Geometric Structures

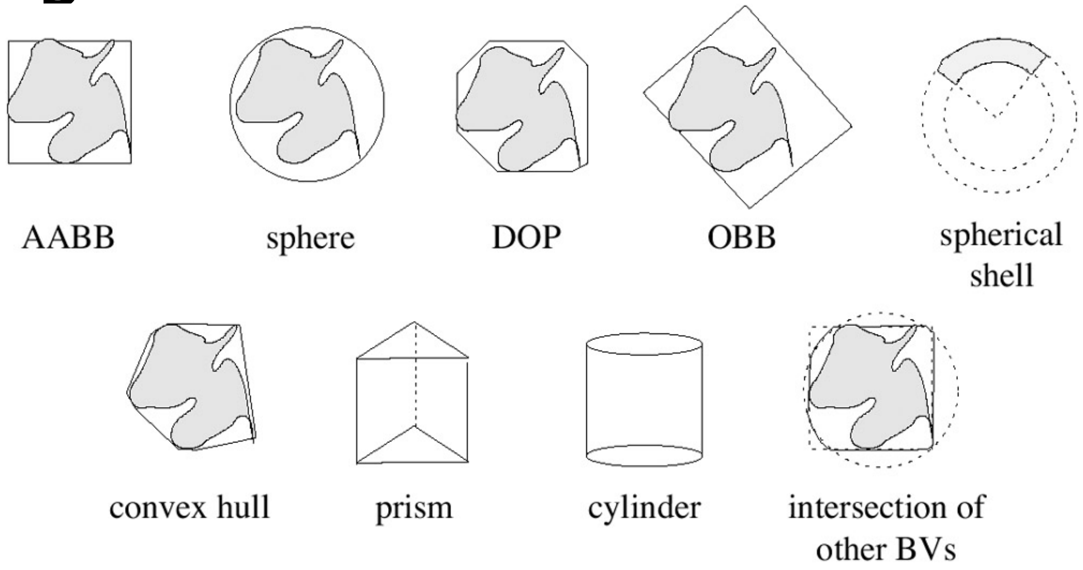
4. Bounding Volumes

Martin Samuelčík

samuelcik@sccg.sk, www.sccg.sk/~samuelcik, I4

Bounding volumes

- Auxiliary set containing one or more objects
- Simple object shape, usually not suitable for exact representation
- Transfer of computation from object to bounding volume



Bounding volumes

- http://www.geometrictools.com/index.html
- Algorithm prerequisites
  - Creating volume from set of points, …
  - Volume approximation precision
  - Intersection with other bounding volume
  - Intersection with line
  - Point location
  - Rigid body object transformations (rotation, scale, translation)
  - Union of two volumes, creating new volume

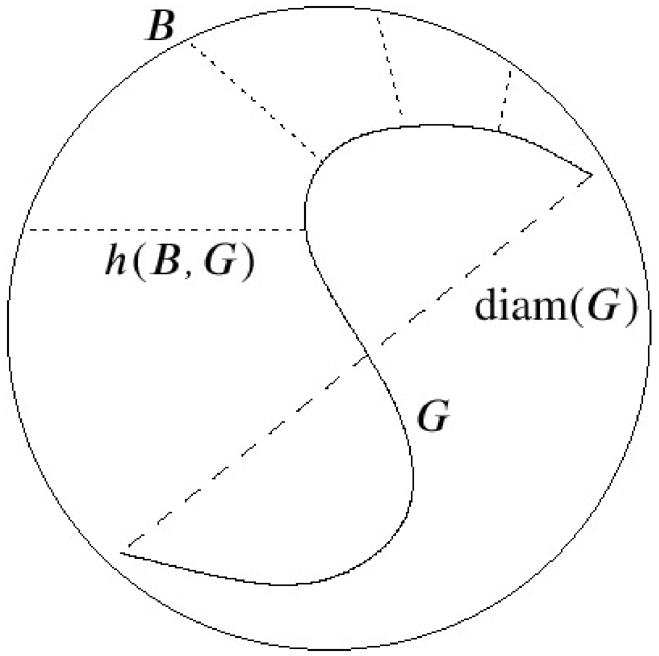
Approximation precision

- Cost, distance function between object and its bounding volume
- Using Hausdorff distance
- Object volume matters

$$h(B, G) = \max_{b \in B} \min_{g \in G} d(b, g)$$

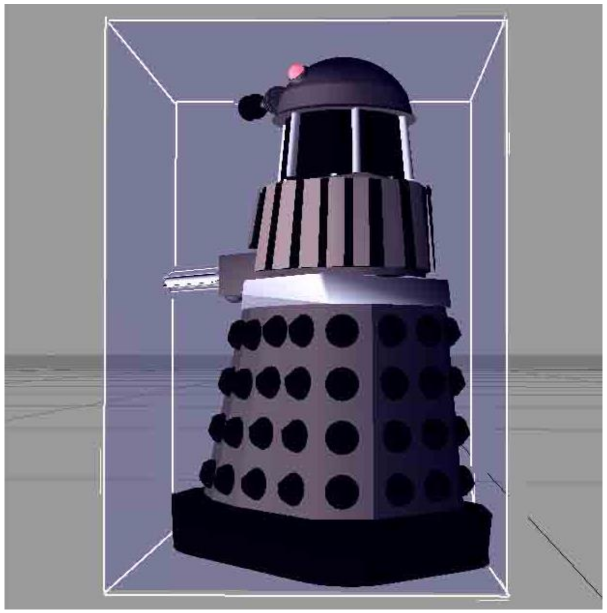
$$\mathrm{diam}(G) = \max_{g, f \in G} d(g, f)$$

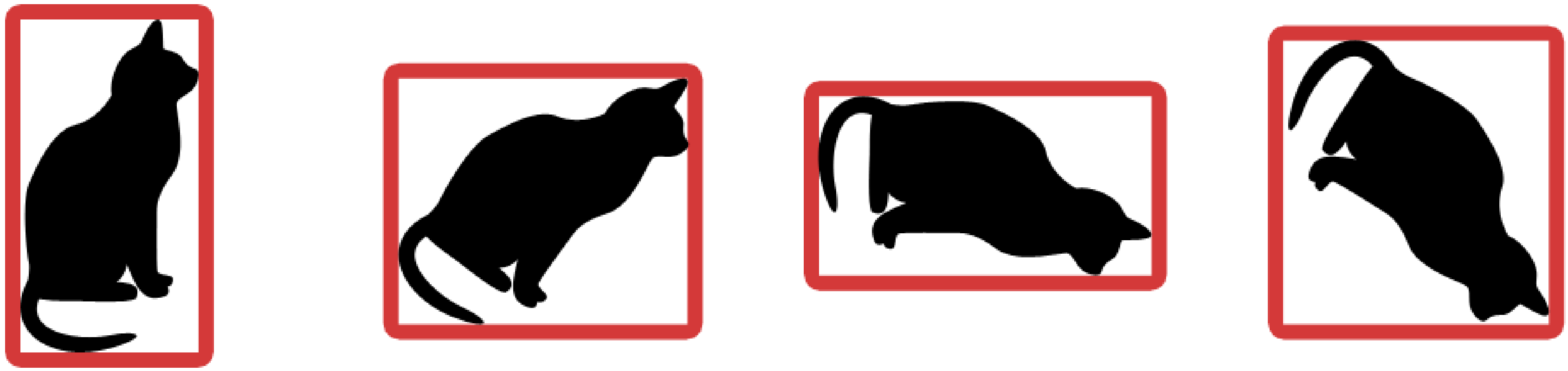
$$\tau := \frac{h(B, G)}{\mathrm{diam}(G)}.$$



AABB

- Axis-aligned bounding box
- d-dimensional interval
- Simple volume creation
- Simple intersections
- Complex transformations
- Worst approximation, max. 0.5





AABB

- Creation – finding min-max values
- Intersection – comparing min-max values
- Transformation – recomputations needed

