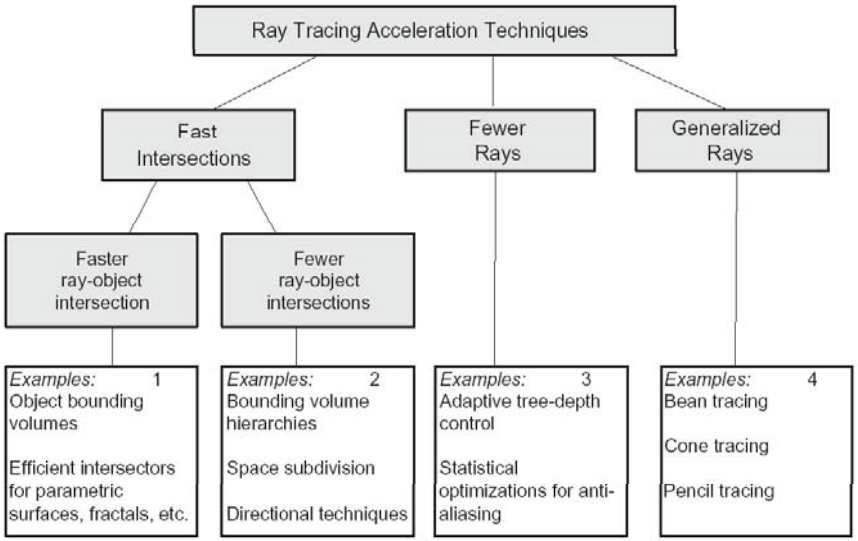
Acceleration

Digital Image Synthesis

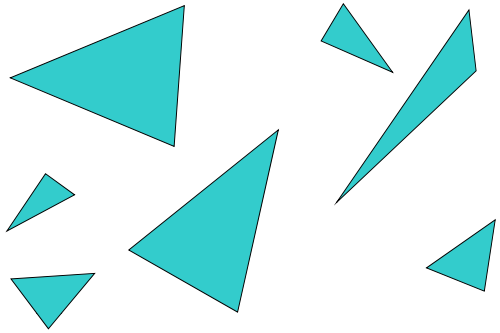
Yung-Yu Chuang

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

Acceleration techniques

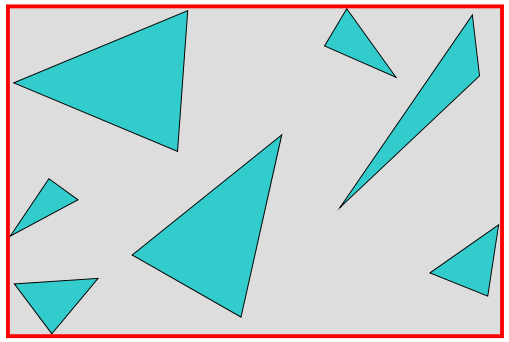


Bounding volume hierarchy



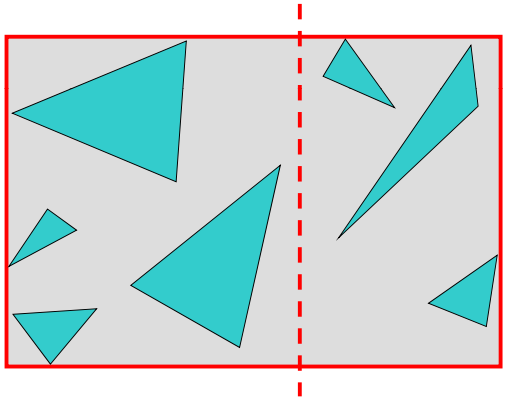
Bounding volume hierarchy

1) Find bounding box of objects



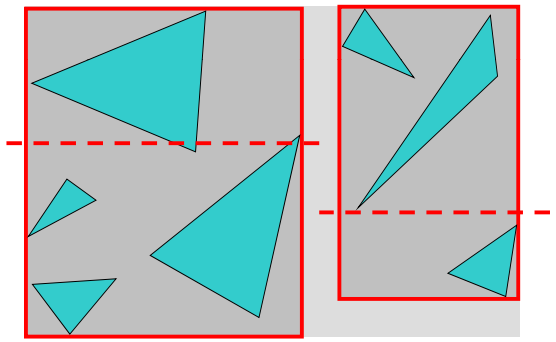
Bounding volume hierarchy

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



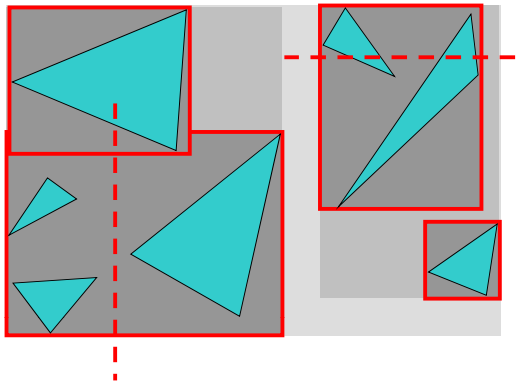
Bounding volume hierarchy

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



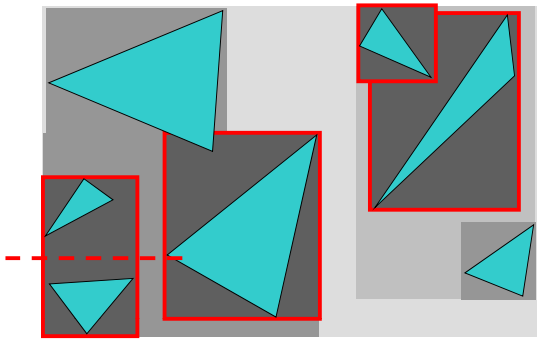
Bounding volume hierarchy

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



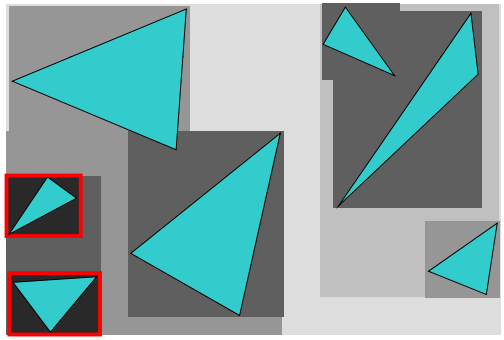
Bounding volume hierarchy

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



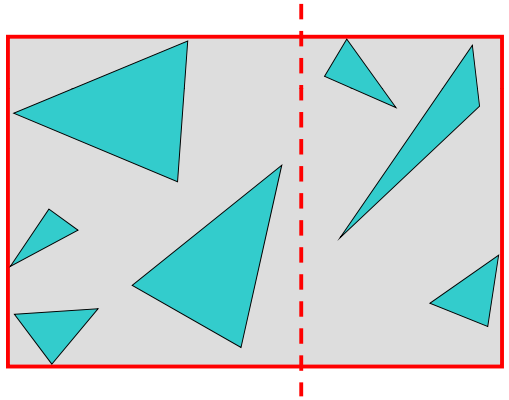
Bounding volume hierarchy

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



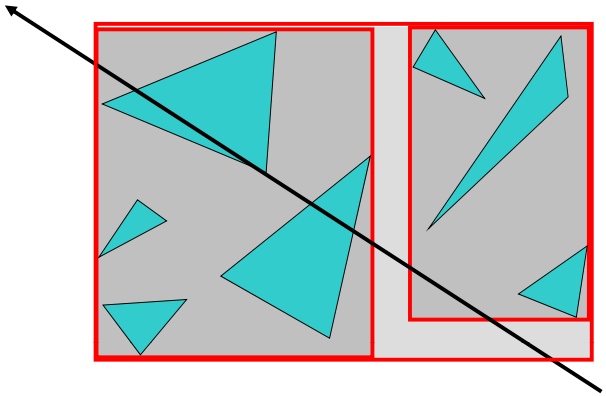
Where to split?

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



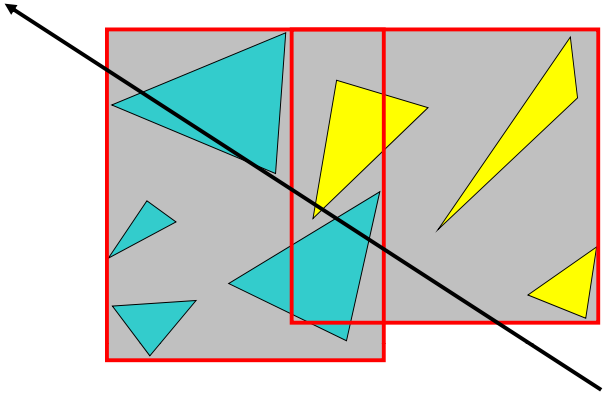
BVH traversal

- If hit parent, then check all children



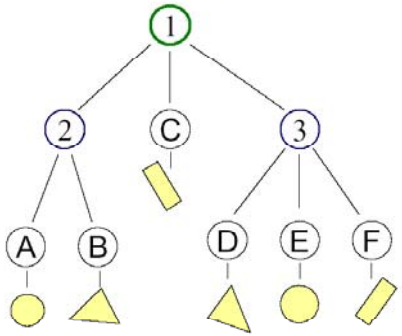
BVH traversal

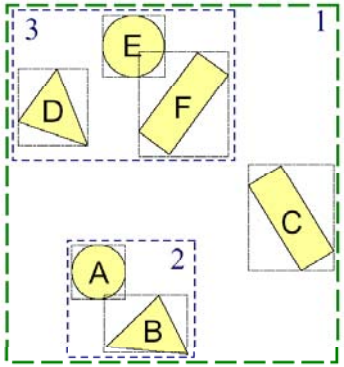
- Don't return intersection immediately because the other subvolumes may have a closer intersection



Bounding volume hierarchy

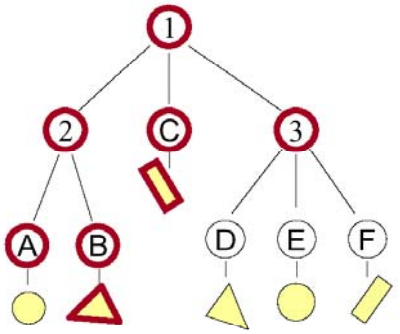
- Build hierarchy of bounding volumes
  - Bounding volume of interior node contains all children

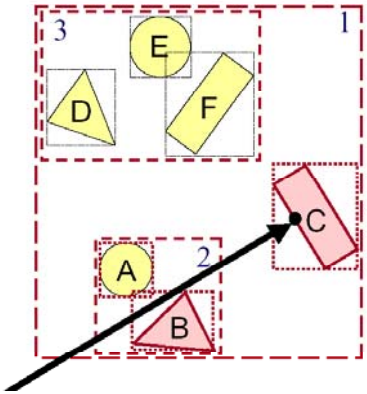




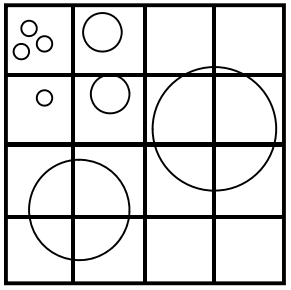
Bounding volume hierarchy

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

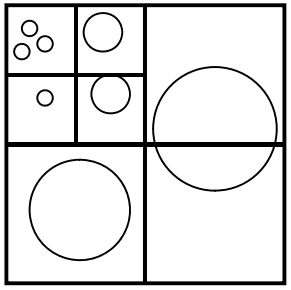




Space subdivision approaches

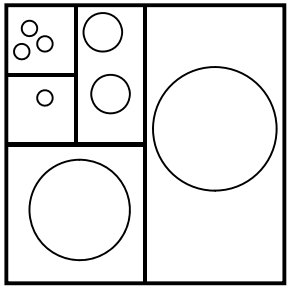


Unifrom grid

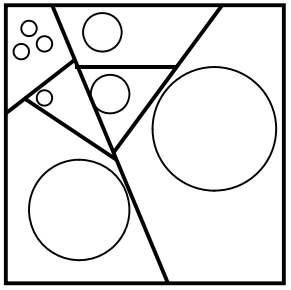


Quadtree (2D)
Octree (3D)

