Real-time graphics II

Contents

- Visibility processing of complex scenes
  - OCTREE
  - BSP-trees
  - OBB, AAOB
- Shadows
- Ray intersect collision

Visibility processing

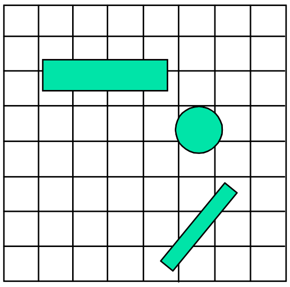
- Scene partitioning important for view frustum intersection
- We don't want to compare every little detail/polygon with the VF.
- During visibility processing an object may intersect the VF
  - Parts of it is inside and part outside
  - We might decide we want to subdivide the object.
- Compare to WTK:s scene graph, either an object is inside (partially or full) and is rendered or it is outside and is ignored.
- Step one is always to group the objects in an spatially optimized way (see RG I lecture)

Why trees?

- In all cases we build a tree as a pre-process.
- Then descend this tree in real time to find out:
  - Of two objects occupy the same spatial partitions, collision
  - To find out if the space of an object and the VF are disjoint, which eliminates the entire object from further processing.
- Most of these operations reduce to tree descent or traversal – a fast linear time operation

Space subdivision hierarchies

- Brute force
  - Divide all world space into regular or cubic voxel and label it.
  - Each voxel that contain an object is given the identity of the object that occupies it.



  - Very memory consuming
  - Classic for ray tracing.
    - Instead of asking the expensive question, does this ray intersect with any object in the scene?
    - We pose the question which objects are intersected as we track the ray trough the voxel space.

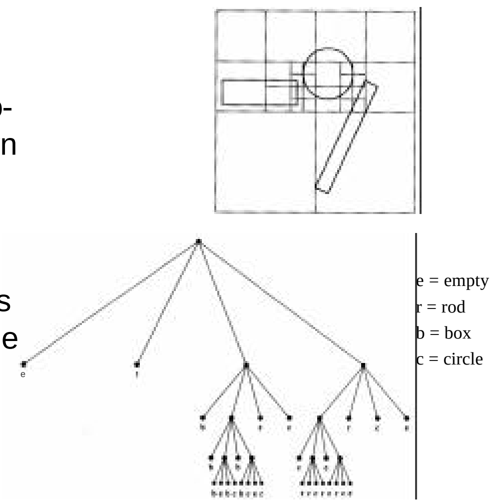
OCTREE

- Describes how the objects in the scene are distributed throughout the tree-dimensional space occupied by the scene.
- Organizes the voxels into a hierarchy.
- The process of creating a OCTREE:
  1. Root of tree is a cube region covering the whole scene (bounding box).
  2. Because the box is occupied by objects, it is subdivided into 8 sub-regions.
  3. Any region that is occupied by an object is further subdivided.
  4. Termination:
     1. No objects within a region.
     2. Cells of minimum size that are occupied by part of an object.

OCTREE

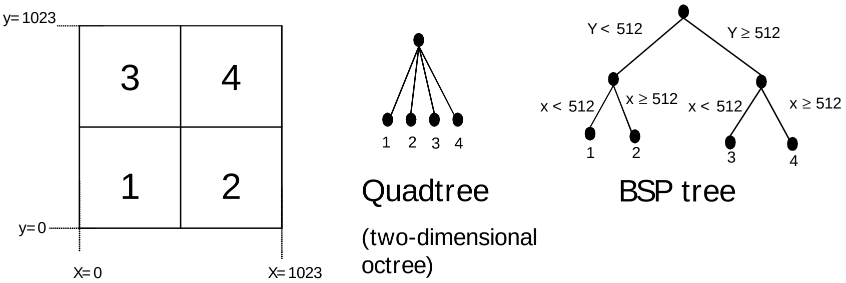
- To divide the whole scene into the smallest component (polygons, or even vertices) would take too much memory
- Instead, the objects are stored in a normal way (vertex, face list) and the OCTREE is used for the distribution of the objects in the scene.

- Quadtree (2-dimensional OCTREE) representation of a two-dimensional scene down to the level of cells containing at most a single node.
- Terminal nodes for cells contains a pointer to the data structure representing the object



BSP trees

- Binary space partitioning tree
- Partitions by dividing two parts at each level by using a splitting plane.



BSP trees

- Using cubic cells makes BSP perform about the same as OCTREE
- Selecting another splitting scheme makes it potentially perform much better.
- The process of building a BSP tree takes time, usually pre computed before rendering.
- Due to this, it suits static objects, buildings, walls, etc.