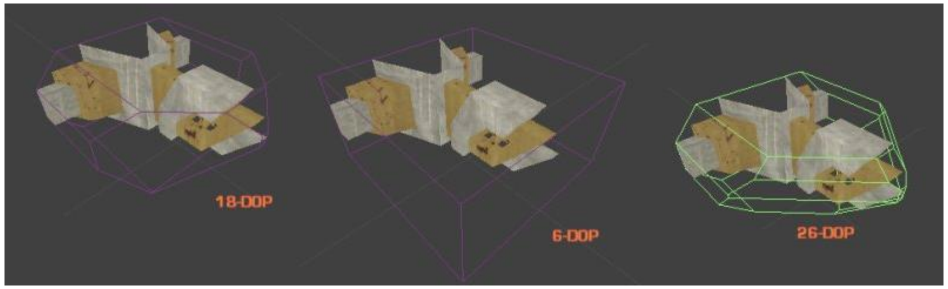
```
struct AABB
{
   float fMinX, fMaxX;
   float fMinY, fMaxY;
   float fMinZ, fMaxZ;
}
```

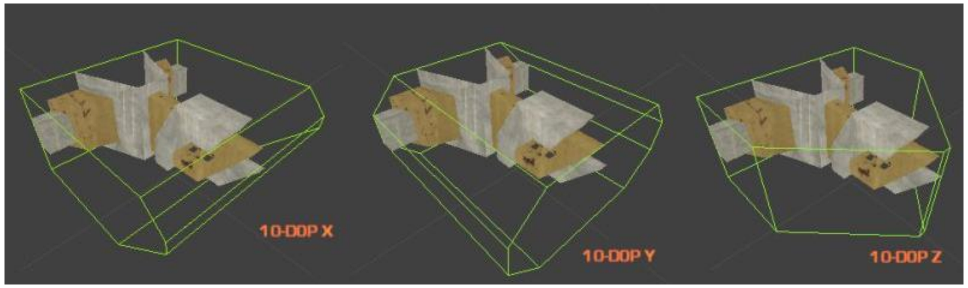
```
AABBIntersection(AABB v1, AABB v2)
{
   if (v1.fMaxX < v2.fMinX) return false;
   if (v1.fMinX > v2.fMaxX) return false;
   if (v1.fMaxY < v2.fMinY) return false;
   if (v1.fMinY > v2.fMaxY) return false;
   if (v1.fMaxZ < v2.fMinZ) return false;
   if (v1.fMinZ > v2.fMaxZ) return false;
   return true;
}
```

```
CreateAABB(vector<Point3D> points)
{
   AABB result;
   result.fMinX = result.fMinY = result,fMinZ = MAX_FLOAT;
   result.fMaxX = result.fMaxY = result.fMaxZ = MIN_FLOAT;
   for (int i = 0;i < points.size();i++)
   {
      if (points[i].x < result.fMinX) result.fMinX = points[i].x;
      if (points[i].x > result.fMaxX) result.fMaxX = points[i].x;
      if (points[i].y < result.fMinY) result.fMinY = points[i].y;
      if (points[i].y > result.fMaxY) result.fMaxY = points[i].y;
      if (points[i].z < result.fMinZ) result.fMinZ = points[i].z;
      if (points[i].z > result.fMaxZ) result.fMaxZ = points[i].z;
   }
   return result;
}
```

k-DOP

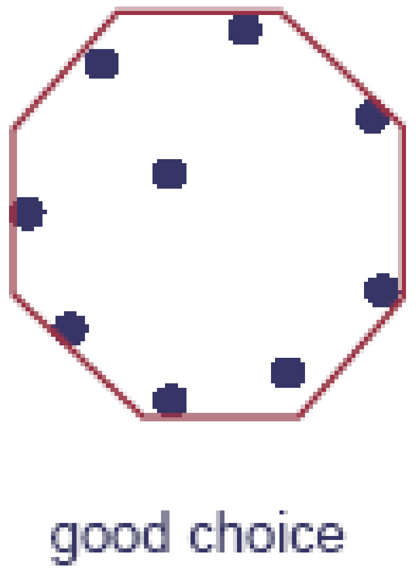
- Discrete Oriented Polytope, Polytop – polygon, polyhedron...
- Intersection of k half-spaces, convex hull is smallest k-DOP
- Usually defined by k / 2 directions and represented by min and max values in each direction
- AABB is k-DOP with directions given by coordinate axis

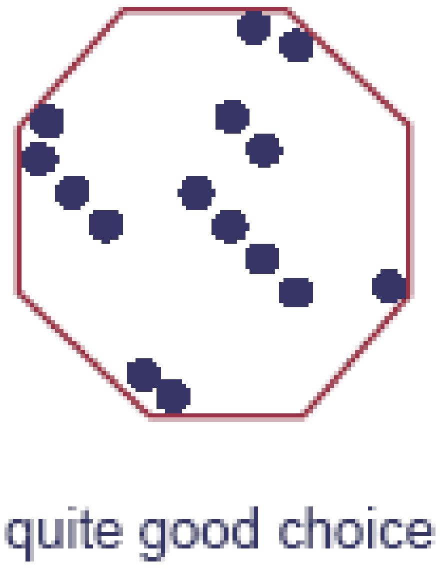




k-DOP

- Better approximation, depends on data
- 2D: k = 4, 8; 3D: k = 6, 18, 26
- Similar algorithms complexity as for AABB





```
struct kDOP
{
   static vector<Vector3> directions;
   Point3 center;
   vector<float> min_values;
   vector<float> max_values;
}
```

k-DOP

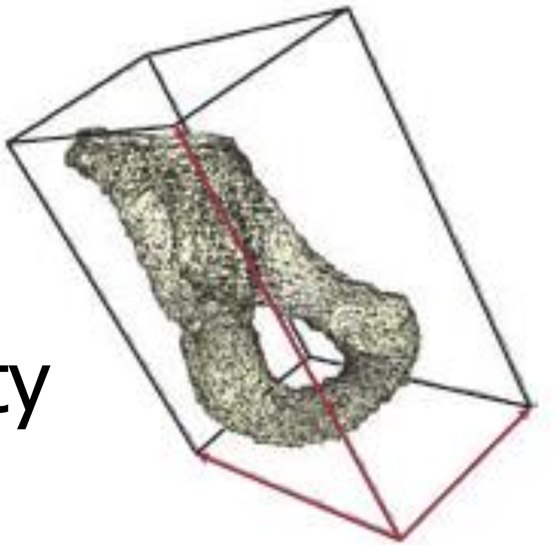
```
CreatekDOP(vector<Point3D> points)
{
   kDOP result;
   result.center.x = result.center.y = result.center.z = 0;
   for (int i = 0;i < points.size();i++)
   {
       result.center.x += points[i].x / points.size();
       result.center.y += points[i].y / points.size();
       result.center.z += points[i].z / points.size();
   }
   for (int j = 0;j < result.directions.size();j++)
   {
       result.min_values[j].push_back(MAX_FLOAT);
       result.max_values[j].push_back(MIN_FLOAT);
   }
   for (int i = 0;i < points.size();i++)
   {
      Point3 X = points[i];
      for (int j = 0;j < result.directions.size();j++)
      {
          Vector3 v = result.directions[j]; Point3 O = result.center;
          float t = (X.x*v.x+X.y*v.y+X.z*v.z-O.x*v.x-O.y*v.y-O.z*v.z);
          t = t / (v.x*v.x+v.y*v.y+v.z*v.z);
          if (t < result.min_values[j]) result.min_values[j] = t;
          if (t > result.max_values[j]) result.max_values[j] = t;
      }
   }  return result;
}
```