Space subdivision approaches

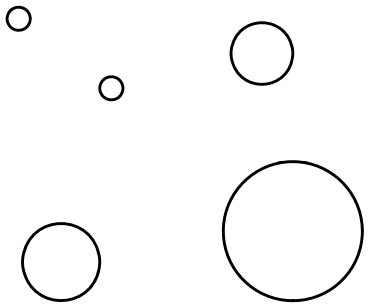


KD tree

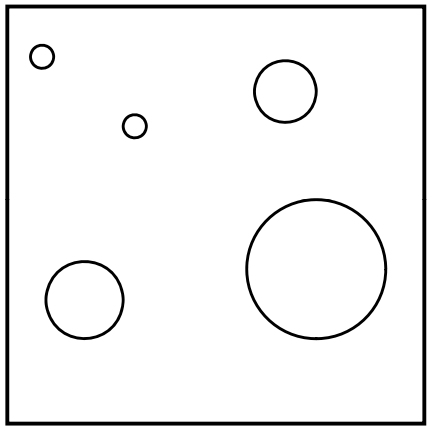


BSP tree

Uniform grid



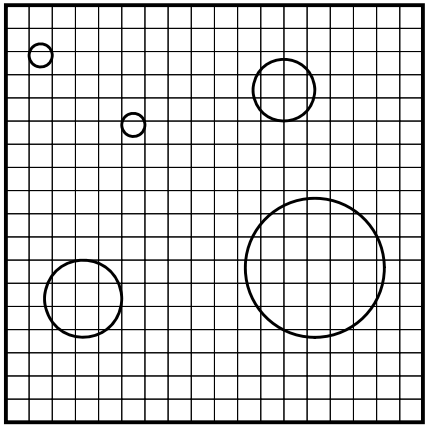
Uniform grid



Preprocess scene

1. Find bounding box

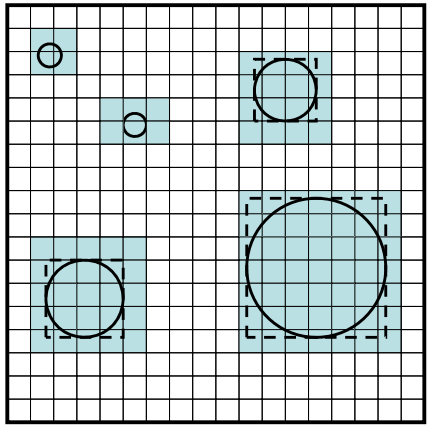
Uniform grid



Preprocess scene

1. Find bounding box
2. Determine grid resolution

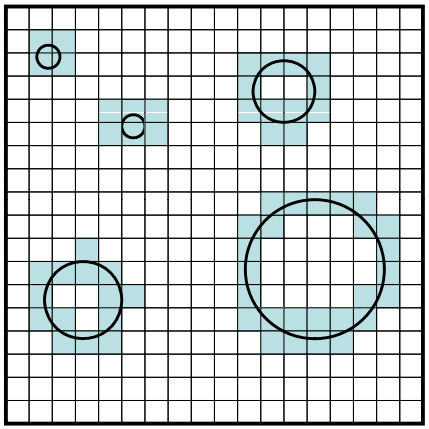
Uniform grid



Preprocess scene

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

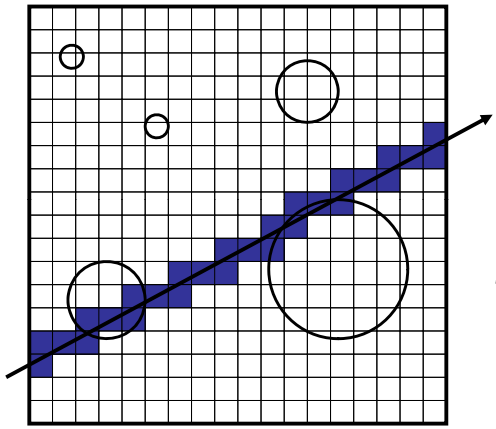
Uniform grid



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 (Digital Differential Analyzer)

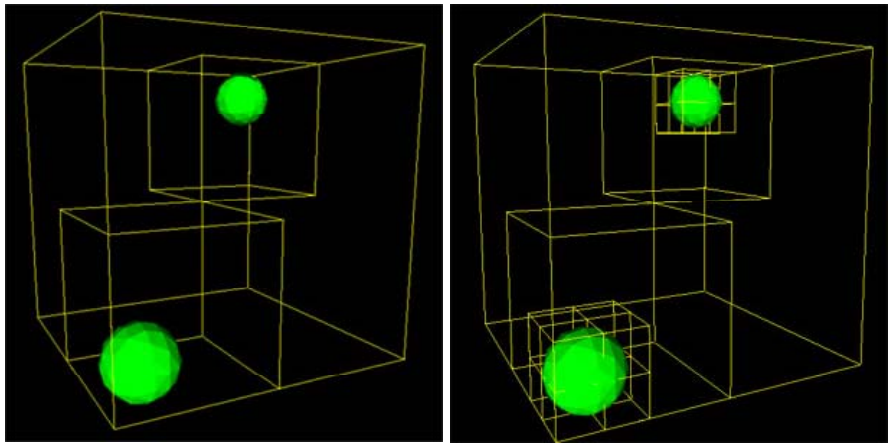
$$y = mx + b \qquad m = \frac{y_2 - y_1}{x_2 - x_1}$$

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

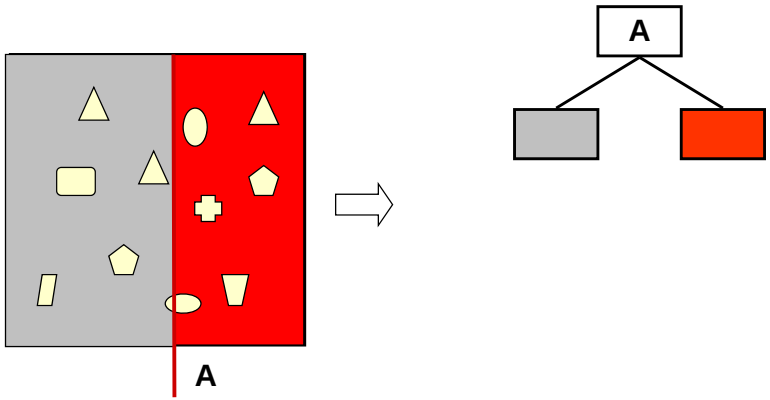
$$y_{i+1} = mx_{i+1} + b \quad \text{naive}$$

$$y_{i+1} = y_i + m \quad \text{DDA}$$

Octree

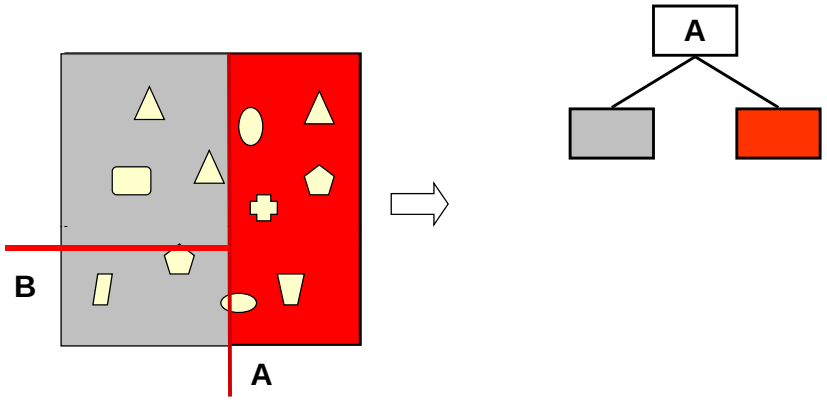


K-d tree



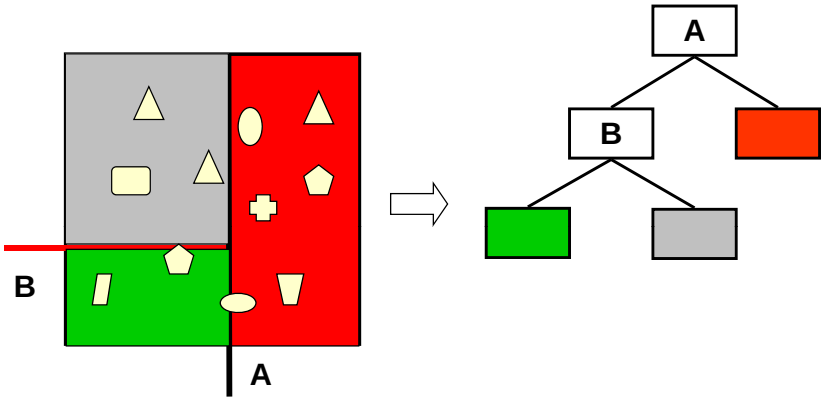
Leaf nodes correspond to unique regions in space

K-d tree



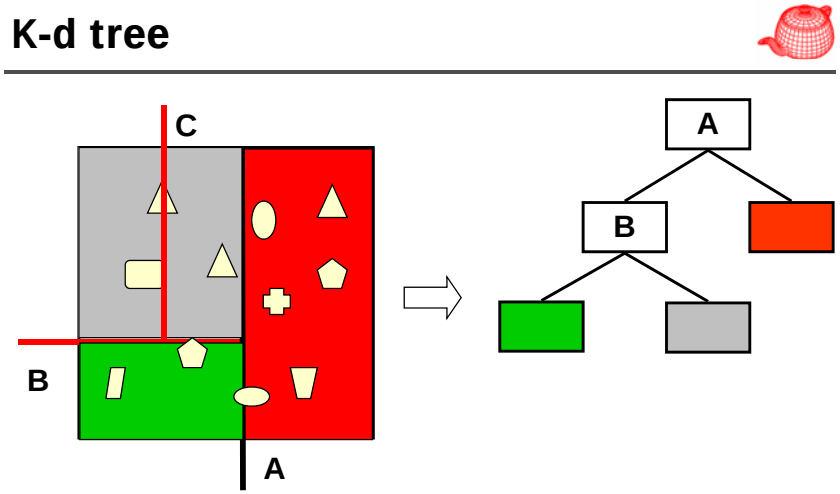
Leaf nodes correspond to unique regions in space

