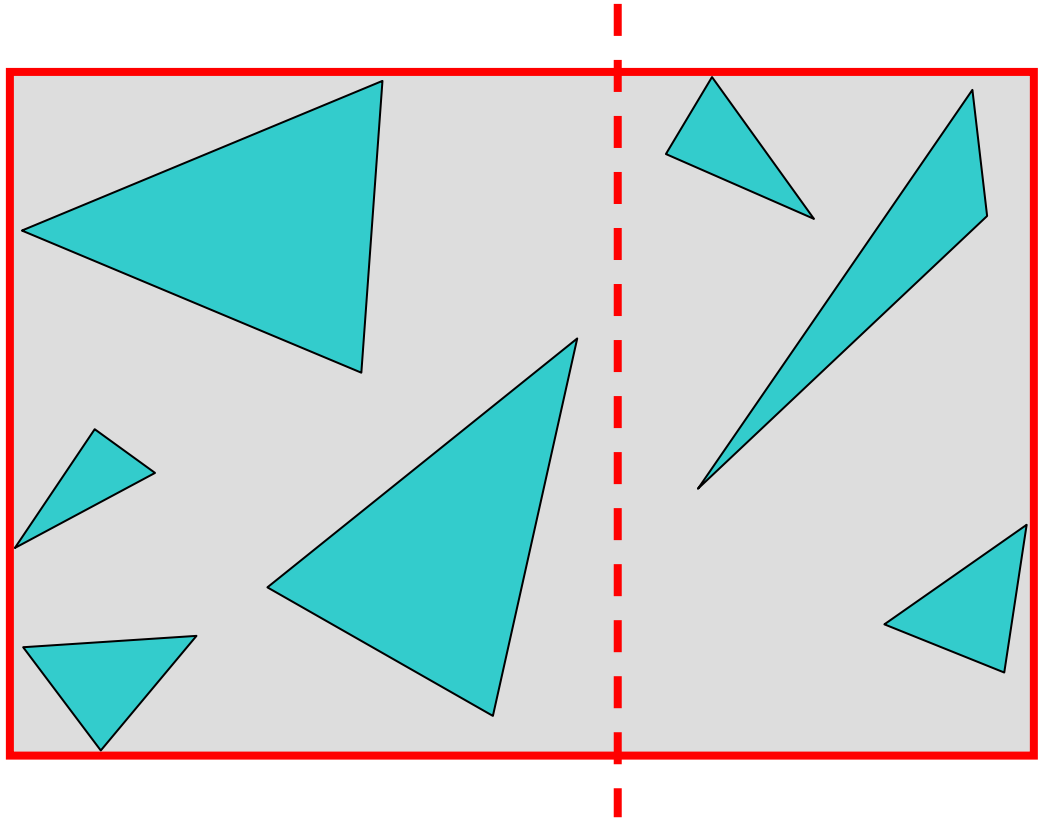
1) Find bounding box of objects



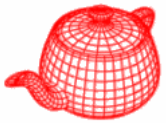
Bounding volume hierarchy



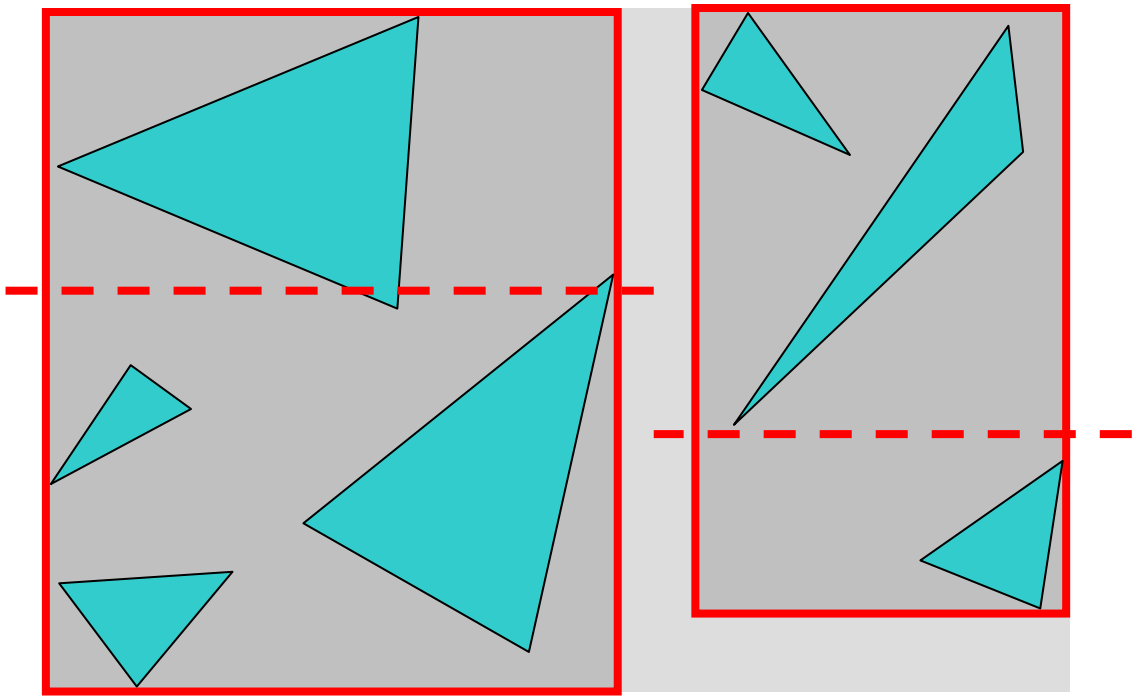
1) Find bounding box of objects
2) Split objects into two groups



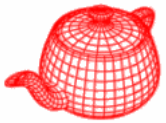
Bounding volume hierarchy



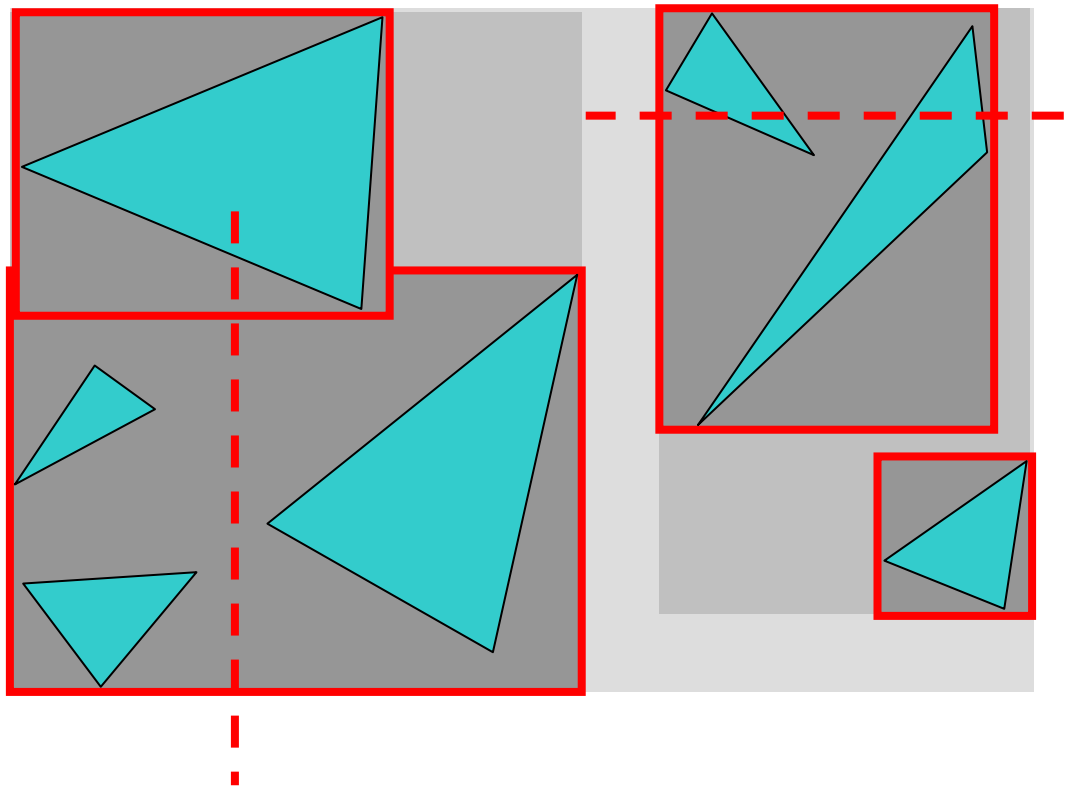
1) Find bounding box of objects
2) Split objects into two groups
3) Recurse



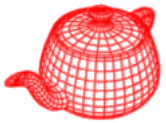
Bounding volume hierarchy



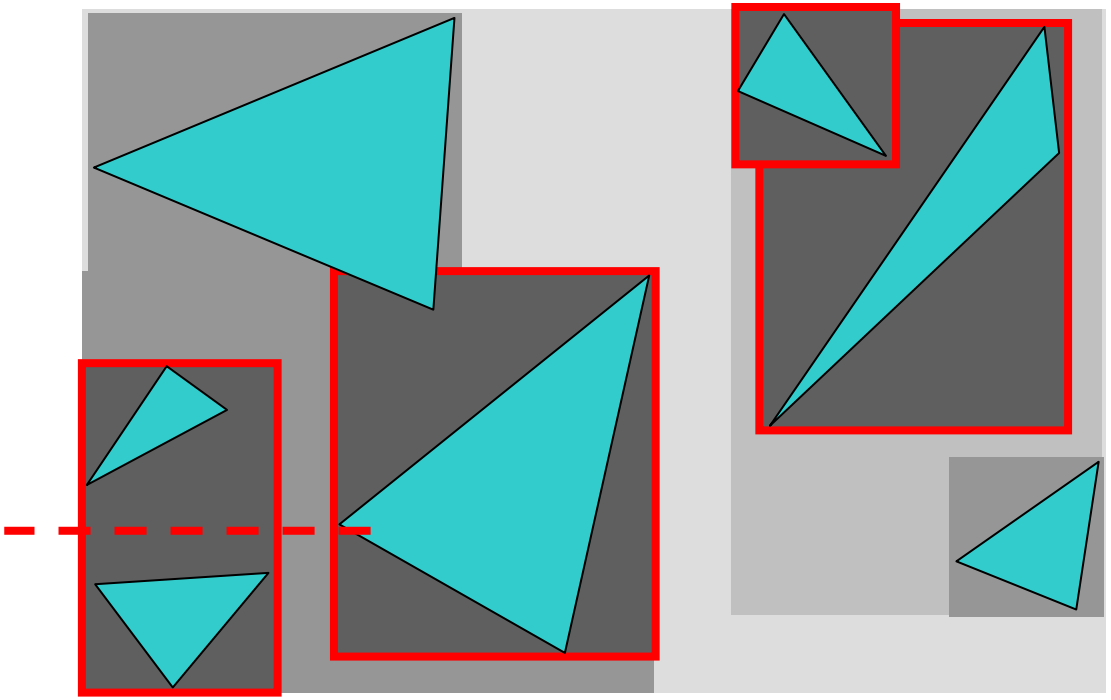
1) Find bounding box of objects
2) Split objects into two groups
3) Recurse



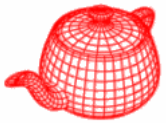
Bounding volume hierarchy



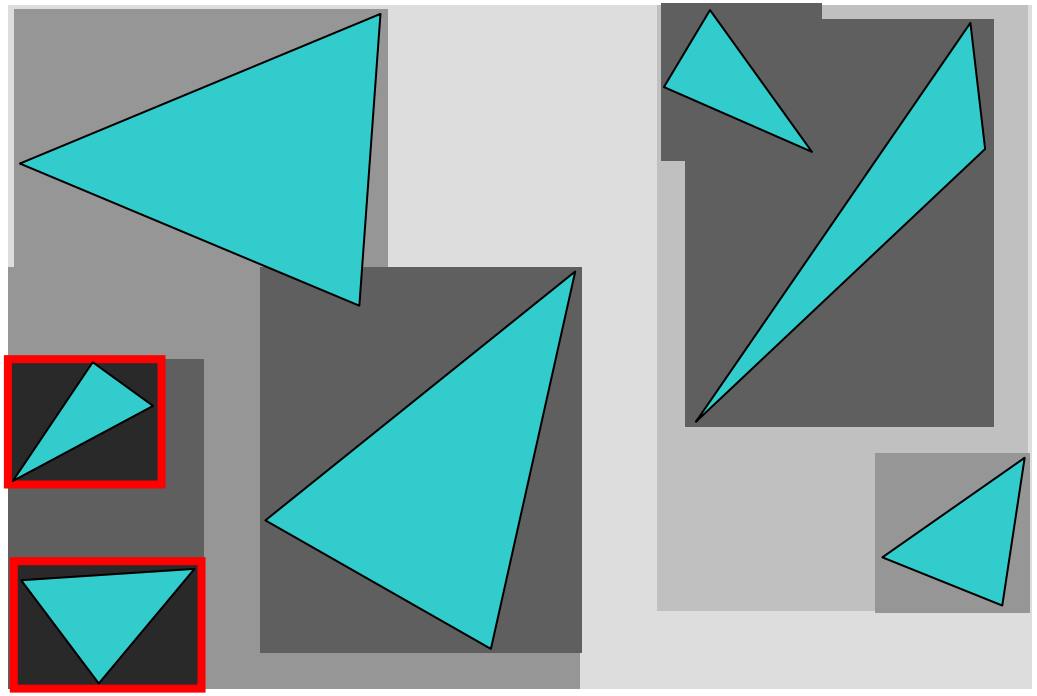
1) Find bounding box of objects
2) Split objects into two groups
3) Recurse



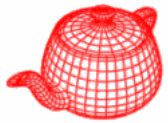
Bounding volume hierarchy



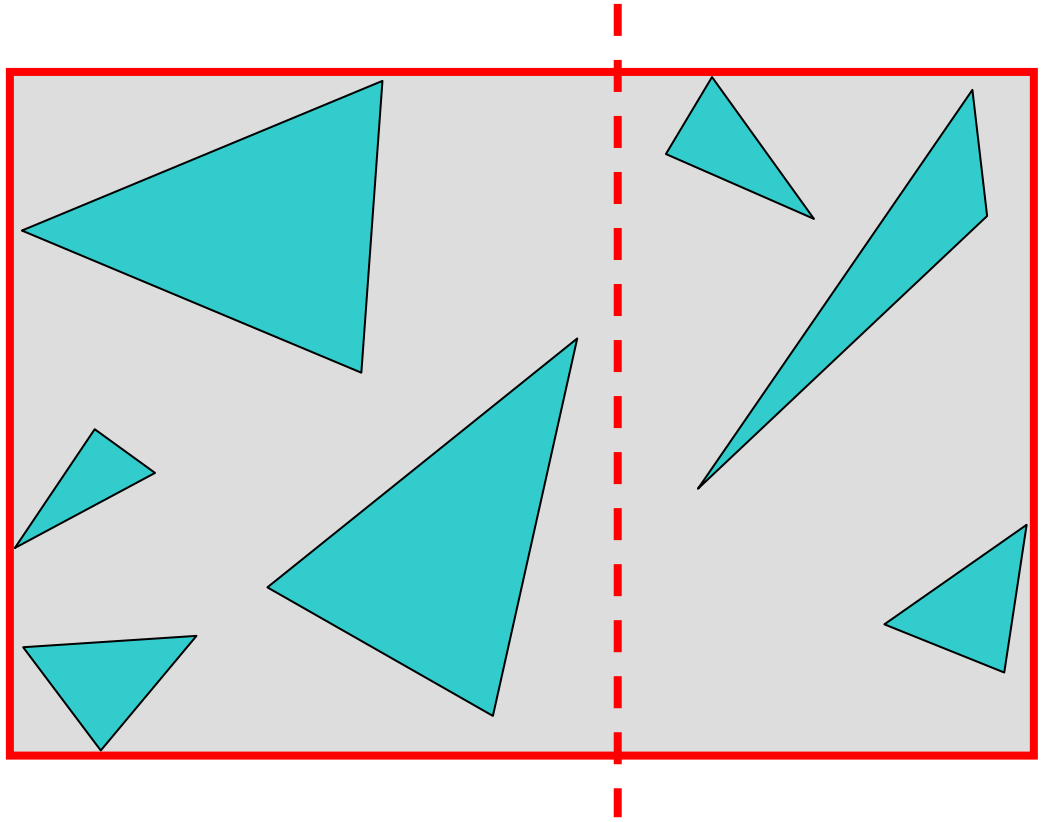
1) Find bounding box of objects
2) Split objects into two groups
3) Recurse



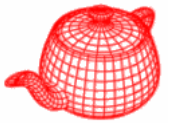
Where to split?



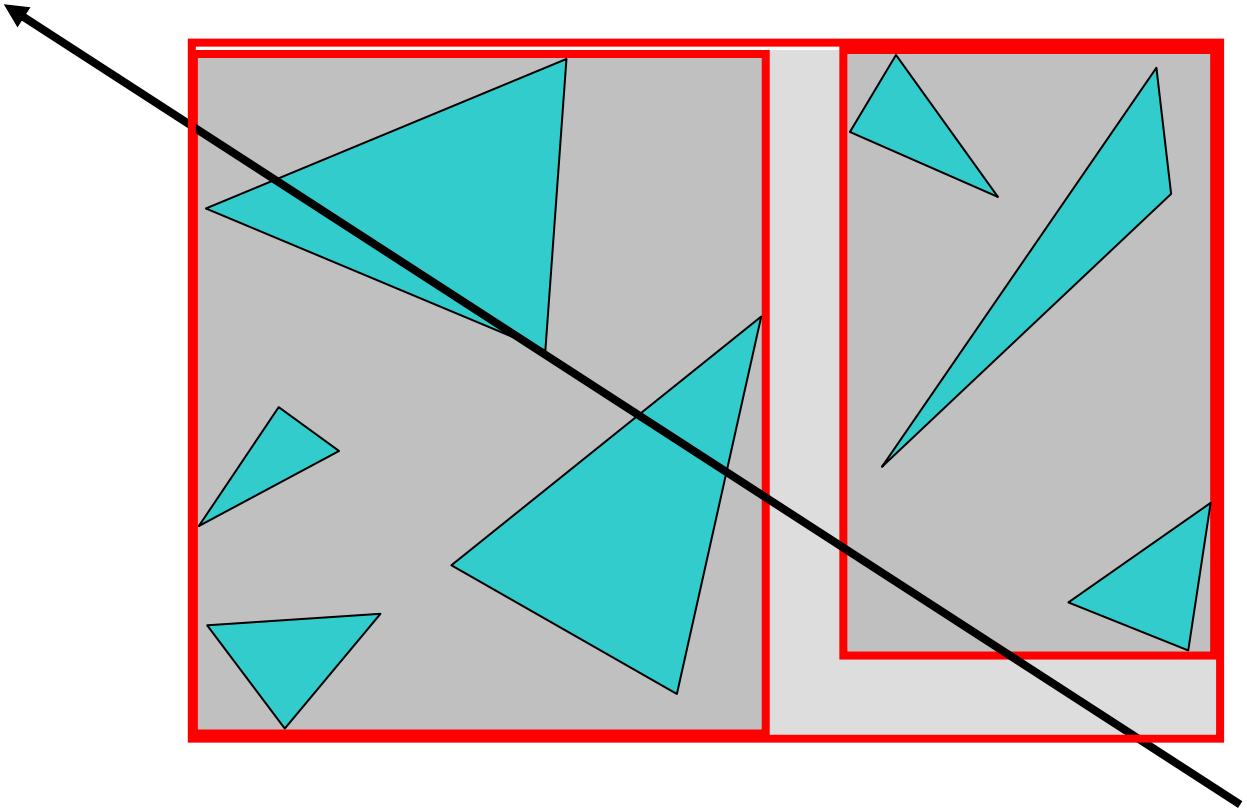
- At midpoint
- Sort, and put half of the objects on each side
- Use modeling hierarchy



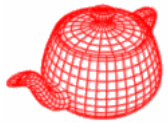
BVH traversal



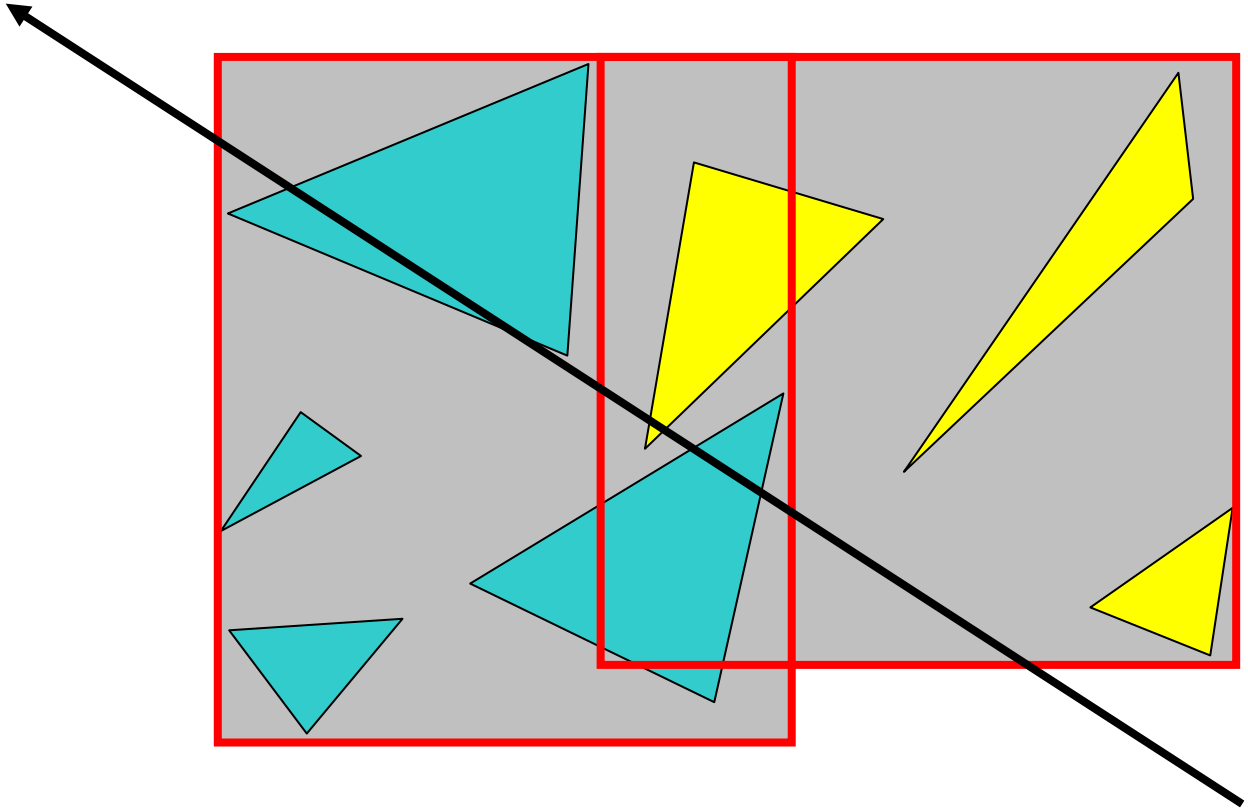
- If hit parent, then check all children



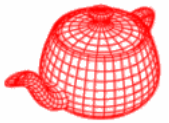
BVH traversal



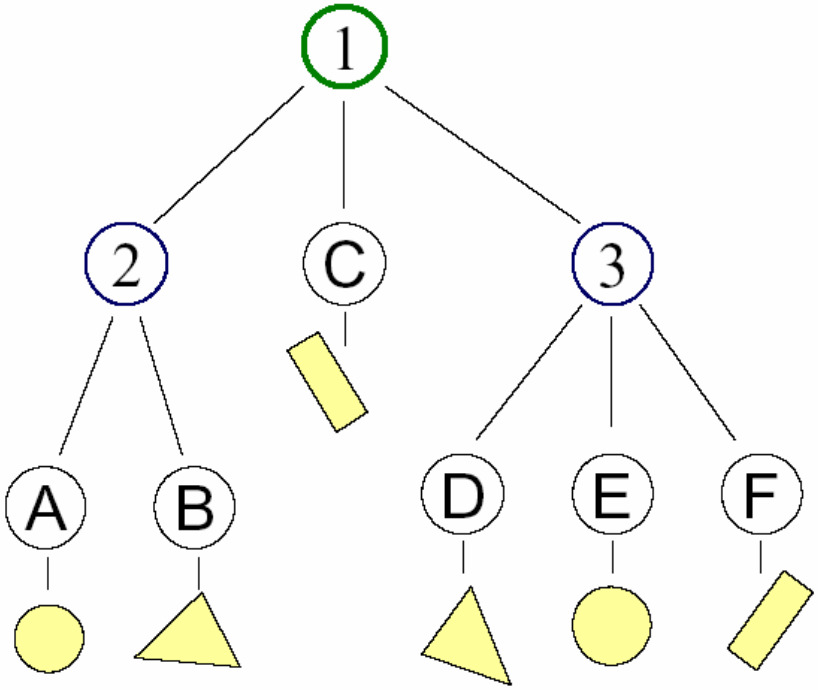
- Don't return intersection immediately because the other subvolumes may have a closer intersection

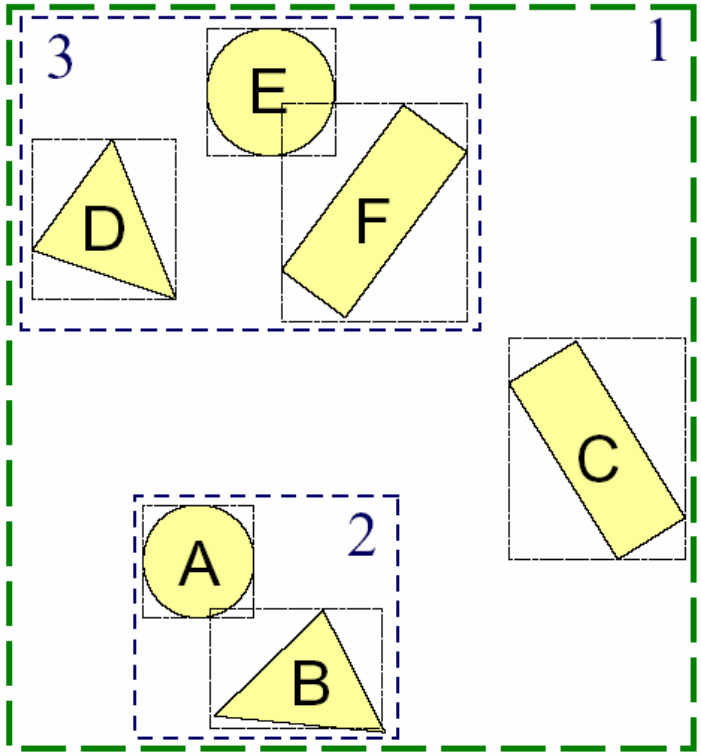


Bounding volume hierarchy

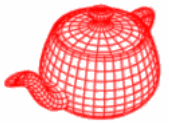


- Build hierarchy of bounding volumes
  - Bounding volume of interior node contains all children

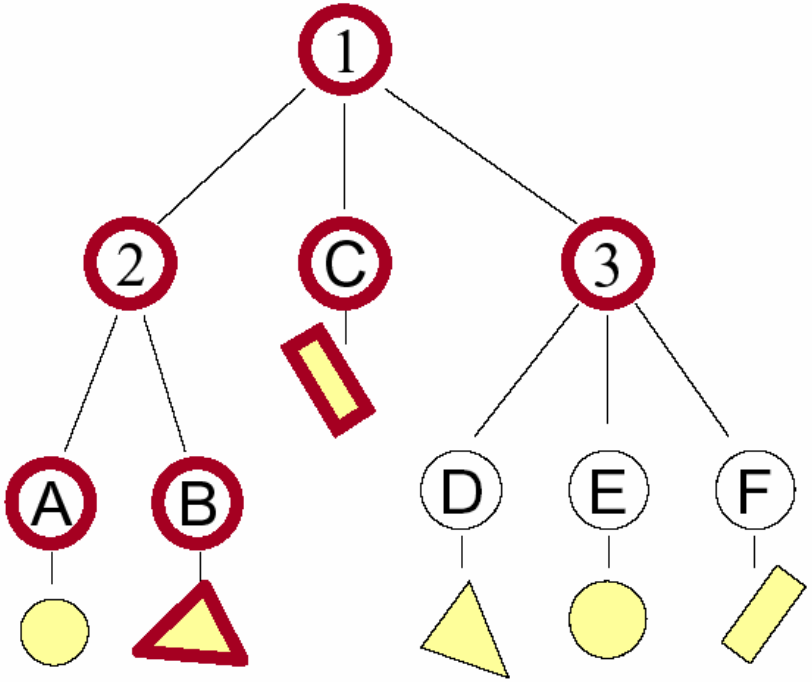


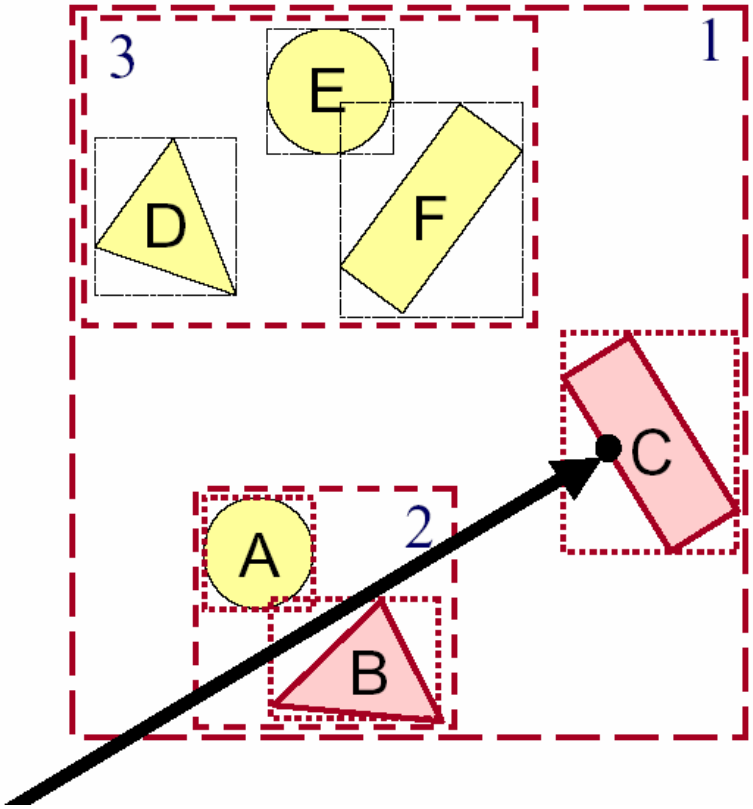


Bounding volume hierarchy

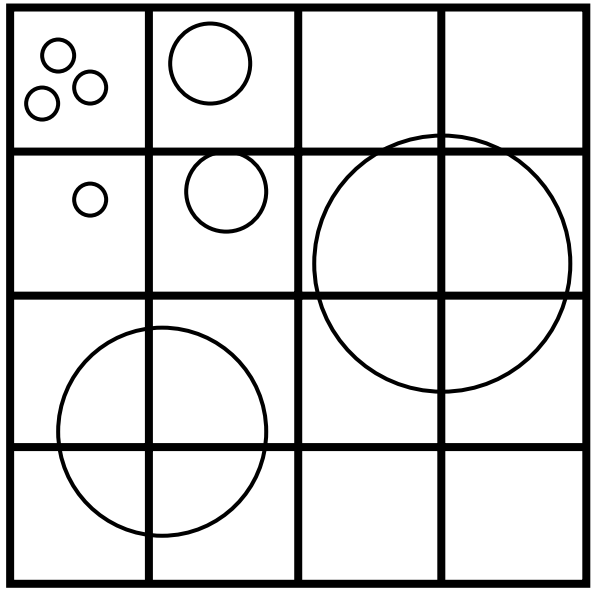


- Use hierarchy to accelerate ray intersections
  - Intersect node contents only if hit bounding volume