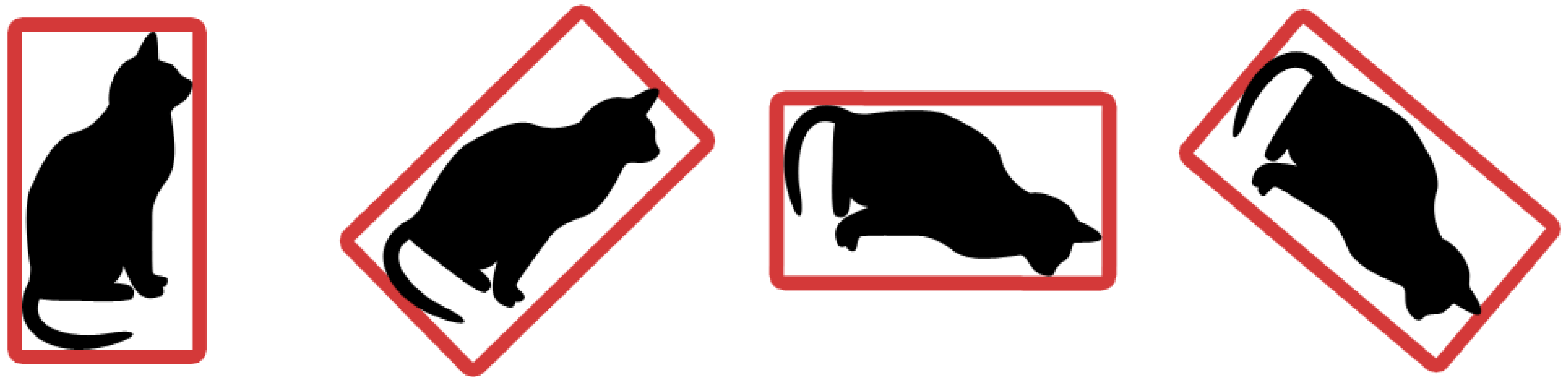
```
kDOPIntersection(kDOP v1, kDOP v2)
{
   Point3 O1 = v1.center; Point3 O2 = v2.center;
   for (int j = 0;j < v1.directions.size();j++)
   {
       Vector3 v = v1.directions[j];
       float t = (O2.x*v.x+O2.y*v.y+O2.z*v.z-O1.x*v.x-O1.y*v.y-O1.z*v.z);
       t = t / (v.x*v.x+v.y*v.y+v.z*v.z);
       if (v1.min_values[j] > (v2.max_values[j] + t)) return false;
       if (v1.max_values[j] < (v2.min_values[j] + t)) return false;
   }
   return true;
}
```

```
PointInsidekDOP(kDOP v,Point3 P)
{
   Point3 O = v.center;
   for (int j = 0;j < v.directions.size();j++)
   {
       Vector3 v = v.directions[j];
       float t = (P.x*v.x+P.y*v.y+P.z*v.z-O.x*v.x-O.y*v.y-O.z*v.z);
       t = t / (v.x*v.x+v.y*v.y+v.z*v.z);
       if (v.min_values[j] > t) return false;
       if (v.max_values[j] < t) return false;
   }
   return true;
}
```

OBB

- Oriented Bounding Box
- Rotated d-dimensional interval
- Better approximation
- Test with higher time complexity
- Better transformations





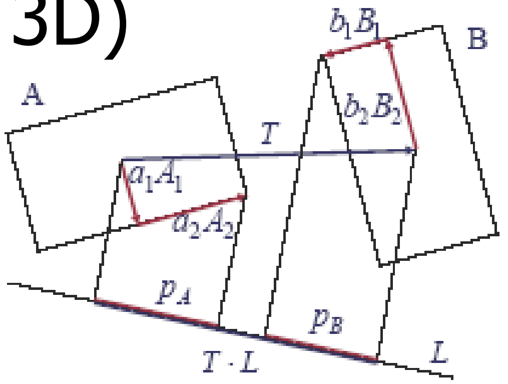
OBB

- Intersection of two OBB – finding „separating axis“
  - Line on which projections of both OBB do not intersect
- Two objects *A* and *B* do not intersect if there is some line *p* such that projections of *A* and *B* onto line p do not intersect
- Direction *p* is defined using boundary polygons orientations of objects *A* and *B* and using cross product of edges from *A* and *B* (in 3D)

$$p_A = |a_1 A_1 L| + |a_2 A_2 L|$$

$$p_B = |b_1 B_1 L| + |b_2 B_2 L|$$

$\exists L : |T \cdot L| > p_A + p_B$ or $\exists L \in \{A_1, A_2, B_1, B_2\} : |T \cdot L| > p_A + p_B$



OBB

```
struct OBB
{
   Point3 center;
   Vector3 v1, v2, v3;
   float fMin1, fMin2, fMin3;
   float fMax1, fMax2, fMax3;
}
```

```
ComputeDirections(vector<Point3> points)
{
   Point3 center;
   center.x = center.y = center.z = 0;
   for (int i = 0;i < points.size();i++)
   {
       center.x += points[i].x / points.size();
       center.y += points[i].y / points.size();
       center.z += points[i].z / points.size();
   }
   Matrix3 C; C[0][0] = C[1][0] = C[2][0] = C[0][1] = C[1][1] = … = 0;
   for (int i = 0;i < points.size();i++)
   {
       C[0][0] += (points[i].x – center.x)*(points[i].x – center.x) / points.size();
       C[1][0] += (points[i].y – center.y)*(points[i].x – center.x) / points.size();
       C[2][0] += (points[i].z – center.z)*(points[i].x – center.x) / points.size();
       C[0][1] += (points[i].x – center.x)*(points[i].y – center.y) / points.size();
       ......
   }
   return C.Eigenvectors();
}
```

```
CreateOBB(vector<Point3> points)
{
   OBB result;
   result.center.x = result.center.y = result.center.z = 0;
   for (int i = 0;i < points.size();i++)
   {
       result.center.x += points[i].x / points.size();
       result.center.y += points[i].y / points.size();
       result.center.z += points[i].z / points.size();
   }
   (result.v1, result.v2, result.v3) = ComputeDirections(points);
   result.fMin1 = result.fMin2 = result.fMin3 = MAX_FLOAT;
   result.fMax1 = result.fMax2 = result.fMax3 = MIN_FLOAT;
   for (int i = 0;i < points.size();i++)
   {
       Point3 X = points[i];
       float t1 = (X.x*v1.x+X.y*v1.y+X.z*v1.z);
       t1 = t1 / (v1.x*v1.x+v1.y*v1.y+v1.z*v1.z);
       if (t1 < result.fMin1) result.fMin1 = t1;
       if (t1 > result.fMax1) result.fMax1 = t1;
       // …. Same for other two directions
   }
   return result;
}
```

OBB

```
PointInsideOBB(Point3 P, OBB v)
{
   Point3 O = v.origin;
   float t = (P.x*v.v1.x+P.y*v.v1.y+P.z*v.v1.z-O.x*v.v1.x-O.y*v.v1.y-O.z*v.v1.z);
   t = t / (v.v1.x*v.v1.x+v.v1.y*v.v1.y+v.v1.z*v.v1.z);
   if (t < v.fMin1) return false;
   if (t > v.fMax1) return false;
   t = (P.x*v.v2.x+P.y*v.v2.y+P.z*v.v2.z-O.x*v.v2.x-O.y*v.v2.y-O.z*v.v2.z);
   t = t / (v.v2.x*v.v2.x+v.v2.y*v.v2.y+v.v2.z*v.v2.z);
   if (t < v.fMin2) return false;
   if (t > v.fMax2) return false;
   t = (P.x*v.v3.x+P.y*v.v3.y+P.z*v.v3.z-O.x*v.v3.x-O.y*v.v3.y-O.z*v.v3.z);
   t = t / (v.v3.x*v.v3.x+v.v3.y*v.v3.y+v.v3.z*v.v3.z);
   if (t < v.fMin3) return false;
   if (t > v.fMax3) return false;
   return true;
}
```