K-d tree

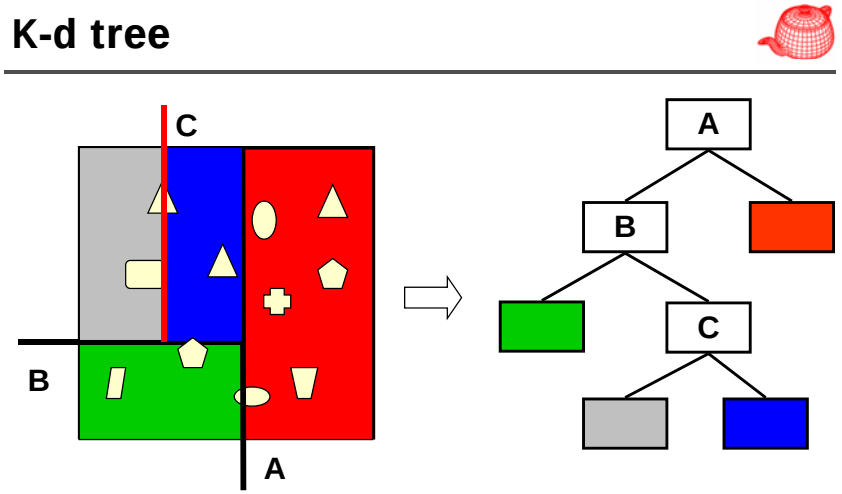


Leaf nodes correspond to unique regions in space

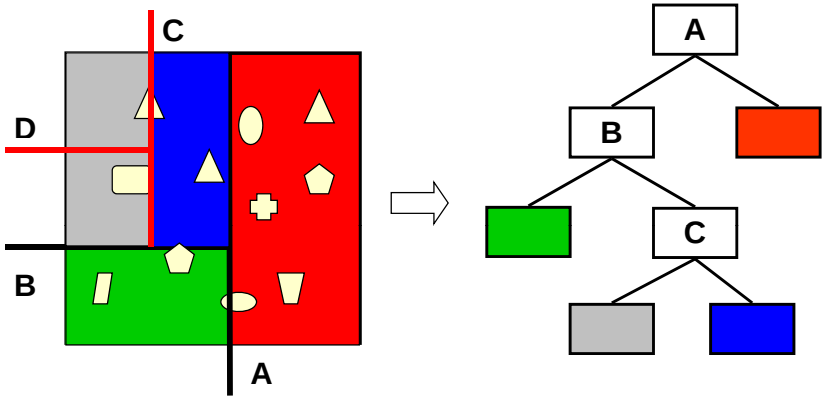
K-d tree



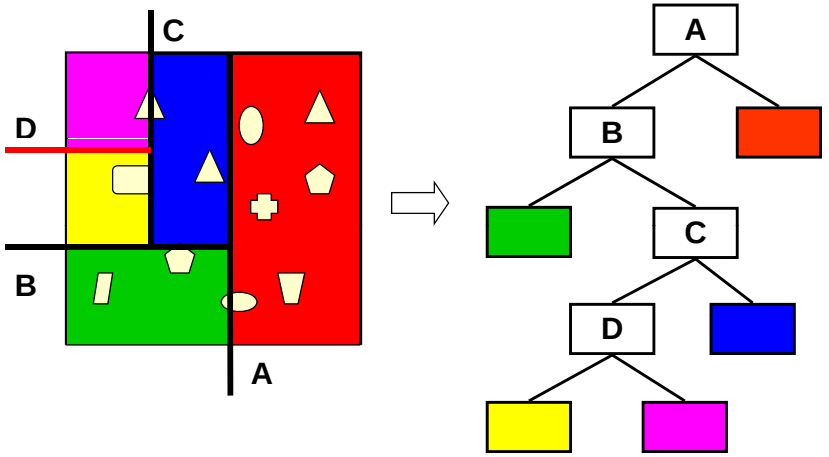
K-d tree



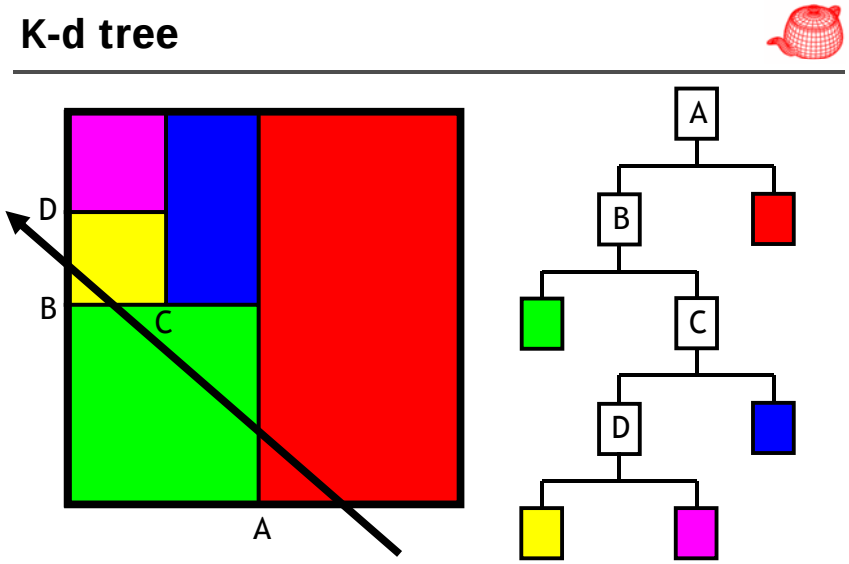
K-d tree



K-d tree

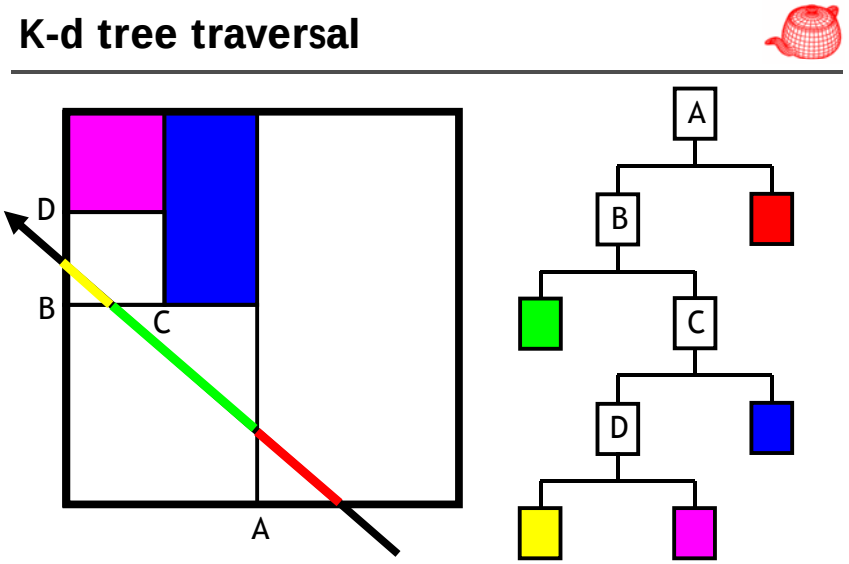


K-d tree



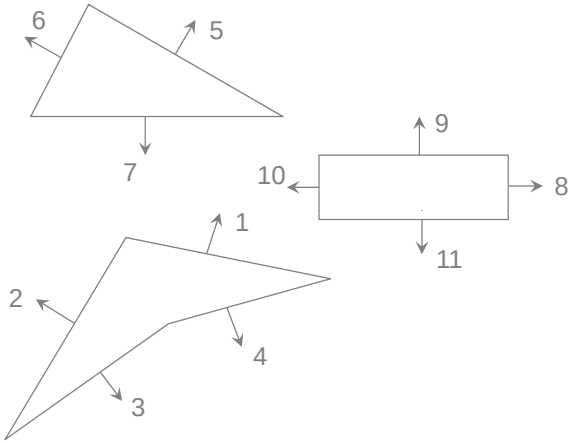
Leaf nodes correspond to unique regions in space

K-d tree traversal

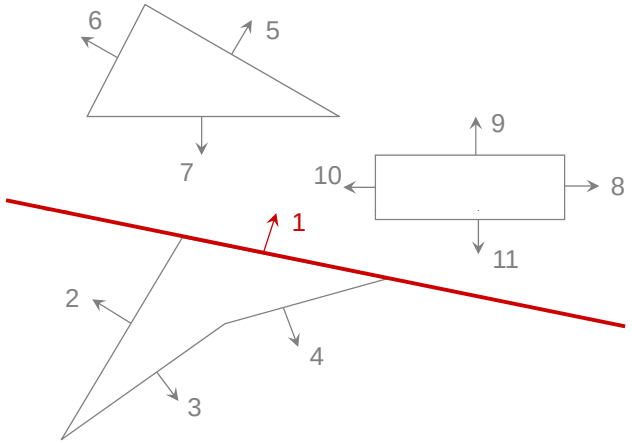


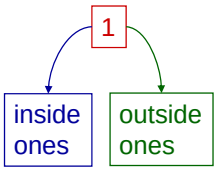
Leaf nodes correspond to unique regions in space

