《Fundamentals of Computer Graphics》

Lecture 8、Ray tracing

Part 4: Computational efficiency

Yong-Jin Liu

liuyongjin@tsinghua.edu.cn

Material by S.M.Lea (UNC)

Using Extents

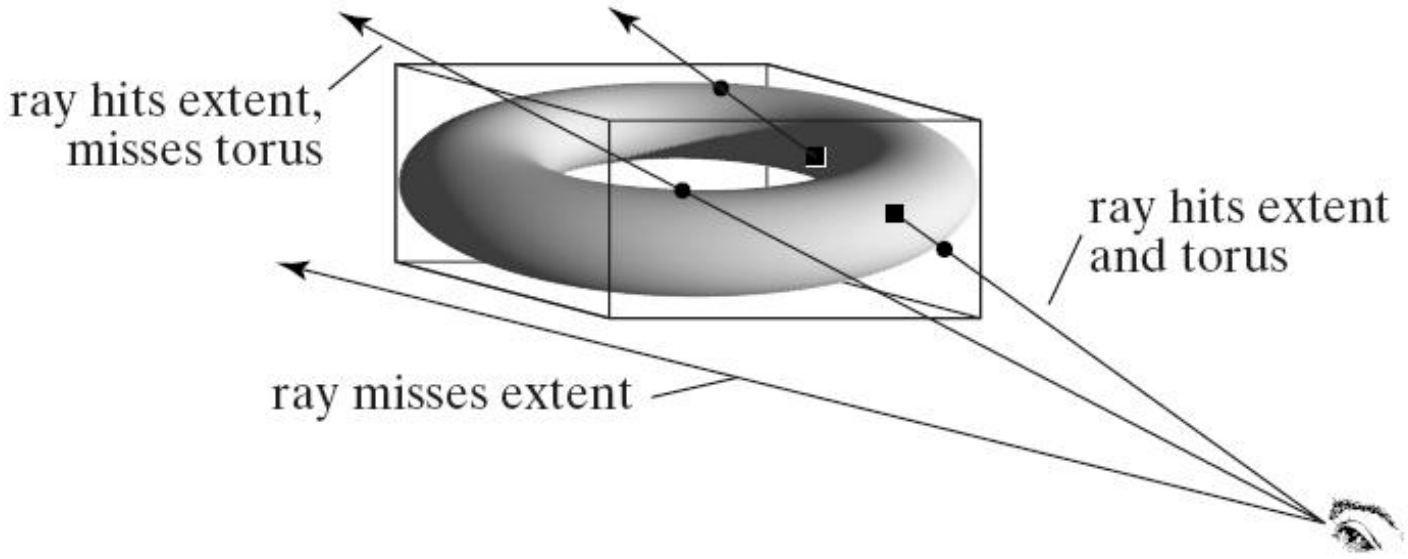
- Ray tracing is very repetitive, performing the same set of operations again and again for a very large number of rays.
- Each ray must be intersected with every object, amounting to an enormous number of intersection calculations.
- Matters will get much worse when we incorporate shadows, reflections, and refractions.
- The use of extents can speed up the ray tracing process significantly.

Using Extents (2)

- An **extent** of an object is a shape that encloses that object.
- It accelerates the ray tracing process by quickly revealing when the current ray *could not possibly hit* a particular object.
  - The notion is that if a ray misses the extent, it must perforce miss the object.
- If the extent has a simple shape, it may be inexpensive to intersect a ray with it, whereas it may be very expensive to intersect a ray with the enclosed object.

Using Extents (3)

- This figure shows an example where a torus (expensive to intersect) is enclosed in a box-like extent (inexpensive to intersect).



Using Extents (4)

- When the ray is tested against the extent, three things can happen:
  - The ray misses the box. Therefore the test against the torus is skipped;
  - The ray hits the box, so the full test against the torus is performed, revealing that the ray misses the torus.
  - The ray hits the box, so the full test against the torus is performed, revealing that the ray hits the torus.
- If the cost of the full hit test is much greater than that of the extent test, it is well worth making the extra test for those frequent cases where the ray does miss the extent.

The Cost Saving

- Suppose it costs *T* time units to test a ray against the extent and *mT* time units to test it against the torus.
- Further, suppose that *N* rays are cast to create the image and only fraction *f* of them hit the box.
- It follows that *N* tests are made against the extent, at a cost of *NT*, and *fN* tests are made against the torus, at a cost of *fNmT*. So the total cost is *NT(1 + fm)*.

The Cost Saving (2)

- On the other hand, if extents are not used, all *N* rays must be tested against the torus, at a total cost of *mNT*. This yields the ratio

$$speed\ ratio = \frac{\cos t\ without\ extents}{\cos t\ with\ extents} = \frac{m}{1 + f \cdot m}$$

- For example, if *m* = 20 and *f* = 1/40 (each object in a typical scene covers only a small fraction of the image area), this ratio is about 13, so it would take 13 times as long to trace this scene if extent testing is disabled.

The Cost Saving (3)

- Note from this analysis that we want *f* to be as small as possible.
- That is, we want each extent to be as small as possible so that the fewest rays will hit it.
- This agrees with our intuition that extents should be as tight fitting about their parent objects as possible.

Using Extents (5)

- Where does extent testing fit into the ray tracing process?
- We focus on improving the basic hit() method that every object in the scene possesses.
- Recall the basic routine getFirstHit() that must test the current ray against every object, using the specific hit() method for the object:

```
for (each object, obj)
{    if (!obj->hit(ray,inter)) continue;
     compare this hit time with the best so far,etc.
}
```

Using Extents (6)

- We shall use extent testing inside hit() to accelerate each individual test.
- If we can quickly determine that the ray does *not* hit the object before we perform the entire intersection calculation, the overall speed of the ray tracer will increase significantly.
- One advantage of putting extent testing inside each hit() method is that the testing can be fine-tuned to the specific geometric shape in question.