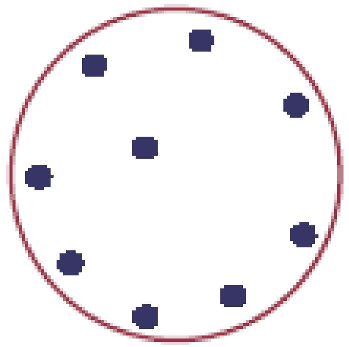
```
RayPlaneIntersection(Ray r, Vector3 normal, float plane4)
{
   Point3 result;
   float t = -plane4 – normal.x*r.start.x – normal.y*r.start.y – normal.z*r.start.z;
   t = t / (normal.x*v.x+normal.y*v.y+normal.z*v.z);
   result.x = r.start.x + t * v.dir.x;
   result.y = r.start.y + t * v.dir.y;
   result.z = r.start.z + t * v.dir.z;
   return result;
}
```

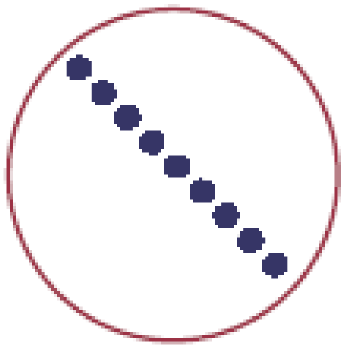
```
struct Ray
{
   Point3 start;
   Vector3 dir;
}
```

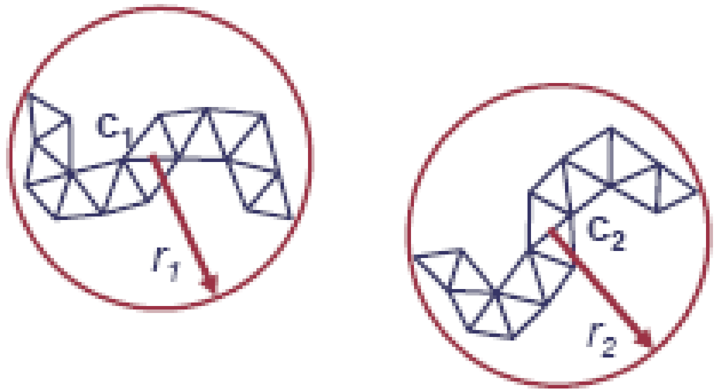
```
RayOBBIntersection(Ray r, OBB v)
{
   Point3 plane;
   plane.x = v.origin.x + v.fMin1*v.v1.x;
   plane.y = v.origin.y + v.fMin1*v.v1.y;
   plane.z = v.origin.z + v.fMin1*v.v1.z;
   float d = -v.v1.x*plane.x-v.v1.y*plane.y-v.v1.z*plane.z;
   Point3 I = RayPlaneIntersection(r, v.v1, d);
   if (PointInsideOBB(I, v))
      return true;

   plane.x = v.origin.x + v.fMax1*v.v1.x;
   plane.y = v.origin.y + v.fMax1*v.v1.y;
   plane.z = v.origin.z + v.fMax1*v.v1.z;
   float d = -v.v1.x*plane.x-v.v1.y*plane.y-v.v1.z*plane.z;
   I = RayPlaneIntersection(r, v.v1, d);
   if (PointInsideOBB(I, v))
      return true;
   // same for remaining two directions v2, v3
   ........
   return false;
}
```

Sphere, ellipse

- Good for transformations
- Simple computations, intersections
- Construction: smallest bounding sphere, sphere bounding AABB, OBB


good choice


bad choice



$$(\mathbf{c}_1-\mathbf{c}_2)(\mathbf{c}_1-\mathbf{c}_2) > (r_1+r_2)^2$$

Sphere

```
struct BoundingSphere
{
    Point3 center;
    float radius;
}
```

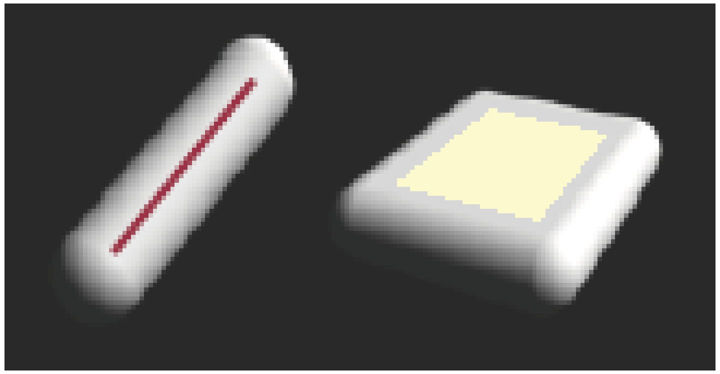
```
CreateBoundingSphere(vector<Point3> points)
{
    BoundingSphere result;
    result.center.x = result.center.y = result.center.z = 0;
    for (int i = 0;i < points.size();i++)
    {
        result.center.x += points[i].x / points.size();
        result.center.y += points[i].y / points.size();
        result.center.z += points[i].z / points.size();
    }
    result.radius = 0;
    for (int i = 0;i < points.size();i++)
    {
        float t = (points[i].x - result.center.x)^2 +
                  (points[i].y - result.center.y)^2 +
                  (points[i].z - result.center.z)^2;
      if (t > result.radius) result.radius = t;
    }
    result.radius = sqrt(result.radius);
    return result;
}
```

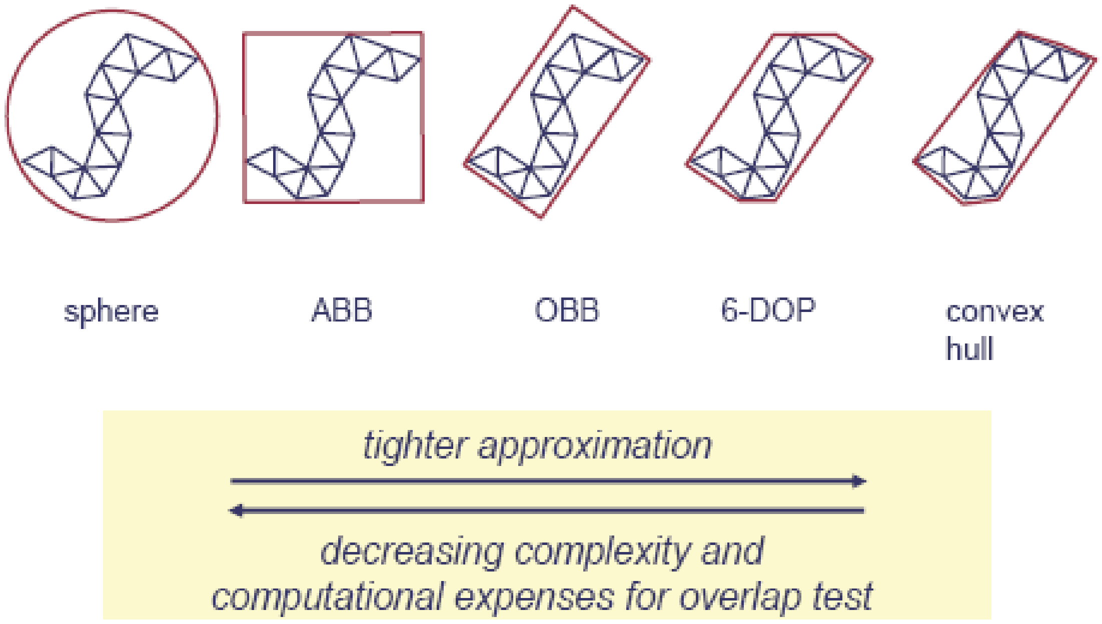
```
BoundingSphereIntersection(BoundingSphere v1, BoundingSphere v2)
{
    float d = (v1.center.x – v2.center.x)^2 +
              (v1.center.y – v2.center.y)^2 +
              (v1.center.z – v2.center.z)^2;
    if ((v1.radius + v2.radius) > sqrt(d))
        return true;
    else
        return false;
}
```

```
RayBoundingSphereIntersection(BoundingSphere s, Ray r)
{
    Point3 S = s.center;
    Vector3 v = r.dir;
    Point3 O = r.start;
    float dis = v.x*(O.x-S.x)+v.y*(O.y-S.y)+v.z*(O.z-S.z);
    dis = dis*dis;
    dis = dis – (v.x*v.x+v.y*v.y+v.z*v.z)*
               ((O.x-S.x)^2+(O.y-S.y)^2+(O.z-S.z)^2-s.radius^2);
    if (dis < 0)
        return false;
    else
        return true;
}
```