BSP tree

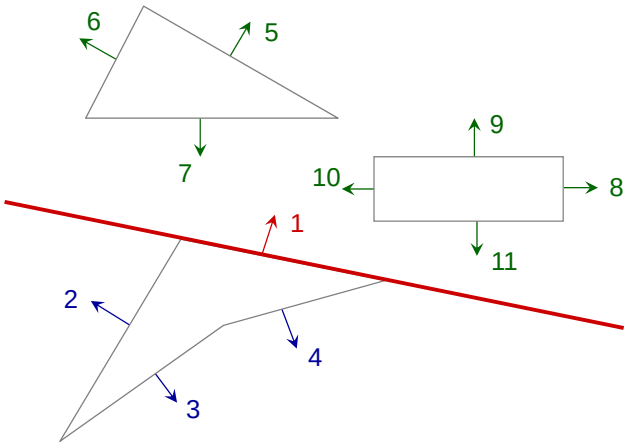


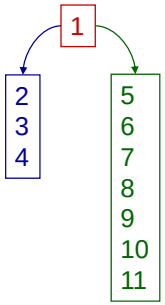
BSP tree



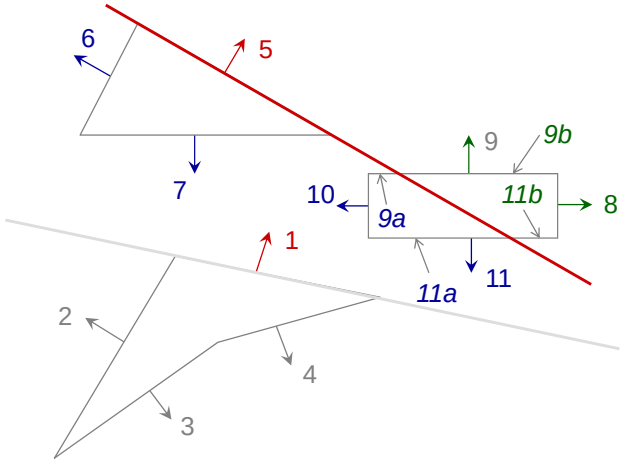


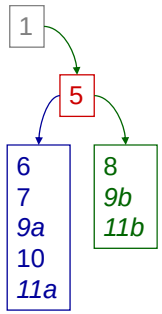
BSP tree



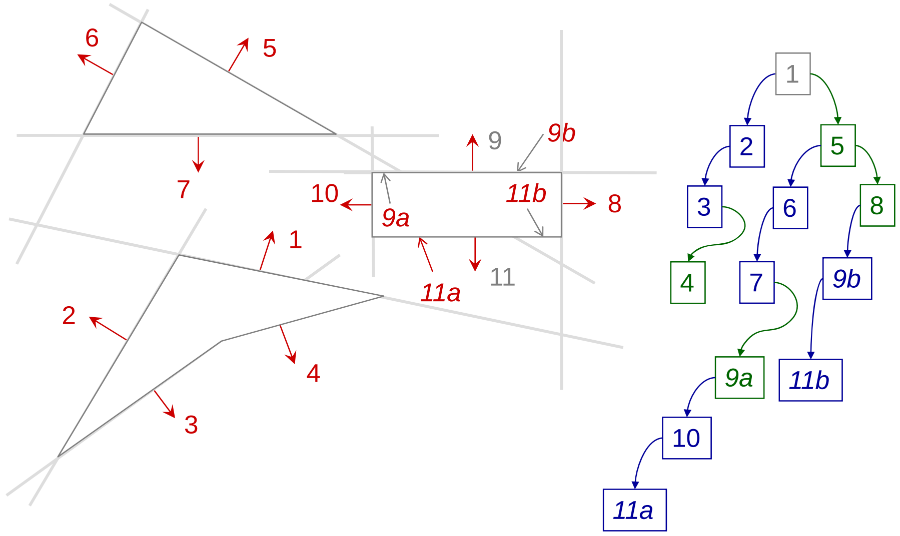


BSP tree

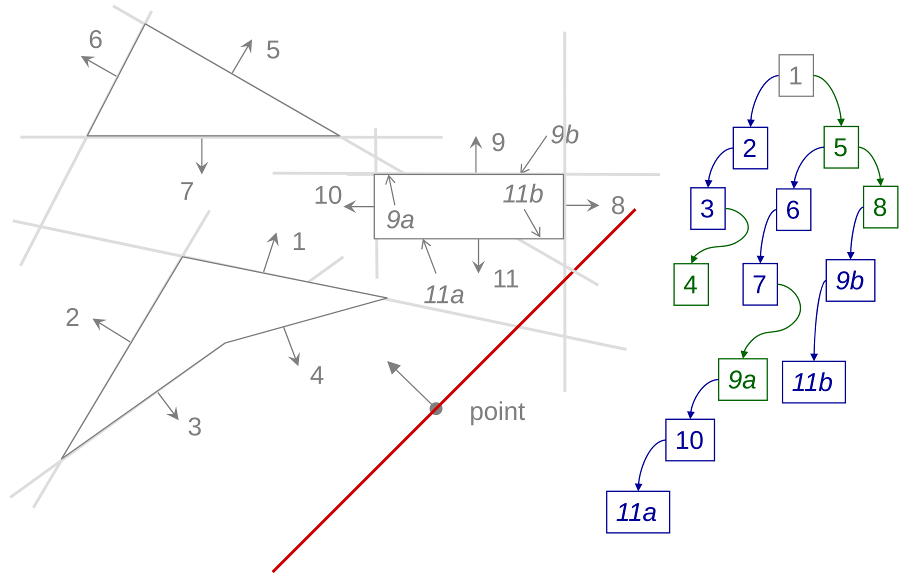




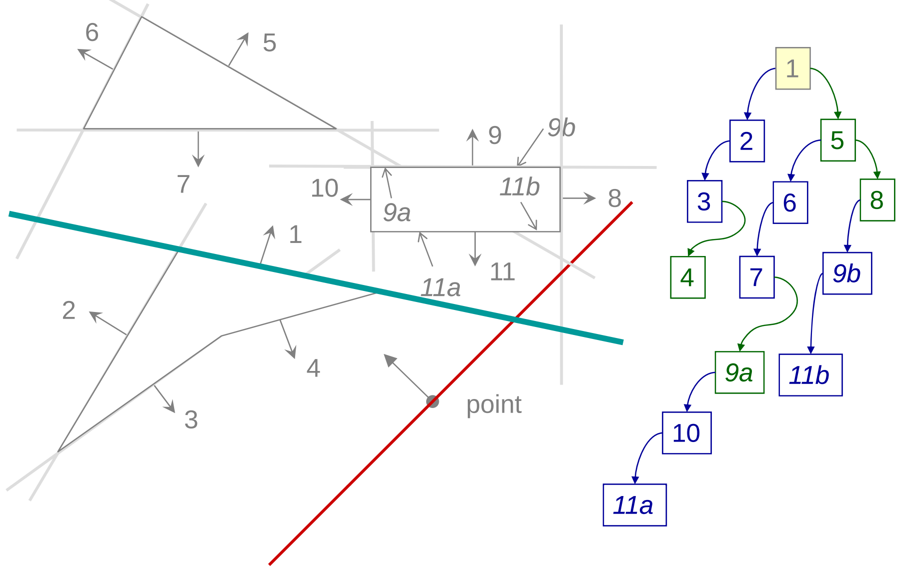
BSP tree



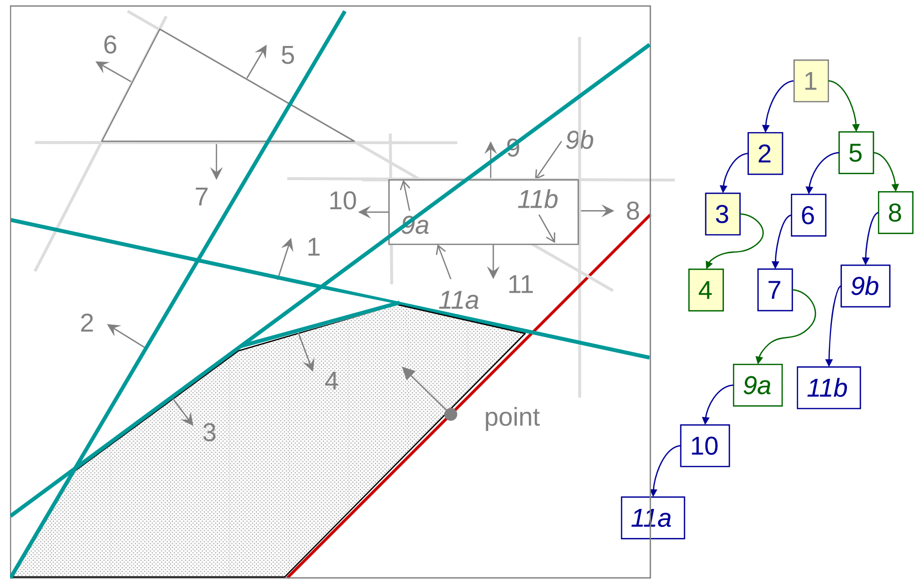
BSP tree traversal



BSP tree traversal



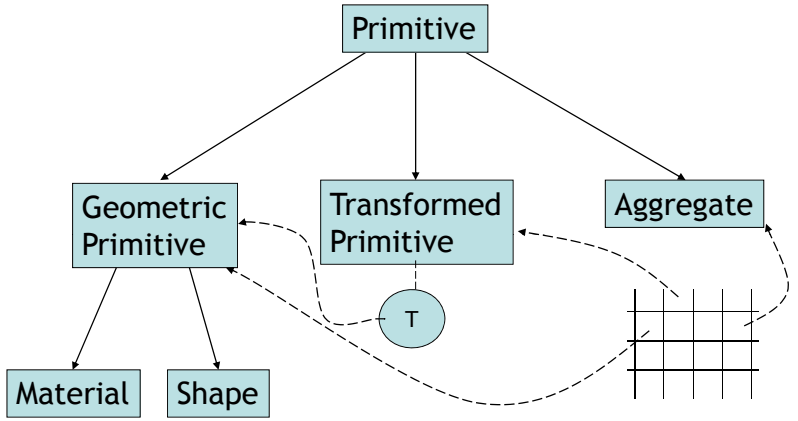
BSP tree traversal



Classes

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

Hierarchy



Support both Instancing and animation

Store intersectable primitives, call Refine If necessary

Primitive

```
class Primitive : public ReferenceCounted {
  <Primitive interface>
  const int primitiveId;
  static int nextprimitiveId;
}

class TransformedPrimitive: public Primitive
{
  …
  Reference<Primitive> instance;
}
```

Interface

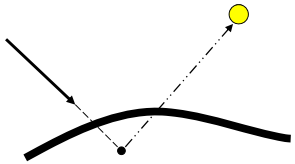
```
BBox WorldBound();                                  geometry
bool CanIntersect();
bool Intersect(const Ray &r,          // update maxt
               Intersection *in);
bool IntersectP(const Ray &r);
void Refine(vector<Reference<Primitive>> &refined);
void FullyRefine(vector<Reference<Primitive>> &refined);
```

```
AreaLight *GetAreaLight();                          material
BSDF *GetBSDF(const DifferentialGeometry &dg,
              Transform &WorldToObject);
BSSRDF *GetBSSRDF(DifferentialGeometry &dg,
              Transform &WorldToObject);
```

Intersection

```
struct Intersection {
  <Intersection interface>
  DifferentialGeometry dg;
  const Primitive *primitive;
  Transform WorldToObject, ObjectToWorld;
  int shapeId, primitiveId;
  float rayEpsilon;
};
```

adaptively estimated



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

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;   // BRDF
AreaLight *areaLight;           // emittance
```

- Most operations are forwarded to shape

GeometricPrimitive

```
bool Intersect(Ray &r,Intersection *isect) {
  float thit, rayEpsilon;
  if (!shape->Intersect(r, &thit,
                &rayEpsilon, &isect->dg))
    return false;
  isect->primitive = this;
  isect->WorldToObject = *shape->WorldToObject;
  isect->ObjectToWorld = *shape->ObjectToWorld;
  isect->shapeId = shape->shapeId;
  isect->primitiveId = primitiveId;
  isect->rayEpsilon = rayEpsilon;
  r.maxt = thit;
  return true;
}
```

Object instancing

61 unique plants, 4000 individual plants, 19.5M triangles
With instancing, store only 1.1M triangles, 11GB->600MB

TransformedPrimitive

```
Reference<Primitive> primitive;
AnimatedTransform WorldToPrimitive;
```

for instancing and animation

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

TransformedPrimitive

```
bool Intersect(Ray &r, Intersection *isect){
  Transform w2p;
  WorldToPrimitive.Interpolate(r.time,&w2p);
  Ray ray = w2p(r);
  if (!primitive->Intersect(ray, isect))
    return false;
  r.maxt = ray.maxt;
  isect->primitiveId = primitiveId;
  if (!w2p.IsIdentity()) {
 // Compute world-to-object transformation for instance
   isect->WorldToObject=isect->WorldToObject*w2p;
   isect->ObjectToWorld=Inverse(
                              isect->WorldToObject);
```

TransformedPrimitive

```
 // Transform instance's differential geometry to world space
  Transform PrimitiveToWorld = Inverse(w2p);
  isect->dg.p = PrimitiveToWorld(isect->dg.p);
  isect->dg.nn = Normalize(
                 PrimitiveToWorld(isect- >dg.nn));
  isect->dg.dpdu=PrimitiveToWorld(isect->dg.dpdu);
  isect->dg.dpdv=PrimitiveToWorld(isect->dg.dpdv);
  isect->dg.dndu=PrimitiveToWorld(isect->dg.dndu);
  isect->dg.dndv=PrimitiveToWorld(isect->dg.dndv);
 }
 return true;
}
```

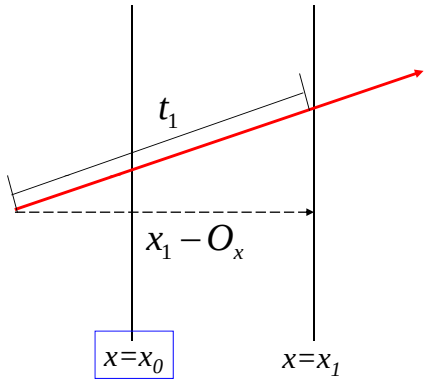
Aggregates

- 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

Ray-Box intersections

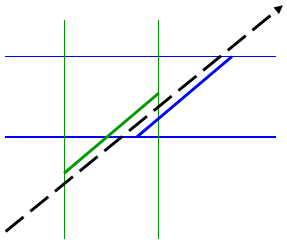
- Almost all acclerators require it
- Quick rejection, use enter and exit point to traverse the hierarchy
- AABB is the intersection of three slabs

Ray-Box intersections



$$\frac{1}{D_x}$$

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



Ray-Box intersections

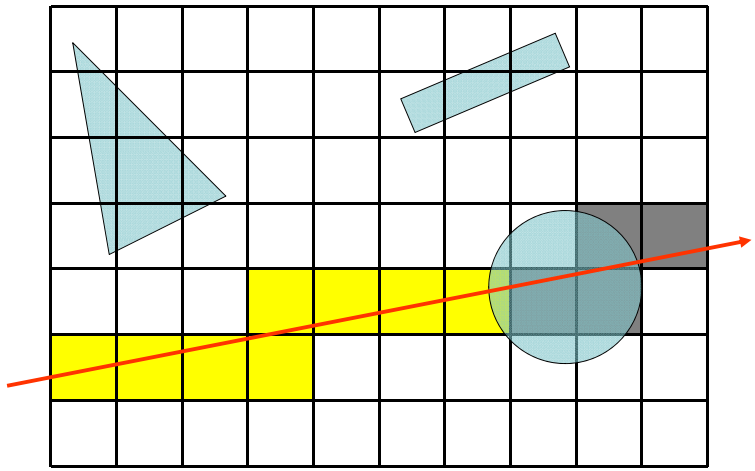
```
bool BBox::IntersectP(const Ray &ray,
                      float *hitt0, float *hitt1)
{
  float t0 = ray.mint, t1 = ray.maxt;
  for (int i = 0; i < 3; ++i) {
    float invRayDir = 1.f / ray.d[i];
    float tNear = (pMin[i] - ray.o[i]) * invRayDir;
    float tFar  = (pMax[i] - ray.o[i]) * invRayDir;