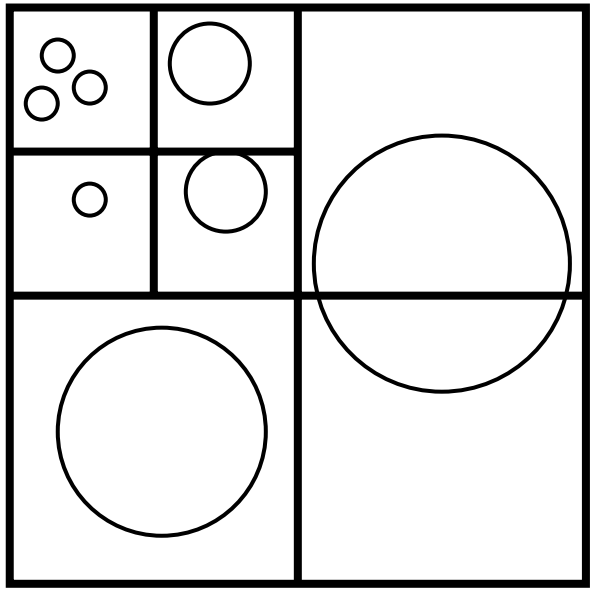




Space subdivision approaches

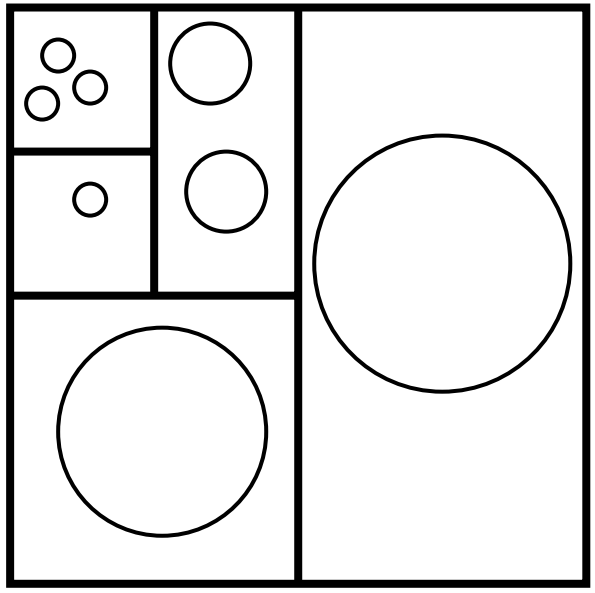


Unifrom grid

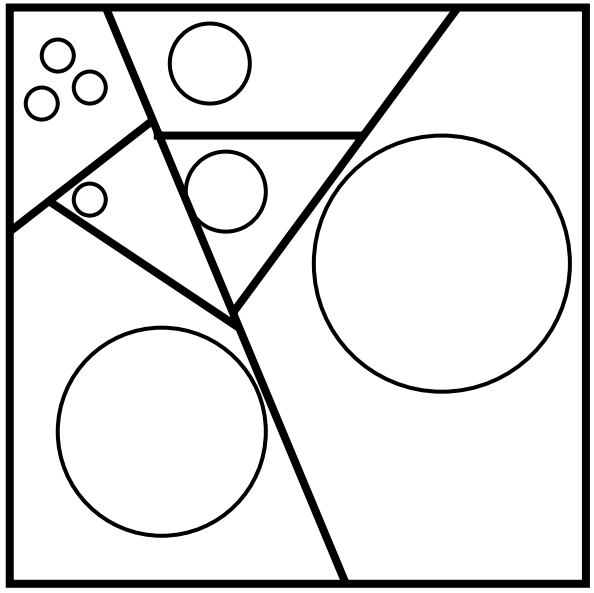


Quadtree (2D)
Octree (3D)