Box and Sphere Extents

- The two shapes most often used for extents are the sphere and an aligned box:
- **Sphere extent**: a sphere that completely encloses the given object; it is specified by (*C*, *r*), its center point *C,* and radius *r.*
- **Box extent**: a rectangular parallelepiped whose sides are aligned with the coordinate axes. It is specified by six numbers: (*left*, *top*, *right*, *bottom*, *front*, *back*).
- Intersecting each of these shapes with a ray is reasonably fast.

Box and Sphere Extents (2)

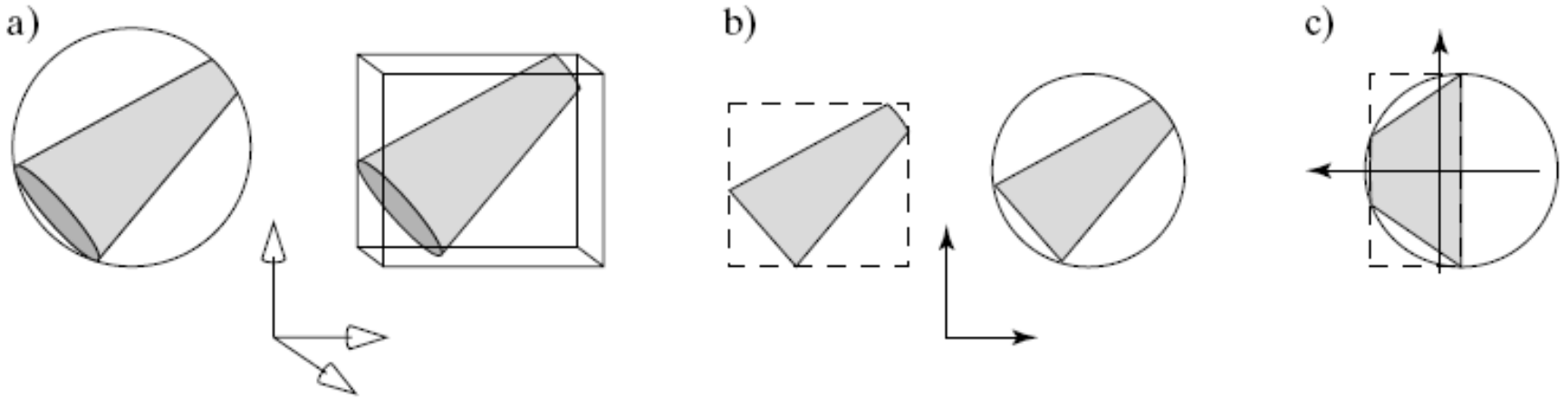
- For the sphere, we must form a quadratic equation and test whether the discriminant is positive.
- For an aligned box, we must intersect the ray with six planes and test whether the candidate interval (t_{in}, t_{out}) vanishes.
  - Each of the plane intersections is very fast since no dot products need to be formed.
  - In addition, an early out frequently occurs, where the candidate interval vanishes after only a few planes have been tested.

Box and Sphere Extents (3)

- We can do extent testing in world coordinates or generic coordinates.
- Generic: We can place an extent about the generic object and test the inverse transformed ray against it. If the ray misses this extent, a full intersection test is avoided.
- World: We place an extent about the object in the scene itself and test the current ray against it. If the ray misses this extent, the generic ray need not be computed, which saves a transformation.

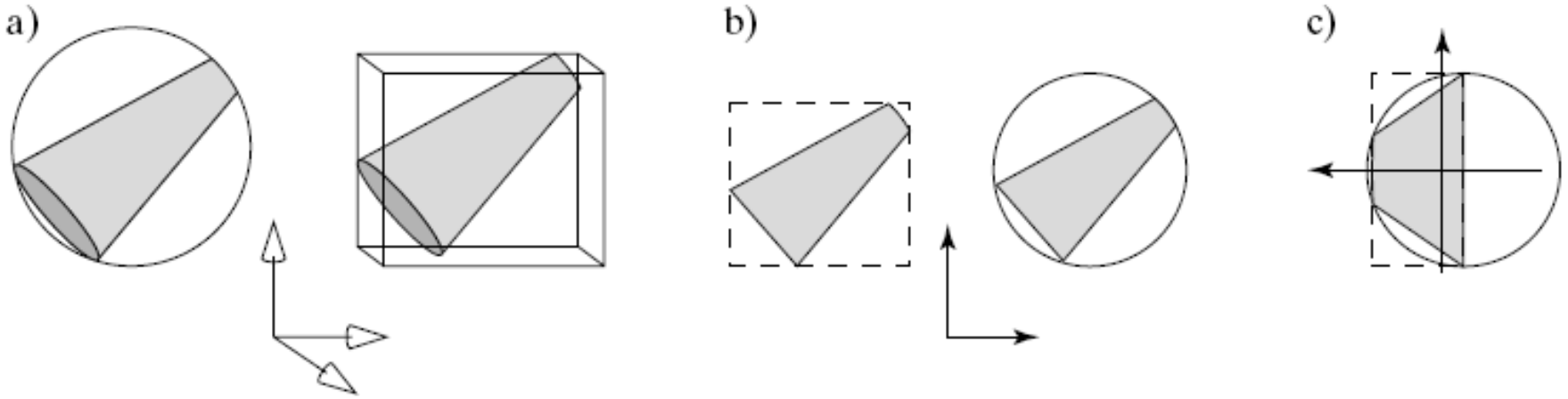
Box and Sphere Extents (4)

- Extents in world coordinates might not fit particularly well about their objects.
- The sphere extent doesn’t fit a long thin cylinder particularly closely, and using it would lead to many false alarms.
- The box extent fits closely about the cylinder even if it is long and thin, unless the cylinder is rotated away from alignment with the coordinate axes.



Box and Sphere Extents (5)

- Extents in generic coordinates usually fit more snugly.
- Part c shows the generic tapered cylinder, surrounded by a sphere extent and a box extent. The sphere extent fits reasonably snugly, and the box extent fits very tightly about the cylinder.