    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

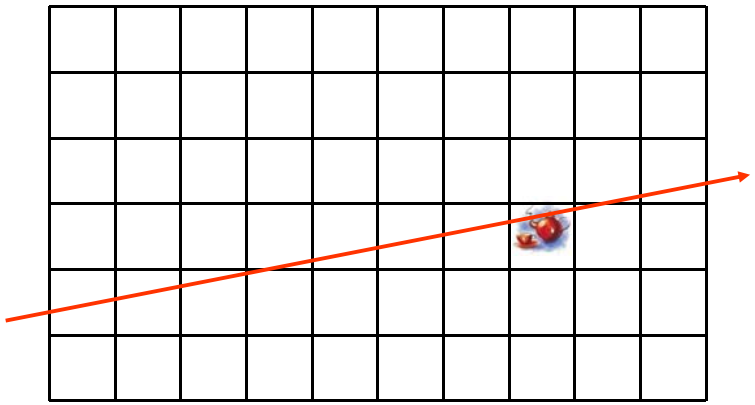
Grid accelerator

- Uniform grid



Teapot in a stadium problem

- Not adaptive to distribution of primitives.
- Have to determine the number of voxels. (problem with too many or too few)

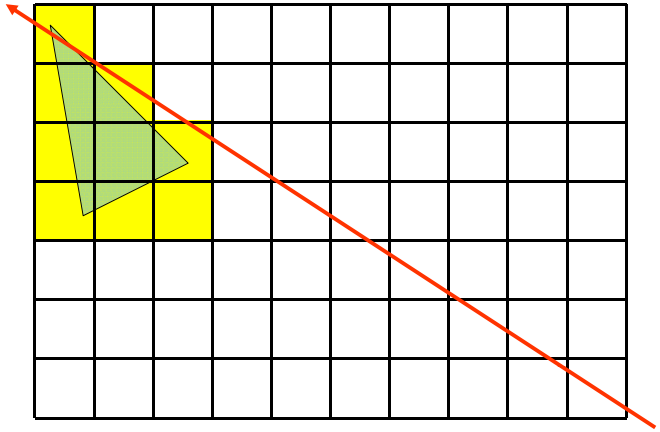


GridAccel

```
Class GridAccel:public Aggregate {
  <GridAccel methods>
  u_int nMailboxes;
  MailboxPrim *mailboxes;
  vector<Reference<Primitive>> primitives;
  int nVoxels[3];
  BBox bounds;
  Vector Width, InvWidth;
  Voxel **voxels;
  MemoryArena voxelArena;
  RWMutex *rwMutex;
}
```

mailbox

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



GridAccel

```
GridAccel(vector<Reference<Primitive> > &p,
    bool forRefined, bool refineImmediately)
  : gridForRefined(forRefined) {
// Initialize with primitives for grid
  if (refineImmediately)
     for (int i = 0; i < p.size(); ++i)
          p[i]->FullyRefine(primitives);
  else
     primitives = p;
  for (int i = 0; i < primitives.size(); ++i)
    bounds = Union(bounds,
                primitives[i]->WorldBound());
```

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 $3\sqrt[3]{N}$ partitions (N:number of primitives)

```
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

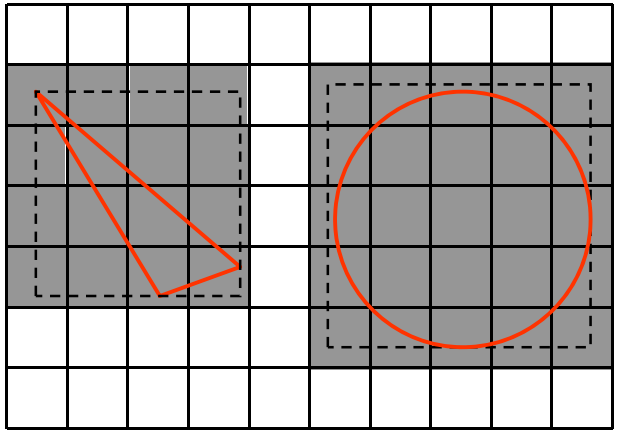
```
for (int axis=0; axis<3; ++axis) {
  nVoxels[axis]=Round2Int(delta[axis]*voxelsPerUnitDist);
  nVoxels[axis]=Clamp(nVoxels[axis], 1, 64);
}

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 nv = nVoxels[0] * nVoxels[1] * nVoxels[2];
voxels=AllocAligned<Voxel *>(nv);
memset(voxels, 0, nv * 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>
}
```



<Find voxel extent of primitive>

```
BBox pb = prims[i]->WorldBound();
int vmin[3], vmax[3];
for (int axis = 0; axis < 3; ++axis) {
  vmin[axis] = posToVoxel(pb.pMin, axis);
  vmax[axis] = posToVoxel(pb.pMax, axis);
}
```

<Add primitive to overlapping voxels>

```
for (int z = vmin[2]; z <= vmax[2]; ++z)
  for (int y = vmin[1]; y <= vmax[1]; ++y)
    for (int x = vmin[0]; x <= vmax[0]; ++x) {
      int o = offset(x, y, z);
      if (!voxels[o]) {
        voxels[o] = voxelArena.Alloc<Voxel>();
        *voxels[o] = Voxel(primitives[i]);
      }
      else {
      // Add primitive to already-allocated voxel
        voxels[o]->AddPrimitive(primitives[i]);
      }
    }
```

Voxel structure

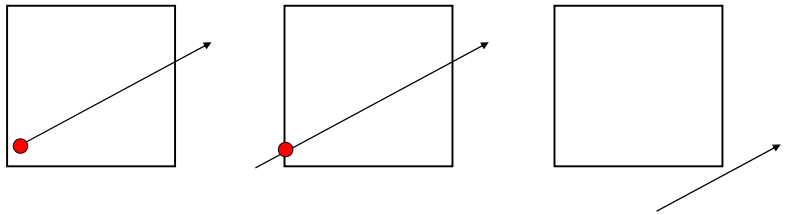
```
struct Voxel {
  <Voxel methods>
  vector<Reference<Primitive>> primitives;
  bool allCanIntersect;
}
Voxel(Reference<Primitive> op) {
  allCanIntersect = false;
  primitives.push_back(op);
}
void AddPrimitive(Reference<Primitive> p) {
  primitives.push_back(p);
}
```

`GridAccel` traversal

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

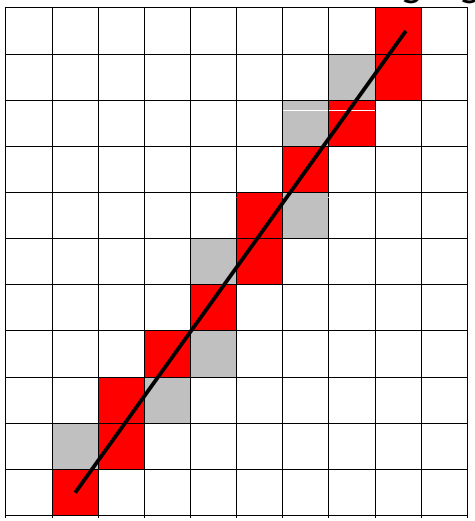
Check against overall bound

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

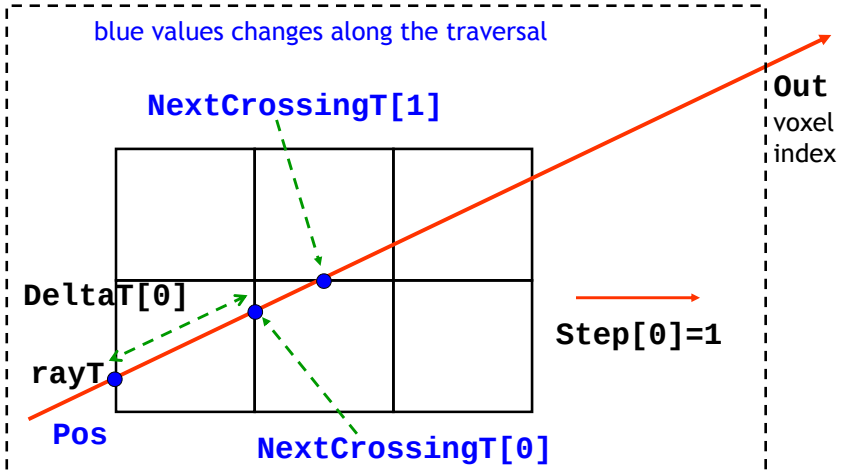


Set up 3D DDA (Digital Differential Analyzer)

- Similar to Bresenhm's line drawing algorithm



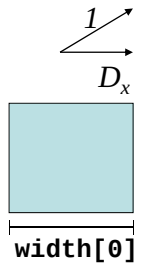
Set up 3D DDA (Digital Differential Analyzer)



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

Set up 3D DDA

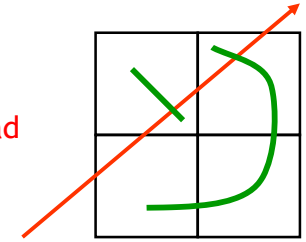
```
for (int axis=0; axis<3; ++axis) {
  Pos[axis]=posToVoxel(gridIntersect, axis);
  if (ray.d[axis]>=0) {
    NextCrossingT[axis] = rayT+
    (voxelToPos(Pos[axis]+1,axis)-gridIntersect[axis])
    /ray.d[axis];

    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
Return when entering a voxel that is beyond the closest found intersection.



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

Bounding volume hierarchies

- Object subdivision. Each primitive appears in the hierarchy exactly once. Additionally, the required space for the hierarchy is bounded.
- BVH v.s. Grid: both are efficient to build, but BVH provides much faster intersection.
- BVH v.s. Kd-tree: Kd-tree could be slightly faster for intersection, but takes much longer to build. In addition, BVH is generally more numerically robust and less prone to subtle round-off bugs.
- accelerators/bvh.*

BVHAccel

```
class BVHAccel : public Aggregate {
  <member functions>
  uint32_t maxPrimsInNode;
  enum SplitMethod { SPLIT_MIDDLE, SPLIT_EQUAL_COUNTS,
                     SPLIT_SAH };
  SplitMethod splitMethod;
  vector<Reference<Primitive> > primitives;
  LinearBVHNode *nodes;
}
```

BVHAccel construction

```
BVHAccel::BVHAccel(vector<Reference<Primitive> > &p,
                   uint32_t mp, const string &sm)
{
  maxPrimsInNode = min(255u, mp);
  for (uint32_t i = 0; i < p.size(); ++i)
    p[i]->FullyRefine(primitives);
  if (sm=="sah")        splitMethod =SPLIT_SAH;
  else if (sm=="middle") splitMethod =SPLIT_MIDDLE;
  else if (sm=="equal") splitMethod=SPLIT_EQUAL_COUNTS;
  else {
    Warning("BVH split method \"%s\" unknown.  Using
      \"sah\".", sm.c_str());
    splitMethod = SPLIT_SAH;
  }
```

BVHAccel construction

<Initialize **buildData** *array for primitives>*

<Recursively build BVH tree for primitives>

<compute representation of depth-first traversal of BVH tree>

```
}
```

It is possible to construct a pointer-less BVH tree directly, but it is less straightforward.

Initialize buildData array

```
vector<BVHPrimitiveInfo> buildData;
buildData.reserve(primitives.size());
for (int i = 0; i < primitives.size(); ++i)
{
  BBox bbox = primitives[i]->WorldBound();
  buildData.push_back(
                BVHPrimitiveInfo(i, bbox));
}
```

```
struct BVHPrimitiveInfo {
    BVHPrimitiveInfo() { }
    BVHPrimitiveInfo(int pn, const BBox &b)
        : primitiveNumber(pn), bounds(b) {
        centroid = .5f * b.pMin + .5f * b.pMax;
    }
    int primitiveNumber;
    Point centroid;
    BBox bounds;
};
```

Recursively build BVH tree

```
MemoryArena buildArena;
uint32_t totalNodes = 0;
vector<Reference<Primitive> > orderedPrims;
orderedPrims.reserve(primitives.size());

BVHBuildNode *root = recursiveBuild(buildArena,
  buildData, 0, primitives.size(), &totalNodes,
  orderedPrims);
```

[start end)

```
primitives.swap(orderedPrims);
```

BVHBuildNode

```
struct BVHBuildNode {
  void InitLeaf(int first, int n, BBox &b) {
    firstPrimOffset = first;
    nPrimitives = n; bounds = b;
  }
  void InitInterior(int axis, BVHBuildNode *c0,
                    BVHBuildNode *c1) {
    children[0] = c0; children[1] = c1;
    bounds = Union(c0->bounds, c1->bounds);
    splitAxis = axis; nPrimitives = 0;
  }
  BBox bounds;
  BVHBuildNode *children[2];
  int splitAxis, firstPrimOffset, nPrimitives;
};
```

The leaf contains primitives from BVHAccel::primitives[firstPrimOffset] to [firstPrimOffset+nPrimitives-1]

recursiveBuild

- Given n primitives, there are in general 2^n-2 possible ways to partition them into two non-empty groups. In practice, one considers partitions along a coordinate axis, resulting in 6n candidate partitions.