Other bounding volumes

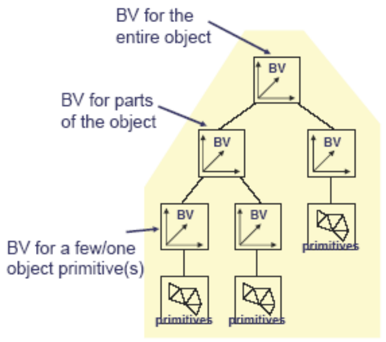
- Ellipsoids
- Cylinders, boxes
- Convex hulls
- Extended spherical volumes
- Combinations



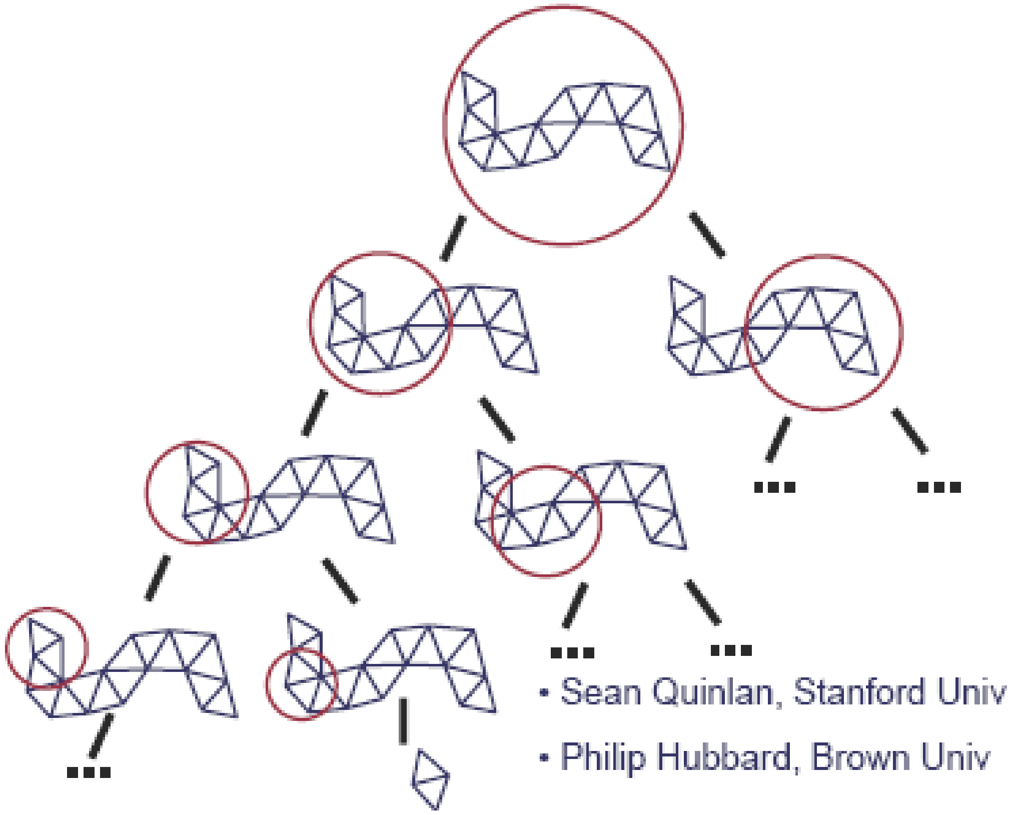

sphere
ABB
OBB
6-DOP
convex hull
tighter approximation
decreasing complexity and computational expenses for overlap test

Bounding Volumes Hierarchy

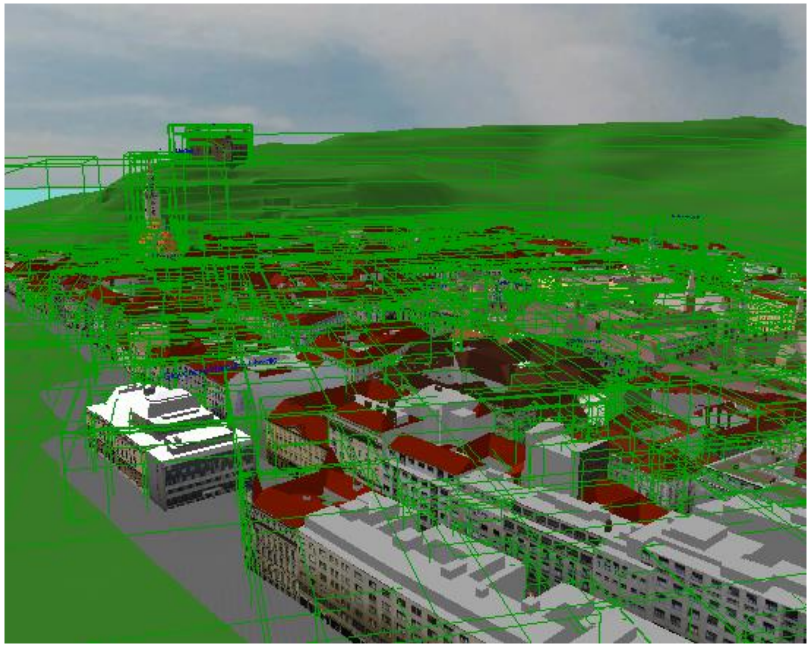
- Tree of bounding volumes
- Let $O = \{o_1, \ldots, o_n\}$ is the set of objects
- Then BVH for set O is defined:
  - if $|O| \leq e$, then BVH(O) is leaf that stores O and also stores bounding volume of O
  - if $|O| > e$, then BVH(O) is node with m siblings v_i, $i=1,\ldots,m$, where v_i are BVH(O_i) created using sets of objects O_i, $O = \cup\ O_i$, *BVH(O)* also stores bounding volume of O

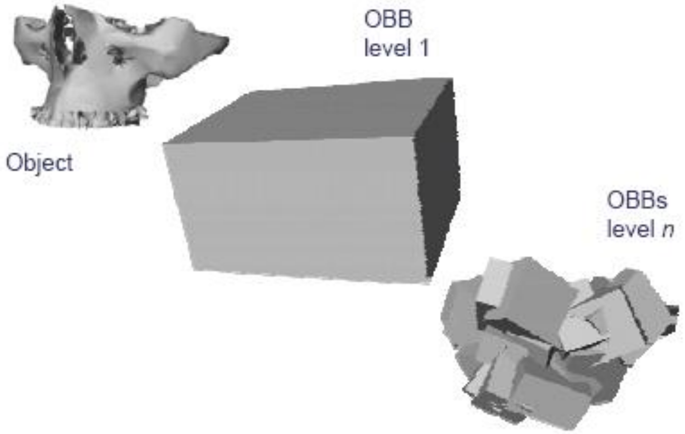


BVH examples



- Sean Quinlan, Stanford Univ
- Philip Hubbard, Brown Univ



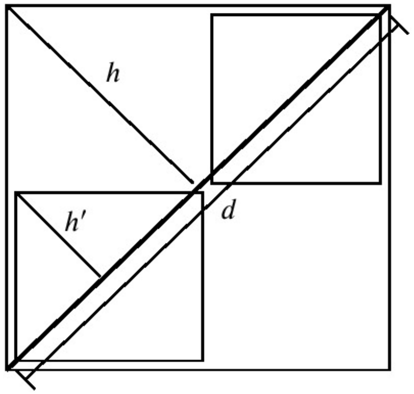


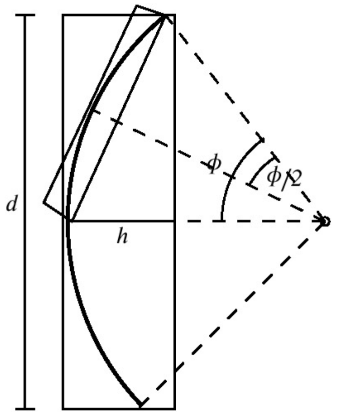
BVH approximation precision

- Volumetric tightness
- *C(v)* are siblings of *v*, *L(v)* are leafs of *v*

$$\tau := \frac{\mathrm{Vol}(v)}{\sum_{v' \in C(v)} \mathrm{Vol}(v')}. \qquad \tau := \frac{\mathrm{Vol}(v)}{\sum_{v' \in L(v)} \mathrm{Vol}(v')},$$