Box and Sphere Extents (6)

- Testing in world coordinates is fast because there is no need to compute the inverse transformed ray. But extents may not fit very tightly about their objects.
- Testing in generic coordinates requires finding the generic ray, but extents fit more tightly.
- It is difficult to say in general which kind of test is superior.
- It is also difficult to generalize on the advantage of sphere extents over box extents or vice versa.

Pseudocode for Adding Extents to hit()

```
bool TaperedCylinder::hit (Ray &r, Intersection &inter)
{
   if (!rayHitsSphereExtent(r,worldSphereExtent)) return
   false;

   Ray genRay; //make the generic ray
   xfrmRay (genRay,invTransf,r);  //expensive

   if (!rayHitsSphereExtent(genRay,genSphereExtent))
   return false;
   …Do expensive full testing with the generic cylinder…
}
```

Pseudocode (2)

- Sphere extent testing is shown, and the test is done in both world and generic coordinates to show where the testing is done. (Only one of the tests would be used in an actual ray tracer.)
- rayHitsSphereExtent() is called to see whether the ray in world coordinates hits the sphere extent in world coordinates; if not, there is an early out from hit().
- If the ray hits this extent it is inverse transformed. The routine is then called a second time to see if the generic ray hits the generic sphere extent.
- If it misses, there is an early out, and the elaborate intersection test of a ray with the generic cylinder is skipped.

Box and Sphere Extents (7)

- We must choose which tests to include in each hit() method for the different shape classes.
- For a Sphere object, there is no sense in doing a sphere extent test in generic coordinates, and similarly you would never do a box extent test in generic coordinates for the Cube object.
- There are no extents for a Plane, so no tests are included in Plane :: hit().
- For a Square, it is meaningful to build an extent, but the intersection test is so simple anyway there is no gain is using an extent test.
- A Mesh object, on the other hand, has a large number of bounding planes and is expensive to intersect, so extent testing can yield significant gains.

Implementing Extent Testing

- It is natural to store the extent information of an object inside the object itself, so we add some fields to the GeomObj class.
- We may want to do any one of the four types of extent testing, so we add fields for each:
- SphereInfo genSphereExtent, worldSphereExtent;
- Cuboid genBoxExtent, worldBoxExtent;

Implementing Extent Testing (2)

- The SphereInfo class holds a description of a sphere in two fields: center that holds the location of the center of the sphere extent and radSq that holds the square of its radius.
  - It could instead hold the radius, but only the square of the radius is required for the extent test.
- The Cuboid class has six fields: left, top, right, bottom, front, back.
  - The box extent is understood to extend from left to right along the *x*-axis, from bottom to top along the *y*-axis, and from back to front along the *z*-axis.

Implementing Extent Testing (3)

- After the object list has been built, but before ray tracing begins, data is placed in these fields for each object in the object list.
- Then, during ray tracing, some combination of the routines rayHitsSphereExtent() and rayHitsBoxExtent() is called inside each hit() method.

Implementing Sphere Extent Testing

```
bool rayHitsSphereExtent (Ray & ray,
   SphereInfo& sph)
{
   double A = dot3D(ray.dir, ray.dir);
   Vector3 diff = ray.start – sph.center;
   double B = dot3D(diff, ray.dir);
   double C = dot3D(diff,diff) - sph.radSq;
   return (B * B >= A * C);
}
```

Building Box and Sphere Extents

- We need to create the sphere and box extents for each given object.
- The generic extents need only be made once for each type of shape, but the world extents must be made for each object instance, taking into account the affine transformation associated with the instance.
- The involvement of a transformation makes the computation of the extent much more difficult.
  - For instance, how do we find a sphere extent for a cylinder that has been scaled and rotated in a complex way?

Building Box and Sphere Extents (2)

- For the cylinder, a particularly straightforward method associates a **point cluster** with each shape. This is a set of points whose convex hull encloses the object.
  - Recall an intuitive way to envision the convex hull of a set of points fixed in space: place a balloon about the points and let the balloon collapse onto the points. The resulting polyhedron is the convex hull.
- With a point cluster in hand, it is easy to construct a sphere that just encloses the cluster. And certainly such a sphere would enclose the original object.

Building Box and Sphere Extents (3)

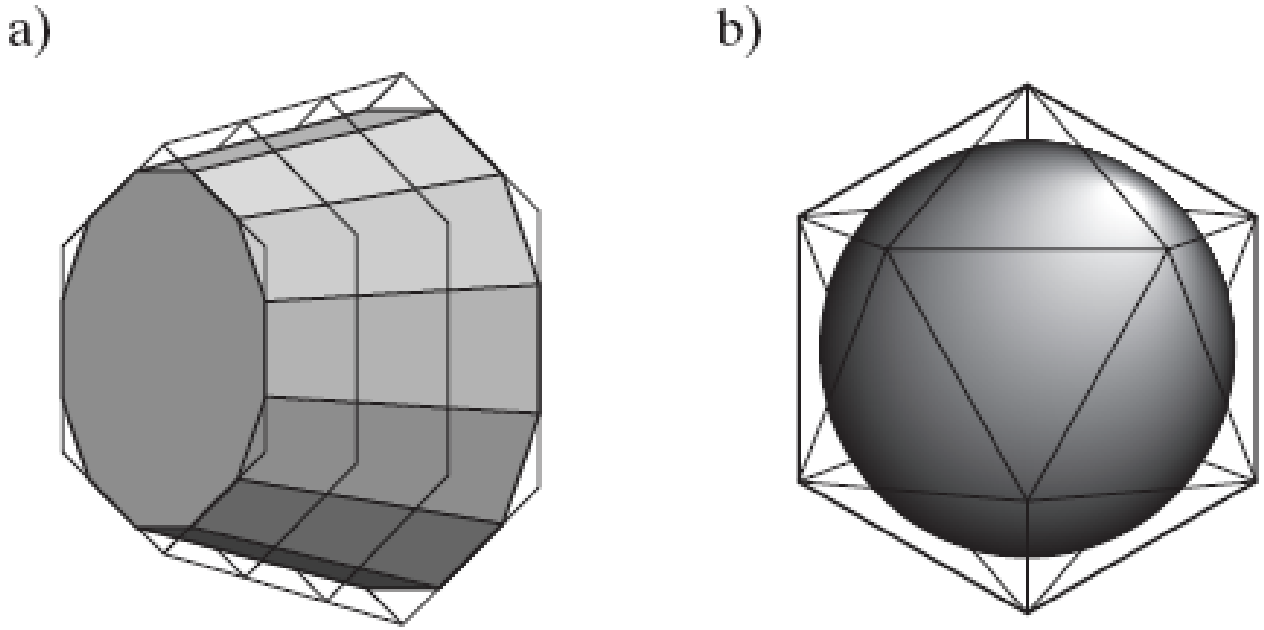
- We can also obtain a convex hull for the object after it is transformed: just transform each point in the point cluster and use the new points to define the transformed convex hull.
- Since affine transformations preserve inside-ness, if object *A* lies inside the convex hull *B*, then the transformed object *T*(*A*) must lie inside the convex hull *T*(*B*) built on the transformed points.