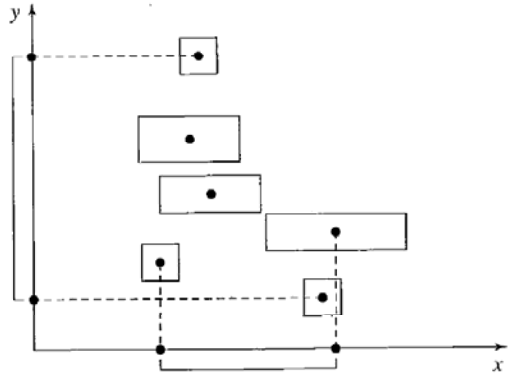
1. Choose axis
2. Choose split
3. Interior(dim,

```
        recursiveBuild(.., start, mid, ..),
        recursiveBuild(.., mid, end, ..)
        )
```

Choose axis

```
BBox cBounds;
for (int i = start; i < end; ++i)
  cBounds=Union(cBounds, buildData[i].centroid);
int dim = centroidBounds.MaximumExtent();
```

If cBounds has zreo volume, create a leaf



Choose split (split_middle)

```
float pmid = .5f * (centroidBounds.pMin[dim] +
                    centroidBounds.pMax[dim]);
BVHPrimitiveInfo *midPtr = std::partition(
  &buildData[start], &buildData[end-1]+1,
  CompareToMid(dim, pmid));
mid = midPtr - &buildData[0];
```

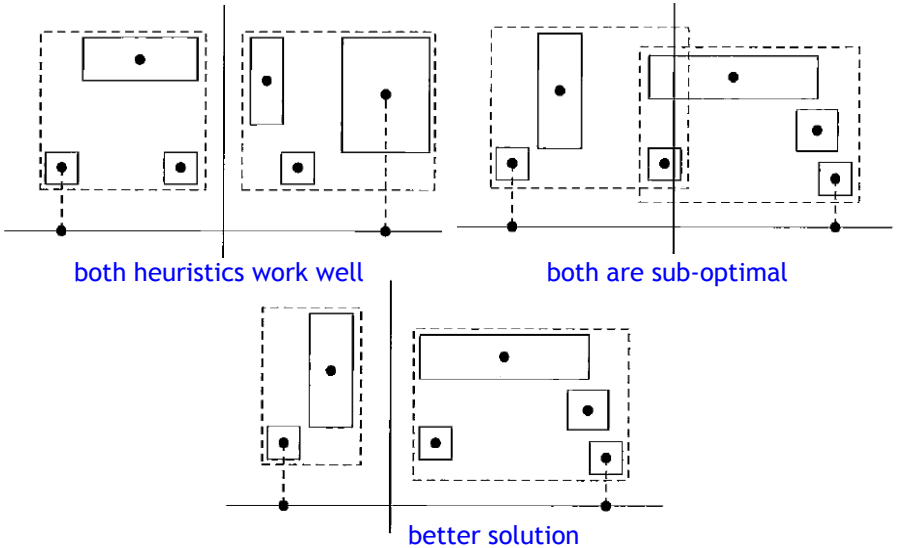
Return true if the given primitive's bound's centroid is below the given midpoint

Choose split (split_equal_count)

```
mid = (start + end) / 2;
std::nth_element(&buildData[start],
     &buildData[mid], &buildData[end-1]+1,
     ComparePoints(dim));
```

It orders the array so that the middle pointer has median, the first half is smaller and the second half is larger in O(n).

Choose split



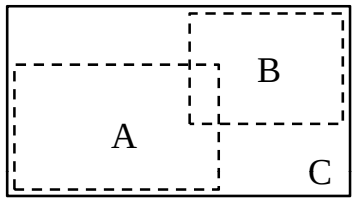
Choose split (split_SAH)

Do not split

$$\sum_{i=1}^{N} t_{isect}(i)$$

split

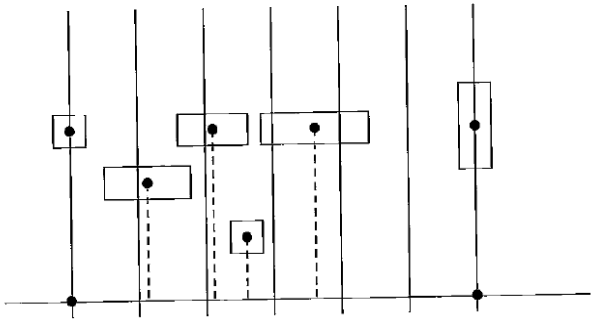
$$c(A,B) = t_{trav} + p_A \sum_{i=1}^{N_A} t_{isect}(a_i) + p_B \sum_{i=1}^{N_B} t_{isect}(b_i)$$



$$p_A = \frac{s_A}{s_C} \qquad p_B = \frac{s_B}{s_C}$$

Choose split (split_SAH)

- If there are no more than 4 primitives, use equal size heuristics instead.
- Instead of testing 2n candidates, the extend is divided into a small number (12) of buckets of equal extent. Only buck boundaries are considered.



Choose split (split_SAH)

```
const int nBuckets = 12;
struct BucketInfo {
  int count;  BBox bounds;
};
BucketInfo buckets[nBuckets];
for (int i=start; i<end; ++i) {
  int b = nBuckets *
((buildData[i].centroid[dim]-centroidBounds.pMin[dim])/
 (centroidBounds.pMax[dim]-centroidBounds.pMin[dim]));
  if (b == nBuckets) b = nBuckets-1;
  buckets[b].count++;
  buckets[b].bounds = Union(buckets[b].bounds,
  buildData[i].bounds);
}
```

Choose split (split_SAH)

```
float cost[nBuckets-1];
for (int i = 0; i < nBuckets-1; ++i) {
  BBox b0, b1;
  int count0 = 0, count1 = 0;
  for (int j = 0; j <= i; ++j) {
    b0 = Union(b0, buckets[j].bounds);
    count0 += buckets[j].count; }
  for (int j = i+1; j < nBuckets; ++j) {
    b1 = Union(b1, buckets[j].bounds);
    count1 += buckets[j].count; }
  cost[i] = .125f + (count0*b0.SurfaceArea() +
  count1*b1.SurfaceArea())/bbox.SurfaceArea();
}
```

Traverse cost : Intersection cost = 1 : 8

Choose split (split_SAH)

```
float minCost = cost[0];  uint32_t minCostSplit = 0;
for (int i = 1; i < nBuckets-1; ++i) {
  if (cost[i] < minCost) {
    minCost = cost[i];
    minCostSplit = i;
  }
}
if (nPrimitives > maxPrimsInNode ||
    minCost < nPrimitives) {
  BVHPrimitiveInfo *pmid =
  std::partition(&buildData[start],&buildData[end-
  1]+1, CompareToBucket(minCostSplit, nBuckets, dim,
  centroidBounds));
  mid = pmid - &buildData[0];
} else <create a leaf>
```

Compact BVH

- The last step is to convert the BVH tree into a compact representation which improves cache, memory and thus overall performance.

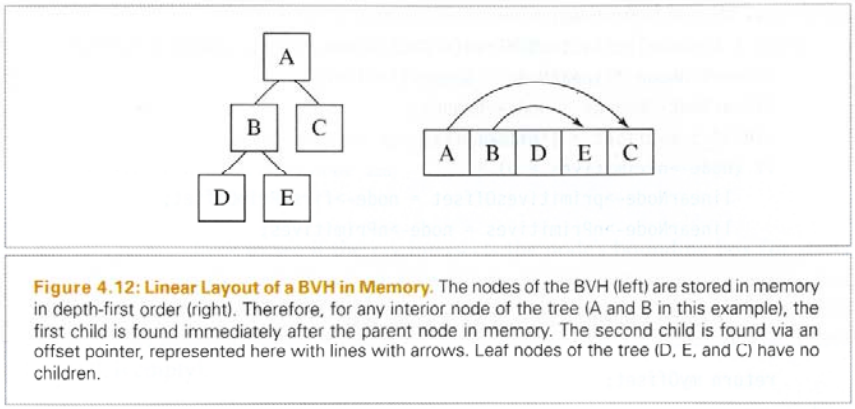


Figure 4.12: Linear Layout of a BVH in Memory. The nodes of the BVH (left) are stored in memory in depth-first order (right). Therefore, for any interior node of the tree (A and B in this example), the first child is found immediately after the parent node in memory. The second child is found via an offset pointer, represented here with lines with arrows. Leaf nodes of the tree (D, E, and C) have no children.

BVHAccel traversal


```
bool BVHAccel::Intersect(const Ray &ray,
  Intersection *isect) const {
  if (!nodes) return false;

  bool hit = false;
  Point origin = ray(ray.mint);
  Vector invDir(1.f / ray.d.x, 1.f / ray.d.y,
                1.f / ray.d.z);
  uint32_t dirIsNeg[3]={ invDir.x < 0, invDir.y < 0,
                         invDir.z < 0 };

  uint32_t nodeNum = 0; offset into the nodes array to be visited
  uint32_t todo[64];  nodes to be visited; acts like a stack
  uint32_t todoOffset = 0; next free element in the stack
```

BVHAccel traversal


```
while (true) {
  const LinearBVHNode *node = &nodes[nodeNum];
  if (::IntersectP(node->bounds,ray,invDir,dirIsNeg)){
    if (node->nPrimitives > 0) {        leaf node
      // Intersect ray with primitives in leaf BVH node
      for (uint32_t i = 0; i < node->nPrimitives; ++i){
        if (primitives[node->primitivesOffset+i]
                        ->Intersect(ray, isect))
          hit = true;
      }
      if (todoOffset == 0) break;
      nodeNum = todo[--todoOffset];
    }
```

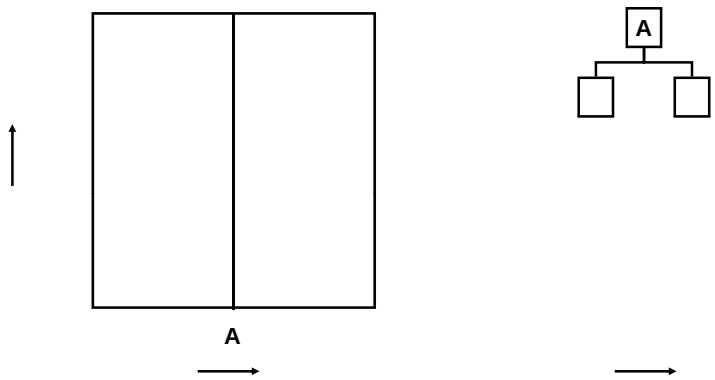
BVHAccel traversal


```
    else {                                   interior node
      if (dirIsNeg[node->axis]) {
        todo[todoOffset++] = nodeNum + 1;
        nodeNum = node->secondChildOffset;
      }
      else {
        todo[todoOffset++] = node->secondChildOffset;
        nodeNum = nodeNum + 1;
      }
    }
  }
  else { Do not hit the bounding box; retrieve the next one if any
    if (todoOffset == 0) break;
    nodeNum = todo[--todoOffset];
  }
```