Space subdivision approaches

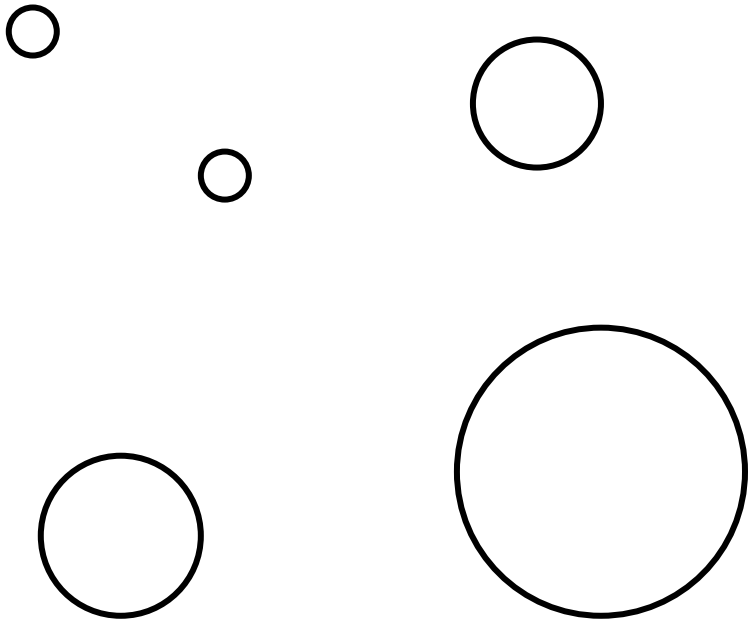


KD tree

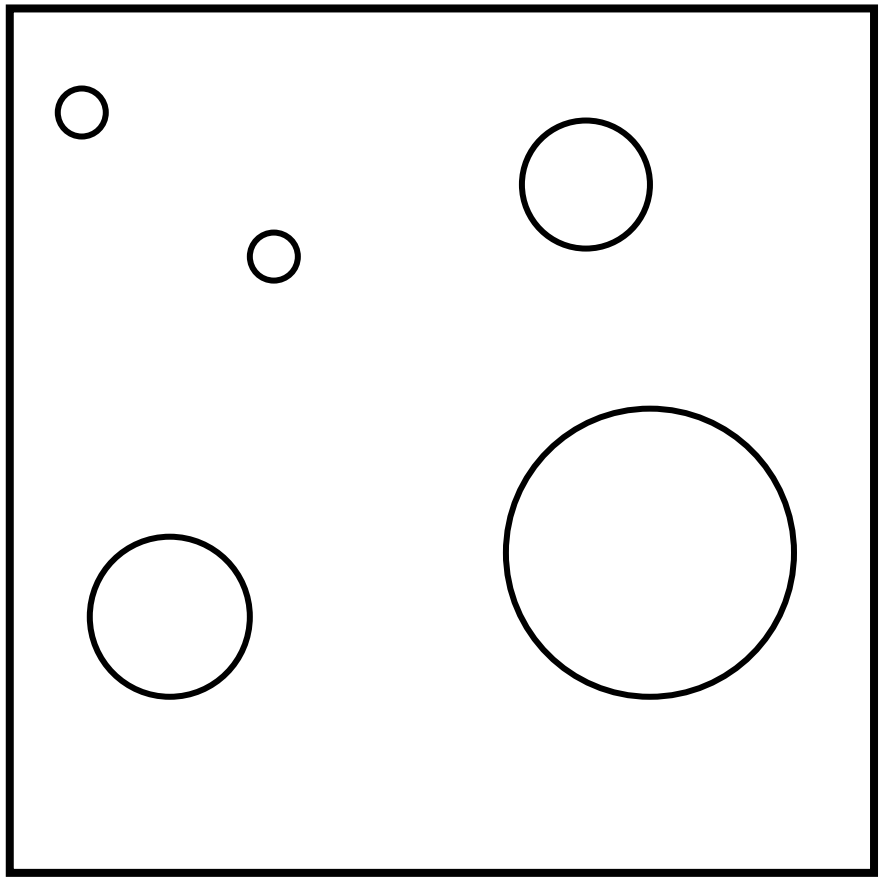


BSP tree

Uniform grid



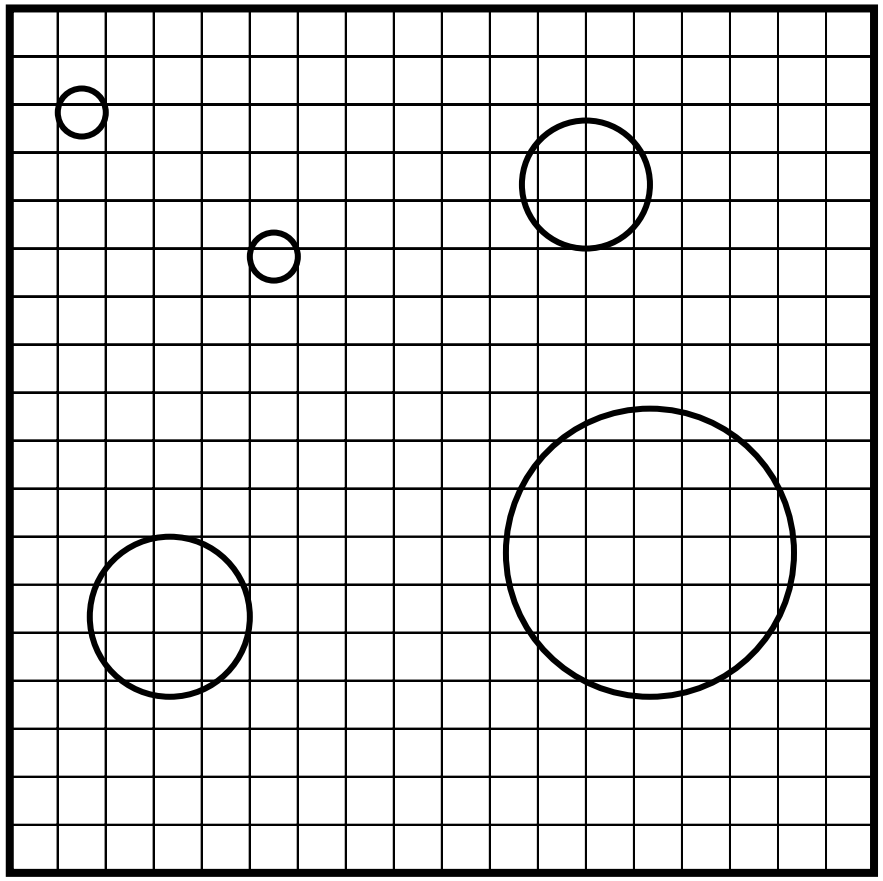
Uniform grid



Preprocess scene

1. Find bounding box

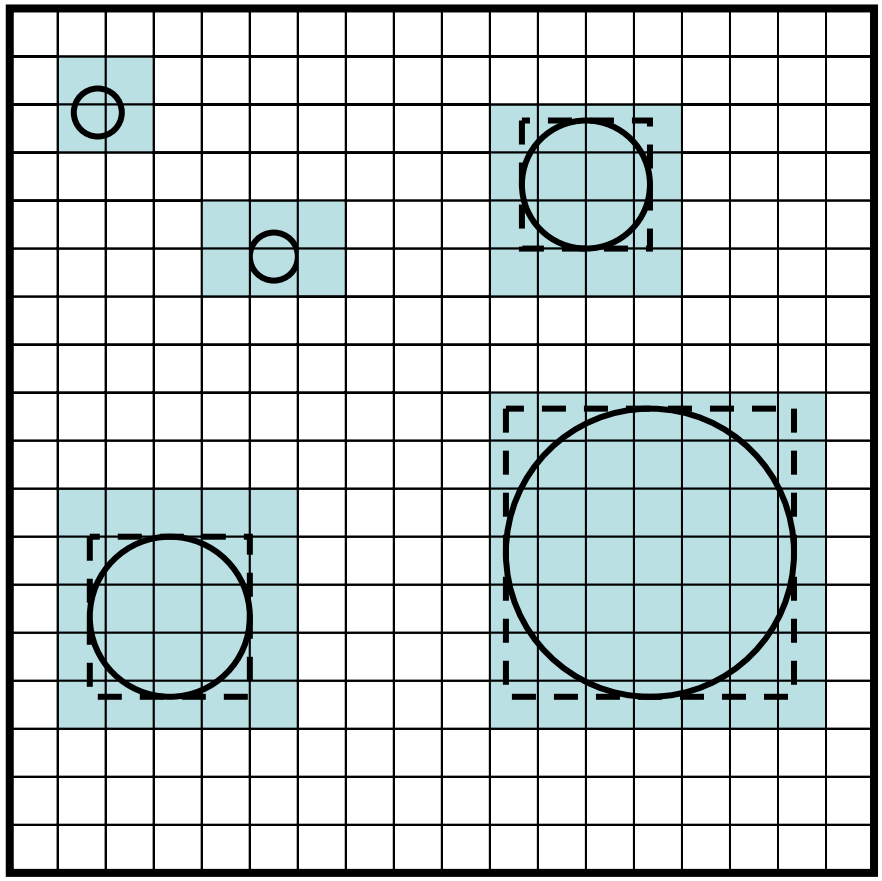
Uniform grid



Preprocess scene

1. Find bounding box
2. Determine grid resolution

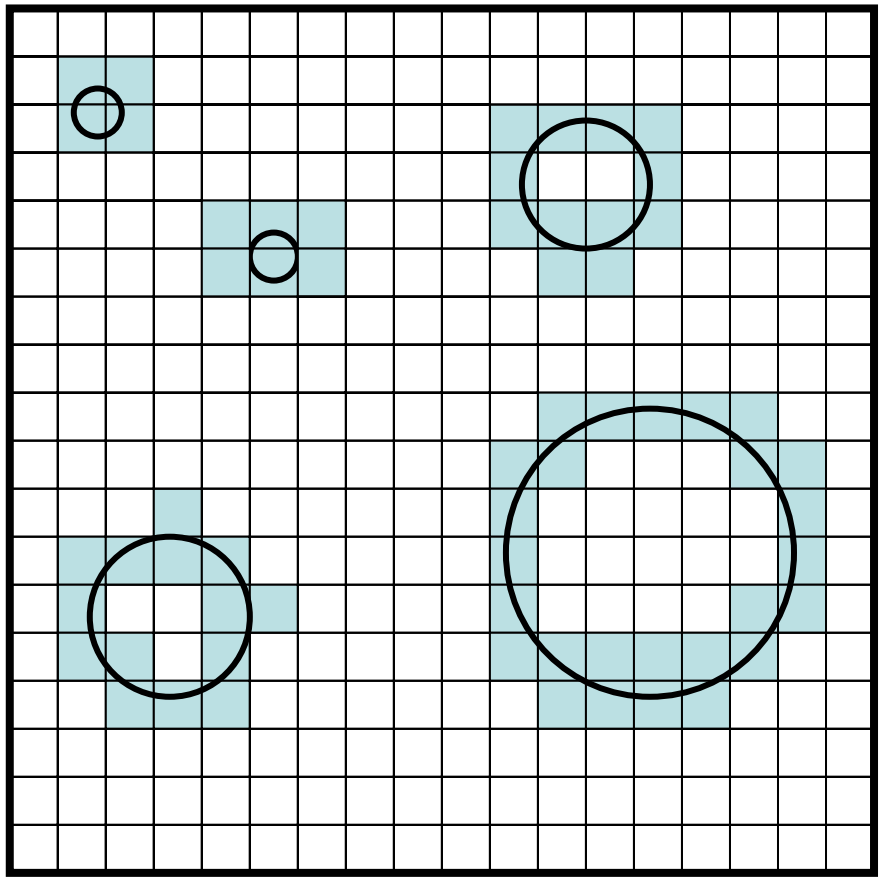
Uniform grid



Preprocess scene

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

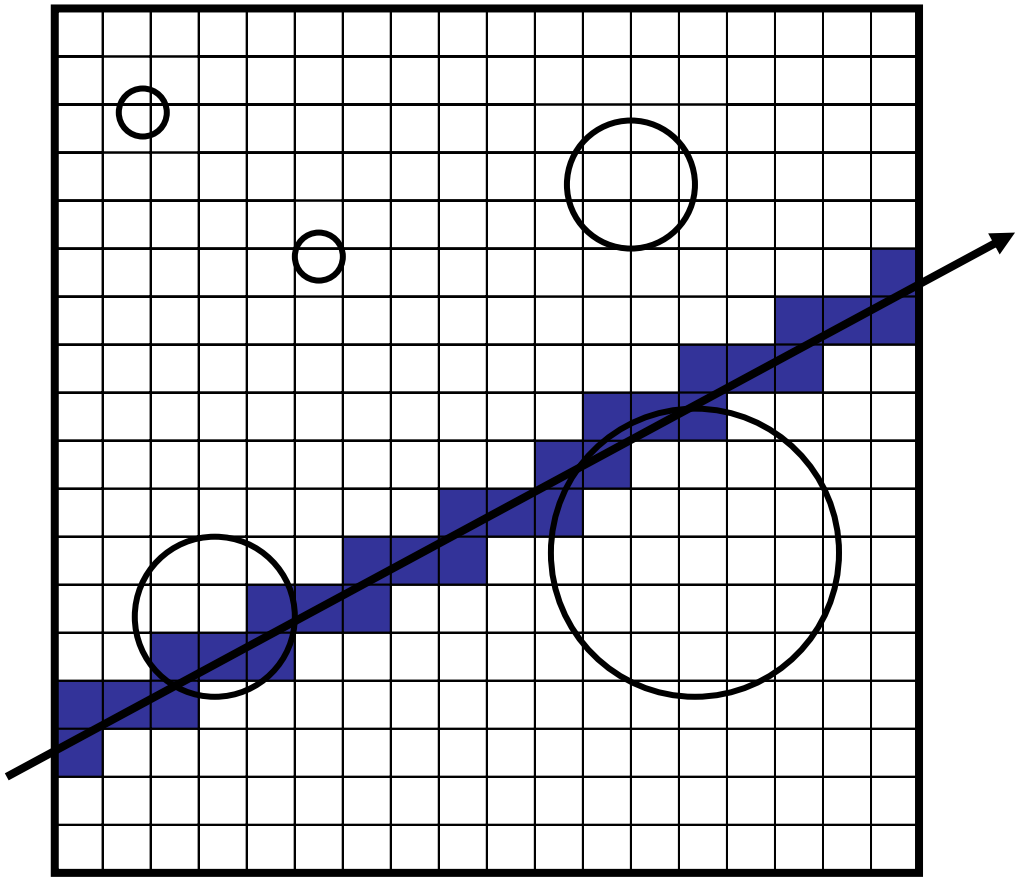
Uniform grid



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

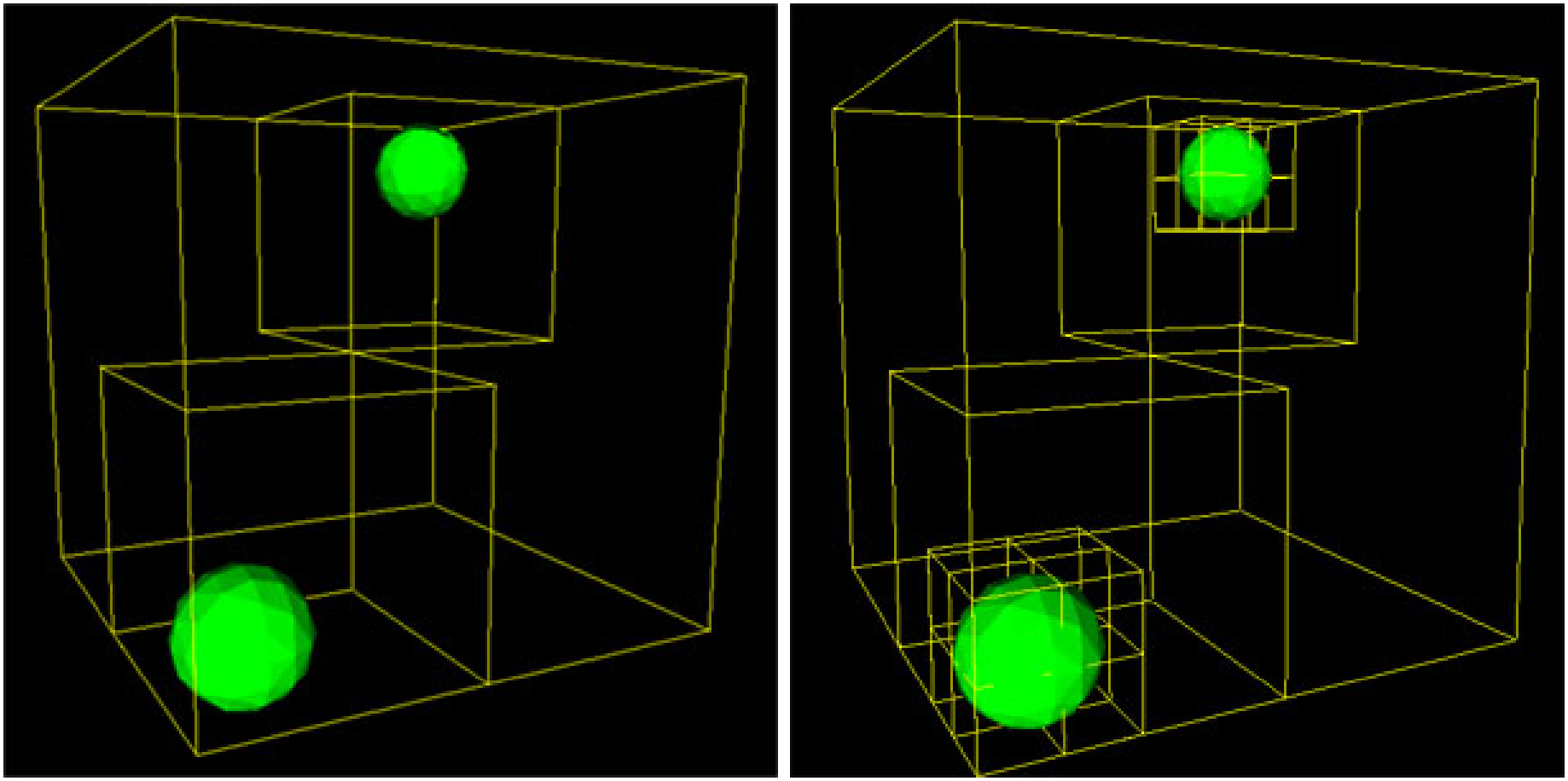


Preprocess scene

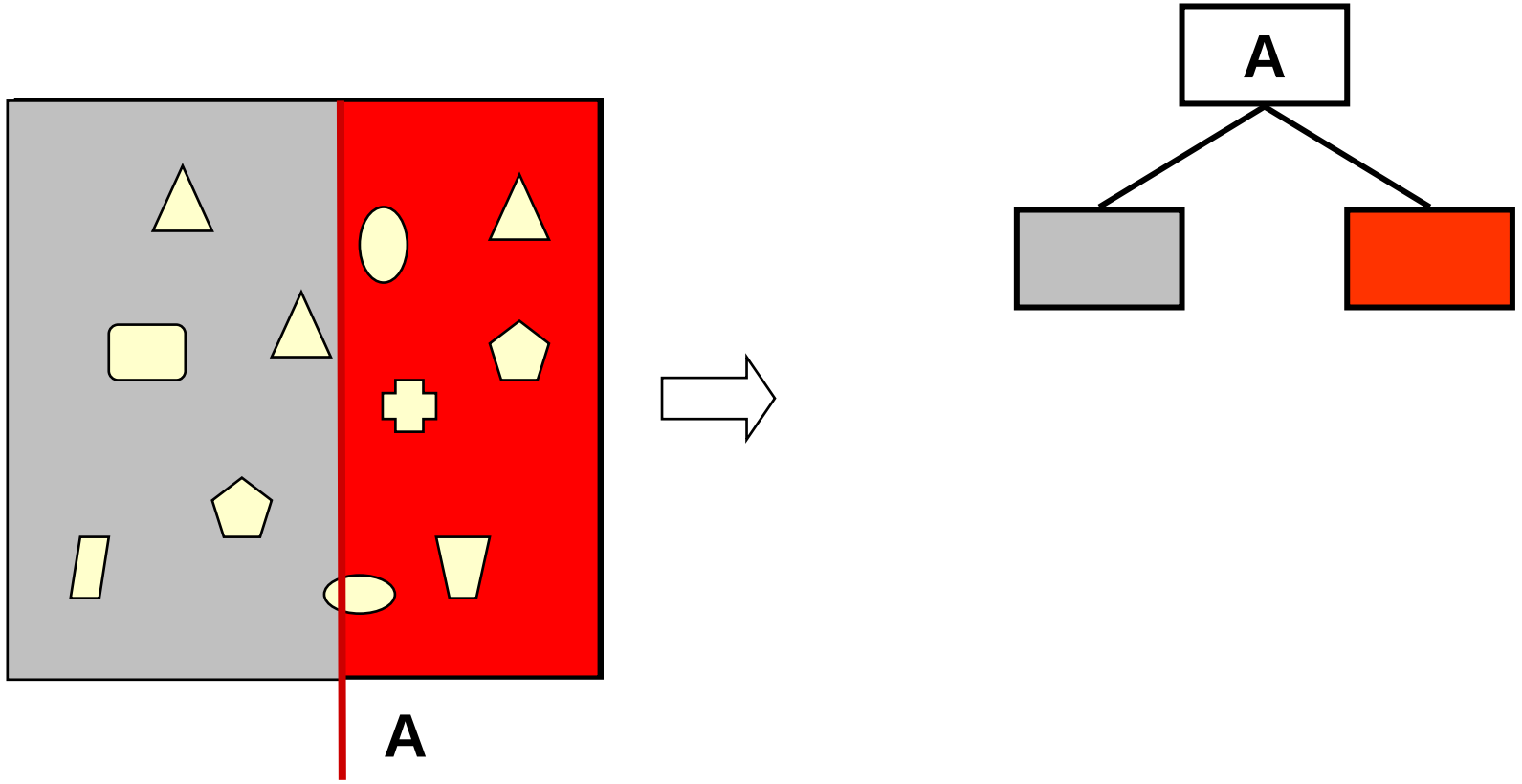
Traverse grid

3D line = 3D-DDA

Octree

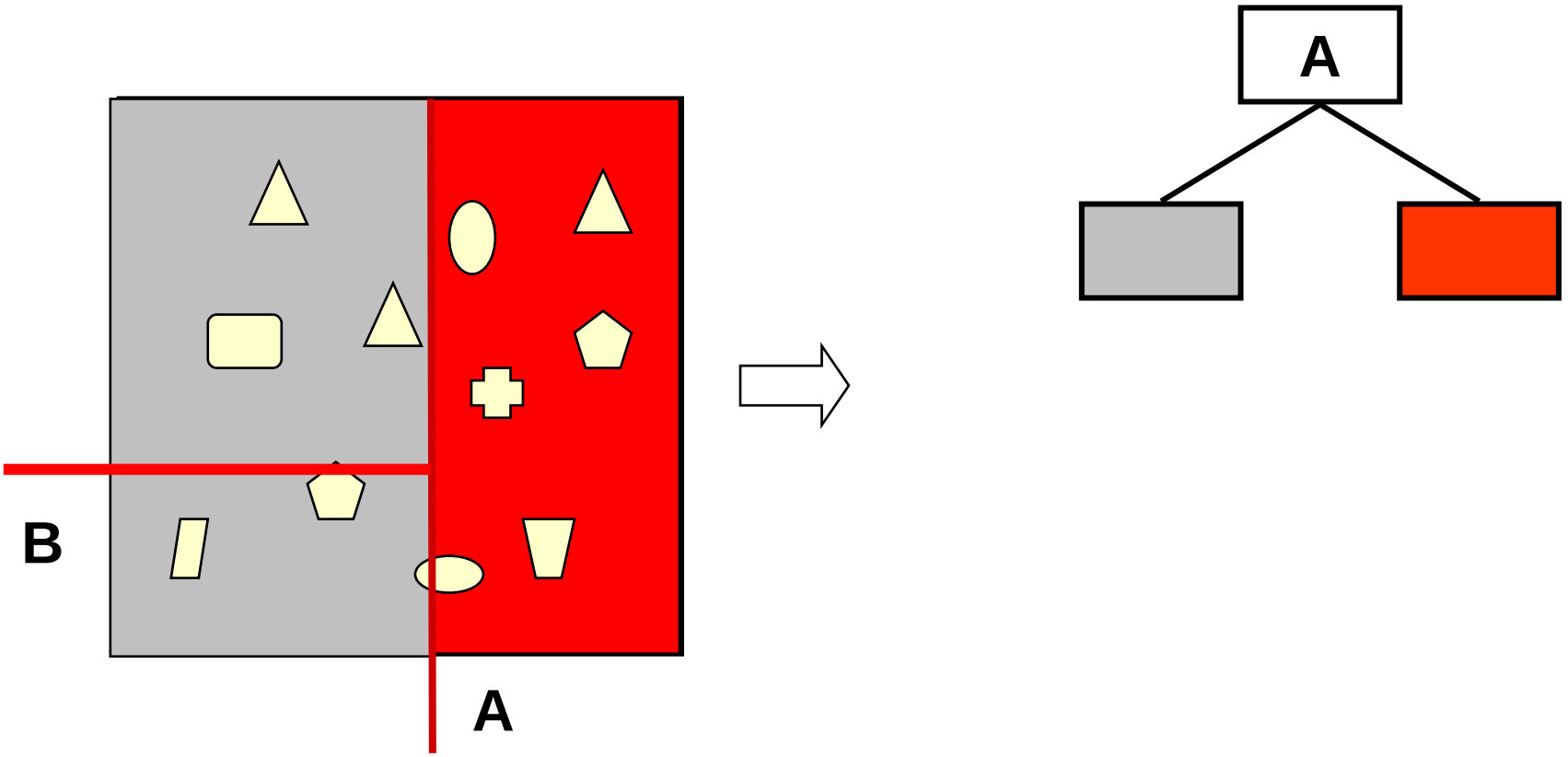


K-d tree



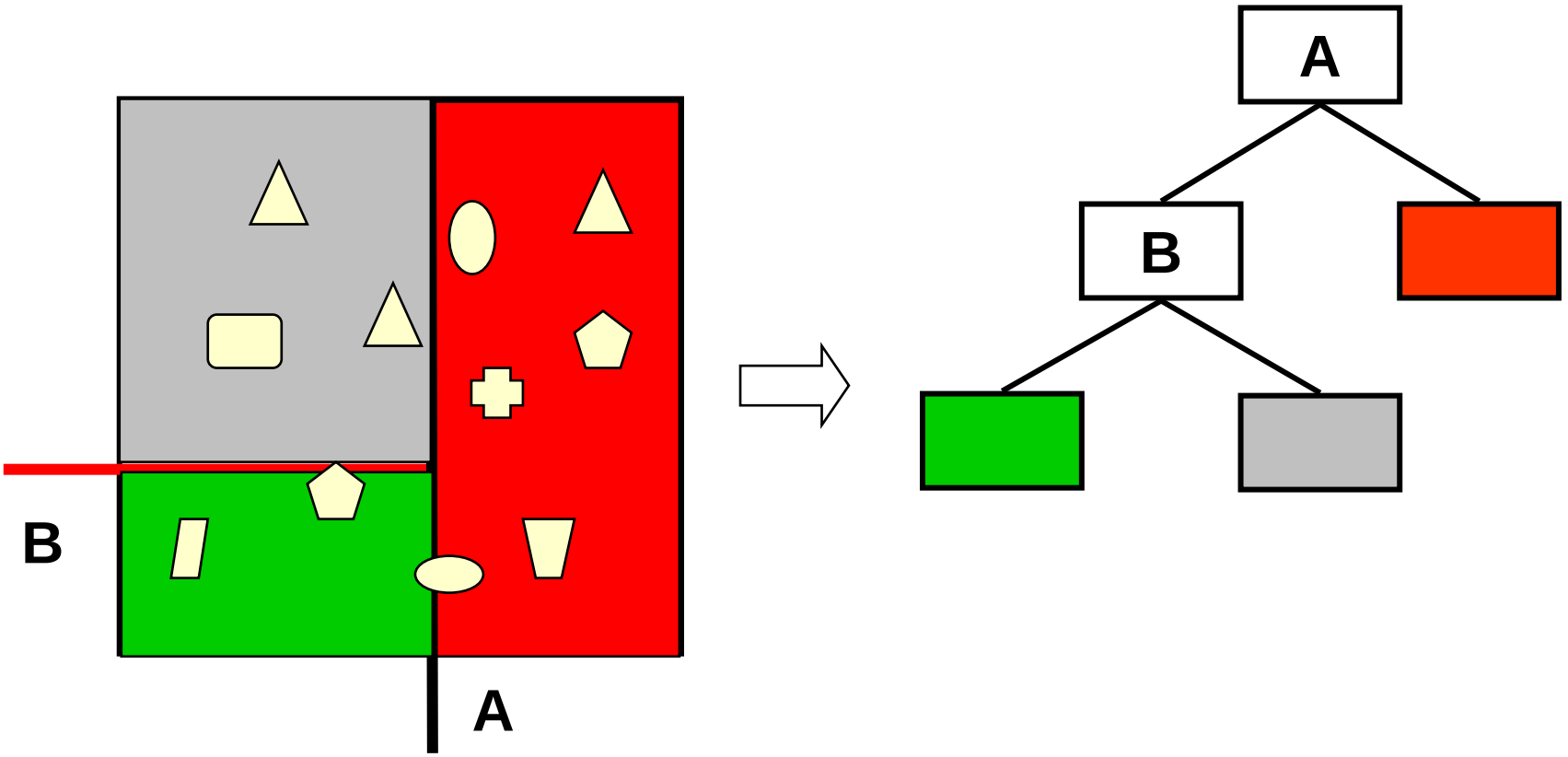
Leaf nodes correspond to unique regions in space

K-d tree



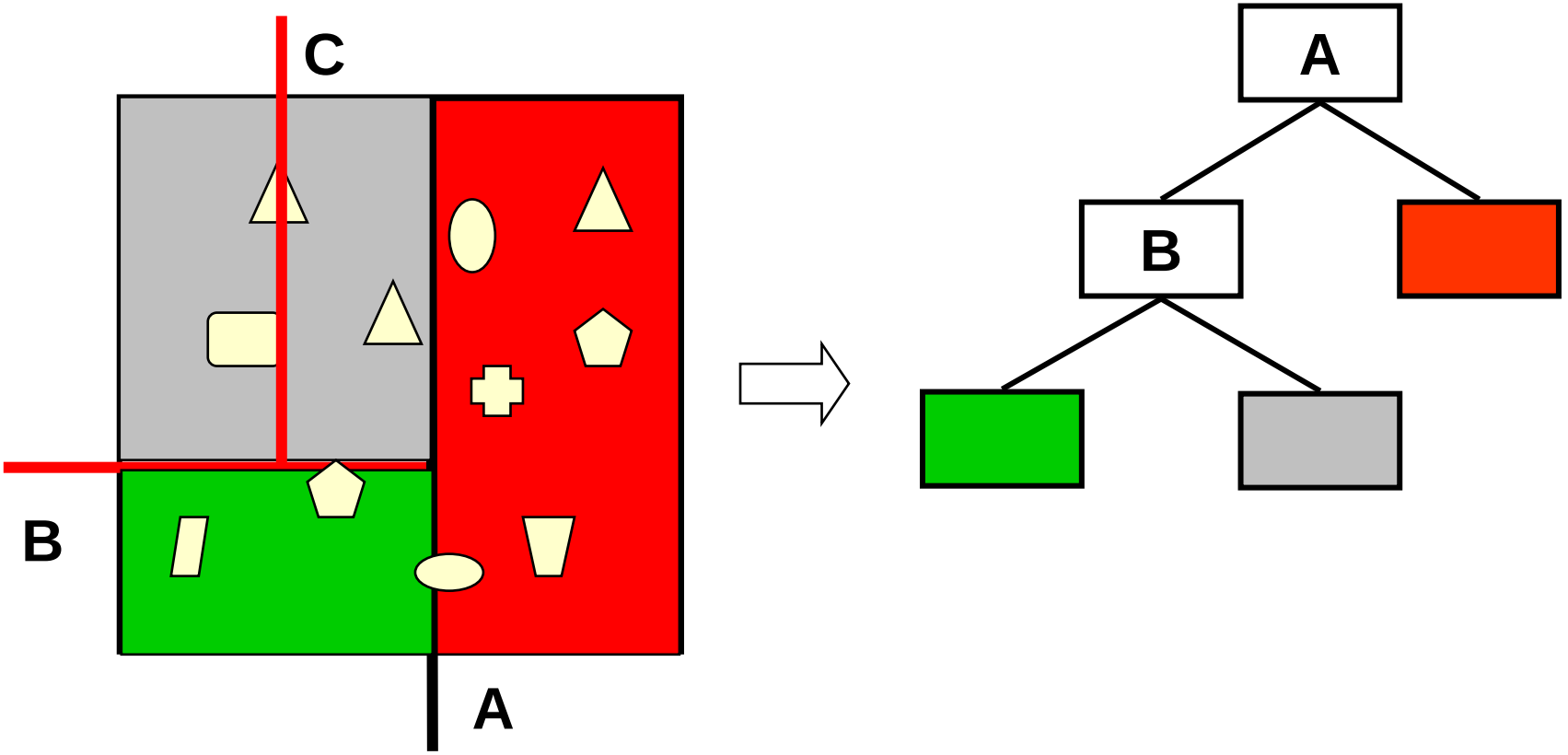
Leaf nodes correspond to unique regions in space

K-d tree

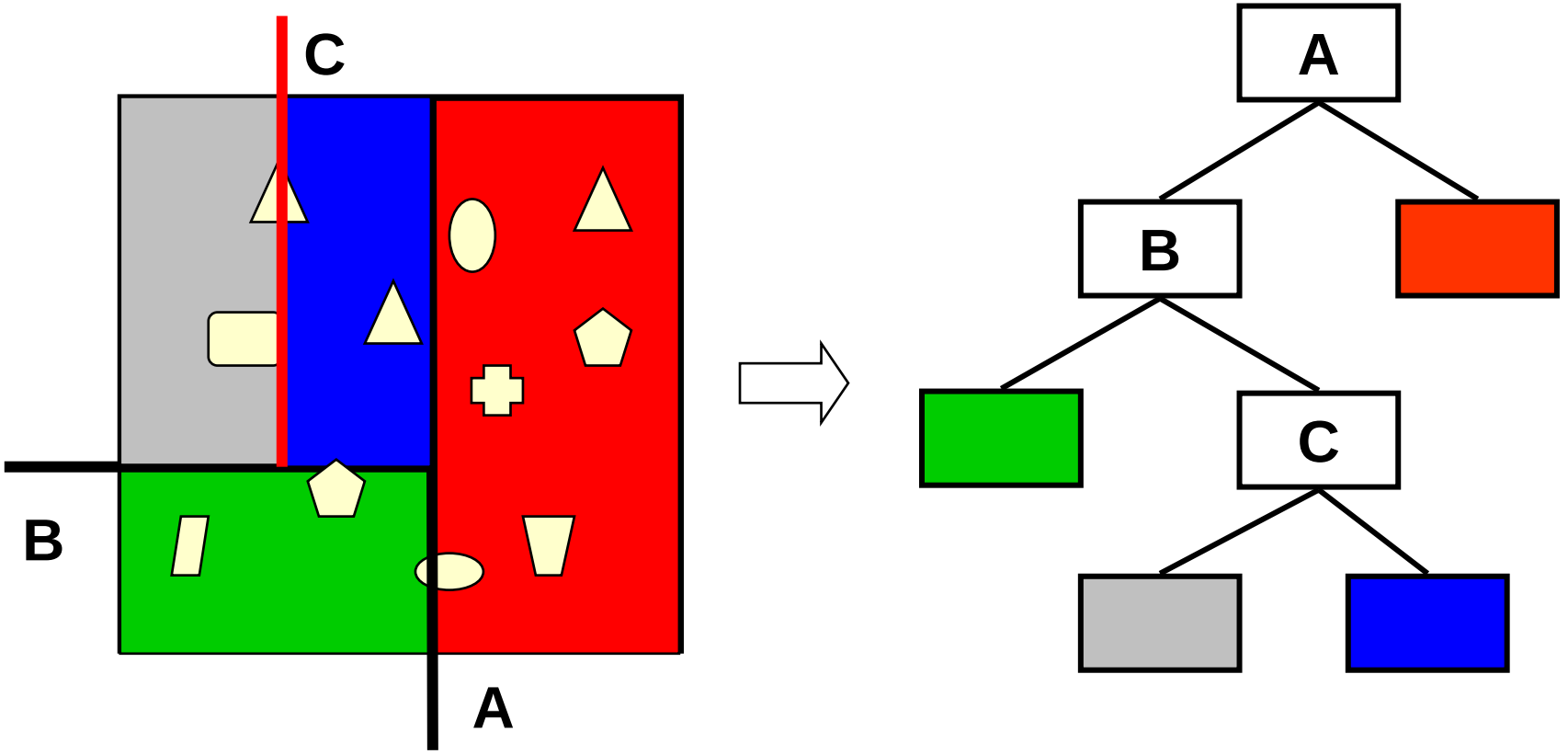


Leaf nodes correspond to unique regions in space

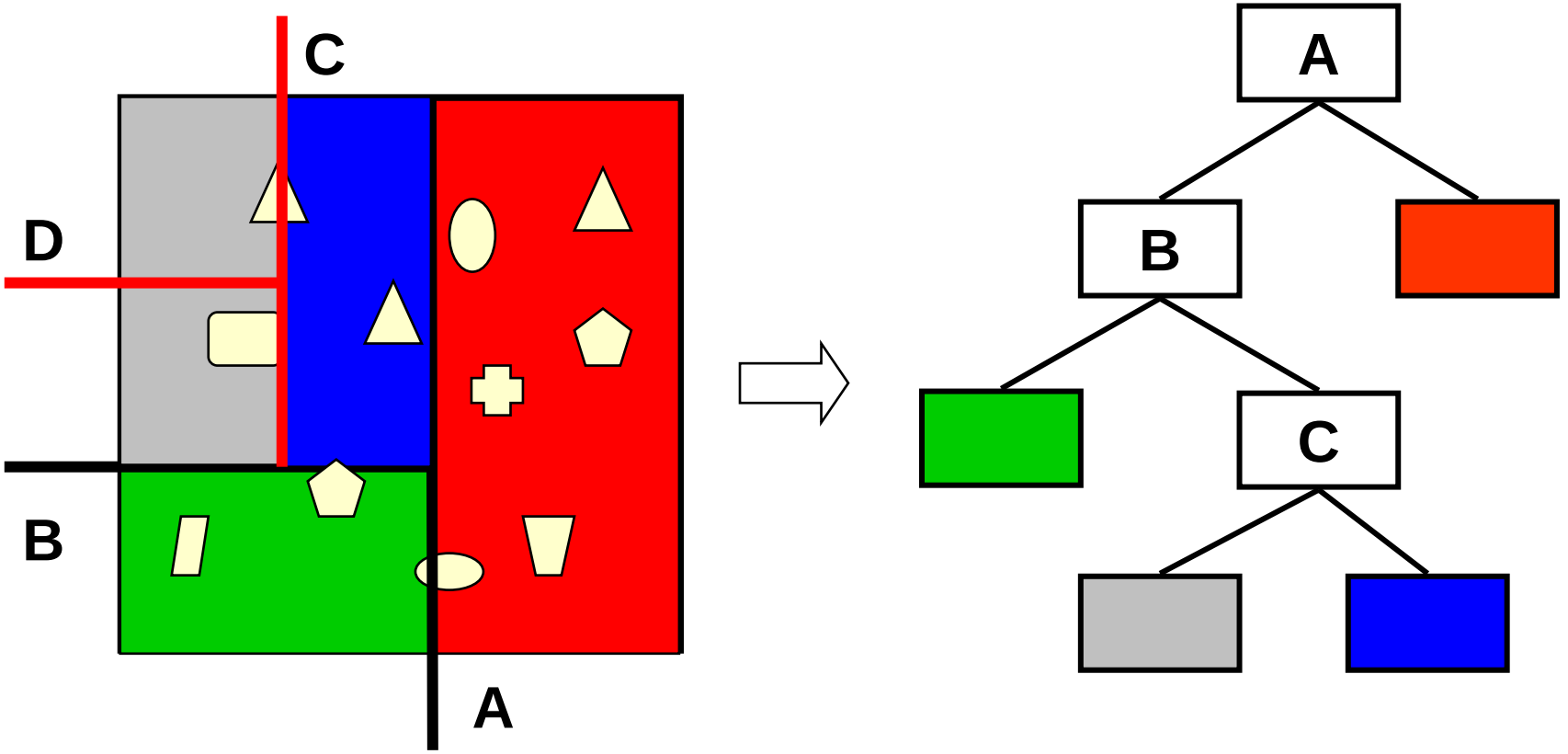
K-d tree



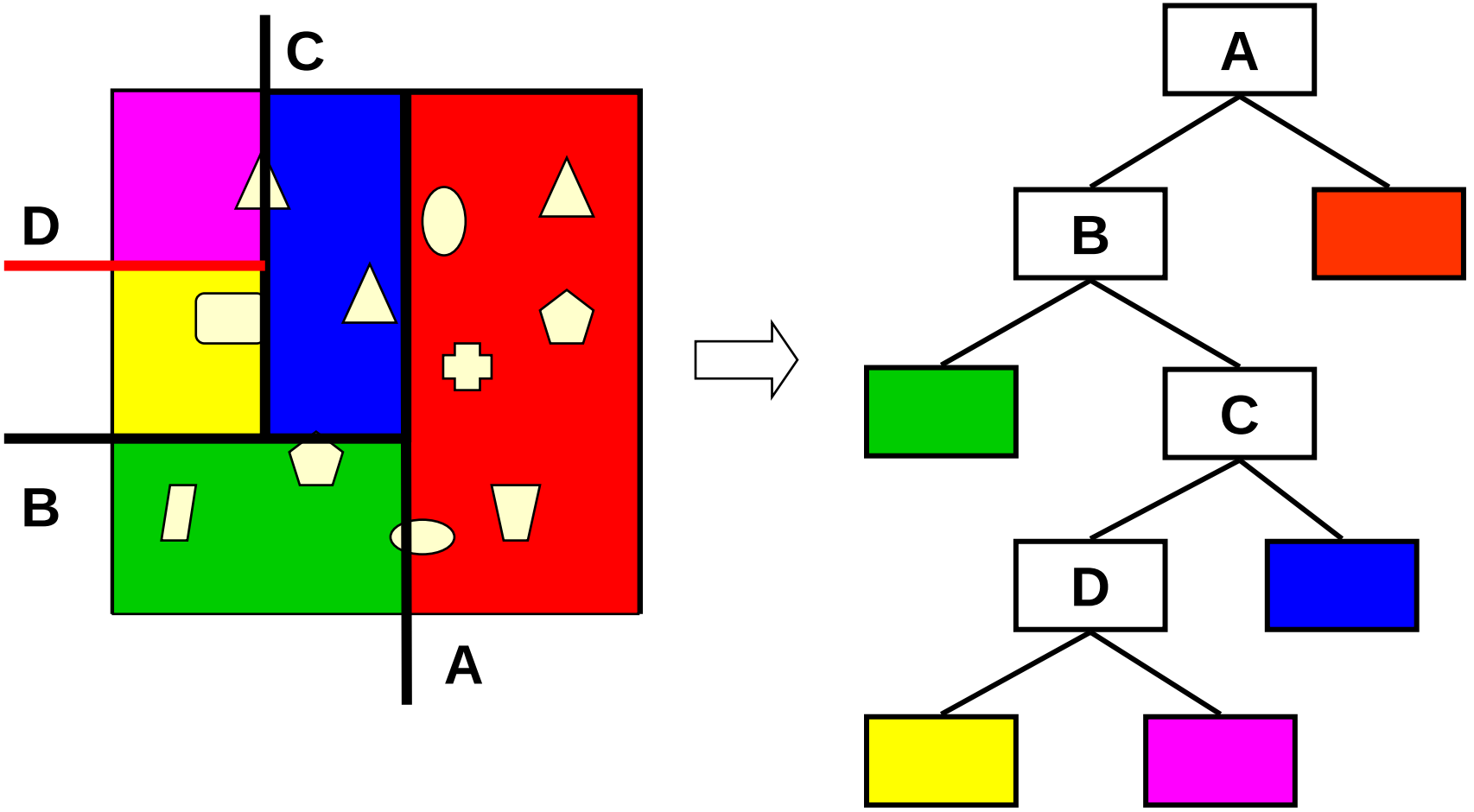
K-d tree



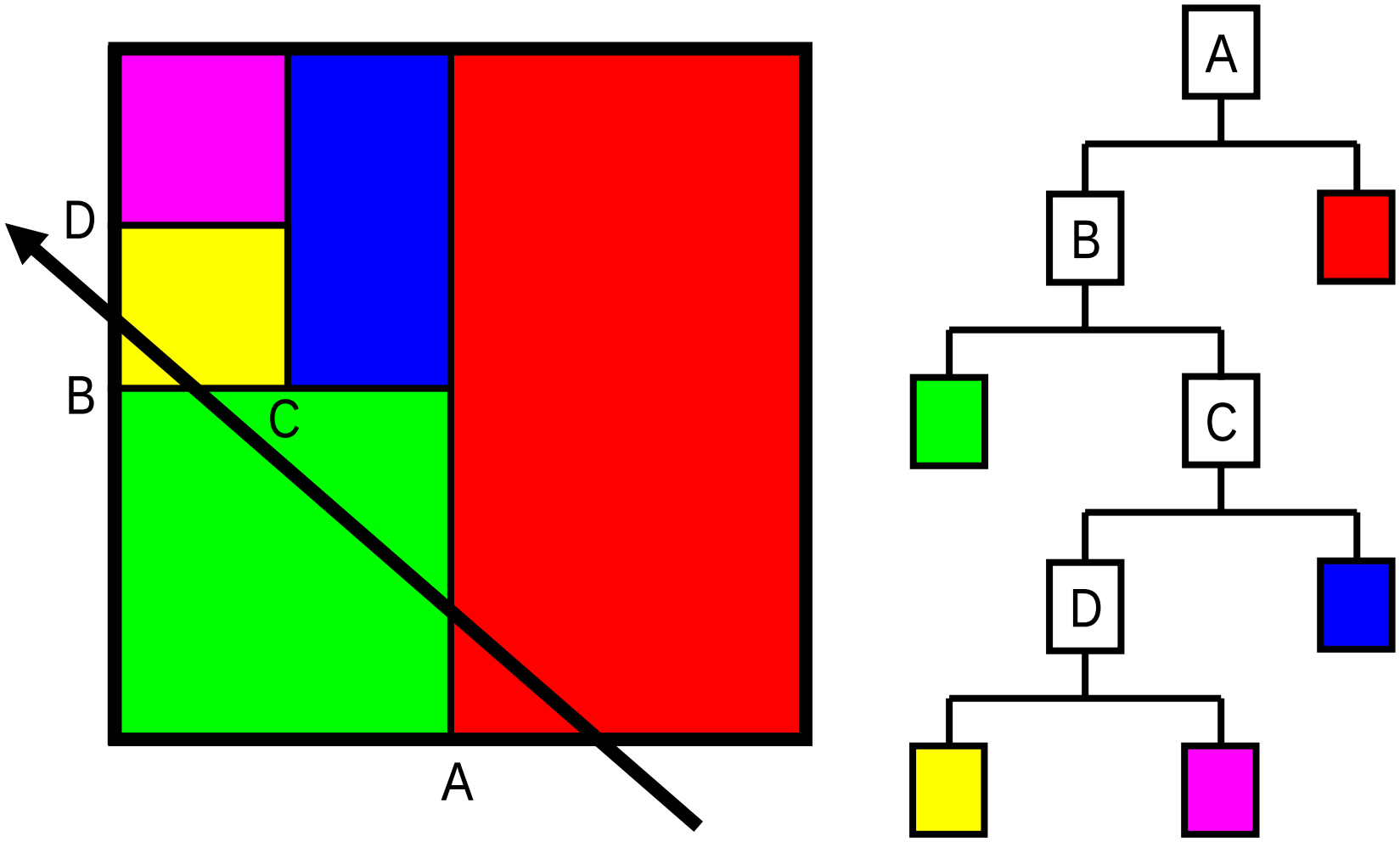
K-d tree



K-d tree

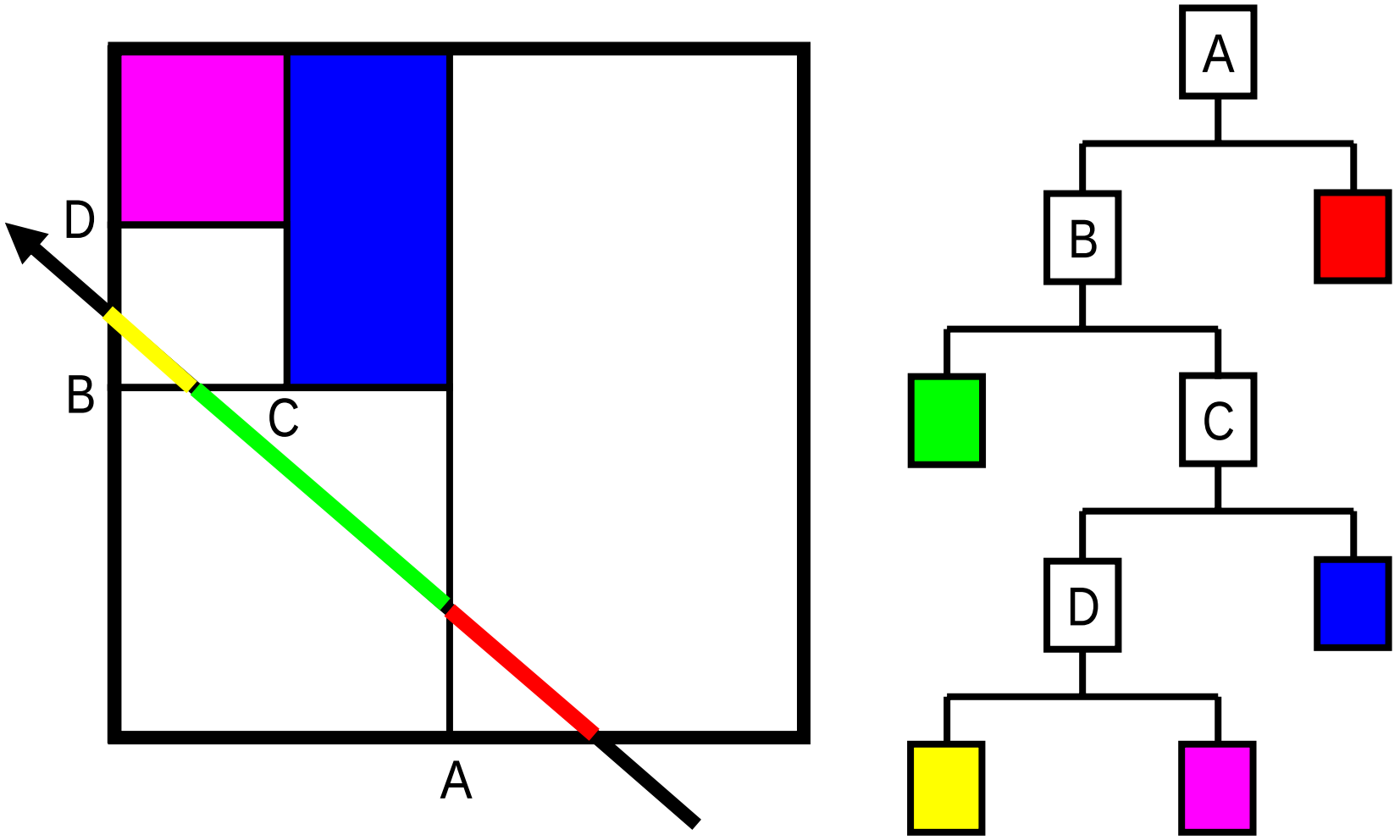


K-d tree



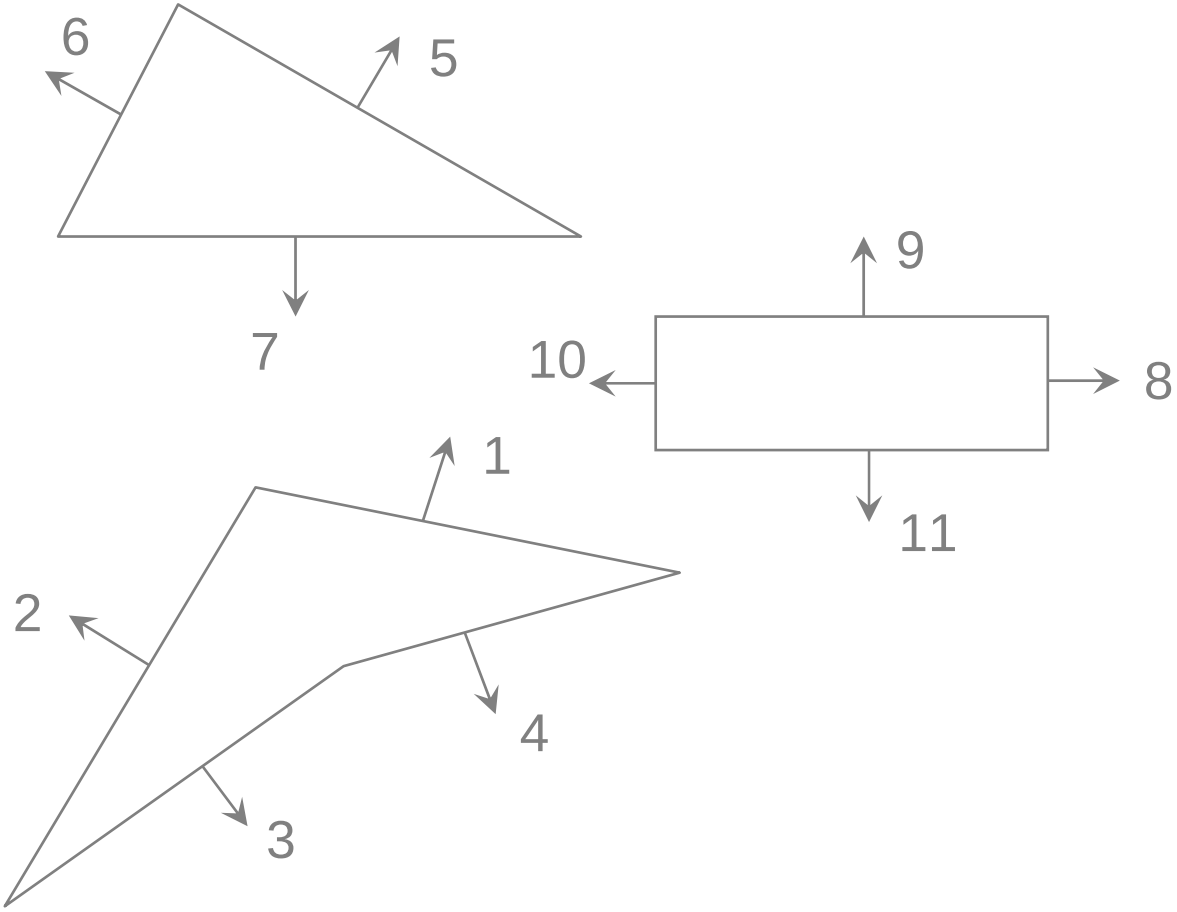
Leaf nodes correspond to unique regions in space