Building Box and Sphere Extents (4)

- We therefore need to build a point cluster for each shape type.
- This is easy for the Cube and the Square; their own vertices provide the points.
- It is also simple for a Mesh: just use the vertex list itself as the point cluster.
- A reasonable approach to the Sphere and the TaperedCylinder wraps each shape in a tightly enclosing polyhedron and uses the vertices of the polyhedron as the point cluster.

Building Box and Sphere Extents (5)

- This figure shows the generic tapered cylinder wrapped in a prism based on a hexagon at each end and the generic sphere wrapped in an icosahedron.



Building Box and Sphere Extents (6)

- The hexagon at the base of the tapered cylinder must have a radius of 4/3 in order to fit snugly about the circular base.
- The hexagon at the cap has a radius of 4/3 times the radius of the cap.
- Note that this point cluster is constructed individually for each instance of a tapered cylinder, and that it is designed specially for the specific shape of the cylinder.
- The vertices of the icosahedron around the sphere must be placed at a radius of about 1.26 to ensure the unit sphere is properly enclosed.

Building Box and Sphere Extents (7)

- The point clusters for each shape type can be stored in a simple PointCluster data structure containing a field num for the number of points in the cluster, and an array pt[ ].
- A method makeExtentPoints (PointCluster& clust) is developed for each shape class and called once during a preprocessing step for each object in the scene.

Building Box and Sphere Extents (8)

- Once the point cluster is available for a generic object, it is easy to find the box and sphere extents.
- Each field of the box extent holds the largest or smallest coordinate found among the cluster points, so all fields can be found within a single loop over the cluster points.
- This would be done in a routine void makeBoxExtent(PointCluster& clust,Cuboid& cub) that takes a point cluster and generates a Cuboid data structure that is then stored in the object.

Building Box and Sphere Extents (9)

- To build the sphere extent, the center of the sphere extent is chosen to be the *centroid* of the point cluster since this is a nice centralized point.
  - To find the centroid, just add up all of the points component-wise, and divide by the number of points.
- Then the radius of the enclosing sphere is found as the largest distance from this center to any one of the points in the cluster.
- Based on this approach, the routine void makeSphereExtent (PointCluster& clust, SphereInfo& sph) is easily fashioned.

Building Box and Sphere Extents (10)

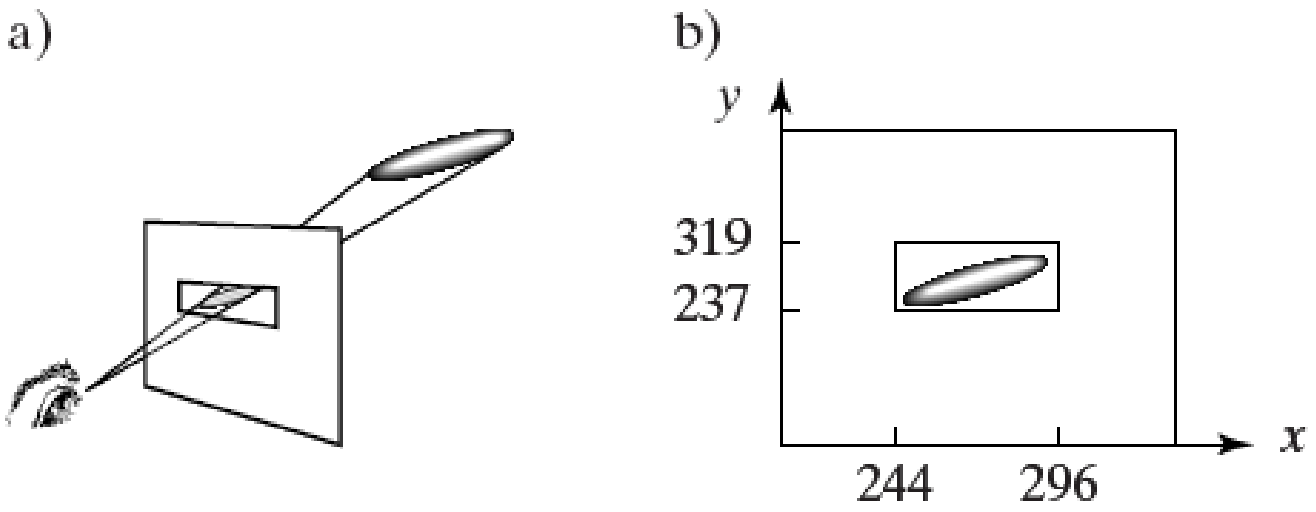
- Finally, it is easy to form the world box extent and world sphere extent.
- Simply transform each of the points of the generic object's point cluster to form a point cluster in world coordinates.
- Then use the same routines as above to form the desired extents.