BSP trees

- The process of building a BSP tree
  1. The first polygon in the scene is read, the plane of the polygon is used as a partitioning plane
  2. The process recurses with each of the two sets of polygons
  3. Termination
     1. a maximum depth is reached
     2. Number of polygons in a leaf node is below a threshold
     3. Only one polygon at a leaf node
  4. Polygons that are at both sides of the partitioning plane is either
     1. Divided into two polygons (generates more polygons)
     2. Insert the polygon in both sub-trees, and flag it during traversal (drawing, collision detection etc.) Saves memory.

Partitioning plane

- To determine a plane from a polygon we evaluate its coefficients as follows. A plane has the equation:

$$Ax + By + Cz + D = 0 \qquad [1]$$

where A, B, C are the coordinate values of its normal vector, calculated from any three (non-collinear vertices).

Cross product of two vectors ***V*** and ***W*** is defined as:

$$\boldsymbol{X}=\boldsymbol{V}´\boldsymbol{X}=(v_2w_3-v_3w_2)\boldsymbol{i} + (v_3w_1-v_1w_3)\boldsymbol{j}-(v_1w_2-v_2w_1)\boldsymbol{k}$$

Where ***i,j*** and ***k*** are the standard unit vectors. *A, B* and *C* are thus:

$A=v_2w_3-v_3w_2$

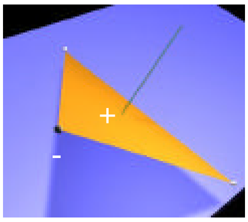
$B=v_3w_1-v_1w_3$

$C=v_1w_2-v_2w_1$

D is obtained by substituting a point known to lie on the plane (a vertex) into eq. 1.

Partitioning plane

- Classifying a point with respect to a plane is the foundation of all simple operations that use BSP trees.
- It is done by substituting the point into eq. 1. (x, y, z)
- If the result is positive
  - On the positive side (with respect to the splitting polygons normal)
- If the result is negative
  - On the negative side (normal)
- If the result is = 0
  - The tested point lies on the plane
- To test if a polygon lies on one side or another, we have to test all vertices (points) for the polygon
  - Either all is on the positive side, negative or the values goes from negative to positive (the polygon is intersecting the split plane).



Partitioning plane

- The pair of vertices of edges that cross the splitting plane (from positive to negative or vice versa) are detected and the intersection points are computed by solving the line/plane equation as follows:

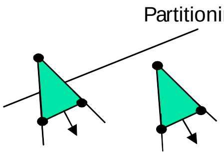
$$t = \frac{Ax_1 + By_1 + Cz_1 + D}{A(x_2 - x_1) + B(y_2 - y_1) + C(z_2 - z_1)}$$

$$x = x_1 + (x_2 - x_1)t$$
$$y = y_1 + (y_2 - y_1)t$$
$$z = z_1 + (z_2 - z_1)t$$

Where (x_1, y_1, z_1) and (x_2, y_2, z_2) are the two vertices of the edge that crosses the plane at intersection point (x, y, z)

Culling against the view frustum

- BSP tree can be used to cull polygons against a view frustum.
- At a node the View Frustum is compared with the partitioning plane. If the appropriate culling condition is fulfilled, the entire sub-tree associated with that node can be eliminated.
- Each vertex of the view frustum (5) is injected into equation 1 and should return the same sign.

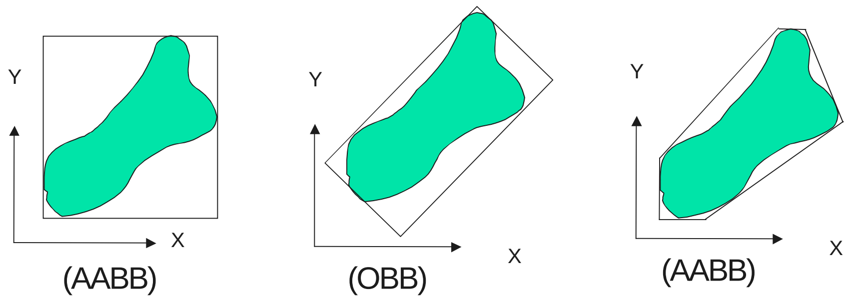


BSP in practice

- Two important choices
  - How do we choose the partition plane?
  - When is the partition terminated?
- A common application is the cell-like scene found in building interiors and games (Quake, etc.)
- Here it makes sense to use wall-aligned partitioning planes.
- Subdividing can continue the space within a room.
- Partitioning can terminate at room level, object level or objects themselves can be subdivided.
- If a leaf contains a cluster of polygons, the BSP tree is used for fast culling of objects outside the VF and exact visibility is performed using standard Z buffer.