K-d tree traversal

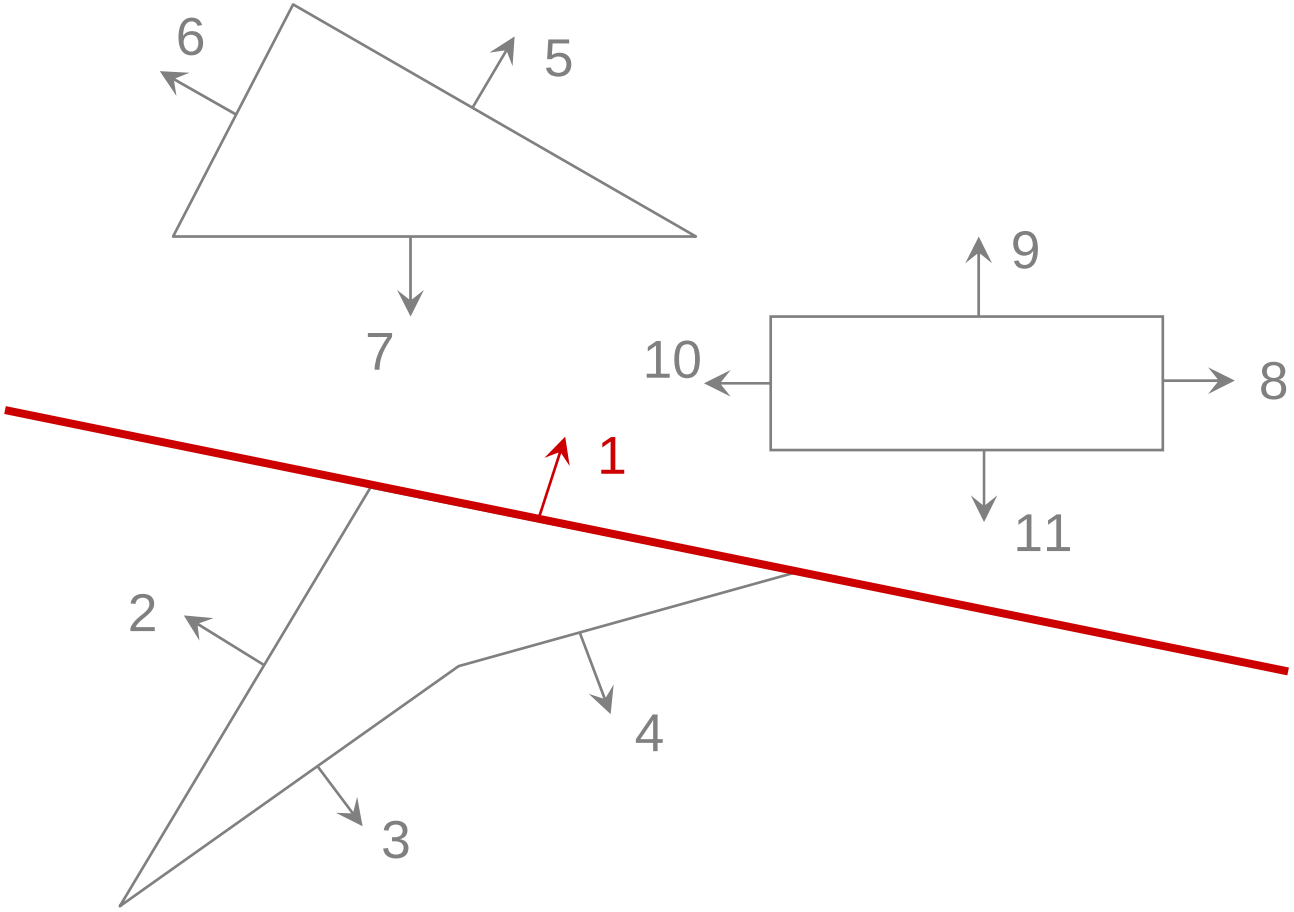


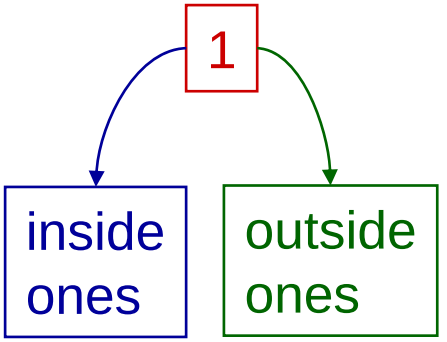
Leaf nodes correspond to unique regions in space