- For AABB, tightness does not change very much, based on geometry orientation
- For OBB, tightness is growing, based on curvature





BVH construction criteria

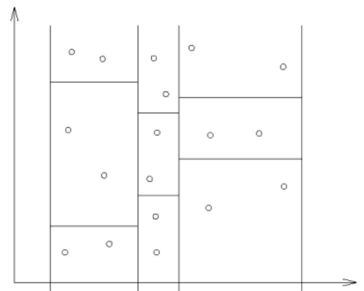
- Balanced tree
- Dividing volumes or objects
- How to divide volumes or objects
- Redundancy minimization (if the object is stored in multiple nodes or not)
- Number of object primitives in leafs
- Type of geometry search
- Cost functions for nodes of tree

BVH construction

- Bottom-up:
  - Starting with separate objects
  - Create bounding volume for each object
  - Recursively cluster bounding volumes and objects into bigger parts
  - First creating leafs of tree, then, traversing up
  - Last node created is root storing bounding volume for all objects

Clustering

- Greedy:
  - Sort volumes based on distance from each other
  - In order choose first *k* volumes and create new parent in hierarchy for those k nodes
- Tiling:
  - For each volume compute center
  - Split the space on $\sqrt[d]{n/k}$ tiles such that in each tile there is same count of primitives *k* (first in x direction, then y)
  - Created tiling has *k* nodes (volumes) in each tile, for each tile cerate new parent node with siblings given by nodes in tile
  - Recursively traverse till root node is processed



BVH construction

- Insertion

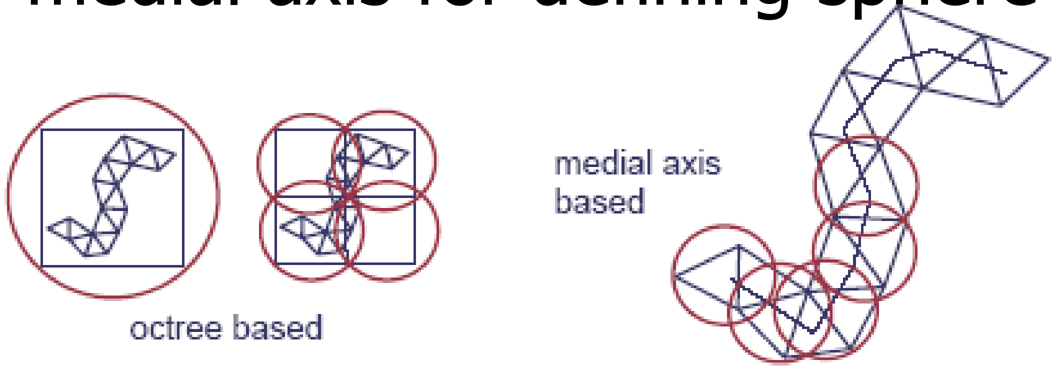
```
void BVH_Insert (B)
{
        while (|B| > 0)
        {
                b = B.pop;  v = root;   // vybratie dobreho *b* je dolezite
                while (!v.IsLeaf())
                {
                        choose child v', so that insertion of b into v' causes minimal increase in the
                        costs of the total tree;
                        v = v';
                }
        }
}
```

BVH construction

- Top-down
  - Starting with whole scene (or object)
  - Creating bounding volume for whole scene
  - Volume (objects) split in *k* parts
  - Recursive subdivision
  - Finishing when current volume (node) has primitives count less than given threshold

Spherical BVH construction

- Creation of top-down BVH using octree structure
- Hubbard, O'Sullivan:
  - Approximating each object or primitive using sphere and create BVH tree bottom-up
  - Create homogenous structure of nodes for redundancy removal
  - Compute medial axis for defining sphere centers



Cost functions

- Raytracing: cost function based on bounding volume area, ɑ = x, y, z, ...

$$k^{\alpha} = \min_{j=0\ldots n} \left\{ \frac{\mathrm{Area}(b_1, \ldots, b_j)}{\mathrm{Area}(B)} j + \frac{\mathrm{Area}(b_{j+1}, \ldots, b_n)}{\mathrm{Area}(B)} (n-j) \right\}$$

- Frustum culling: using volume instead of area
- Collision detection: comparing multiple trees and its parts, minimalizing C(A, B)

$$C(A, B) = 4 + \sum_{i,j=1,2} P(A_i, B_j) \cdot C(A_i, B_j),$$

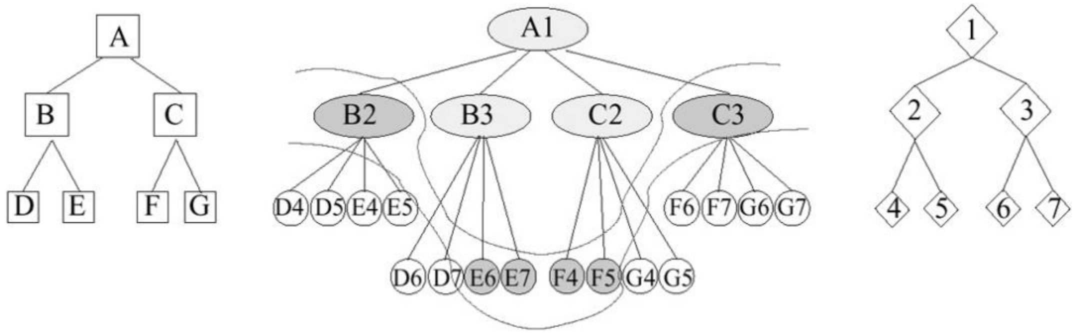
$$C(A, B) \approx 4\big(1 + P(A_1, B_1) + \ldots + P(A_2, B_2)\big).$$

$$P(A_1, B_1) \approx \frac{\mathrm{Vol}(A_1) + \mathrm{Vol}(B_1)}{\mathrm{Vol}(A) + \mathrm{Vol}(B)}.$$

Collision detection

- Checking if some objects intersect→ checking intersection of its simple primitives
- Traversing two trees
- Creating *bounding volume test tree* (BVTT)
- Usage of coherence

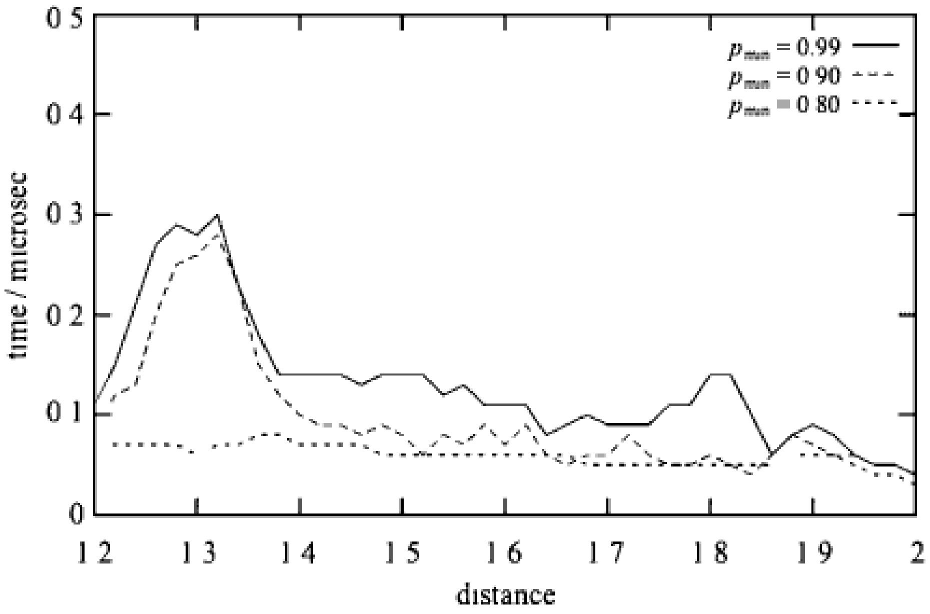
```
traverse(A,B)
{
        if (A ∩ B == 0)
                return NULL;
        if (A.IsLeaf() && B.IsLeaf())
                return A.primitives ∩ B.primitives;
        for (all children A[i] and B[j])
                traverse(A[i],B[j]);
}
```