Interaction methods

- Definitions
  - Axis Aligned Bounding Box (AABB)
    - A lot of empty space enclosed, but easy to calculate
  - Oriented Bounding Box (OBB)
    - Less empty space, more complicated to calculate
  - k-Dop (discrete oriented polytope)
    - Least empty space, and most complicated to calculate



Rules of Thumb

- Perform computations and comparison that might trivially reject or accept various types of intersections to obtain an early escape from further computations
- If possible, exploit the results(s) previous test(s)
- If more than one rejection or acceptance test is used, try changing their internal order, since speed-up may result.
- Postpone expensive calculations (square roots, trigonometric functions and divisions) until truly needed.
- Try to reduce the dimension of the problem from three to two or even one dimension.
- If a single ray or object is being compared to many other objects, look for pre-calculations done only once before the testing begins.
- Make your code robust, (work for all special cases, and insensitive for floating point precision errors).

Ray Sphere Intersection

- A sphere is defined by center point $\boldsymbol{c}$ and a radius r.
- The implicit formula for a sphere is then

  $f(\mathbf{p}) = ||\mathbf{p}-\mathbf{c}|| - r = 0$ [2]

  where $\boldsymbol{p}$ is any point on the sphere's surface.
- To solve a ray/sphere intersection, the $\boldsymbol{p}$ is replaced by the formula for a ray

  $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$
- After simplification we get:

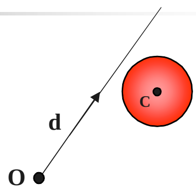
  $t^2+2tb+c=0$ [3]

  where $b=\mathbf{d}\bullet(\mathbf{o}-\mathbf{c})$ and $c=(\mathbf{o}-\mathbf{c})\bullet(\mathbf{o}-\mathbf{c})-r^2$
- The solutions for the second order eq. 3 is:
- If $b^2-c<0$ the ray misses (no real roots).

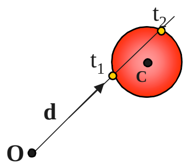
  $t=-b\pm\sqrt{b^2-c}$

Ray Sphere Intersection

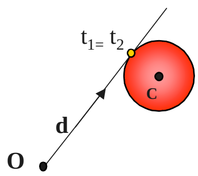
- Ray misses and consequently $(b^2-c<0)$



- Ray intersects sphere at two points $(b^2-c>0)$

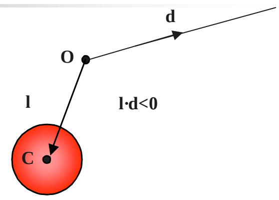


- Ray intersects sphere at one point $(b^2-c=0)$

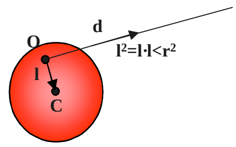


Optimized Ray/Sphere

- Trivially reject
  - Center of sphere is behind origon of ray a hit is impossible



- Trivially accept
  - Origin of ray is within sphere, always a hit



Algorithm for Optimized Ray/Sphere intersection

RaySphereIntersect(o, **d**, **c**, r)

Returns ({REJECT, INTERSECT}, t, **p**)

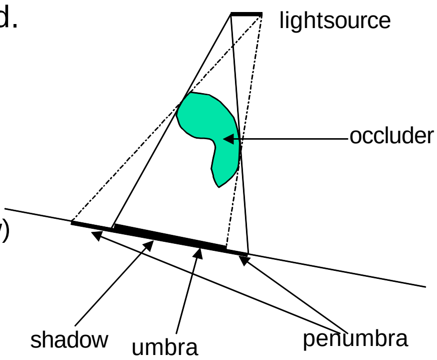
1. **l**=**c**-**o**
2. d=**l** ·**d**
3. l^2=**l** ·**l**
4. if (d<0 and $l^2>r^2$) return (REJECT, 0, **O**)
5. $m^2=l^2-d^2$
6. if($m^2>r^2$) return (REJECT, 0, O)
7. q=sqrt(r^2-m^2)
8. if (l2>r2) t=d-q
9. else $t=d+q$
10. return (INTERSECT, t, **o**+t**d**)

Different kind of intersections

- Ray/Box
- Ray/Triangle
- Ray/Polygon
- Plane/Box
- Triangle/Triangle
- Cube/Polygon
- BV/BV
  - Sphere/Box
  - AABB/AABB
  - K-DOP/k-DOP
  - OBB/OBB
- Line/Line
- 3-Planes intersection

Shadows

- It is better to have an inaccurate shadow, than none at all (Wanger).
- They eye is fairly forgiving about the shape of the shadow.
- A blurred black circle applied as a texture on the floor can anchor a person to the ground.