Using Projection Extents

- A projection extent provides a dramatic gain in the speed of a ray tracer.
- In contrast with box and sphere extents, which operate in 3D space, this extent is a rectangular region on the screen.
- The **projection extent** of an object is an aligned rectangle on the screen that encloses the projection of an object.
- The projection extent is captured by four numbers {*left*, *top*, *right*, *bottom*}.
- It is very easy to use projection extents while ray tracing.

Using Projection Extents (2)

- Each ray passes through the screen at a certain row and column value, say (*r*, *c*). If (*r*, *c*) lies outside of the projection extent of an object, the ray cannot possibly intersect the object.
- For the example in (b), if *c* is less than 244 or larger than 296, or if *r* is less than 237 or larger than 319, the ray definitely misses the object.



Using Projection Extents (3)

- Because projection extents are intimately coupled to the geometry of the camera and the position of the eye, they can only be used for **eye rays**, those that emanate from the eye.
- In later sections we will be generating rays for shadowing, reflections, and refraction, and since these emanate from arbitrary points in the scene, there is no screen on which to place a projection. Projection extents cannot be used in these cases.

Using Projection Extents (4)

- We construct the projection extent for each object in the scene in a preprocessing step and store the extent with the object just as we did with the box and sphere extents.
- To accommodate this, we add an IntRect screenExtent field to the GeomObj class.
- We also extend the Ray class so that a ray knows which row and column on the display it is passing through, and its **recursion level**.
- This will play a big part later when we work with reflection and refraction. Here the level simply keeps track of whether this ray is an eye ray. Eye rays are given a level of 0.

Code to Use Projection Extents

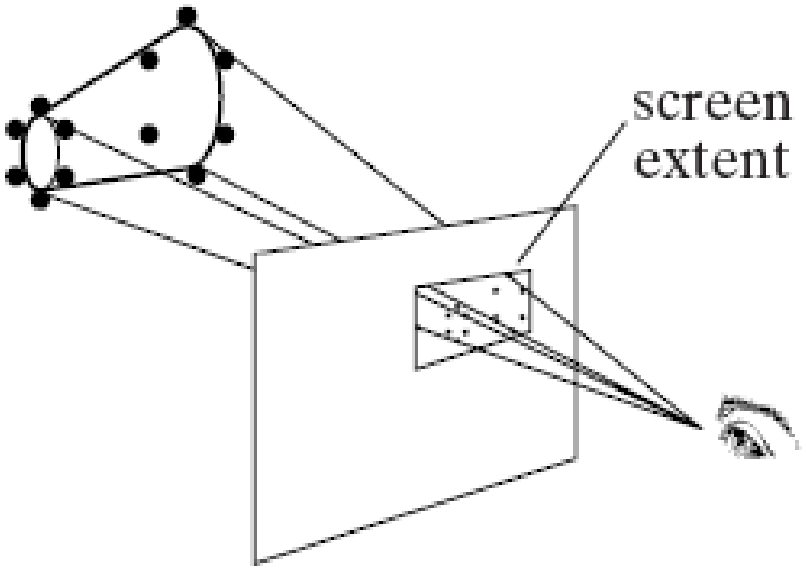
```
bool TaperedCylinder::hit(Ray &r, Intersection &inter)
{if (r.recurseLevel == 0 &&
 r.col < projExtnt.left || r.col > projExtnt.right||
 r.row < projExtnt.bottom || r.row > projExtnt.top )
    return false;  //misses screen extent
 if (!rayHitsSphereExtent(r,worldSphereExtent))
    return false;
// make, and inverse transform, the generic ray
 if (!rayHitsSphereExtent(genRay,genSphereExtent))
    return false;
 …do expensive full testing with the generic cylinder… }
```

Using Projection Extents (5)

- During ray tracing, a new and very fast test is performed at the start of the hit() method for each object type before any sphere or box extent tests.
- This same test is used for all object types – except for the plane.
- The test checks whether this ray is an eye ray, and if so, it tests the ray's row and column against the projection extent stored in the object.
- If the row or column is outside of the projection extent, the ray must miss the object, and hit() immediately returns false.

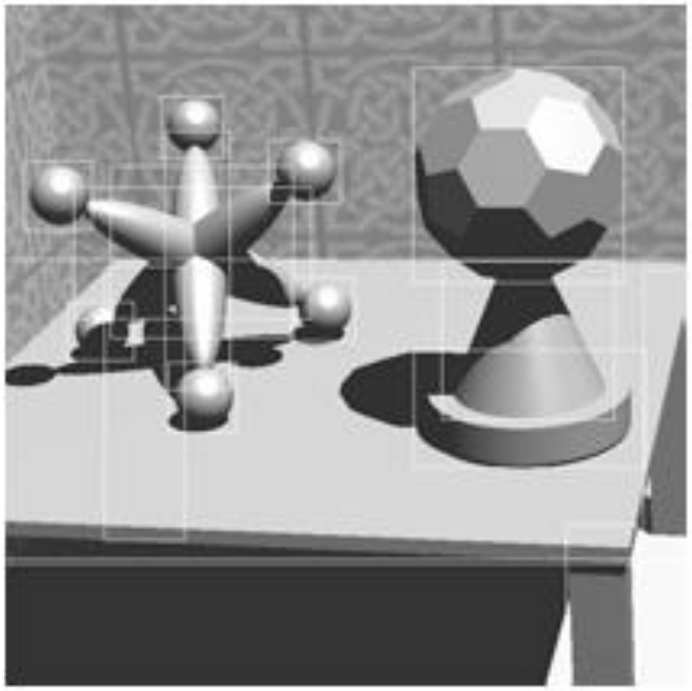
Computing Projection Extents

- The preprocessing step must compute the projection extent for each object in the scene, using the point cluster of the object.
- The point cluster of the generic object is transformed into world coordinates using the object's transformation just as we did for world sphere and box extents.