KD-Tree accelerator

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

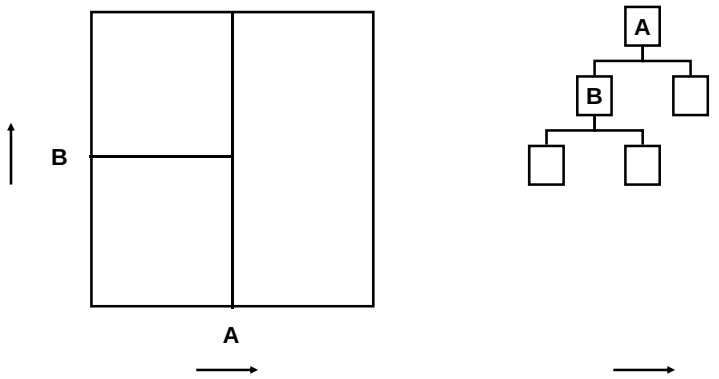
Spatial hierarchies



Letters correspond to planes (A)

Point Location by recursive search

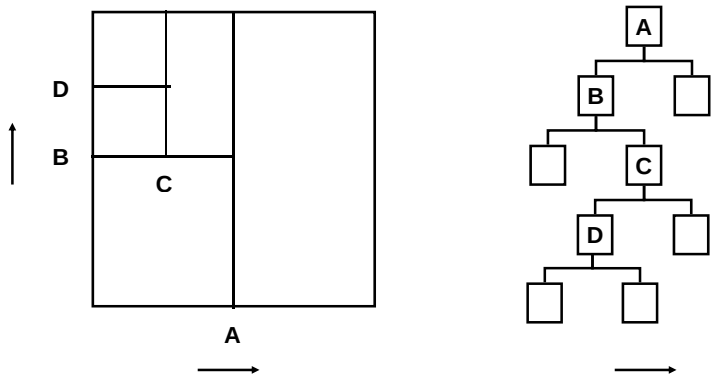
Spatial hierarchies



Letters correspond to planes (A, B)

Point Location by recursive search

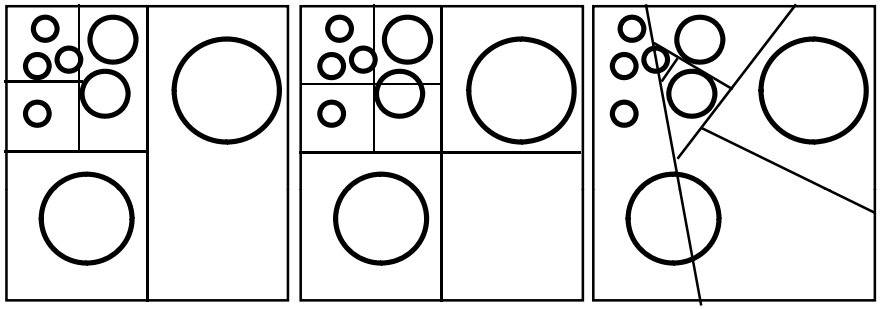
Spatial hierarchies



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

Point Location by recursive search

Variations



kd-tree octree bsp-tree

"Hack" kd-tree building

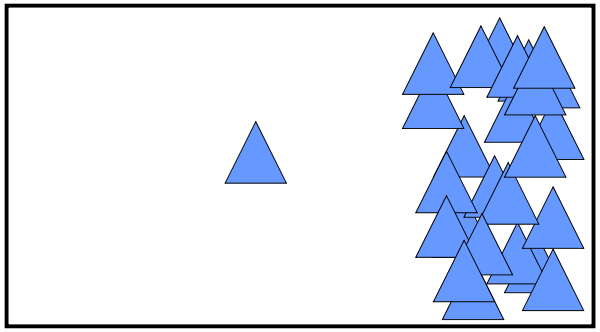
- Split axis
  - Round-robin; largest extent
- Split location
  - Middle of extent; median of geometry (balanced tree)
- Termination
  - Target # of primitives, limited tree depth
- All of these techniques stink.

Building good kd-trees

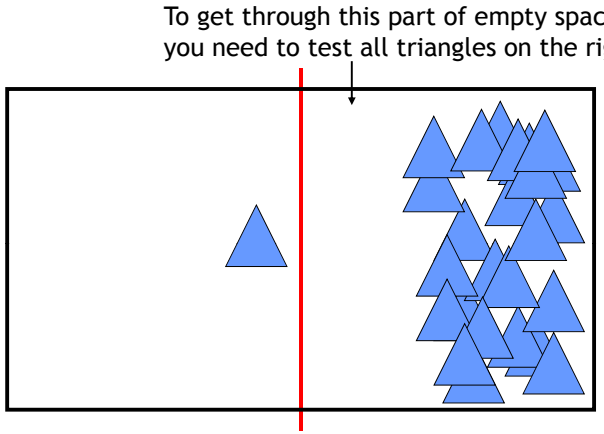
- What split do we really want?
  - Clever Idea: the one that makes ray tracing cheap
  - Write down an expression of cost and minimize it
  - Greedy cost optimization
- What is the cost of tracing a ray through a cell?

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

Splitting with cost in mind

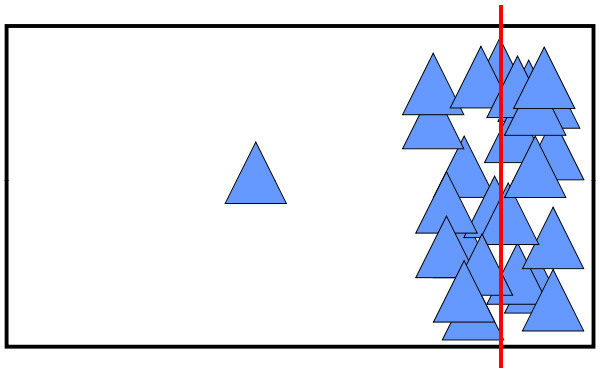


Split in the middle



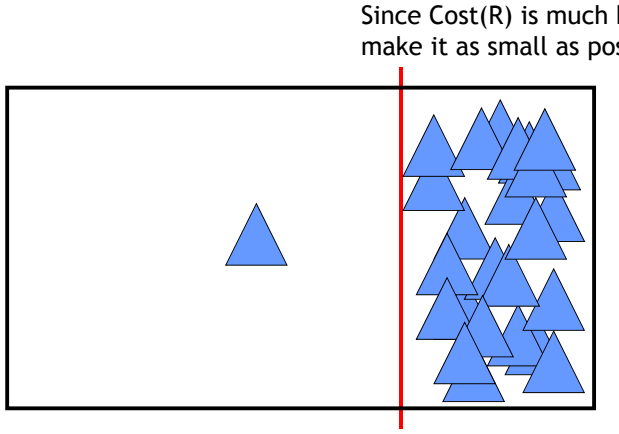
- Makes the L & R probabilities equal
- Pays no attention to the L & R costs

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”

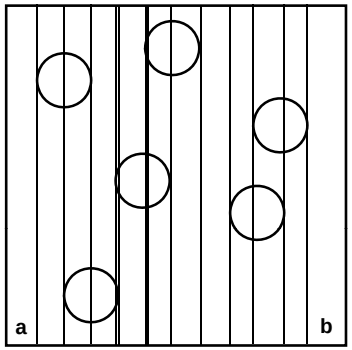
Cost(cell) = C_trav + Prob(hit L) * Cost(L) + Prob(hit R) * Cost(R)

= 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; must coincides with object boundary. Why?

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

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

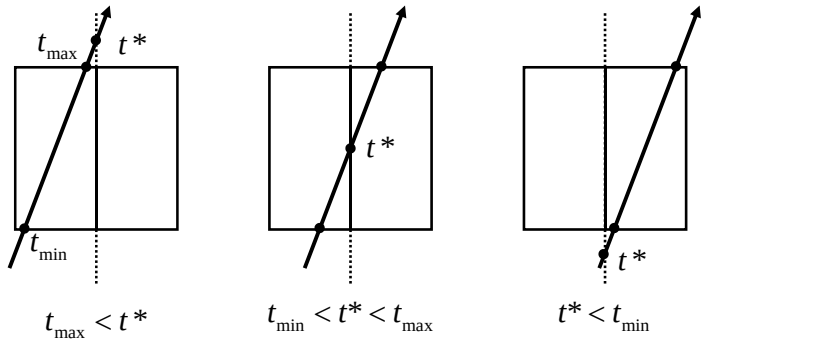
Running time: $T(N) = N \log N + 2T(N/2)$

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

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

Ray traversal algorithm

- Recursive inorder traversal

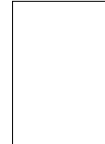


$t_{max} < t*$ | $t_{min} < t* < t_{max}$ | $t* < t_{min}$

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

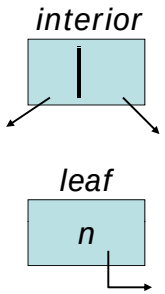
a video for kdtree

Tree representation

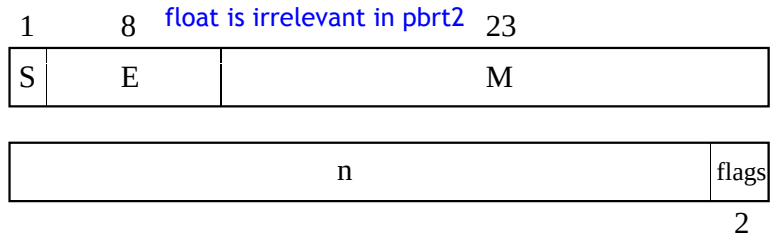


8-byte (reduced from 16-byte, 20% gain)

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



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: $\sum_{k=1}^{N} t_i(k)$

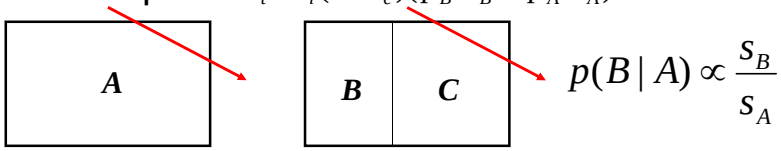
cost of split: $t_t + P_B \sum_{k=1}^{N_B} t_i(b_k) + P_A \sum_{k=1}^{N_A} t_i(a_k)$

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: $t_i N$

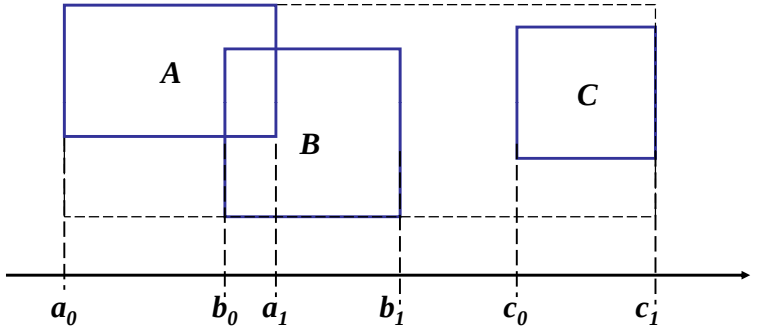
cost of split: $t_t + t_i(1-b_e)(p_B N_B + p_A N_A)$