- Umbra
  - Totally shadowed
- Penumbra
  - Partially shadowed (soft shadow)

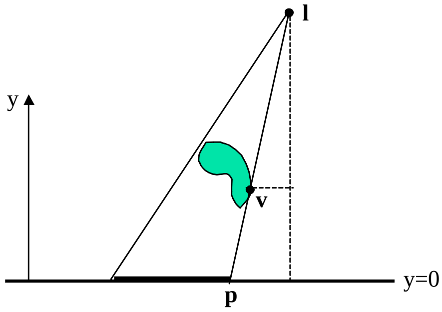
Planar shadows

- A simple method is to project all the occluders vertices onto a flat surface according to the light position and render the resulting geometry black.
- Works for non-intrusive geometries
  - There will be no shadows on geometries other than the shadow plane.
- One light source (or two as long as the shadows dont interfere with each other).

Planar shadows

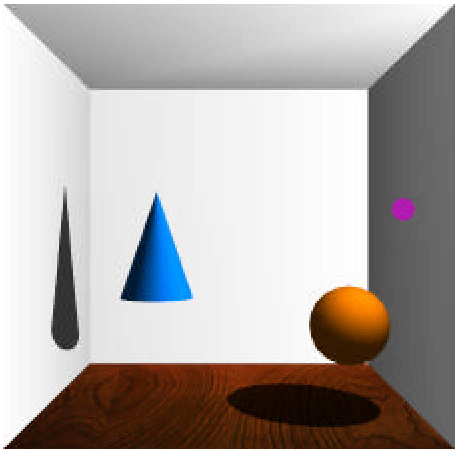
- Light source **l**, casts a shadow onto the plane *y=0*. The vertex **v** is projected onto the plane. The projected point is called **p**.
- Each vertice **v** in geometry **G** is then transformed by the following matrix **M** to render the geometry at the plane *y=0*

$$M = \begin{pmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{pmatrix}$$



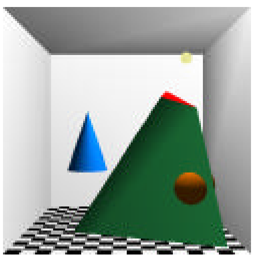
Planar shadows

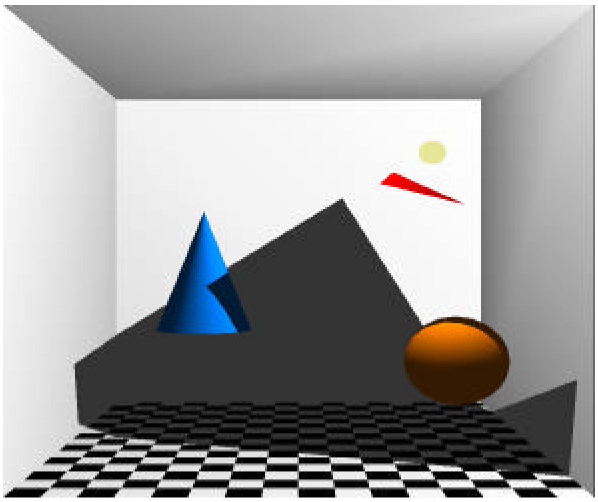
- Two shadow planes.
- The shadow is drawn using only ambient lightning, color {0, 0, 0, 0}



Shadow volumes

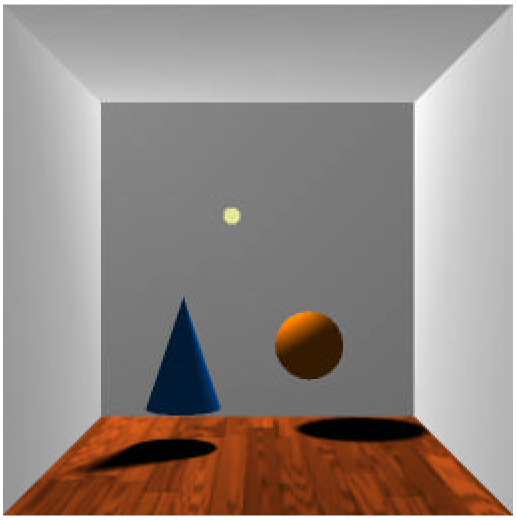
- Hedimann 1991
- Can cast shadows onto arbitrary objects by use of the stencil buffers.





Textured Soft shadows

- Heckbert and Herf
- Gooch et al.



References

- *Real-time rendering,* Möller, Haines, A K Peters, 1999
- *3D Games, Real-time rendering and Software Technology*, Watt, Policarpo, Addison-Wesley, 2000

That's it folks!

- No more THEORY
- Keep on labbing!