BSP tree

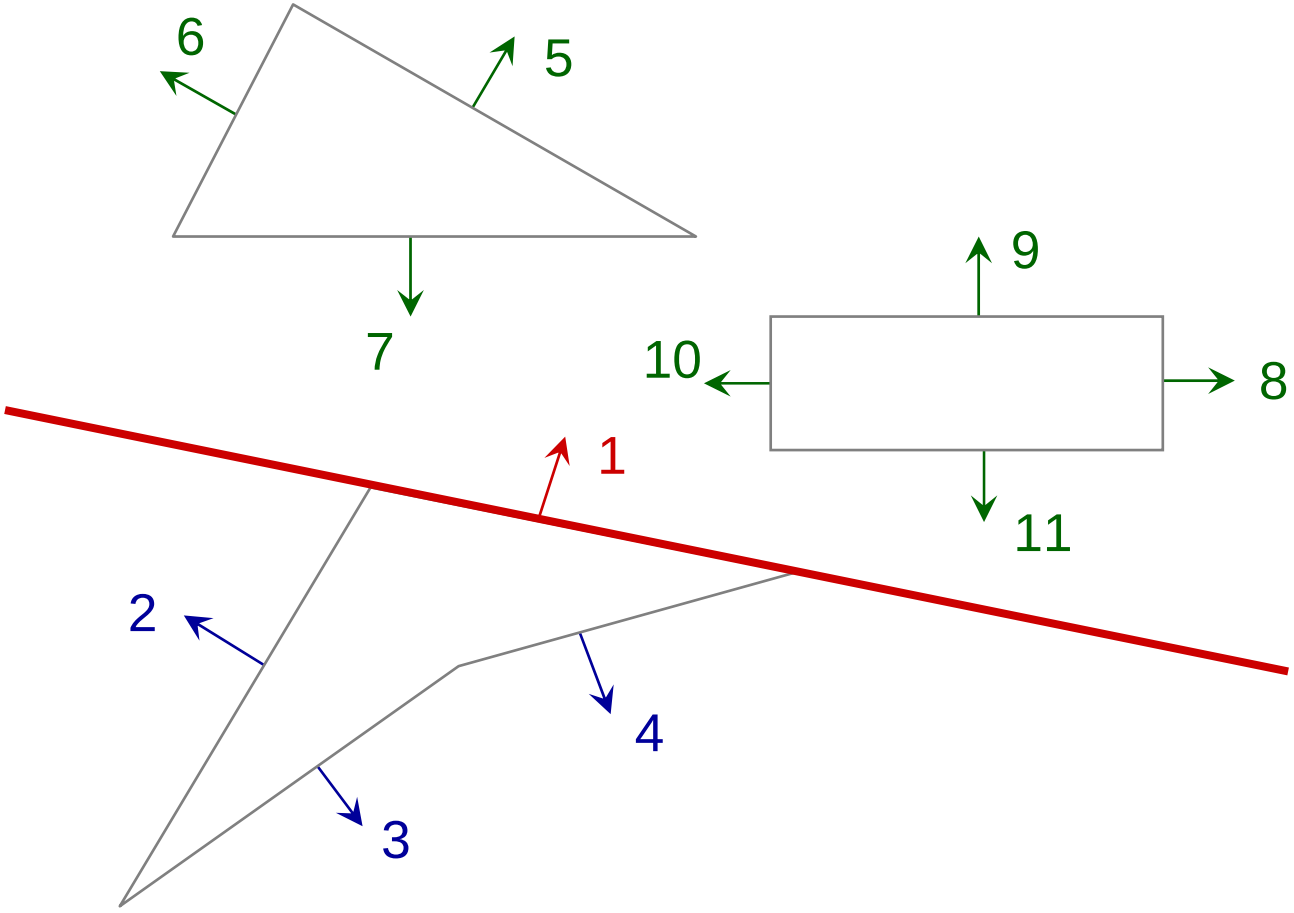


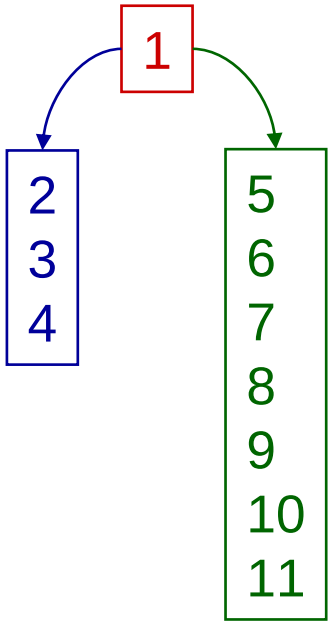
BSP tree



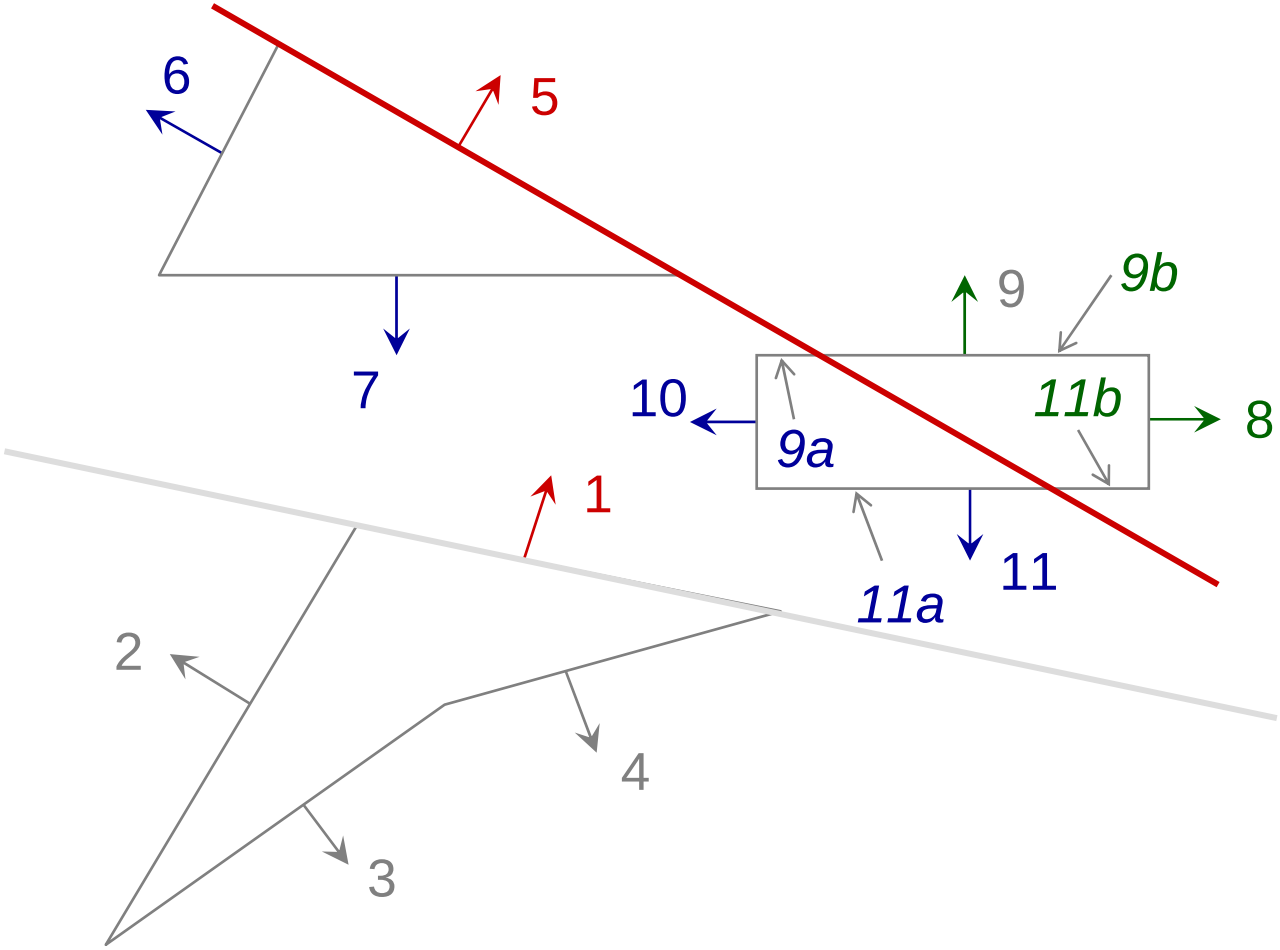


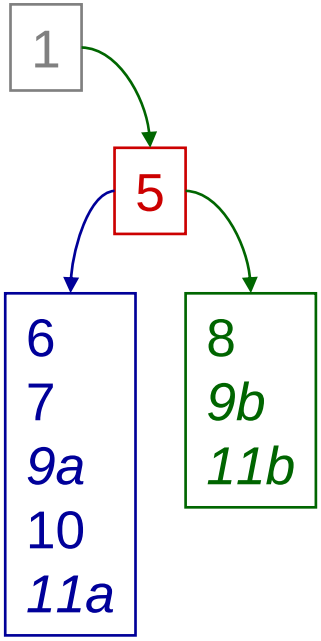
BSP tree



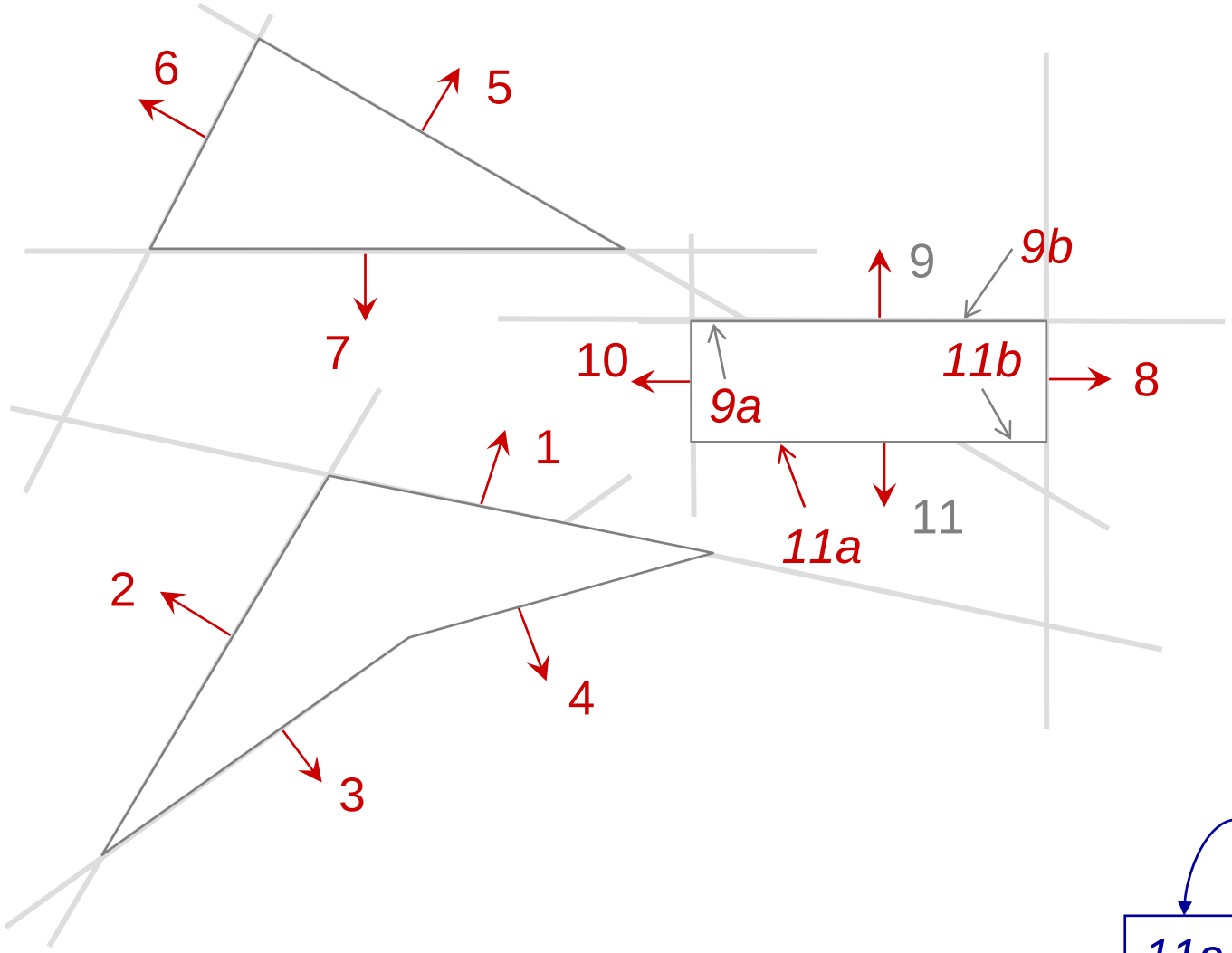


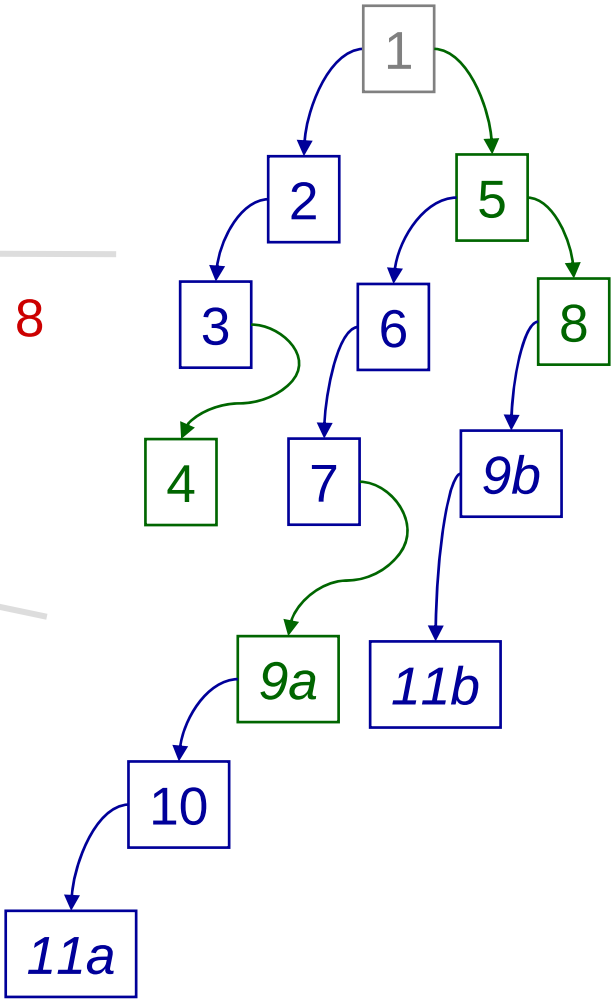
BSP tree



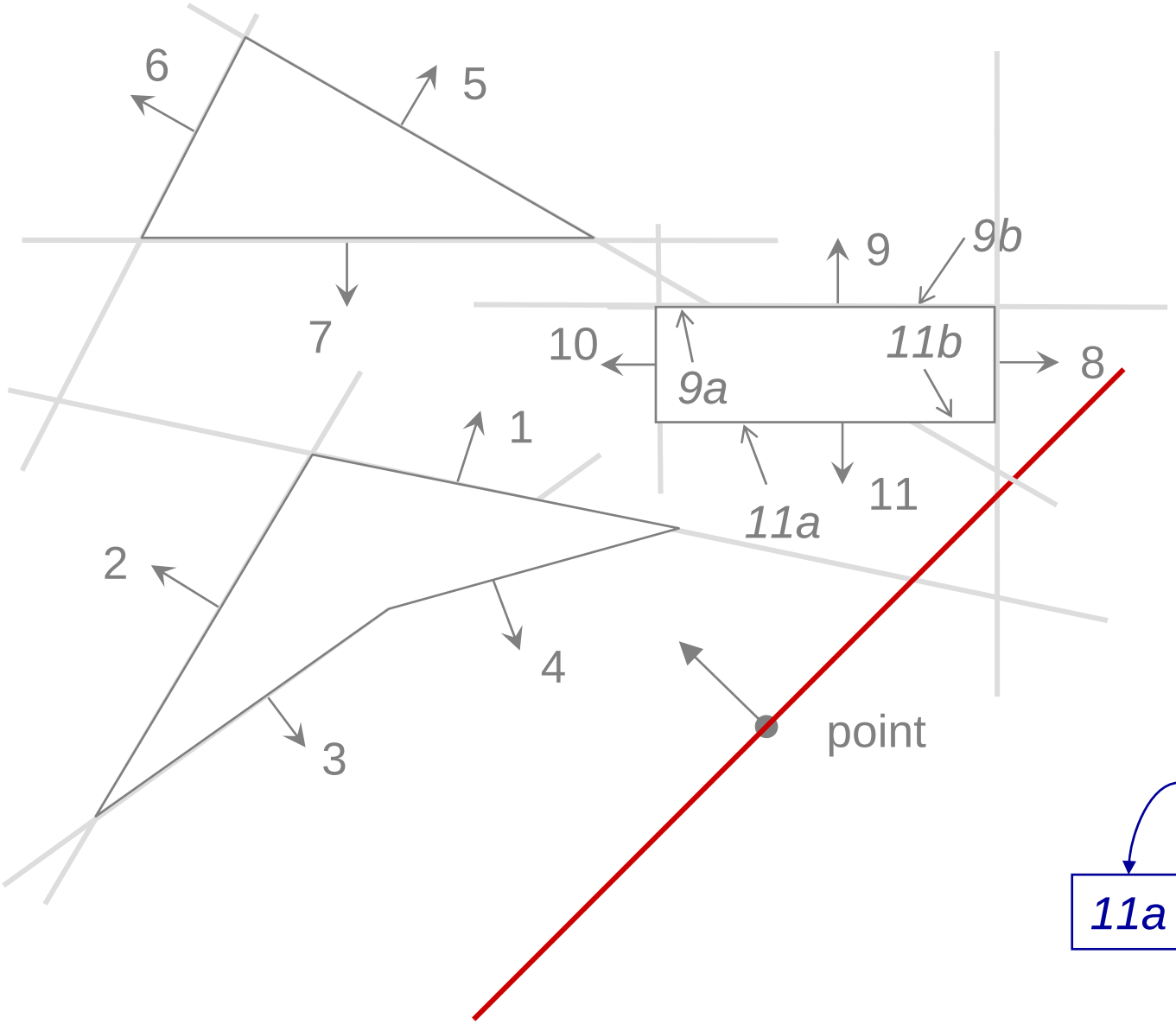


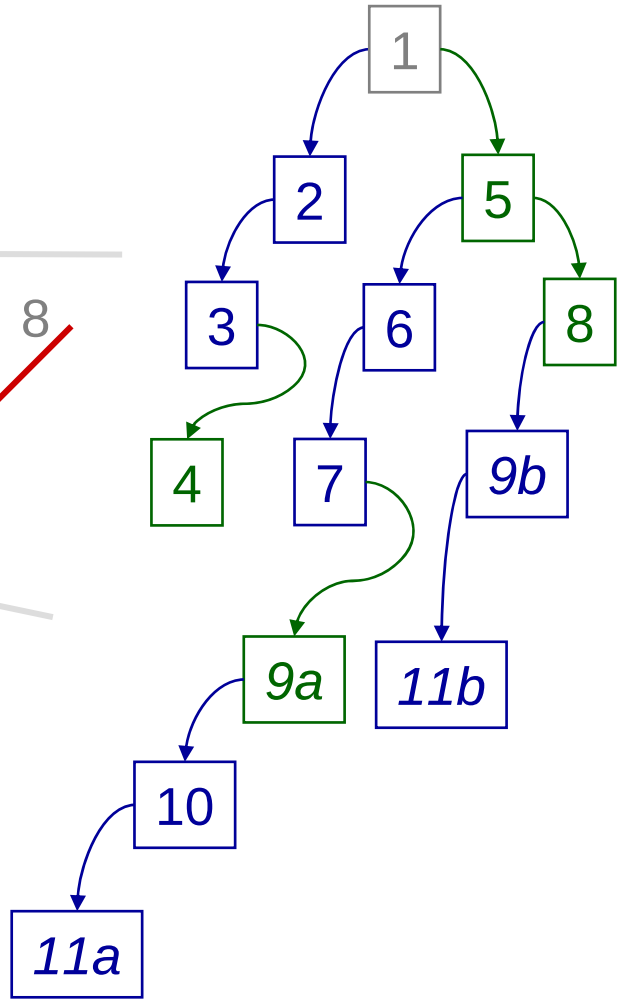
BSP tree



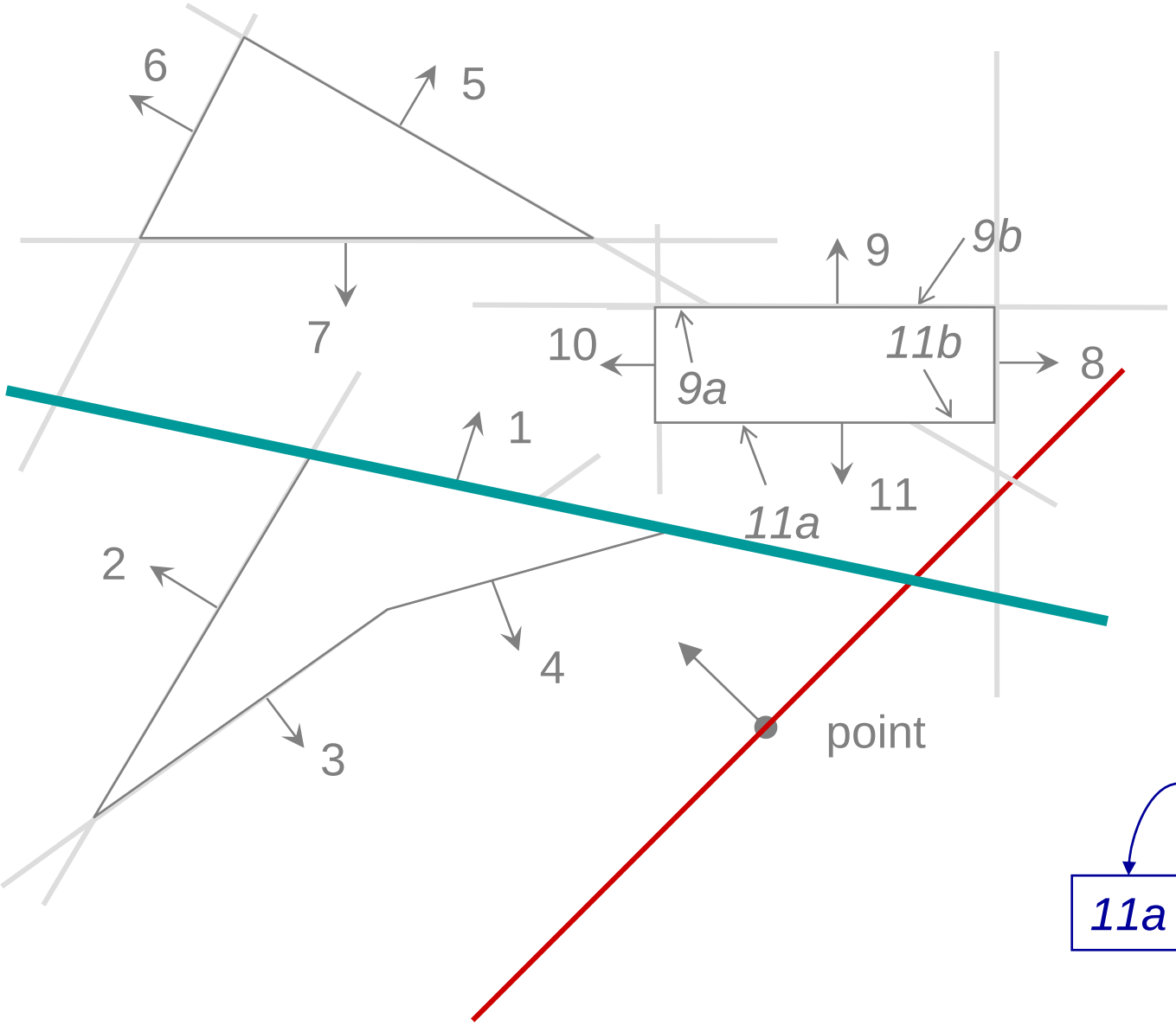


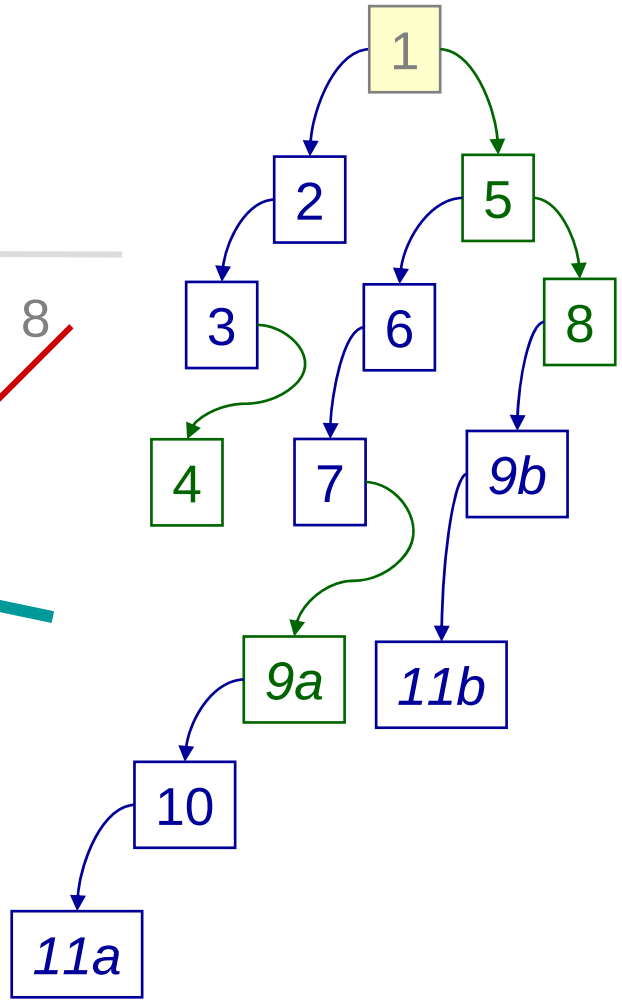
BSP tree traversal



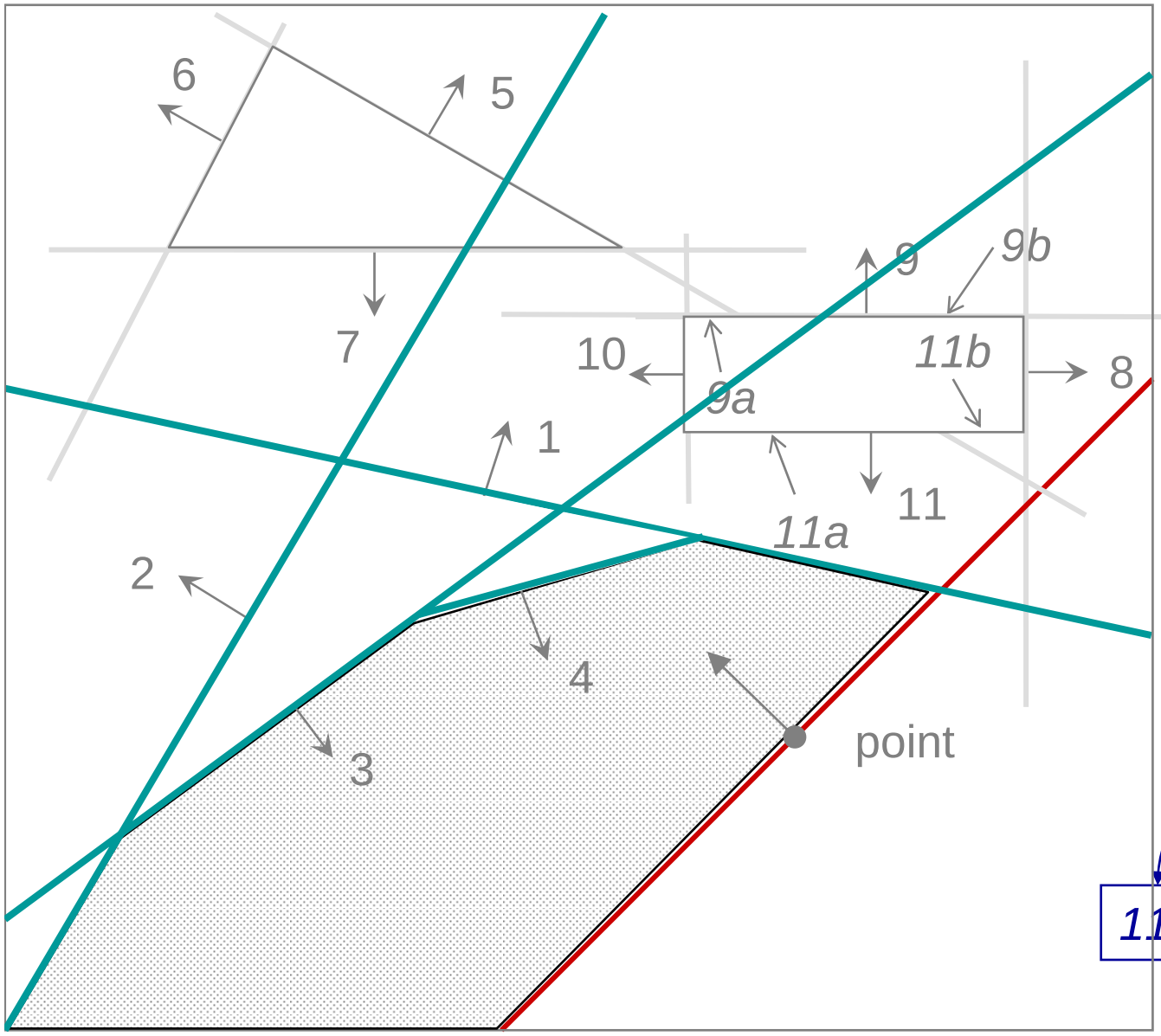


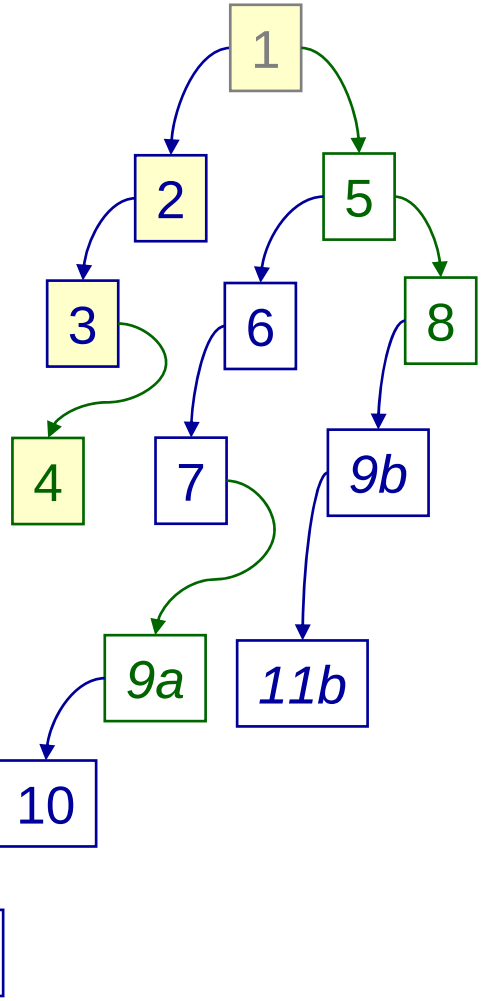
BSP tree traversal



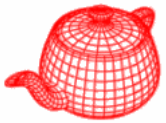


BSP tree traversal



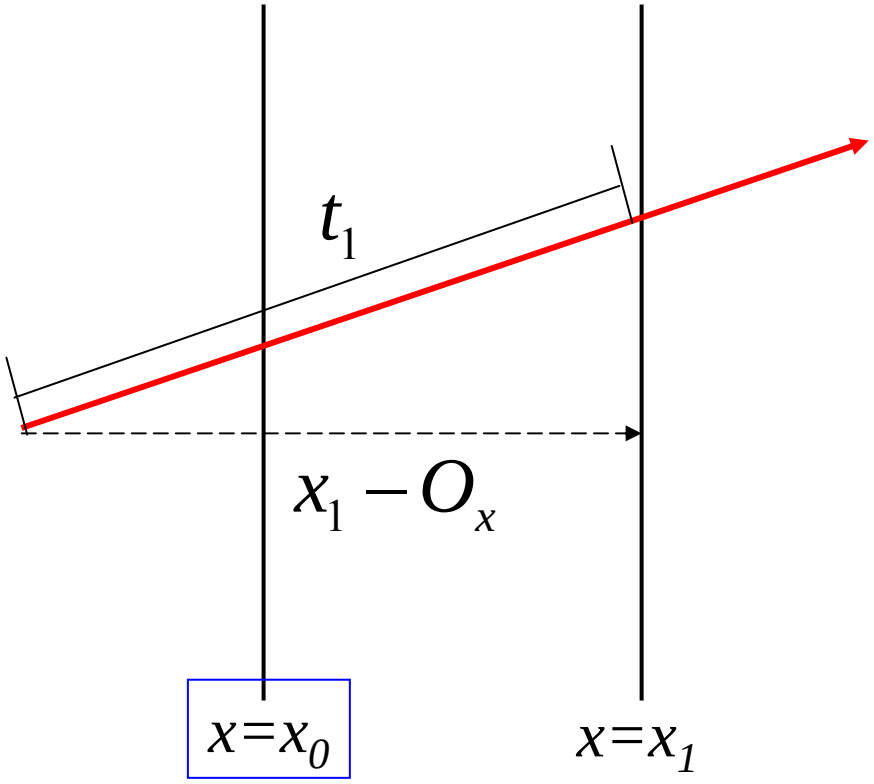


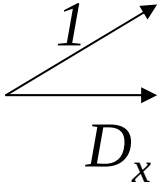
Ray-Box intersections



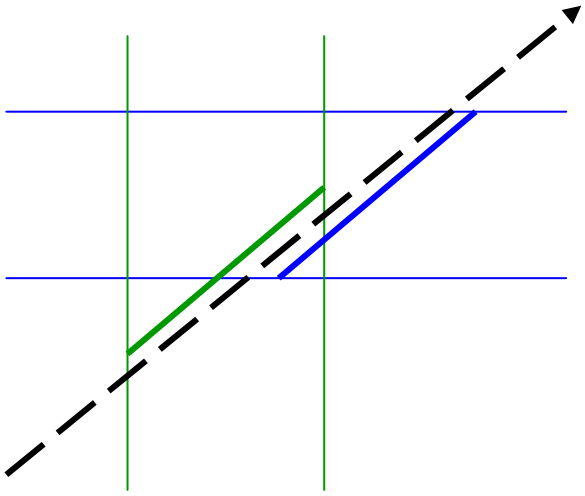
- Both **`GridAccel`** and **`KdTreeAccel`** require it
- Quick rejection, use enter and exit point to traverse the hierarchy
- AABB is the intersection of three slabs

Ray-Box intersections

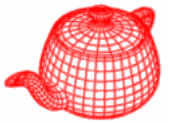




$$t_1 = \frac{x_1 - O_x}{D_x}$$



Ray-Box intersections

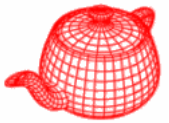


```
bool BBox::IntersectP(const Ray &ray,
                      float *hitt0, float *hitt1)
{
  float t0 = ray.mint, t1 = ray.maxt;
  for (int i = 0; i < 3; ++i) {
    float invRayDir = 1.f / ray.d[i];
    float tNear = (pMin[i] - ray.o[i]) * invRayDir;
    float tFar  = (pMax[i] - ray.o[i]) * invRayDir;

    if (tNear > tFar) swap(tNear, tFar);
    t0 = tNear > t0 ? tNear : t0;
    t1 = tFar  < t1 ? tFar  : t1;
    if (t0 > t1) return false;
  }
  if (hitt0) *hitt0 = t0;
  if (hitt1) *hitt1 = t1;
  return true;
}
```

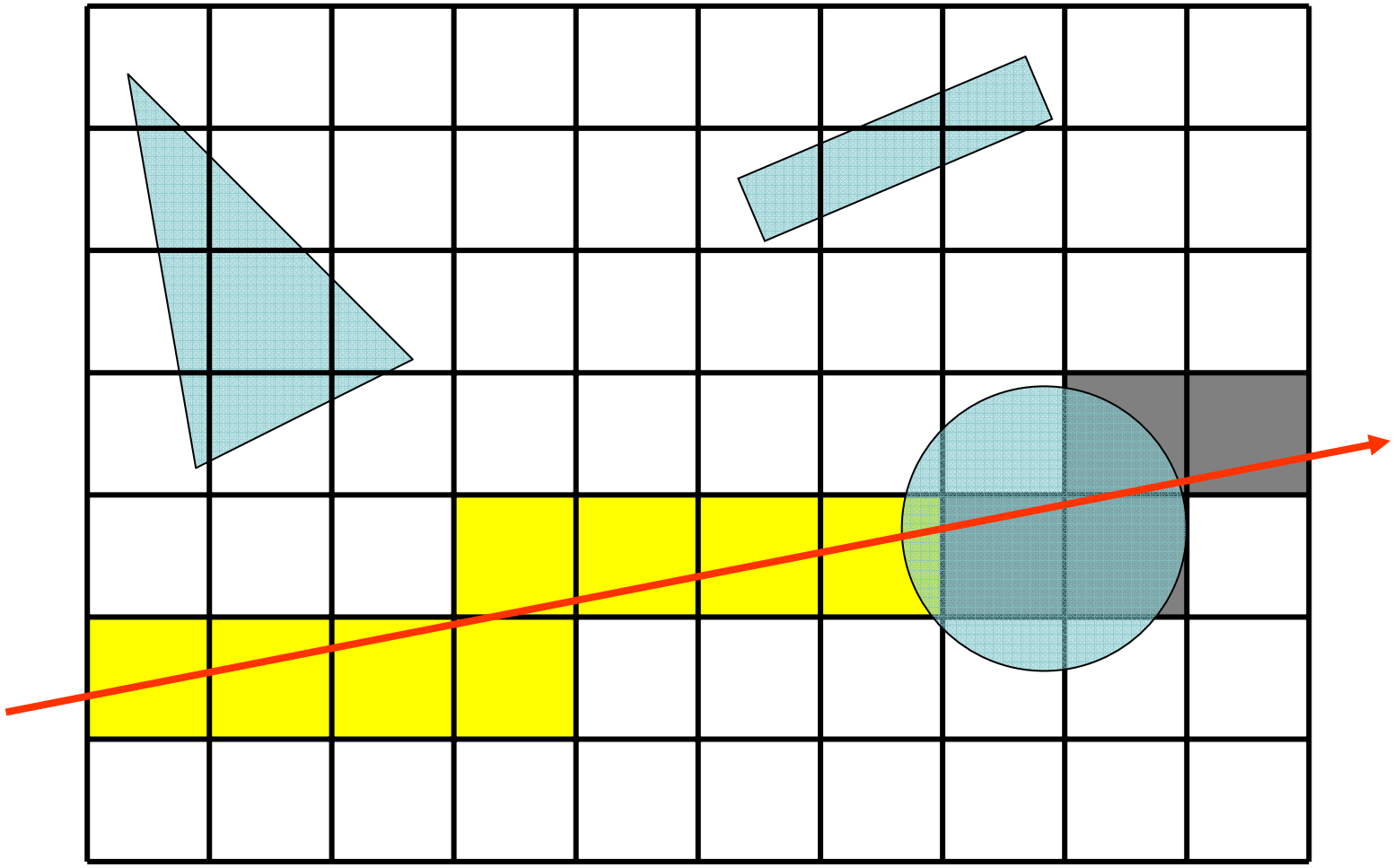
segment intersection

intersection is empty

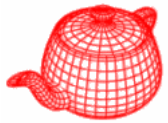
Grid accelerator



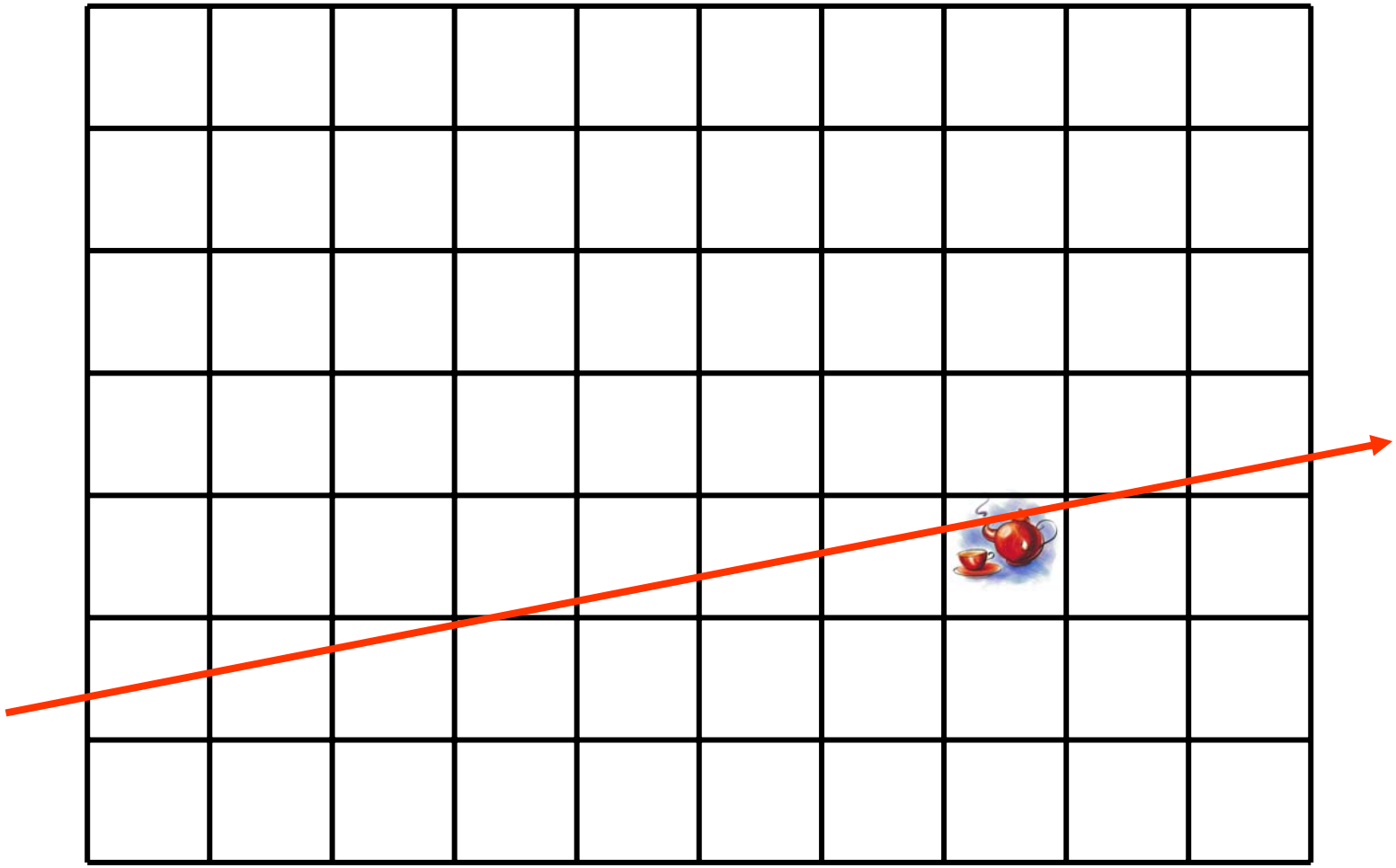
- Unifrom grid



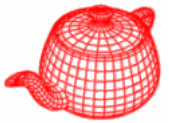
Teapot in a stadium problem



- Not adaptive to distribution of primitives.
- Have to determine the number of voxels.

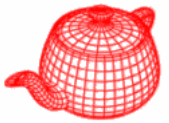


GridAccel

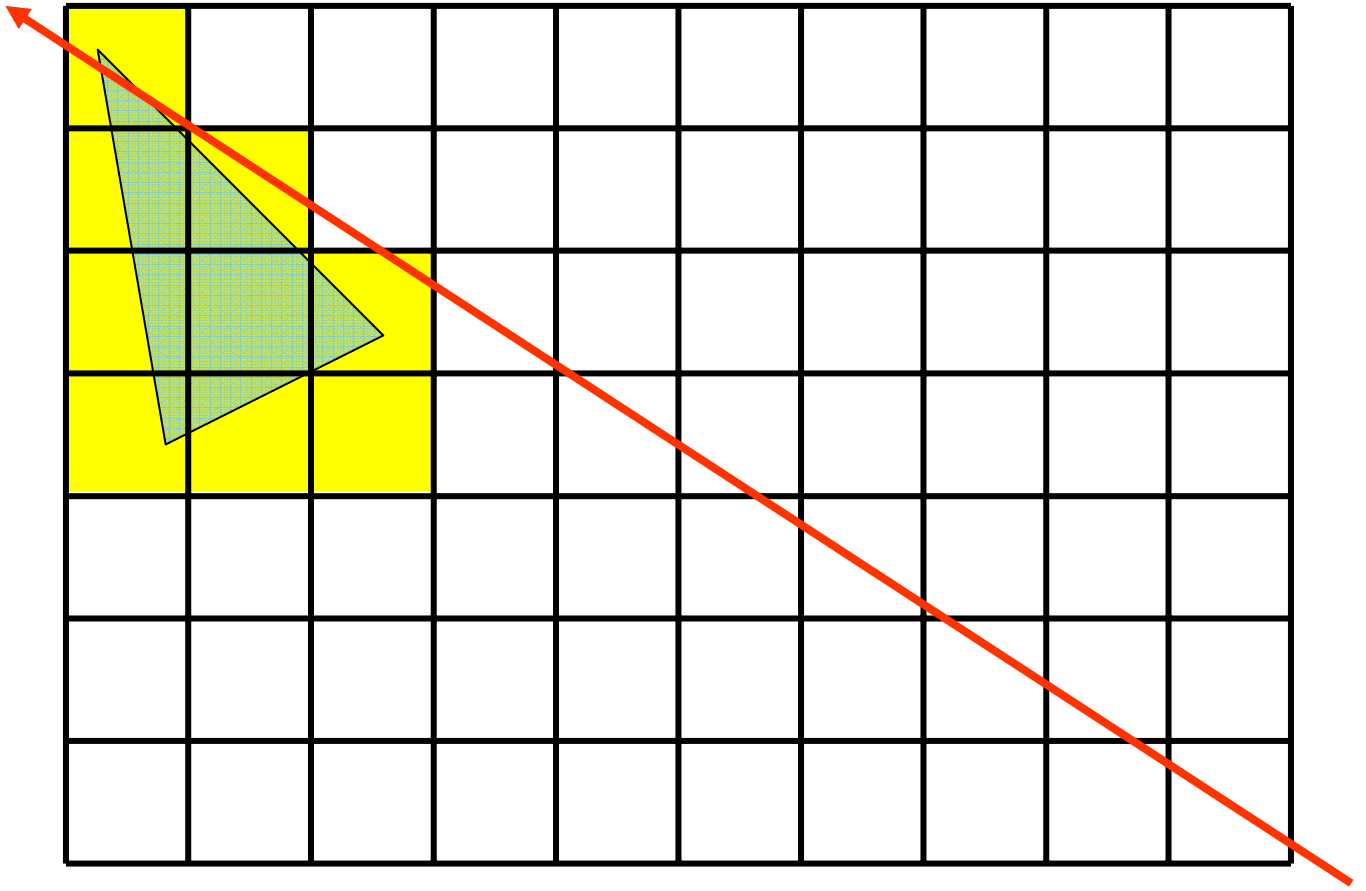


```
Class GridAccel:public Aggregate {
  <GridAccel methods>
  u_int nMailboxes;
  MailboxPrim *mailboxes;
  int NVoxels[3];
  BBox bounds;
  Vector Width, InvWidth;
  Voxel **voxels;
  ObjectArena<Voxel> voxelArena;
  static int curMailboxId;
}
```

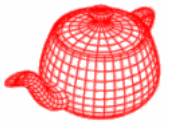
mailbox



```
struct MailboxPrim {
  Reference<Primitive> primitive;
  Int lastMailboxId;
}
```



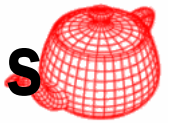
Determine number of voxels



- Too many voxels → slow traverse, large memory consumption (bad cache performance)
- Too few voxels → too many primitives in a voxel
- Let the axis with the largest extent have $3\sqrt[3]{N}$ voxels

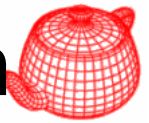
```
Vector delta = bounds.pMax - bounds.pMin;
int maxAxis=bounds.MaximumExtent();
float invMaxWidth=1.f/delta[maxAxis];
float cubeRoot=3.f*powf(float(prims.size()),1.f/3.f);
float voxelsPerUnitDist=cubeRoot * invMaxWidth;
```

Calculate voxel size and allocate voxels



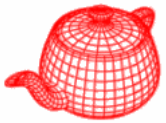
```
for (int axis=0; axis<3; ++axis) {
  NVoxels[axis]=Round2Int(delta[axis]*voxelsPerUnitDist);
  NVoxels[axis]=Clamp(NVoxels[axis], 1, 64);
}

for (int axis=0; axis<3; ++axis) {
  Width[axis]=delta[axis]/NVoxels[axis];
  InvWidth[axis]=
          (Width[axis]==0.f)?0.f:1.f/Width[axis];
}

int nVoxels = NVoxels[0] * NVoxels[1] * NVoxels[2];
voxels=(Voxel **)AllocAligned(nVoxels*sizeof(Voxel *));
memset(voxels, 0, nVoxels * sizeof(Voxel *));
```

Conversion between voxel and position

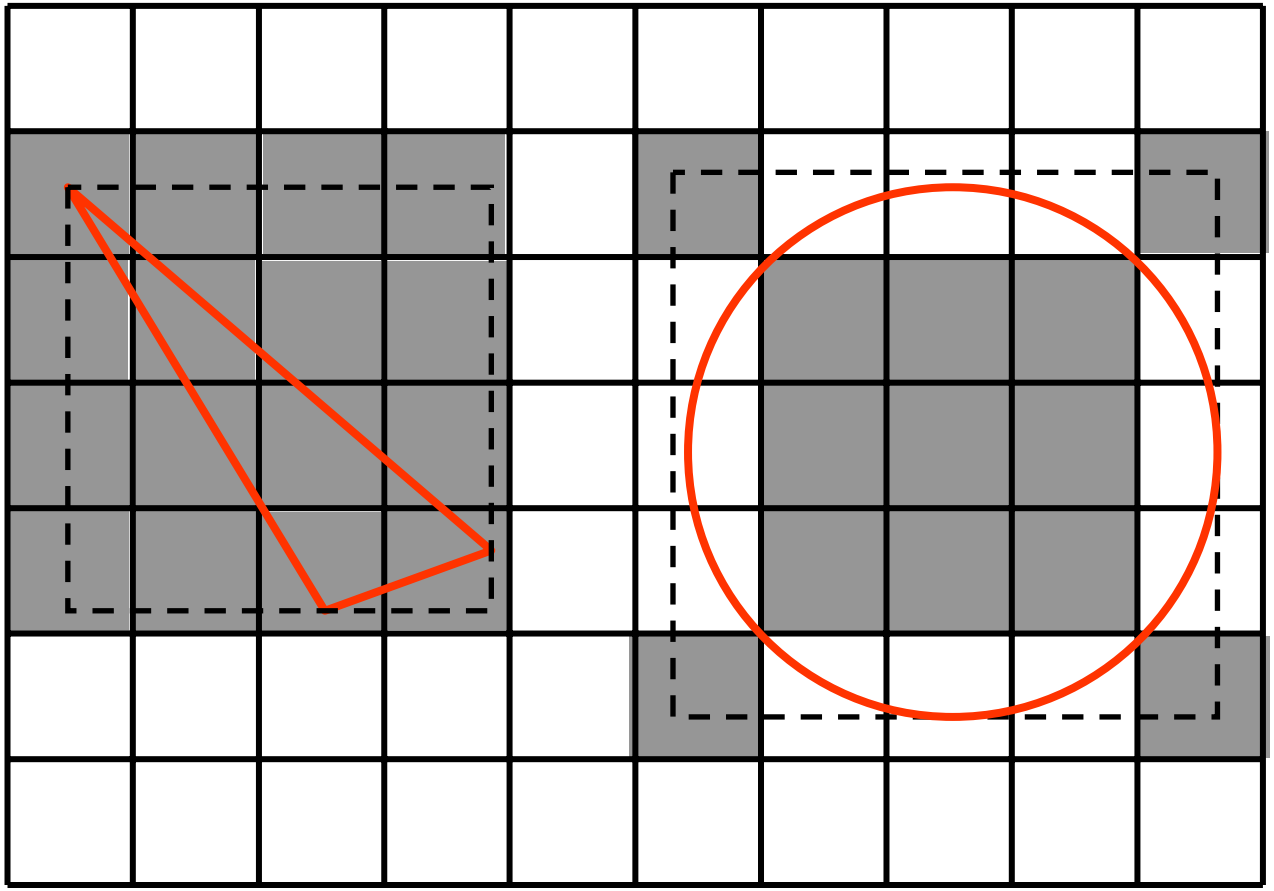


```
int PosToVoxel(const Point &P, int axis) {
  int v=Float2Int(
  (P[axis]-bounds.pMin[axis])*InvWidth[axis]);
  return Clamp(v, 0, NVoxels[axis]-1);
}

float VoxelToPos(int p, int axis) const {
  return bounds.pMin[axis]+p*Width[axis];
}
Point VoxelToPos(int x, int y, int z) const {
  return bounds.pMin+
         Vector(x*Width[0], y*Width[1], z*Width[2]);
}

inline int Offset(int x, int y, int z) {
  return z*NVoxels[0]*NVoxels[1] + y*NVoxels[0] + x;
}
```

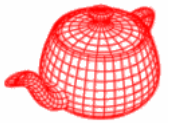
Add primitives into voxels



```
for (u_int i=0; i<prims.size(); ++i) {
  <Find voxel extent of primitive>
  <Add primitive to overlapping voxels>
}
```