Computing Projection Extents (2)

- Each of the points *p* is projected onto the near plane of the camera, based on the location of *p* and the camera geometry. The projected point *p*' is associated with a particular row *r* and column *c.*
- The figure shows a ray traced scene with each object's projection extent superimposed on the object (for debugging).



BSP Trees

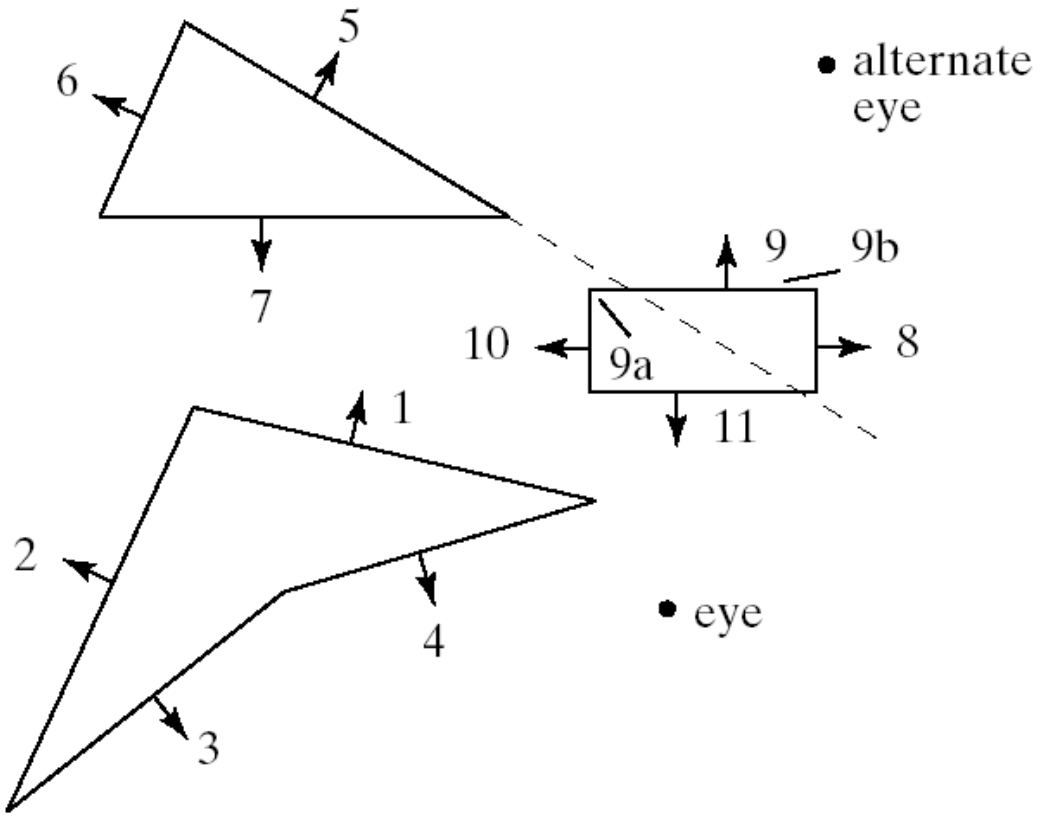
- Ray tracing is very slow because of the large number of rays that must be cast and the large number of objects that are typically in a scene.
- The problem is clearly the large number of intersection tests that must be performed for each ray and each object.
- The use of extents helps somewhat to increase efficiency, but is of little service for secondary (spawned) rays.
- We therefore seek other ways for accelerating ray tracing. One elegant method for accelerating ray tracing makes use of Binary Space Partition (BSP) trees.

BSP Trees (2)

- We build a hierarchical data structure for each ray that eliminates a large number of intersection tests by quickly determining which objects could not possibly intersect that ray.
- Binary space partition trees (BSP) are data structures which can be constructed rapidly and then traversed (by visiting each of its nodes in turn, in some order), and testing for an intersection between the ray and the node object.
- At each node, the ray is intersected with the object but needs only a coarse intersection test.

BSP Trees (3)

- We introduce BSP trees using a 2D example involving edges. Because the organization of the tree does not depend on the dimensionality of the objects contained in the tree, the extension to a BSP of 3D polygons is immediate.

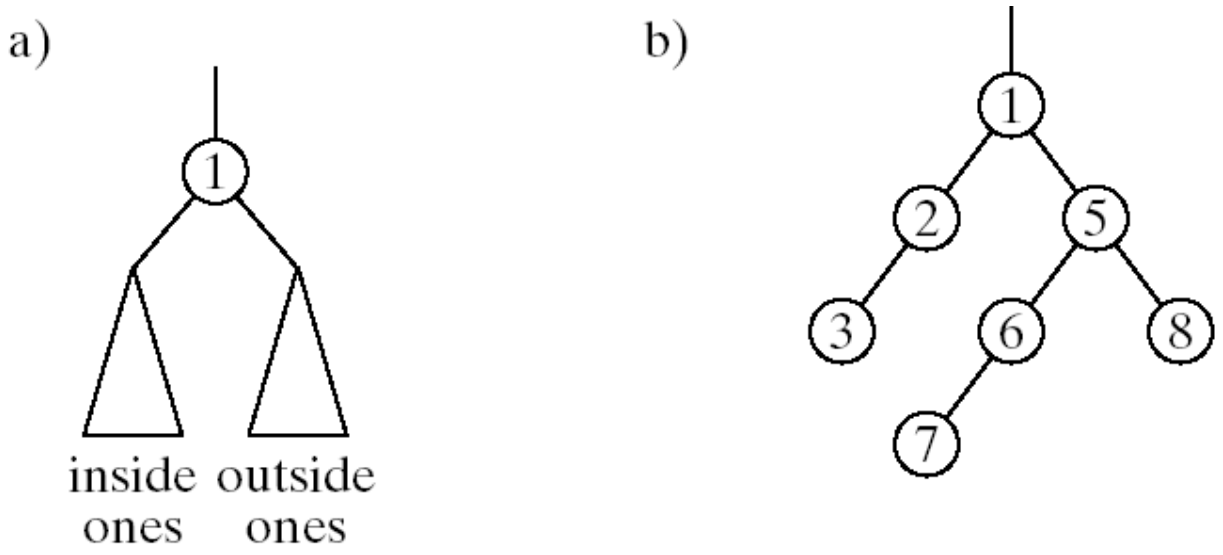


BSP Trees (4)