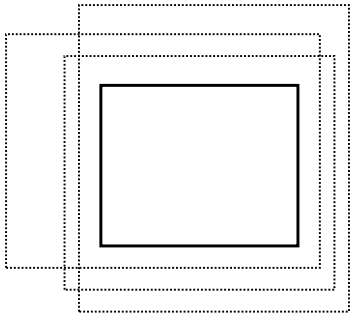
$$p(B \mid A) \propto \frac{s_B}{s_A}$$

Choose split axis position

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

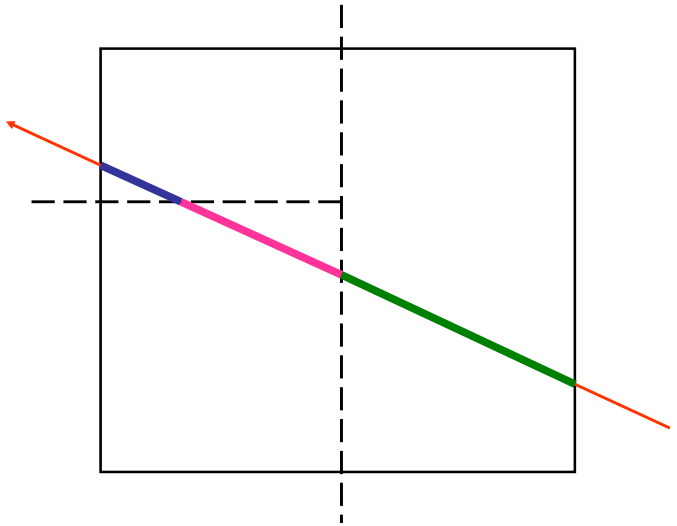


Choose split axis position

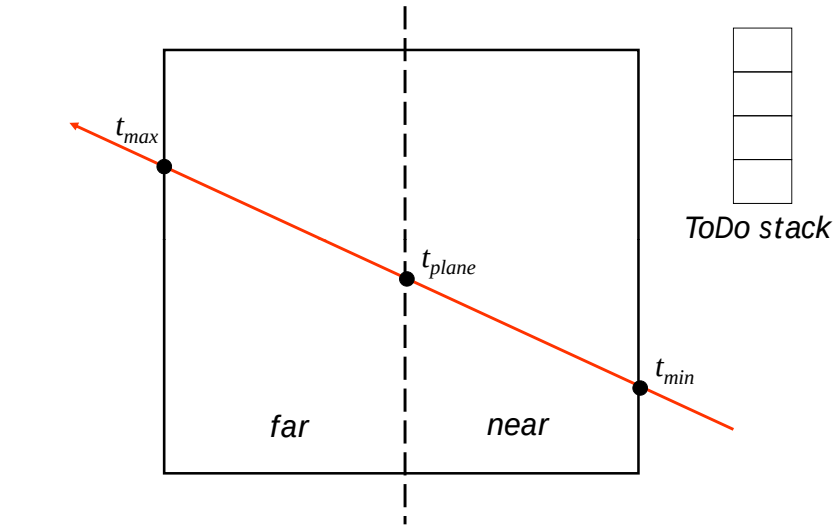


If there is no split along this axis, try other axes.
When all fail, create a leaf.

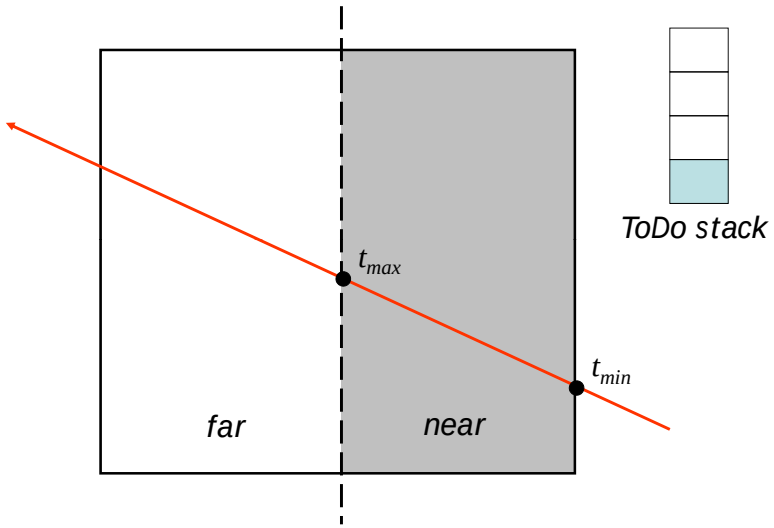
`KdTreeAccel` traversal



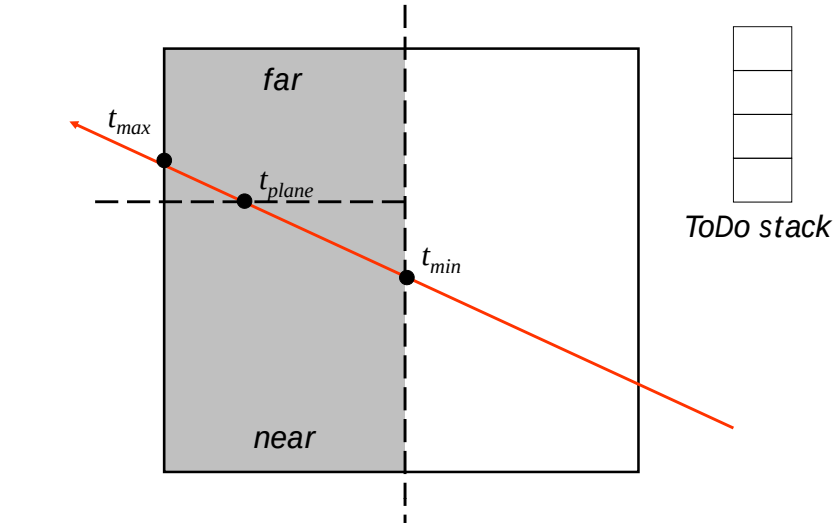
KdTreeAccel traversal



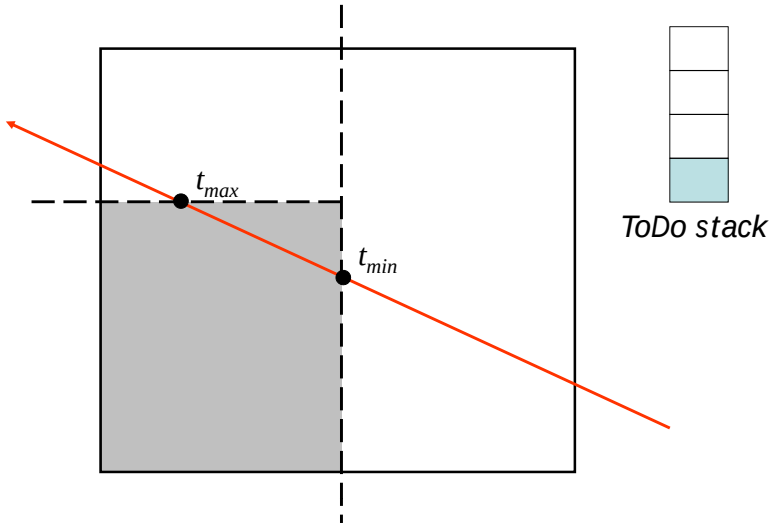
KdTreeAccel traversal



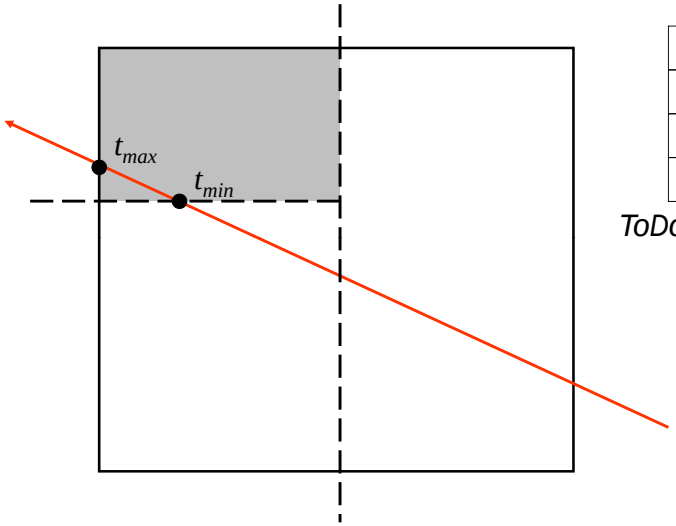
KdTreeAccel traversal

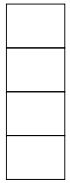


KdTreeAccel traversal



KdTreeAccel traversal

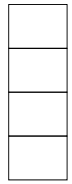




ToDo stack

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>
  }
}
```



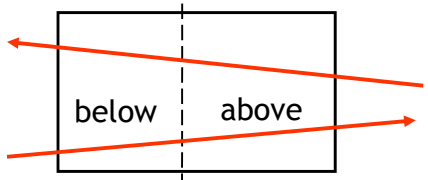
ToDo stack (max depth)

Leaf node

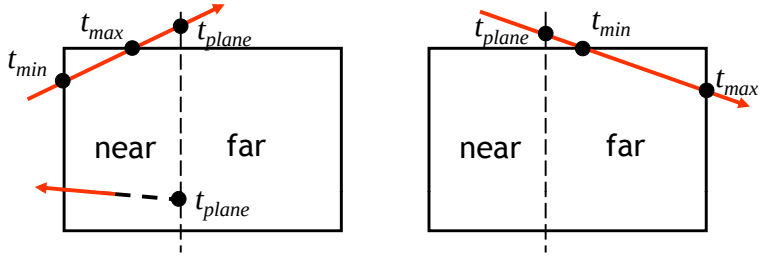
1. Check whether ray intersects primitive(s) inside the node; update ray's **`maxt`**
2. Grab next node from ToDo queue

Interior node

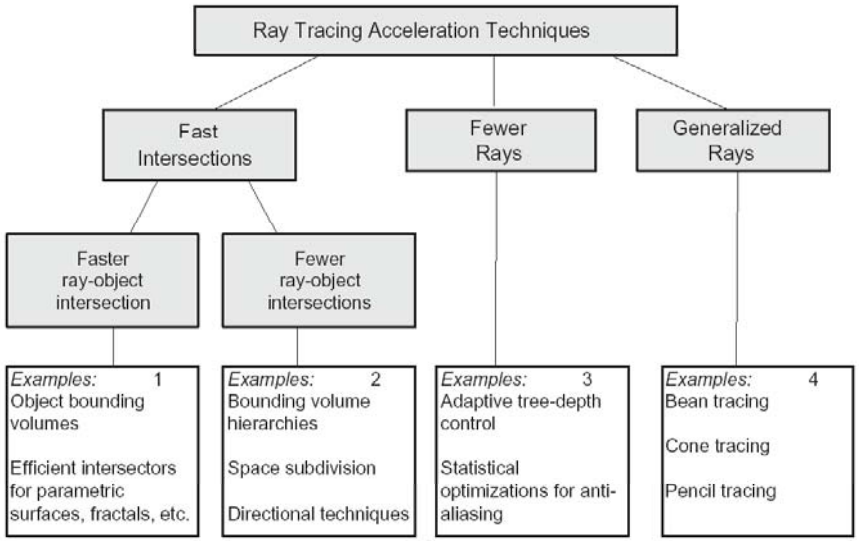
1. Determine near and far (by testing which side O is)



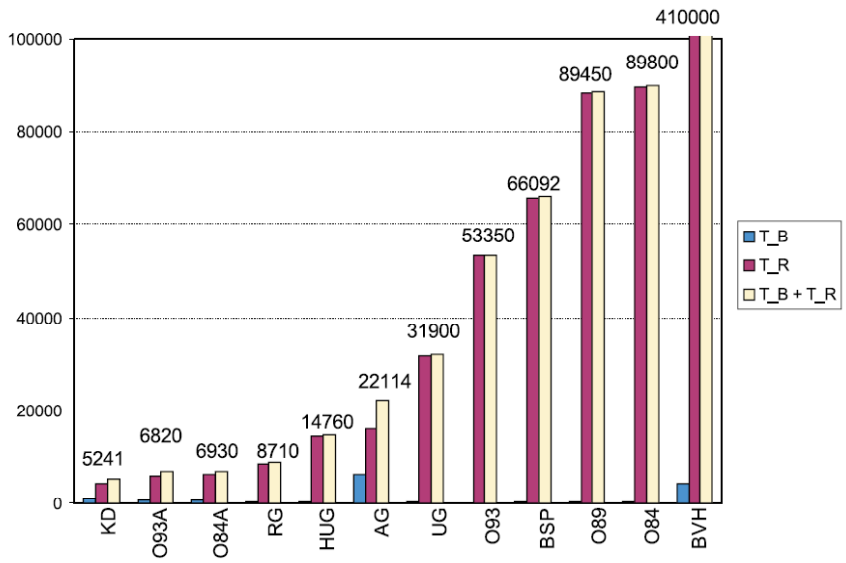
2. Determine whether we can skip a node



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