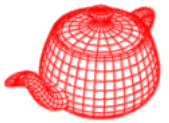
Voxel structure



```
struct Voxel {
  <Voxel methods>
  union {
    MailboxPrim *onePrimitive;
    MailboxPrim **primitives;
  };
  u_int allCanIntersect:1;
  u_int nPrimitives:31;
}
```

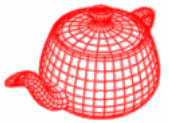
Packed into 64 bits

`GridAccel` traversal



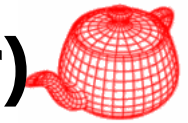
```
bool GridAccel::Intersect(
        Ray &ray, Intersection *isect) {
  <Check ray against overall grid bounds>
  <Get ray mailbox id>
  <Set up 3D DDA for ray>
  <Walk ray through voxel grid>
}
```

Check against overall bound

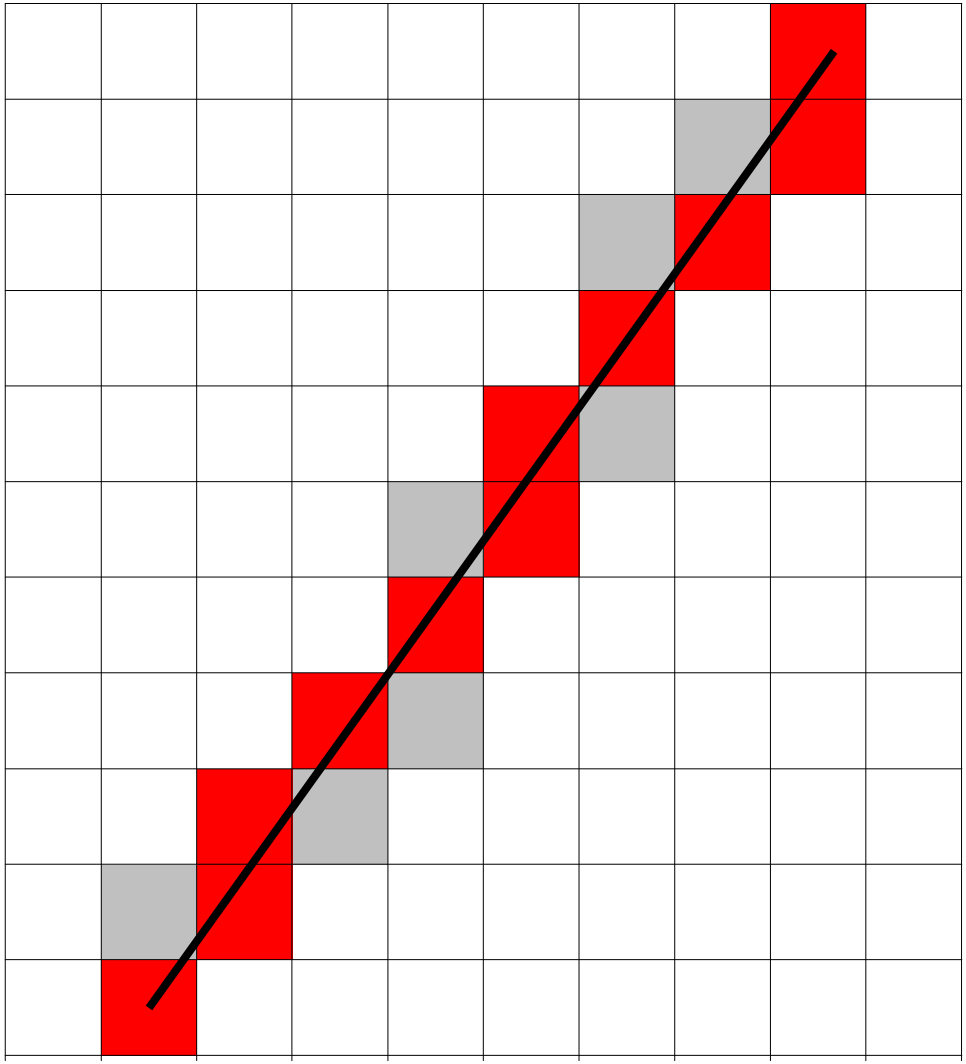


```
float rayT;
if (bounds.Inside(ray(ray.mint)))
  rayT = ray.mint;
else if (!bounds.IntersectP(ray, &rayT))
  return false;
Point gridIntersect = ray(rayT);
```

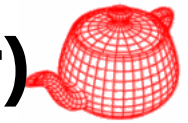
Set up 3D DDA (Digital Differential Analyzer)

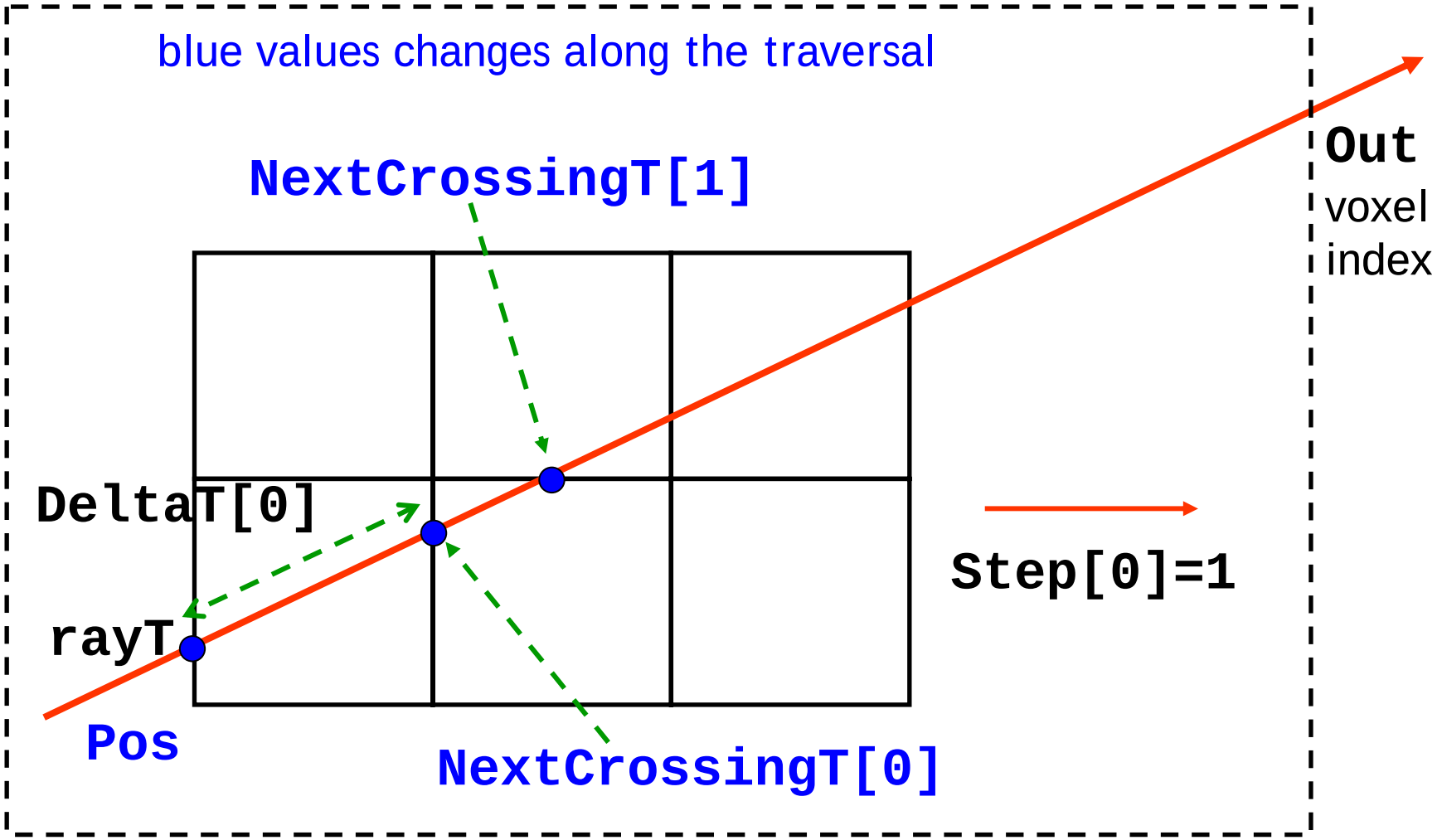


- Similar to Bresenhm's line drawing algorithm



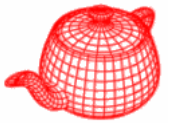
Set up 3D DDA (Digital Differential Analyzer)



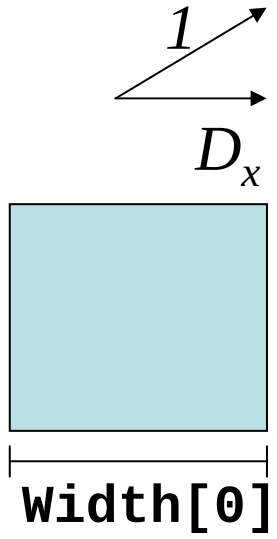


delta: the distance change when voxel changes 1 in that direction

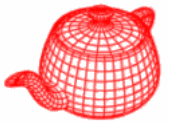
Set up 3D DDA



```
for (int axis=0; axis<3; ++axis) {
  Pos[axis]=PosToVoxel(gridIntersect, axis);
  if (ray.d[axis]>=0) {
    NextCrossingT[axis] = rayT+
    (VoxelToPos(Pos[axis]+1,axis)-gridIntersect[axis])
    /ray.d[axis];

    DeltaT[axis] = Width[axis] / ray.d[axis];
    Step[axis] = 1;
    Out[axis] = NVoxels[axis];
  } else {
    ...
    Step[axis] = -1;
    Out[axis] = -1;
  }
}
```

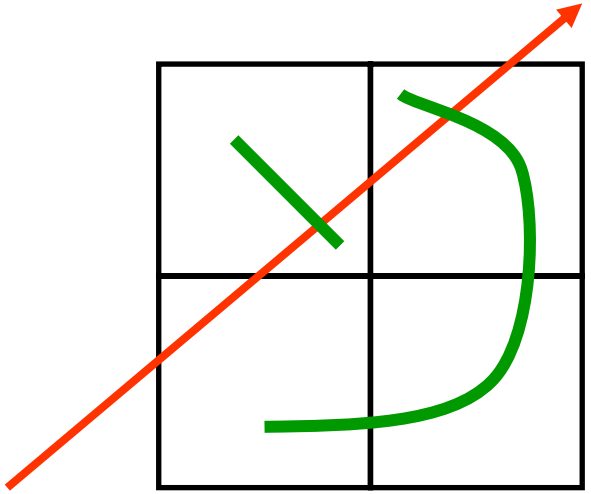


Walk through grid

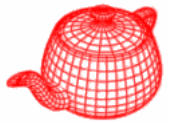


```
for (;;) {
  *voxel=voxels[Offset(Pos[0],Pos[1],Pos[2])];
  if (voxel != NULL)
    hitSomething |=
            voxel->Intersect(ray,isect,rayId);
  <Advance to next voxel>
}
return hitSomething;
```

Do not return; cut tmax instead

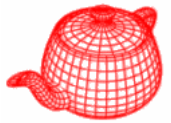


Advance to next voxel



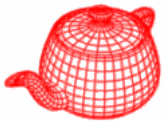
```
int bits=((NextCrossingT[0]<NextCrossingT[1])<<2) +
         ((NextCrossingT[0]<NextCrossingT[2])<<1) +
         ((NextCrossingT[1]<NextCrossingT[2]));
const int cmpToAxis[8] = { 2, 1, 2, 1, 2, 2, 0, 0 };

int stepAxis=cmpToAxis[bits];

if (ray.maxt < NextCrossingT[stepAxis]) break;

Pos[stepAxis]+=Step[stepAxis];

if (Pos[stepAxis] == Out[stepAxis]) break;

NextCrossingT[stepAxis] += DeltaT[stepAxis];
```

conditions



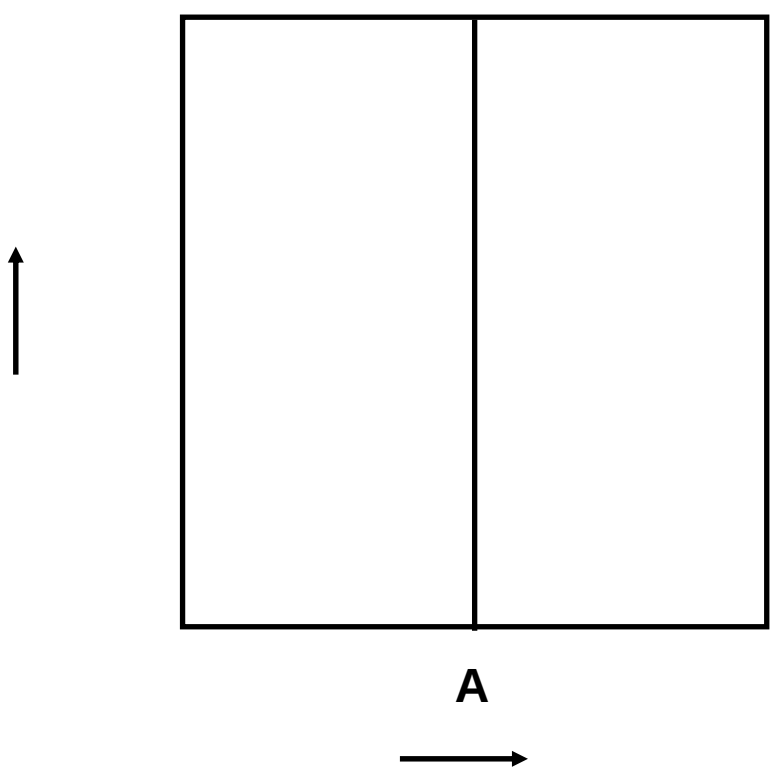
x<y	x<z	y<z		
0	0	0	x≥y≥z	2
0	0	1	x≥z>y	1
0	1	0	-	
0	1	1	z>x≥y	1
1	0	0	y>x≥z	2
1	0	1	-	
1	1	0	y≥z>x	0
1	1	1	z>y>x	0

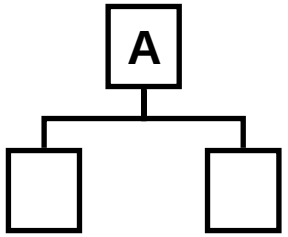
KD-Tree accelerator



- Non-uniform space subdivision (for example, kd-tree and octree) is better than uniform grid if the scene is irregularly distributed.

Spatial hierarchies

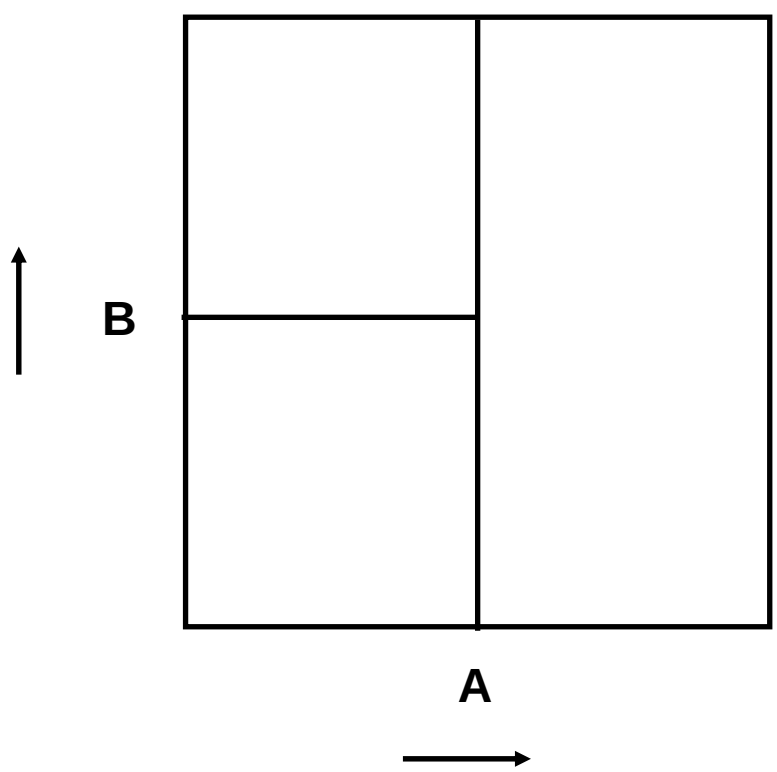


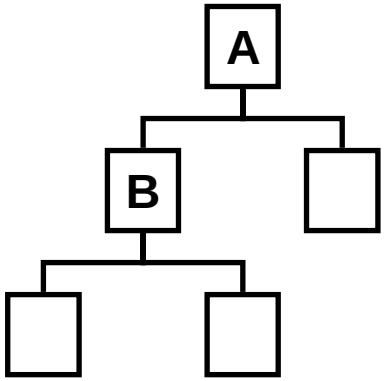


Letters correspond to planes (A)

Point Location by recursive search

Spatial Hierarchies

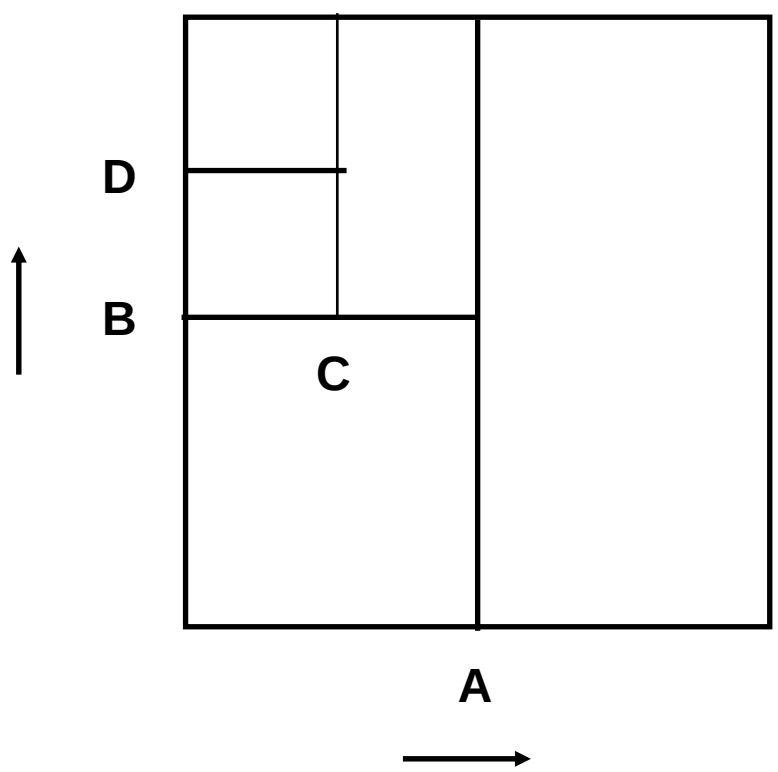


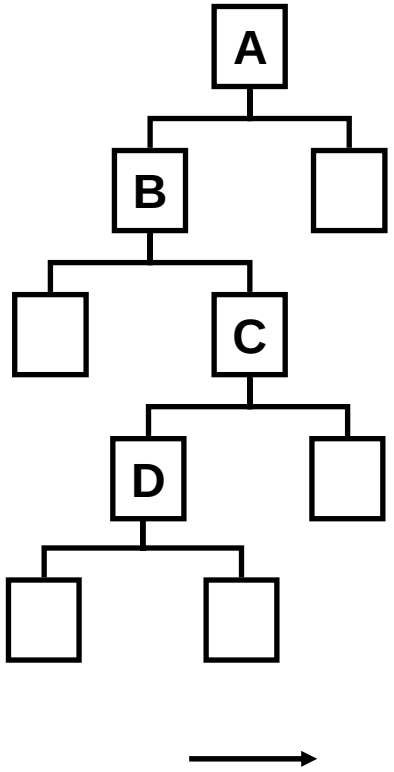


Letters correspond to planes (A, B)

Point Location by recursive search

Spatial Hierarchies

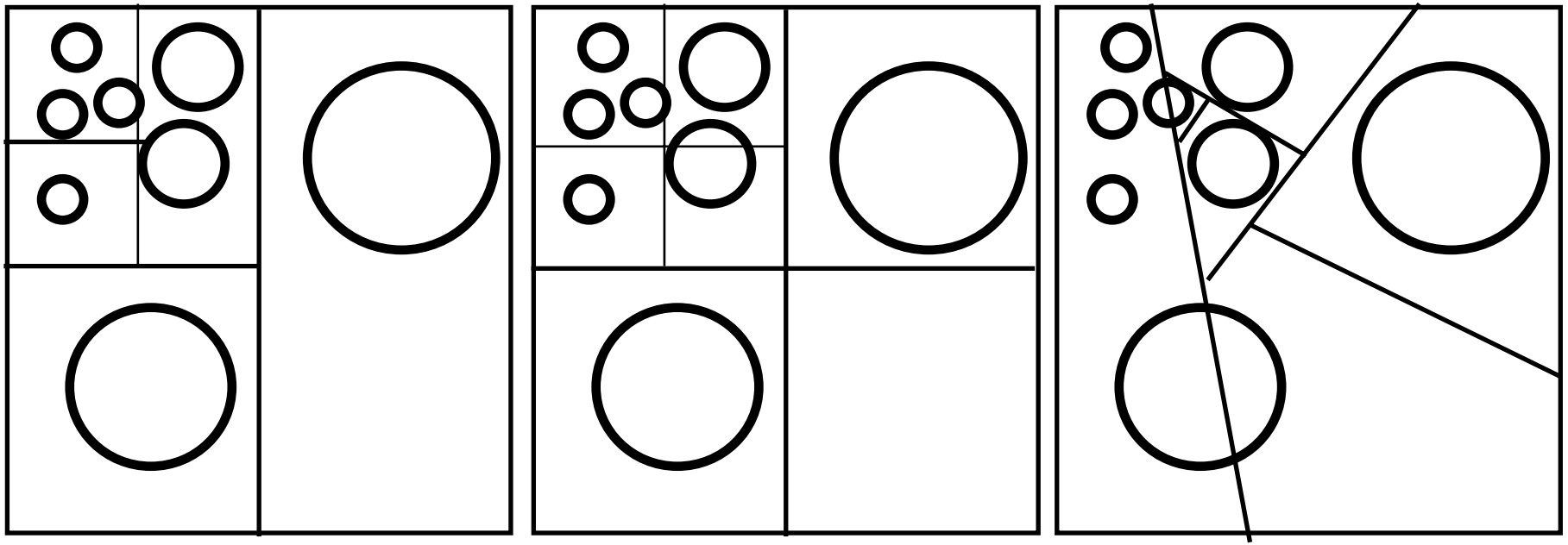




Letters correspond to planes (A, B, C, D)

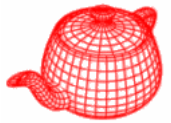
Point Location by recursive search

Variations



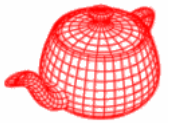
kd-tree octree bsp-tree

“Hack” kd-tree building



- Split Axis
  - Round-robin; largest extent
- Split Location
  - Middle of extent; median of geometry (balanced tree)
- Termination
  - Target # of primitives, limited tree depth
- All of these techniques stink.

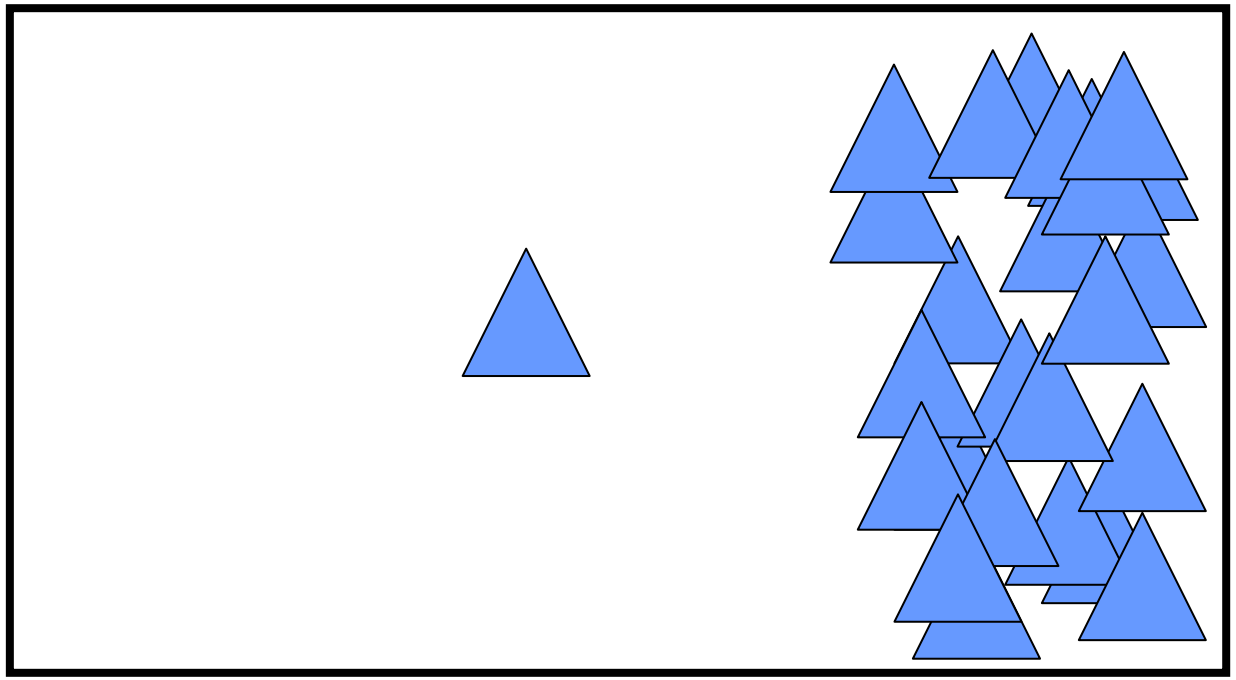
Building good kd-trees



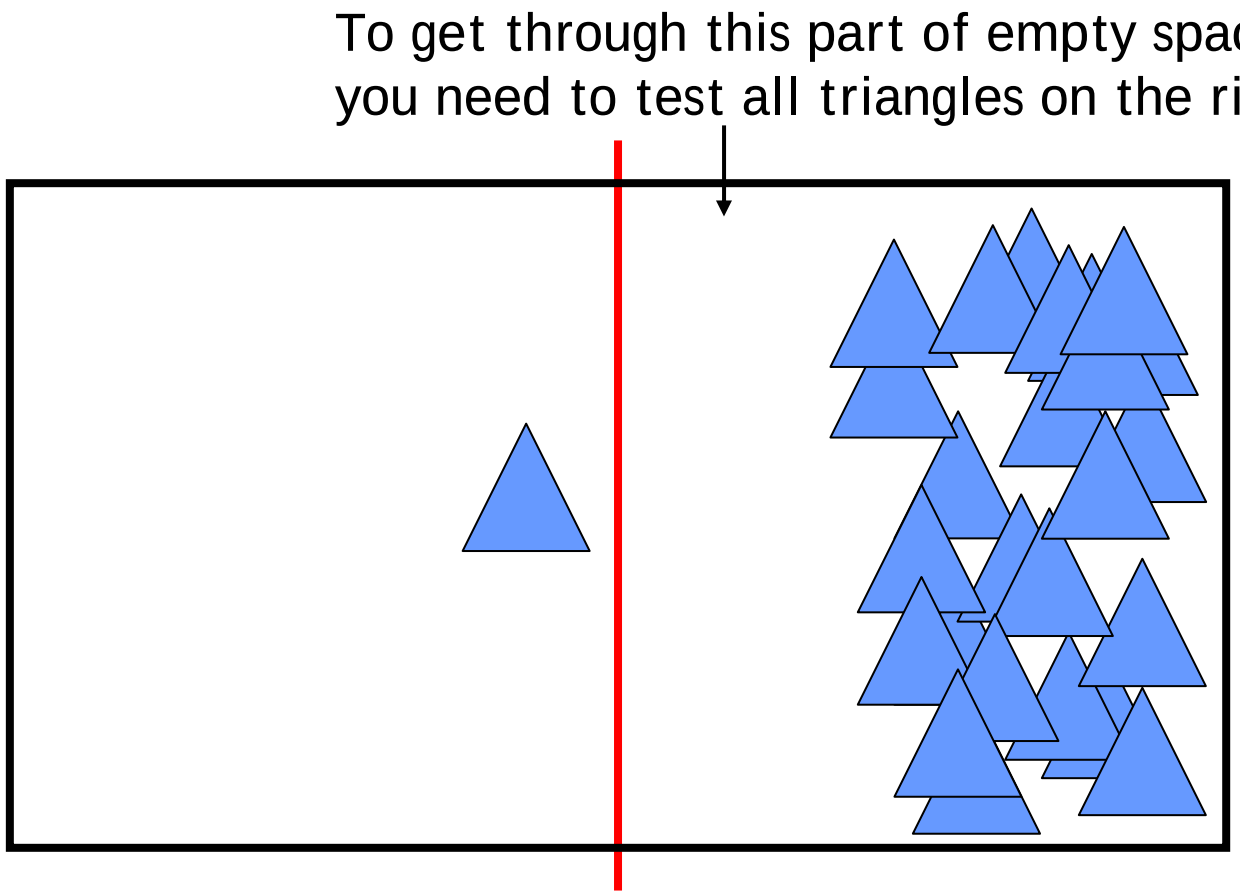
- What split do we really want?
  - Clever Idea: The one that makes ray tracing cheap
  - Write down an expression of cost and minimize it
  - Greedy Cost Optimization
- What is the cost of tracing a ray through a cell?