Stochastic collision detection

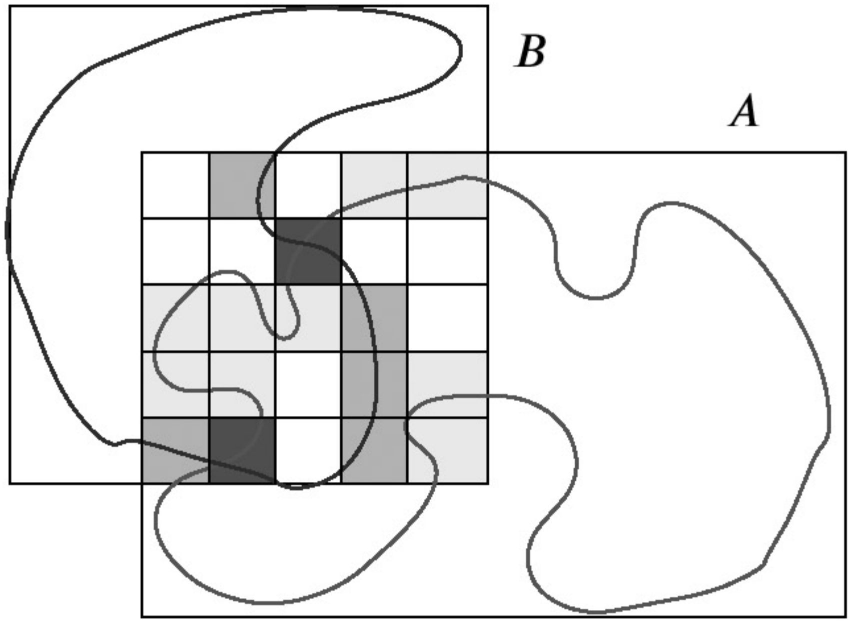
- Finding parts of tree that will be handled with higher priority
- Intersection probability→ external description of primitives in volume

```
traverse(A, B)
{
        q.insert(A, B, 1);
        while (!q.IsEmpty())
       {
                    (A, B) = q.pop();
                    for (all children $A_i$ and $B_j$)
                    {
                              p = P [$A_i$, $B_j$];
                              if (p > thresh)
                                        return “collision”;
                              else if (p > 0)
                                        q.insert($A_i$, $B_j$, p);
                    }
        }
}
```



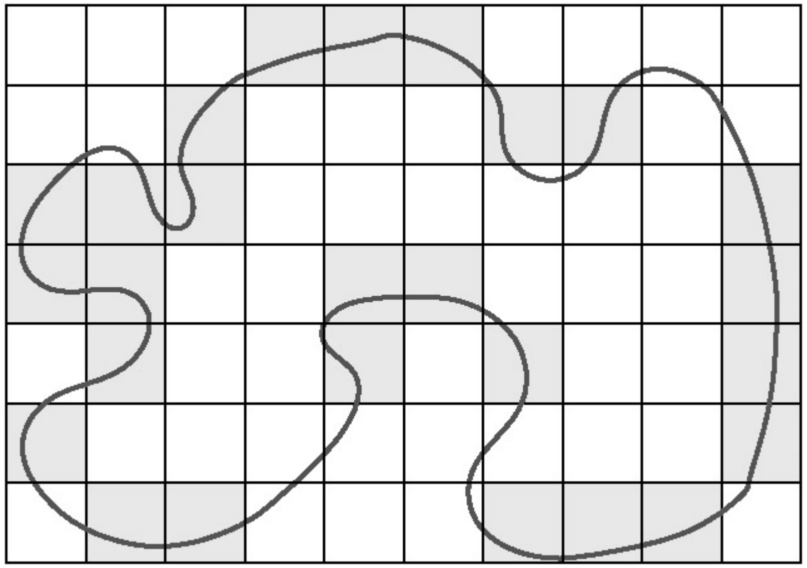
Intersection probability

- Use regular grid in A ∩ B
- In each cell of grid, compute density of primitives of A and B
- Compute count of probable collision cells



Stochastic solution

- When creating volume for one object
- Represent volume using regular grid
- Compute number of cells with higher density and store this value with volume - s_A



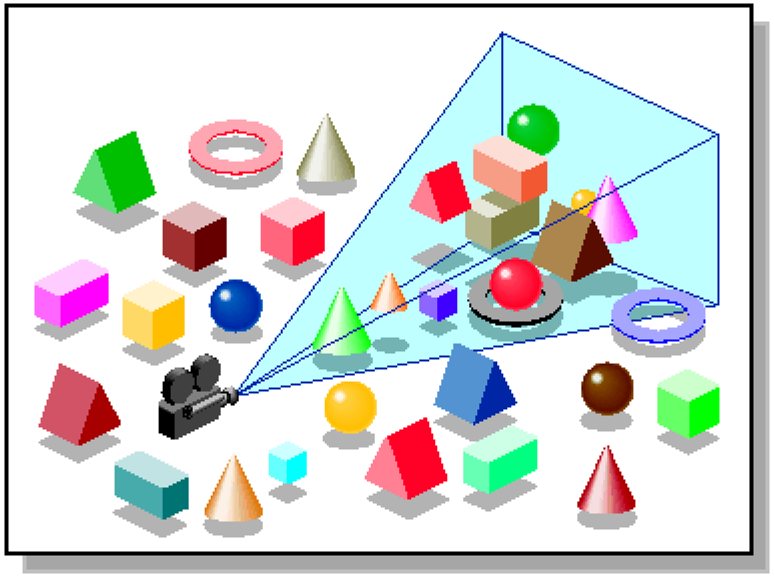
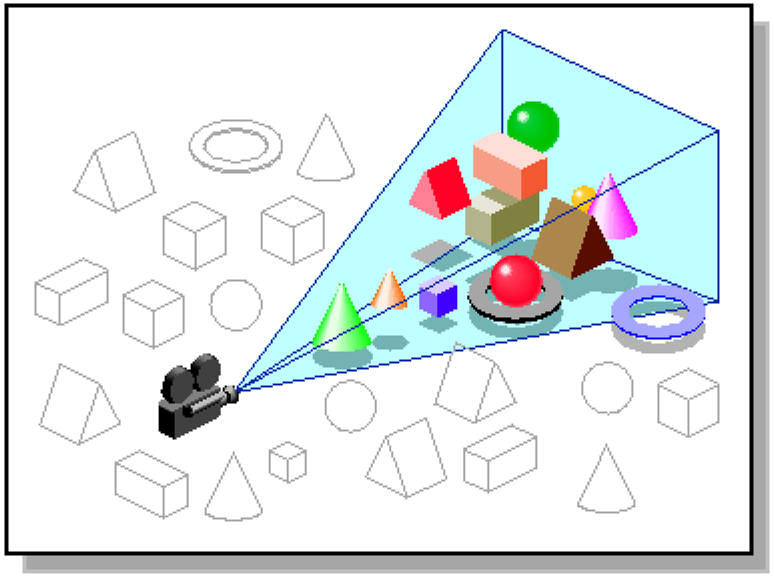
$$s'_A = s_A \frac{\mathrm{Vol}(A)}{\mathrm{Vol}(A \cap B)},$$

$$P[c(A \cap B) \geq x] = 1 - \sum_{t=0}^{x-1} \frac{\binom{s_A}{t}\binom{s - s_A}{s_B - t}}{\binom{s}{s_B}}.$$