- The figure shows 3 polygons in a plane, as well as an eye.
- The question is: what is seen from the ray origin?
- Each edge is given a number, and its outward pointing normal vector is indicated.
- We wish to insert each edge into a tree in such a way that the plane is partitioned (or tesselated) into non-overlapping polygonal regions.

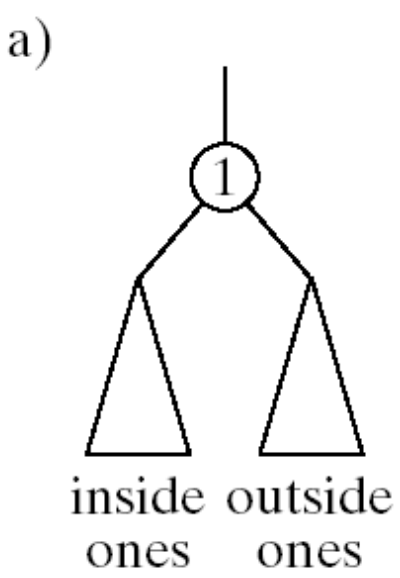
BSP Trees (5)

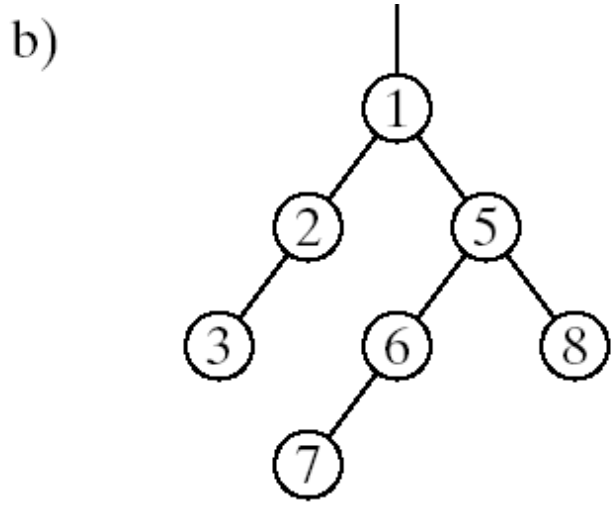
- To build the tree for this set of edges, take edge #1.
- Extending it splits the plane into two half spaces: its outside (the side pointed to by the outward pointing normal) and its inside. We put edge #1 into the root of the tree, as shown in the figure (a).



BSP Trees (6)

- Now we insert each of the other edges into the tree one by one: those that lie in the outside half-space of #1 will go into the right subtree (*outsideOnes*), and those that lie in the inside of #1 will go into the left subtree (*insideOnes*).



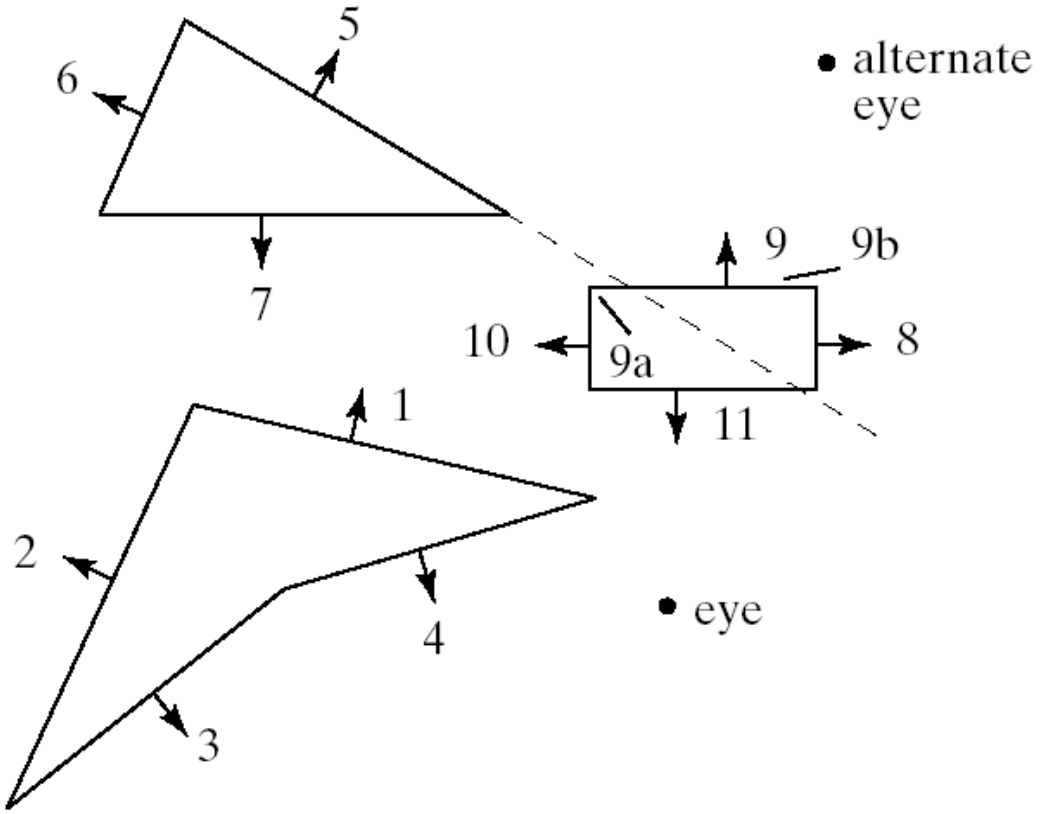


BSP Trees (7)

- As each edge is inserted, it finds its way down the tree, comparing itself with each node it reaches.
- It goes to the *outsideOnes* subtree of the node if it lies on the outside of that node, and to the *insideOnes* subtree otherwise, until it becomes attached to the tree as a new leaf.