Cost(cell) = C_trav + Prob(hit L) * Cost(L) + Prob(hit R) * Cost(R)

Splitting with cost in mind

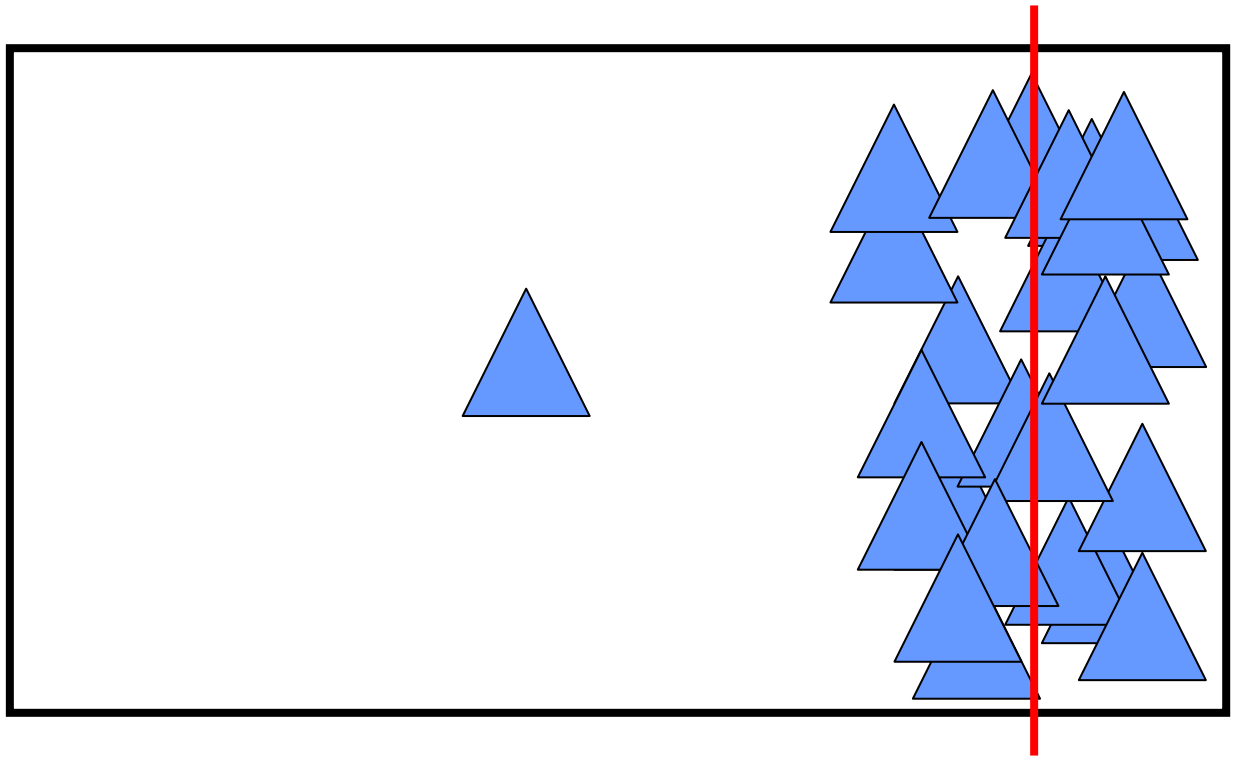


Split in the middle



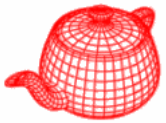
- Makes the L & R probabilities equal
- Pays no attention to the L & R costs

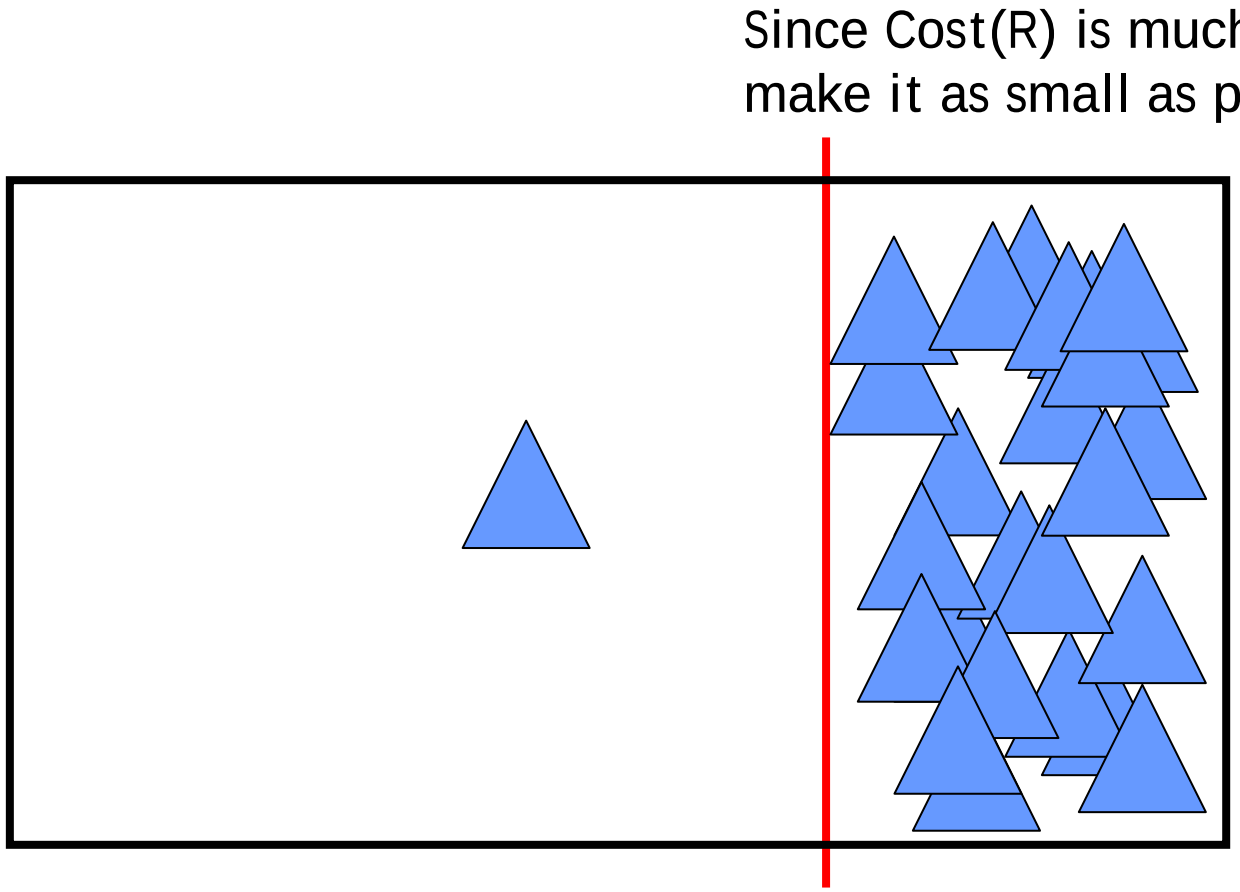
Split at the median



- Makes the L & R costs equal
- Pays no attention to the L & R probabilities

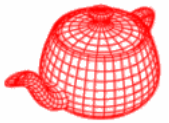
Cost-optimized split





- Automatically and rapidly isolates complexity
- Produces large chunks of empty space

Building good kd-trees



- Need the probabilities
  - Turns out to be proportional to surface area
- Need the child cell costs
  - Simple triangle count works great (very rough approx.)
  - Empty cell “boost”

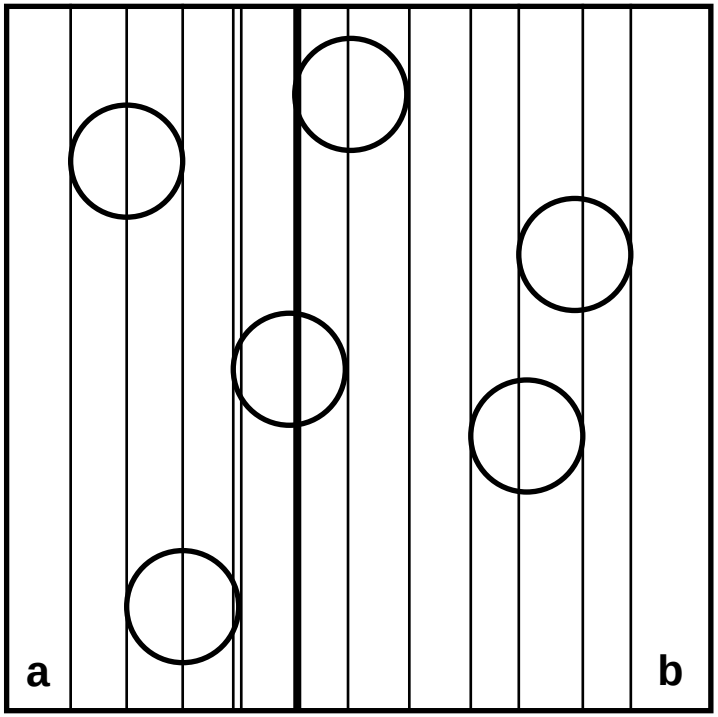
Cost(cell) = C_trav + Prob(hit L) * Cost(L) + Prob(hit R) * Cost(R)

= C_trav + SA(L) * TriCount(L) + SA(R) * TriCount(R)

C_trav is the ratio of the cost to traverse to the cost to intersect

C_trav = 1:80 in pbrt (found by experiments)

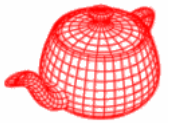
Surface area heuristic



2n splits

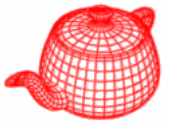
$$p_a = \frac{S_a}{S} \qquad p_b = \frac{S_b}{S}$$

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

Basic building algorithm



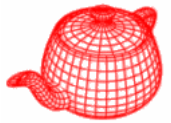
1. Pick an axis, or optimize across all three
2. Build a set of candidate split locations (cost extrema must be at bbox vertices)
3. Sort or bin the triangles
4. Sweep to incrementally track L/R counts, cost
5. Output position of minimum cost split

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

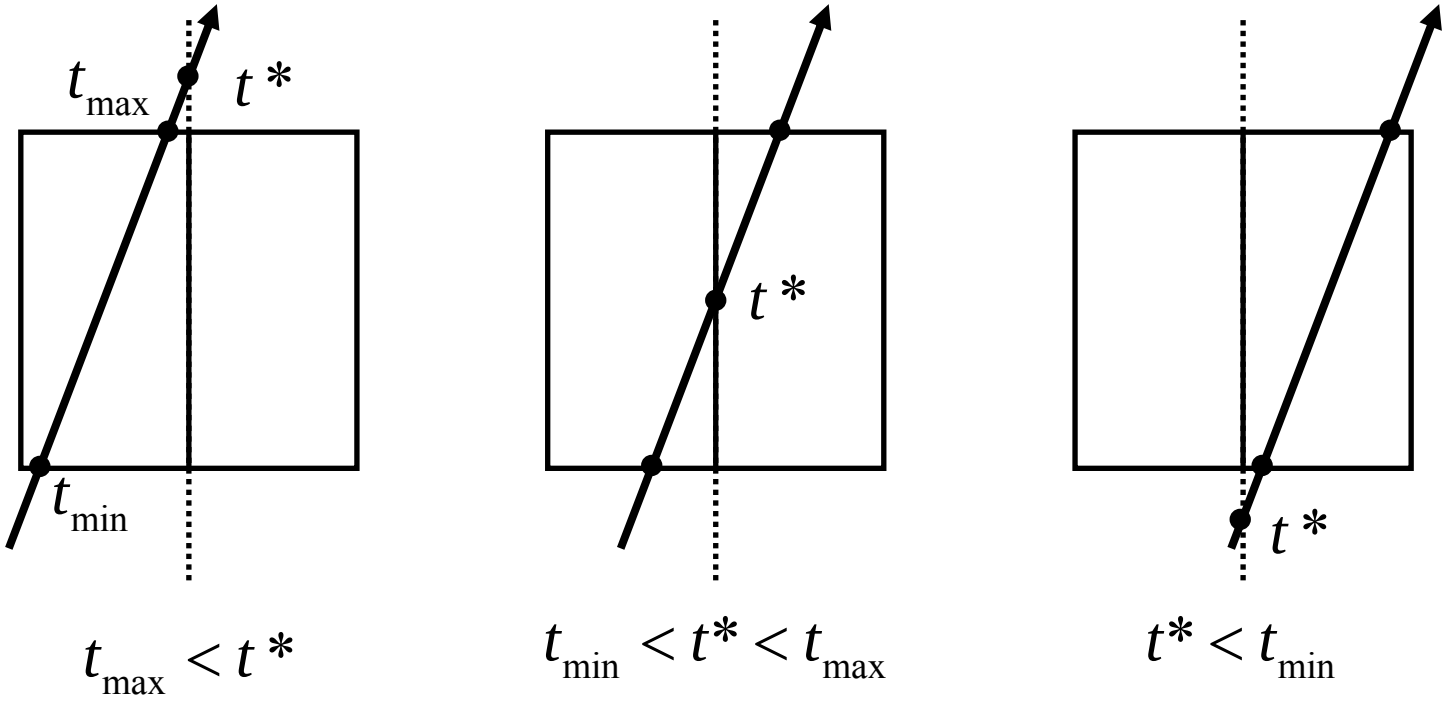
$$T(N) = N \log^2 N$$

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

Ray traversal algorithm



- Recursive inorder traversal

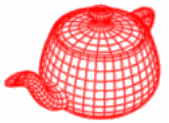


$t_{max} < t*$ $\quad$ $t_{min} < t* < t_{max}$ $\quad$ $t* < t_{min}$

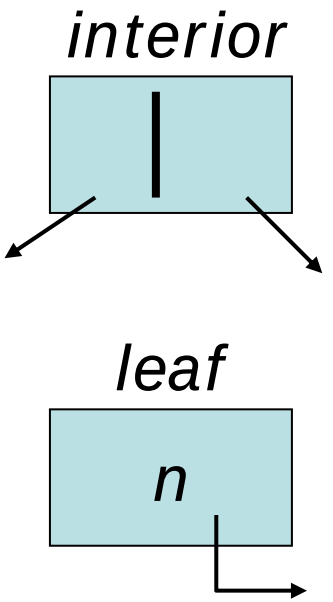
`Intersect(L,tmin,tmax)` `Intersect(L,tmin,t*)` `Intersect(R,tmin,tmax)`
`Intersect(R,t*,tmax)`

a video for kdtree

Tree representation

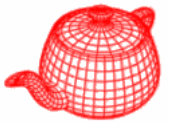


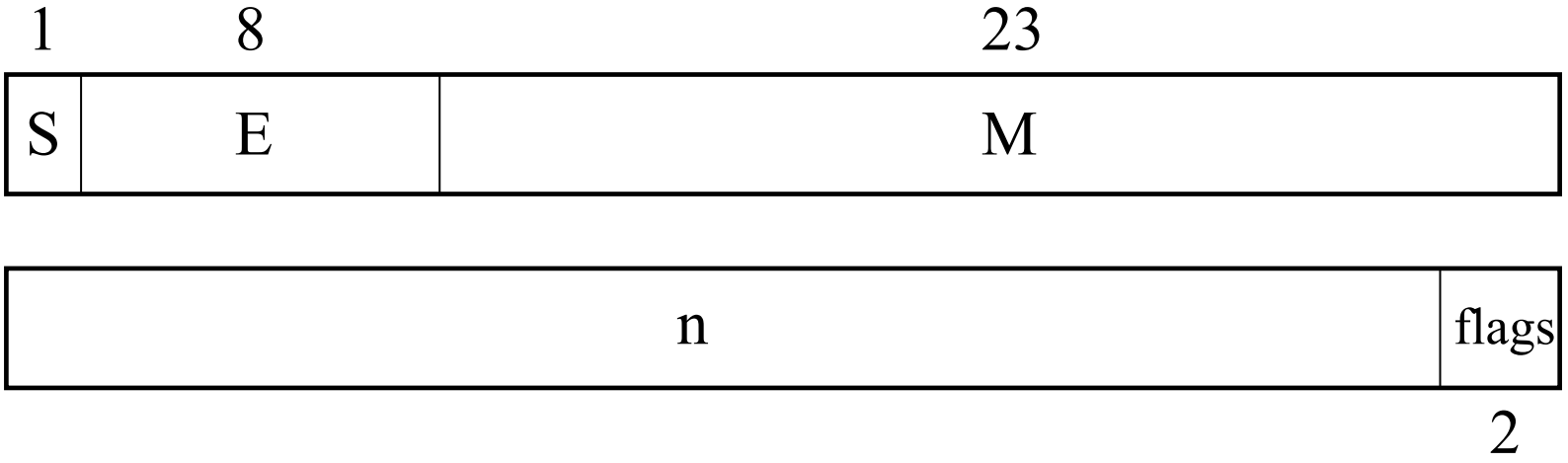
8-byte (reduced from 16-byte, 20% gain)




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

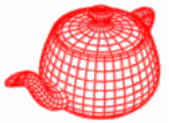
Tree representation





Flag: 0,1,2 (interior x, y, z) 3 (leaf)

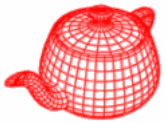
`KdTreeAccel` construction



- Recursive top-down algorithm
- max depth = $8+1.3\log(N)$

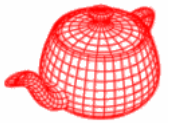
```
If (nPrims <= maxPrims || depth==0) {
  <create leaf>
}
```

Interior node



- Choose split axis position
  - Medpoint
  - Medium cut
  - Area heuristic
- Create leaf if no good splits were found
- Classify primitives with respect to split

Choose split axis position



cost of no split: $\sum_{k=1}^{N} t_i(k)$

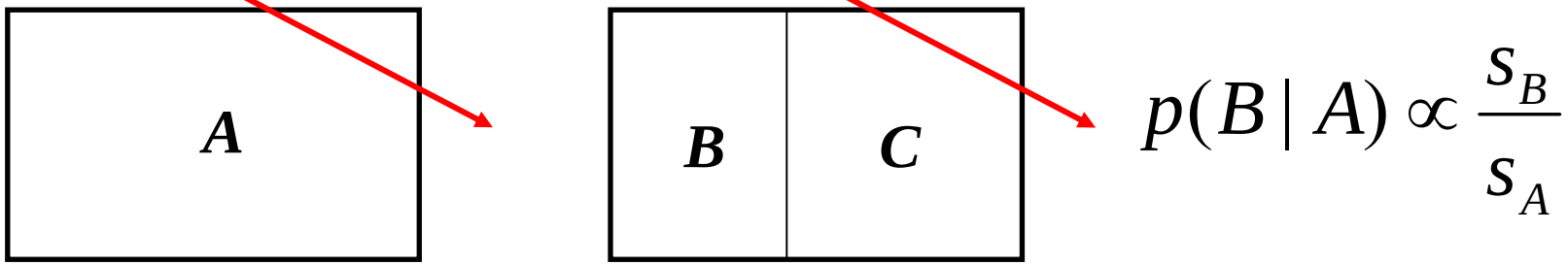
cost of split: $t_t + P_B \sum_{k=1}^{N_B} t_i(b_k) + P_A \sum_{k=1}^{N_A} t_i(a_k)$

assumptions:

1. t_i is the same for all primitives
2. $t_i : t_t$ = 80 : 1 (determined by experiments, main factor for the performance)

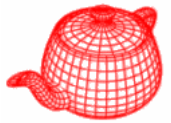
cost of no split: $t_i N$

cost of split: $t_t + t_i(1-b_e)(p_B N_B + p_A N_A)$

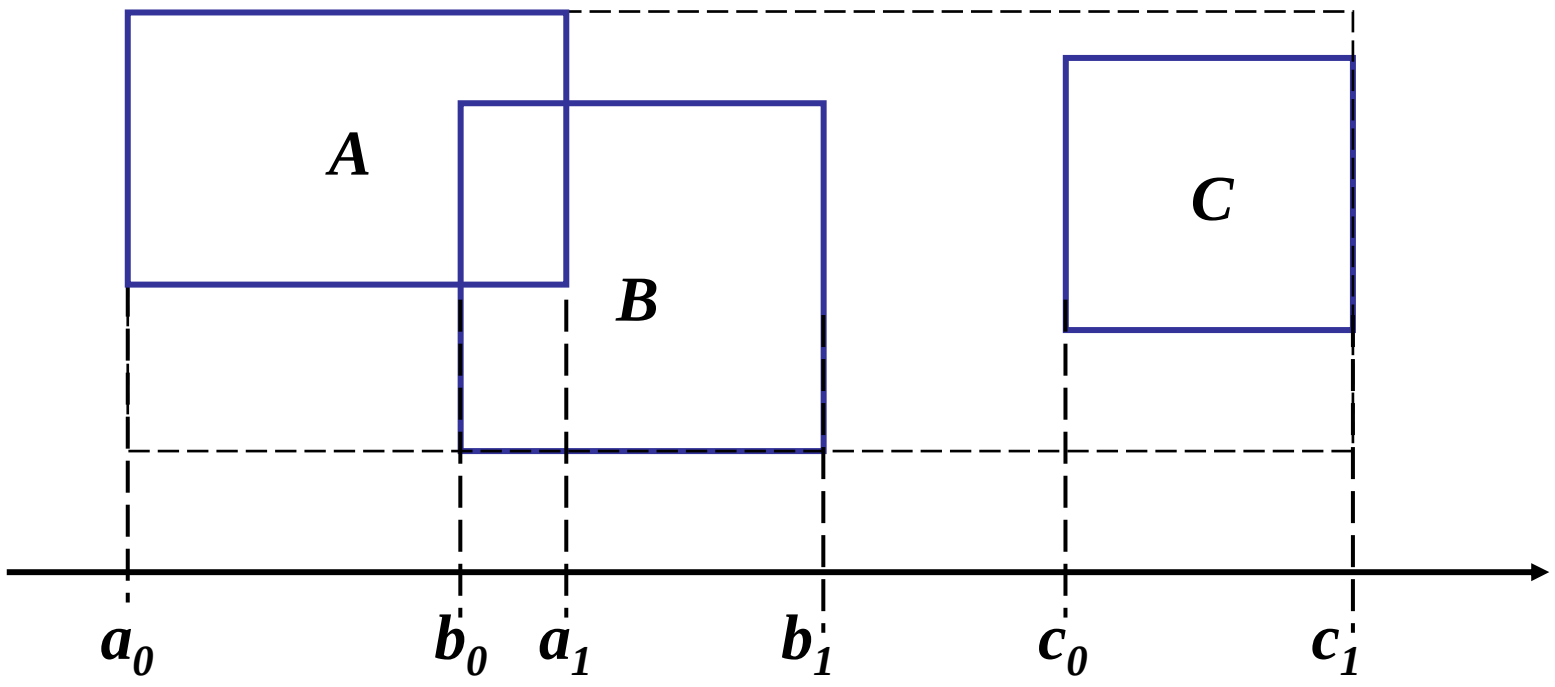


$$p(B \mid A) \propto \frac{S_B}{S_A}$$

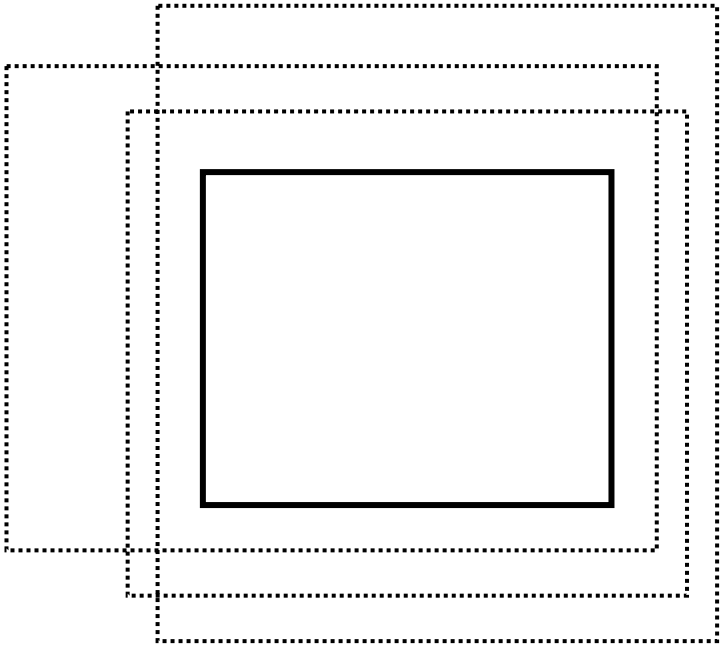
Choose split axis position



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

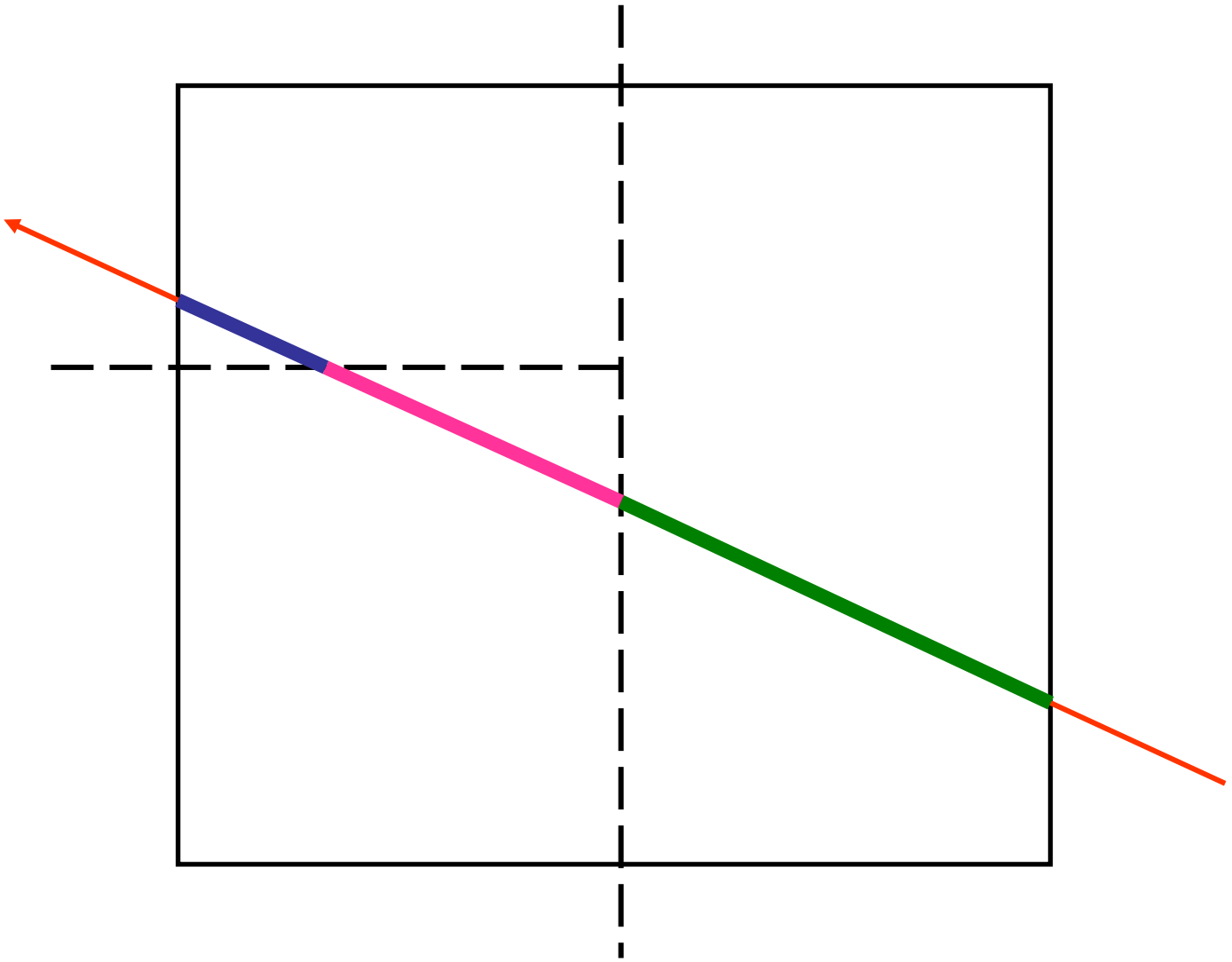


Choose split axis position

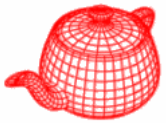


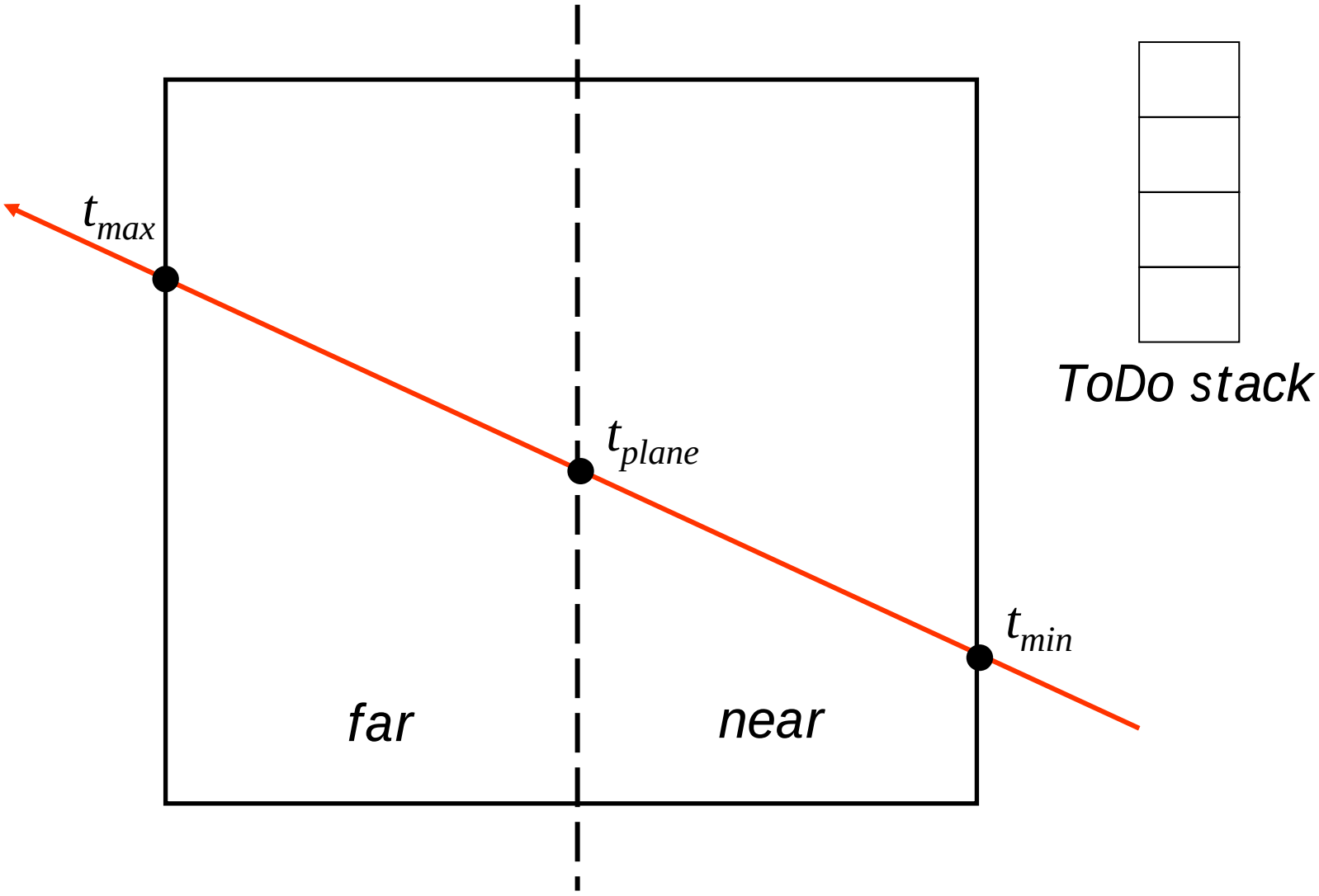
If there is no split along this axis, try other axes.
When all fail, create a leaf.