Probability, that in intersection of A and B is at least x cells with higher density

Frustum culling

- Finding and rendering only objects lying inside viewing frustum volume
- Recursive testing for intersection of bounding volume and viewing frustum

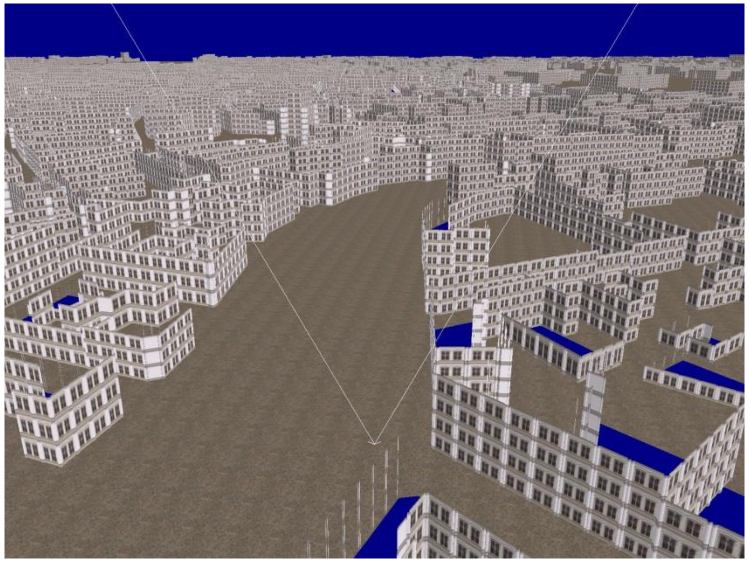



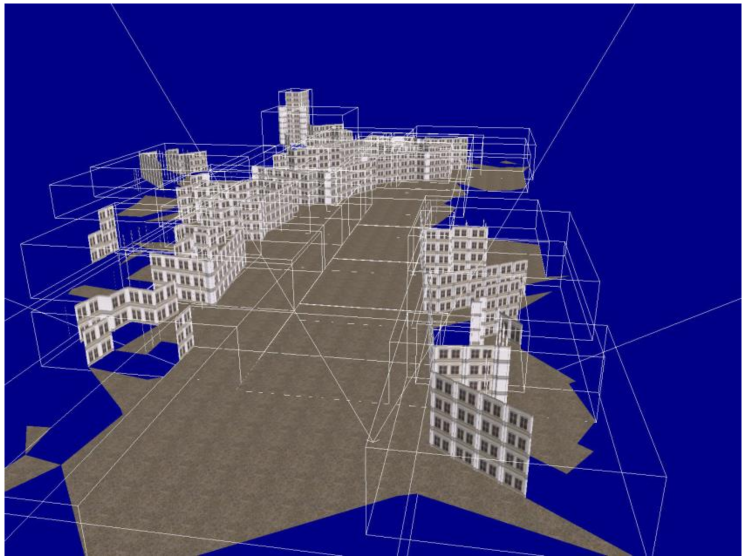
Zdroj: www.sgi.com

Occlusion culling

- Rendering only really visible objects, dropping objects that are fully occluded by other objects
- Needed to compute number of visible pixels on screen for some rendered object
- Algorithm:
  - Render large objects near camera
  - Render only bounding volume of other objects, if volume has some visible pixels on screen, then render also whole object
  - Usually also using front-to-back rendering (BSP)

Occlusion culling



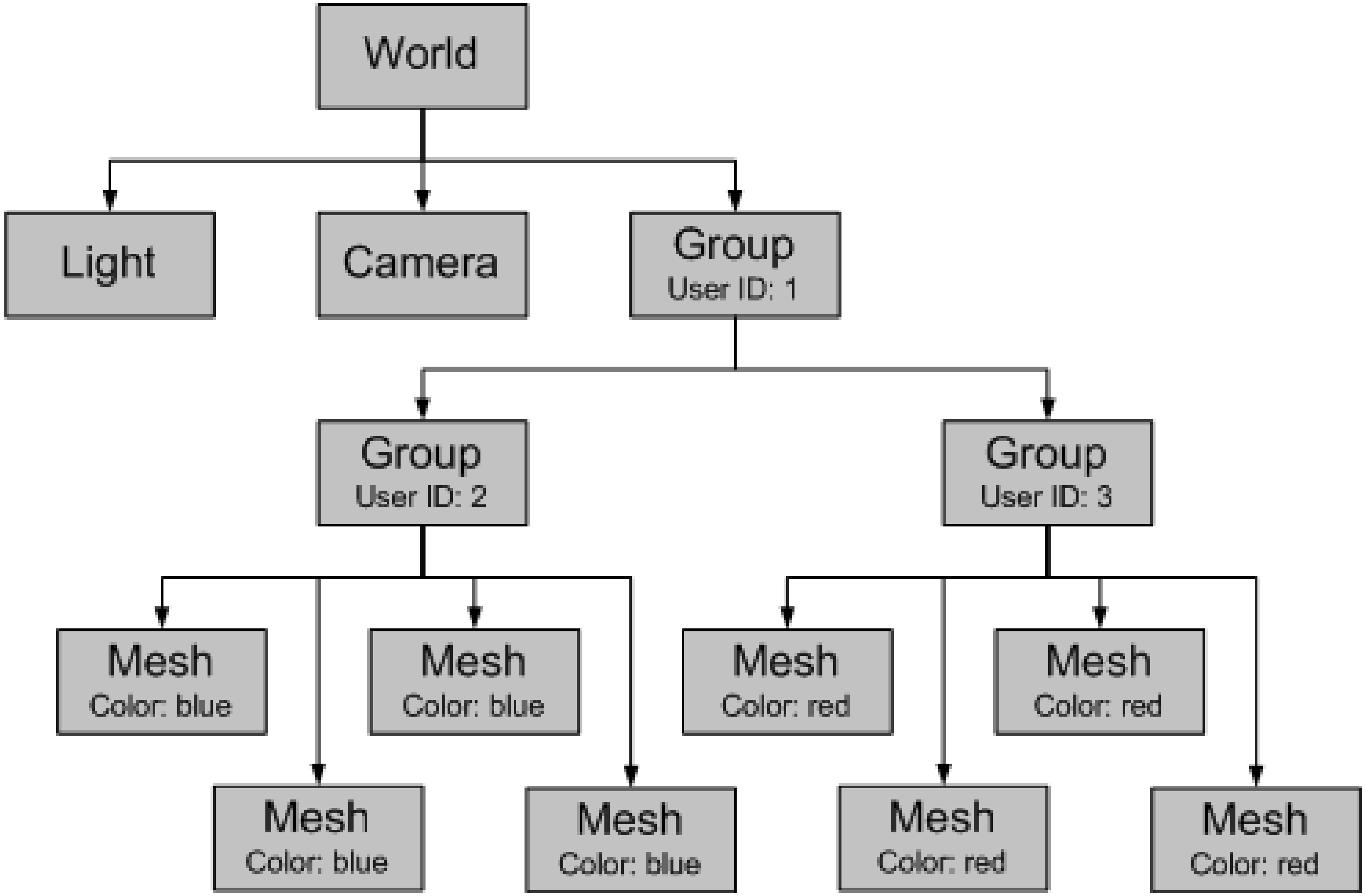


Source: http://www.cg.tuwien.ac.at

Scene graph

- Simulating hierarchies of obejcts in scene
- Better and faster access of data
- Can be combined with bounding volumes
- Dynamic scenes: bottom-up update
- Each scene graph node has several parameters:
  - local, global transformation
  - rendering attributes
  - bounding volume

Graf scény


World
Light
Camera
Group
User ID: 1
Group
User ID: 2
Group
User ID: 3
Mesh
Color: blue
Mesh
Color: blue
Mesh
Color: red
Mesh
Color: red
Mesh
Color: blue
Mesh
Color: blue
Mesh
Color: red
Mesh
Color: red

Questions?