`KdTreeAccel` traversal

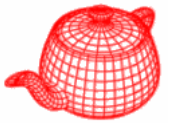


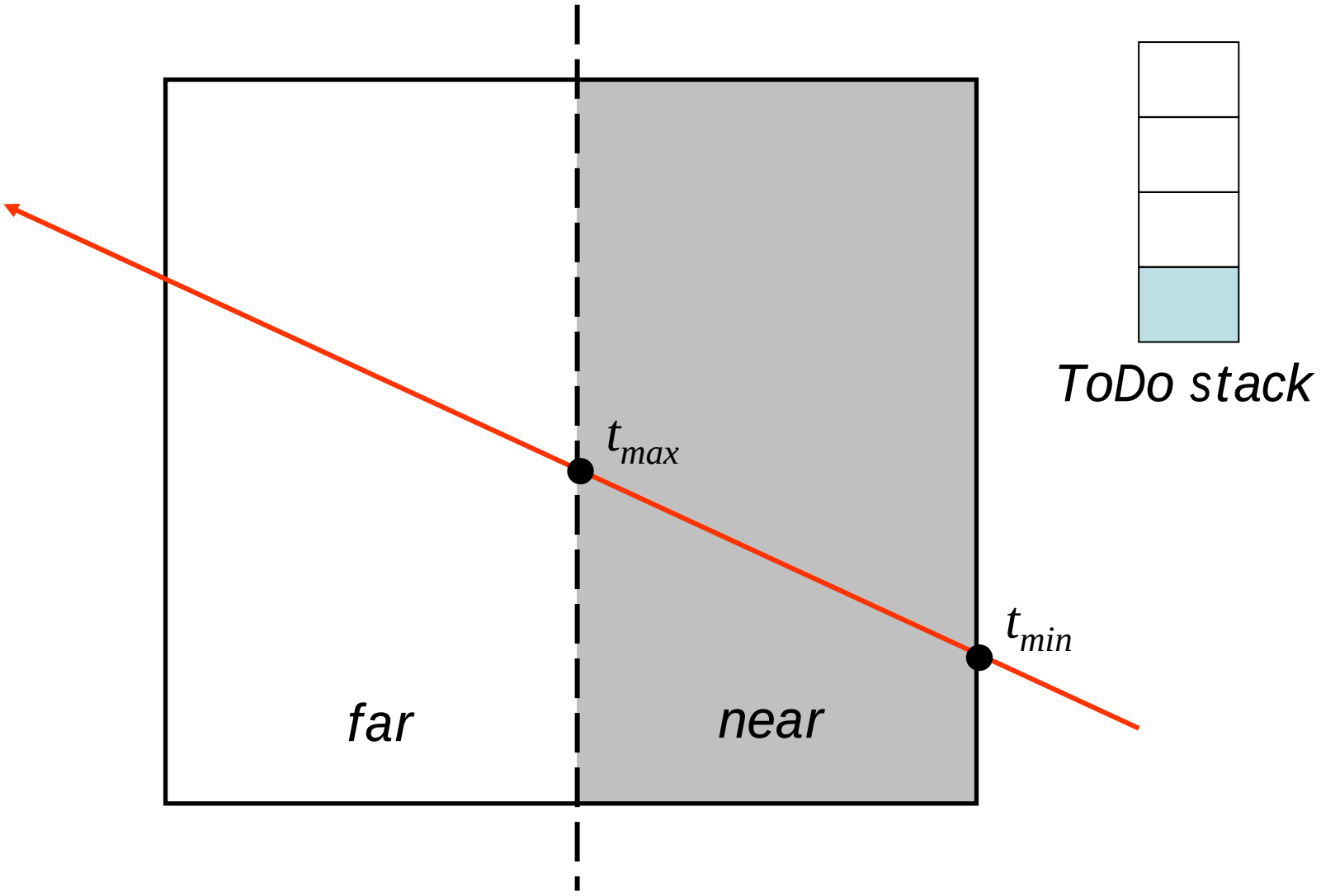
`KdTreeAccel` traversal



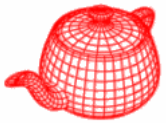


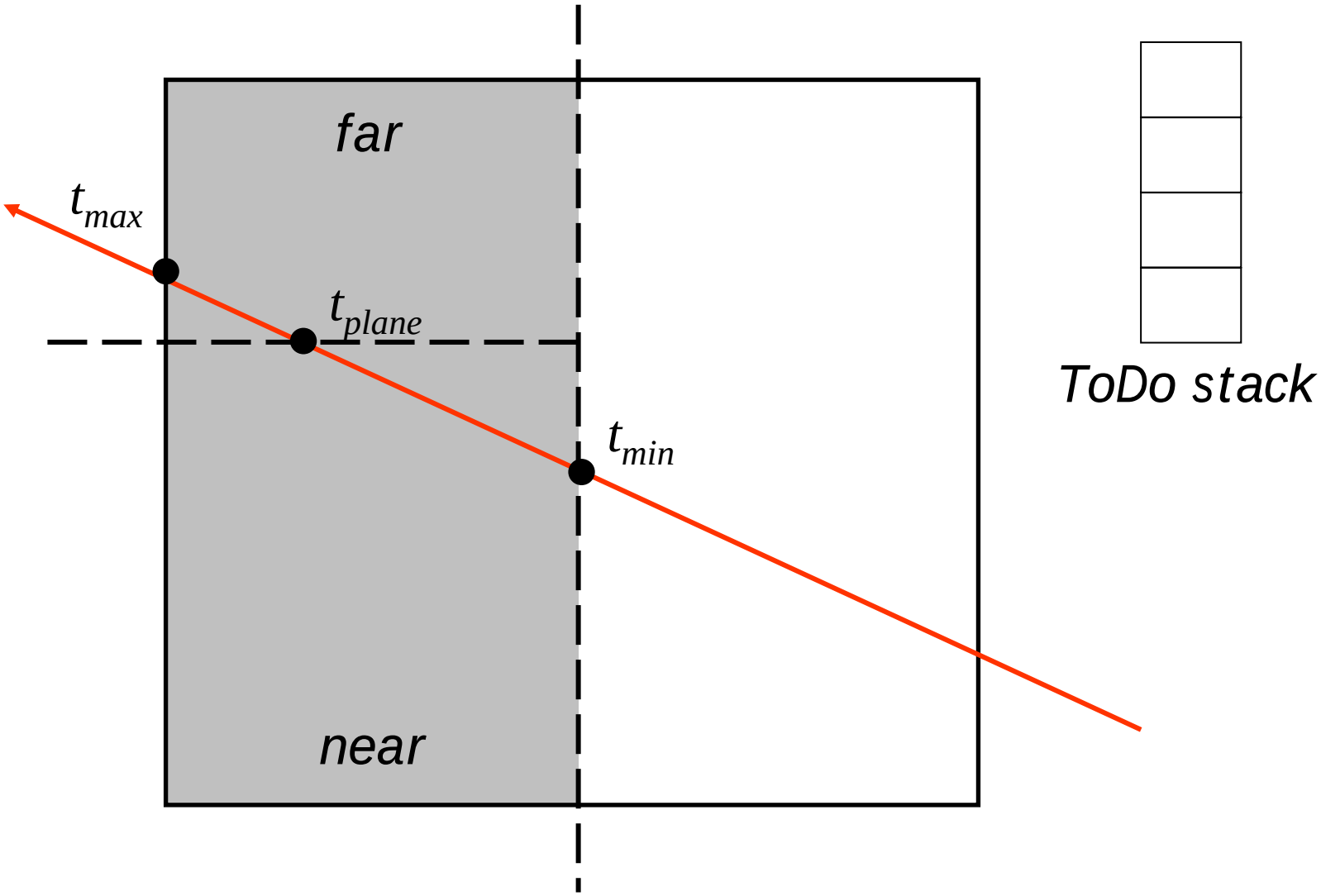
`KdTreeAccel` traversal



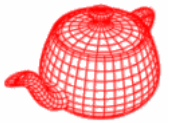


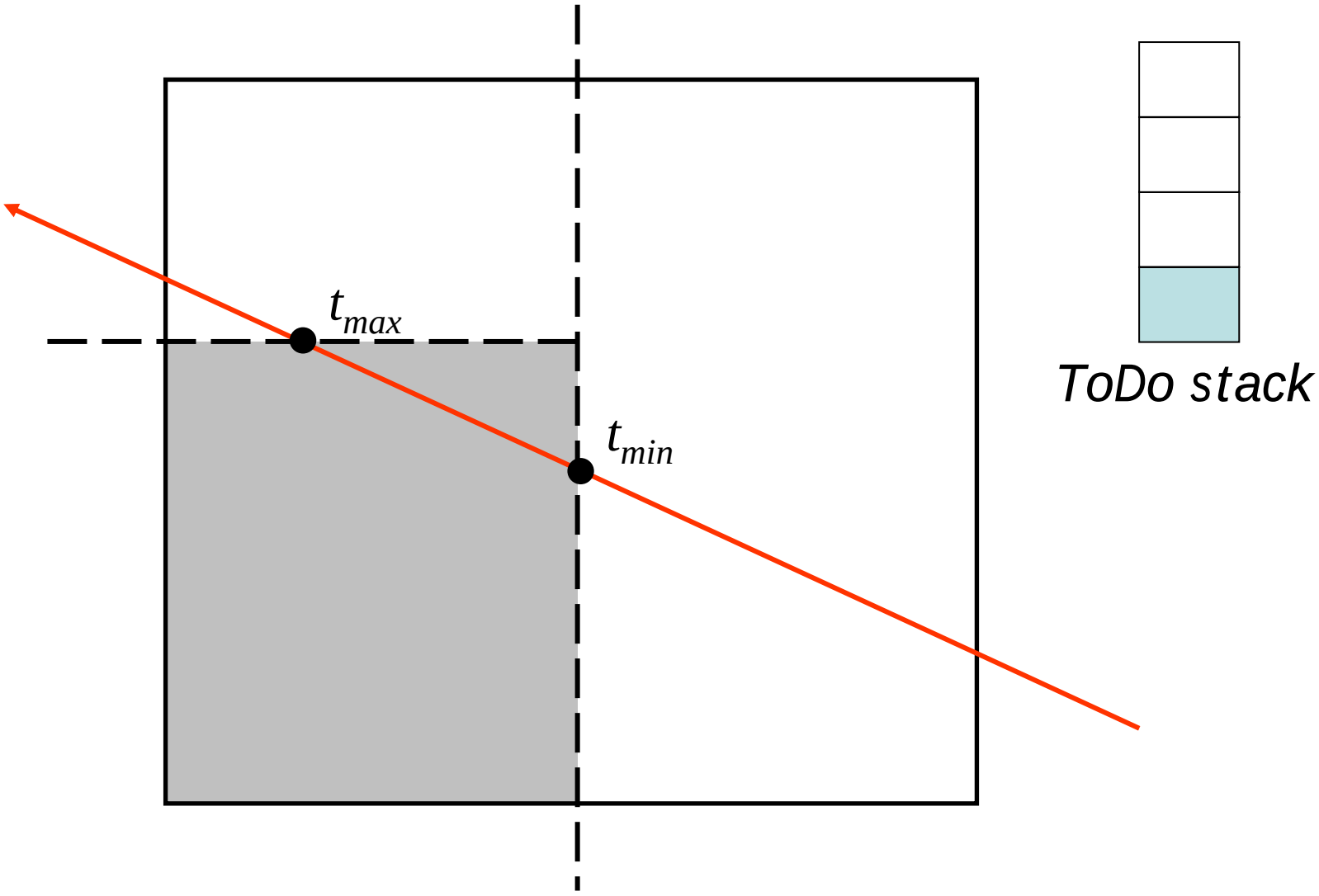
`KdTreeAccel` traversal



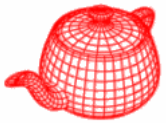


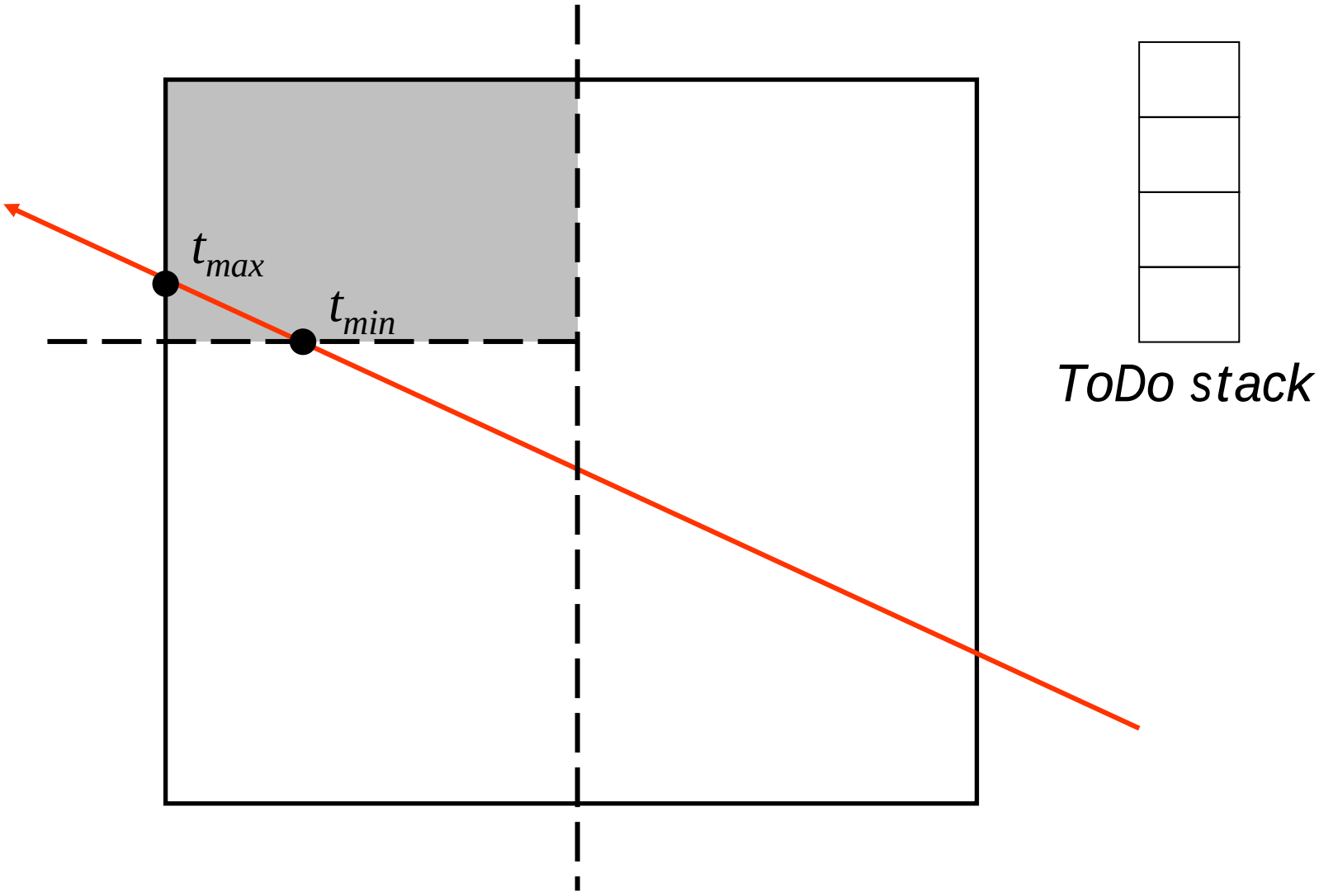
`KdTreeAccel` traversal



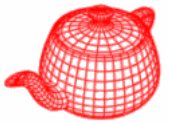


`KdTreeAccel` traversal

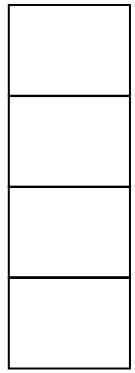




KdTreeAccel traversal

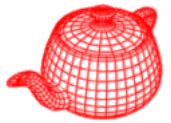


```
bool KdTreeAccel::Intersect
    (const Ray &ray, Intersection *isect)
{
  if (!bounds.IntersectP(ray, &tmin, &tmax))
    return false;

  KdAccelNode *node=&nodes[0];
  while (node!=NULL) {
    if (ray.maxt<tmin) break;
    if (!node->IsLeaf()) <Interior>
    else <Leaf>
  }
}
```



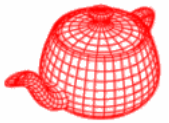
ToDo stack (max depth)

Leaf node

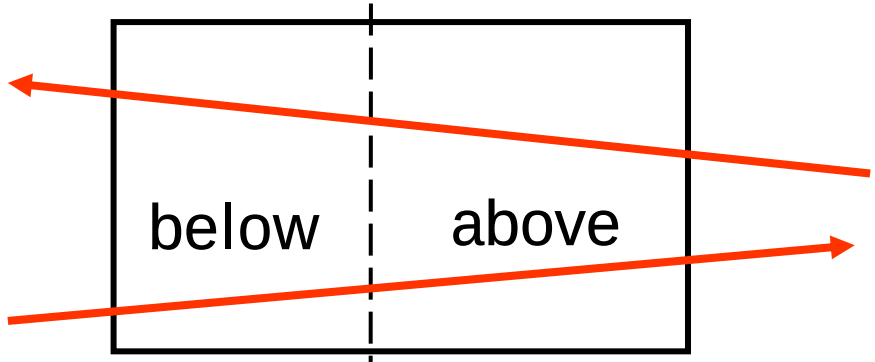


1. Check whether ray intersects primitive(s) inside the node; update ray's **`maxt`**
2. Grab next node from ToDo queue

Interior node

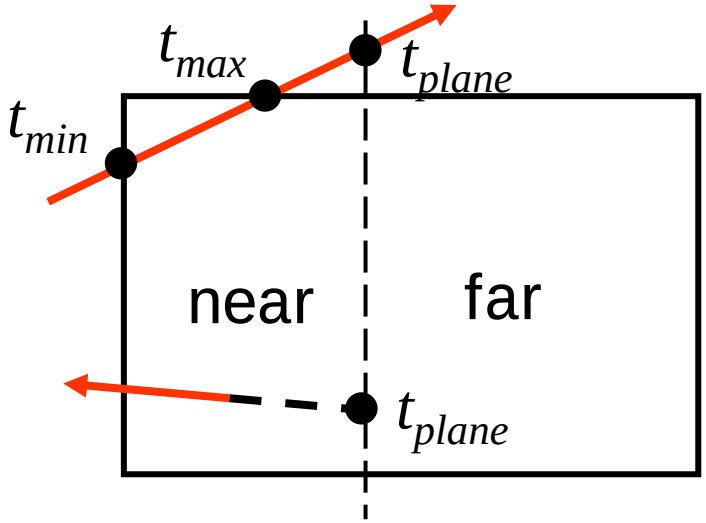


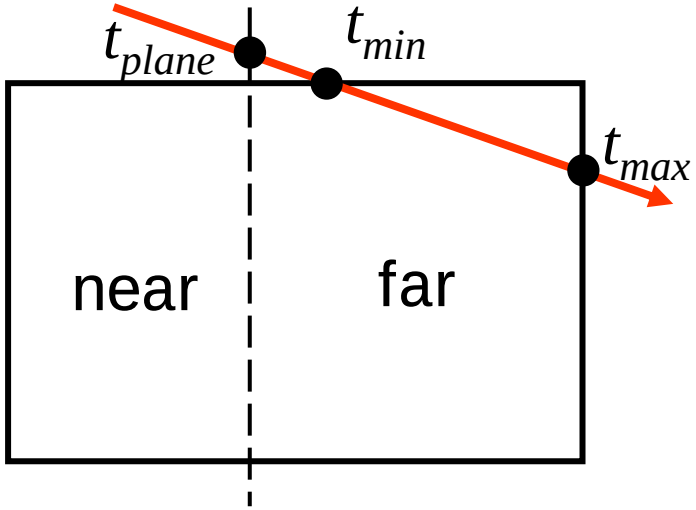
1. Determine near and far



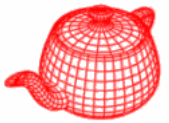
`node+1` **`&(nodes[node->aboveChild])`**

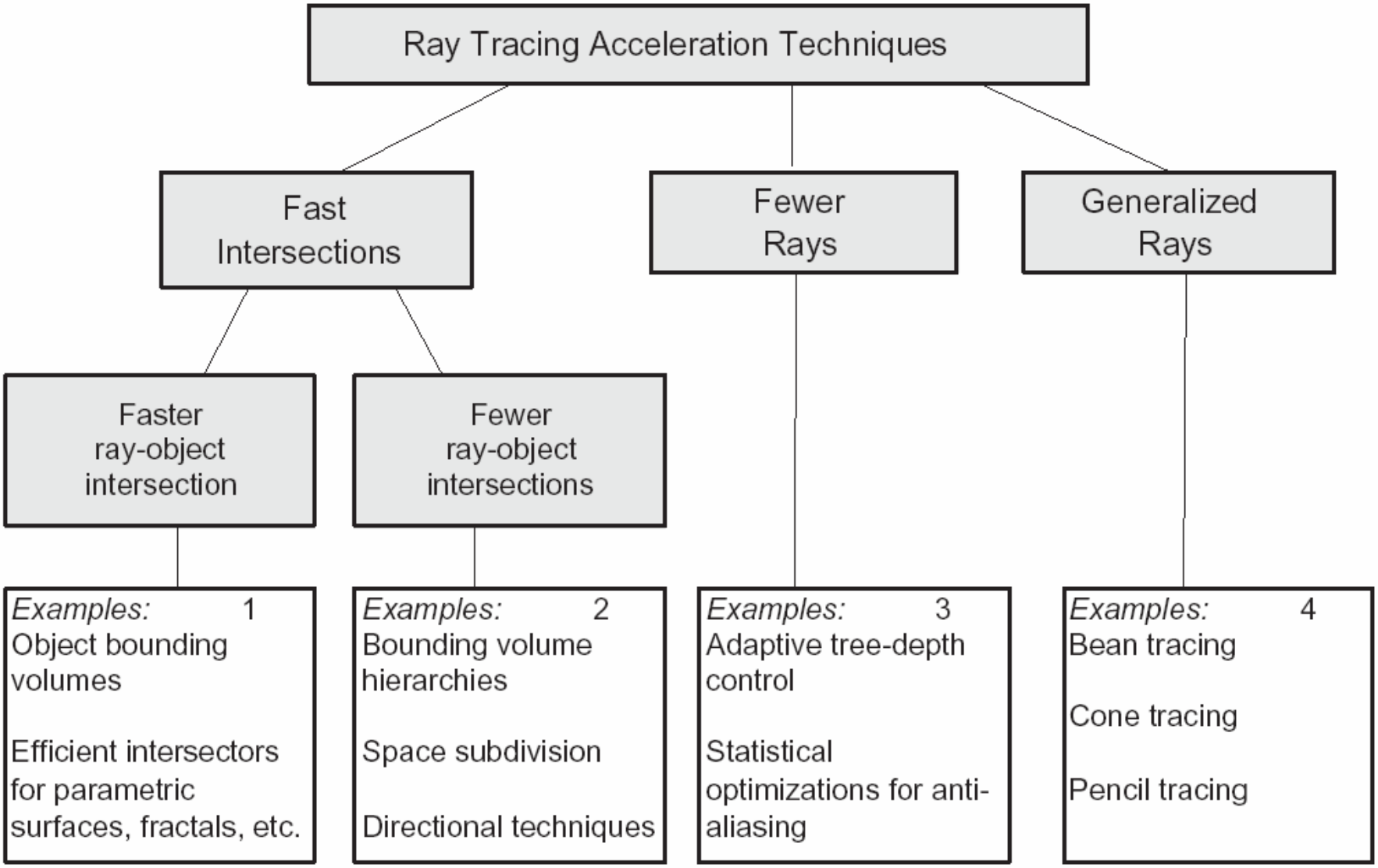
2. Determine whether we can skip a node



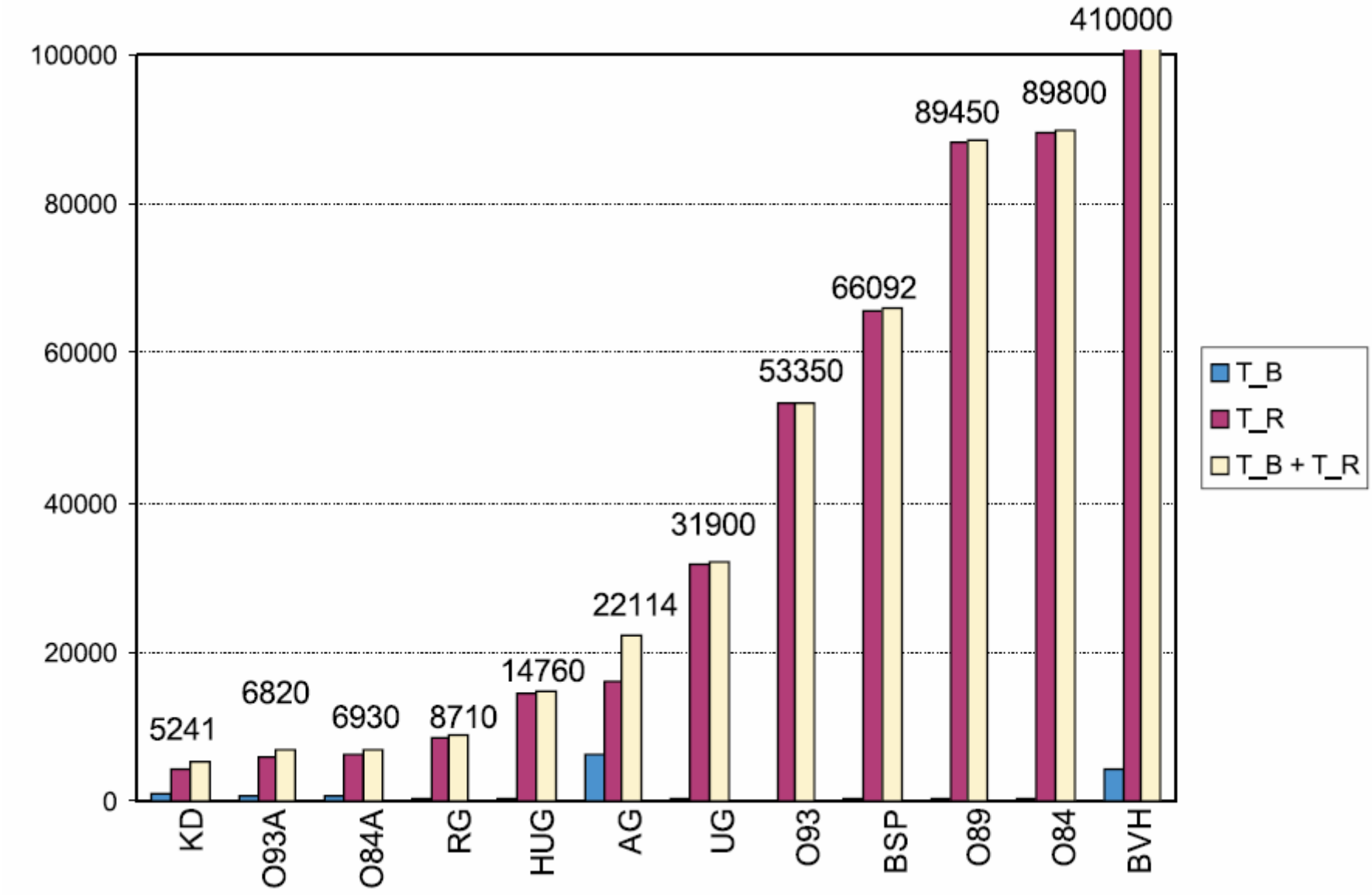


Acceleration techniques

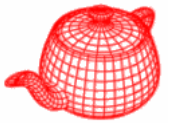




Best efficiency scheme



References



- J. Goldsmith and J. Salmon, Automatic Creation of Object Hierarchies for Ray Tracing, IEEE CG&A, 1987.
- Brian Smits, Efficiency Issues for Ray Tracing, Journal of Graphics Tools, 1998.
- K. Klimaszewski and T. Sederberg, Faster Ray Tracing Using Adaptive Grids, IEEE CG&A Jan/Feb 1999.
- Whang et. al., Octree-R: An Adaptive Octree for efficient ray tracing, IEEE TVCG 1(4), 1995.
- A. Glassner, Space subdivision for fast ray tracing. IEEE CG&A, 4(10), 1984