BSP Trees (8)

- When we insert edge #9, something new happens. It lies on the outside of #1, but lies in both half spaces of #5.
- It is therefore **split** by #5 into two edges: #9a and #9b, as shown.



BSP Trees (9)

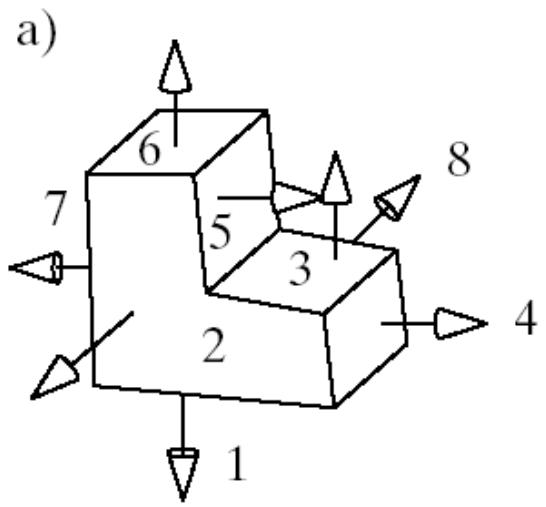
- Each of the pieces finds its way further down in the tree.
- Edge #9a lies on the inside of #5 so it goes down to the left and is tested against #6 and #7.
- Edge #9b lies on the outside of #5 but the inside of #8.
- Edges #10 and #11 are inserted in like manner and don’t require splitting.

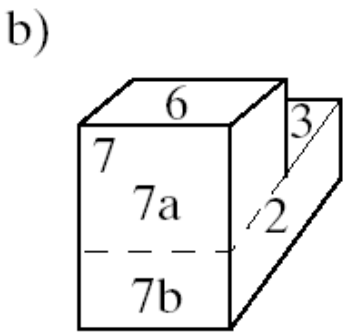
BSP Trees (10)

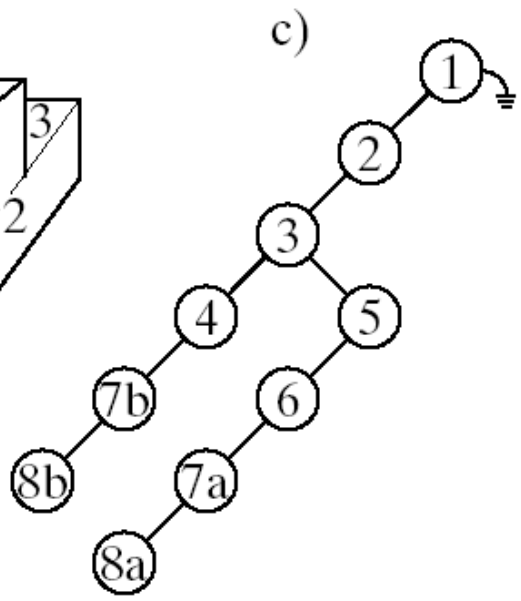
- The BSP tree lists the edges in a particular way, so that for any edge you can quickly tell (by its position in the tree) in which part of space it lies (e.g. edge #6 lies on the outside of #1 and on the inside of #5).
- The order in which edges are inserted in the tree has a profound effect on the “shape” of the final tree.
- Some choices of ordering can make the tree full and “shallow” (a small number of nodes from the root to the deepest leaf), while others make it scrawny and “deep”. Shallow trees are more efficient to deal with.
- The worst case is a BSP tree in the shape of a single chain from top to bottom.

3D BSP Trees

- Consider the block shown. Each of its eight faces is numbered and the outer pointing normal vector is indicated.
- Part b) shows another view of the same block. If the faces are inserted into a BSP tree in the order #1, #2, …, #8 the tree shown in part c) is formed.







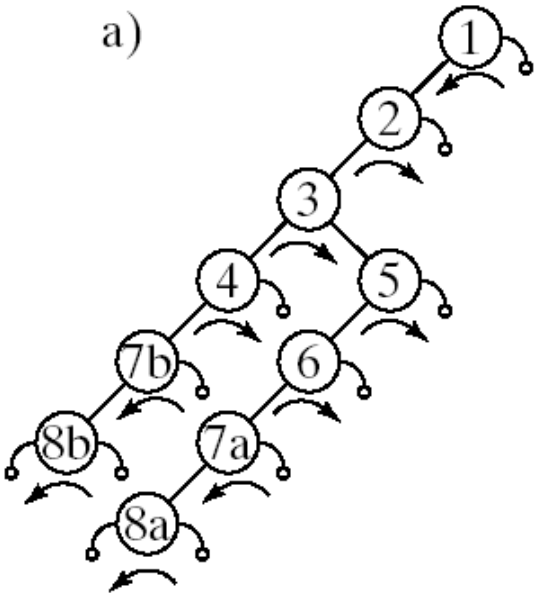
3D BSP Trees (2)

- A face finds its way down the tree in the same way as for the 2D case; if it lies in the outside half-space of the node face it goes to the *outsideOnes* subtree, etc.
- Face #7 is the only one that must be split: it is split into #7a and #7b by the plane of #3, as the dashed line in the figure indicates.

Traversing the Tree (Ray Tracing)

- We know the eye ray hits some object and that hit times are smaller for each left child in the tree than for the right child at the same level.
- Thus, to find the smallest time, simply trace the tree down from the root always moving to the left child. The leaf 8b must be the closest object to the eye.

a)



b)

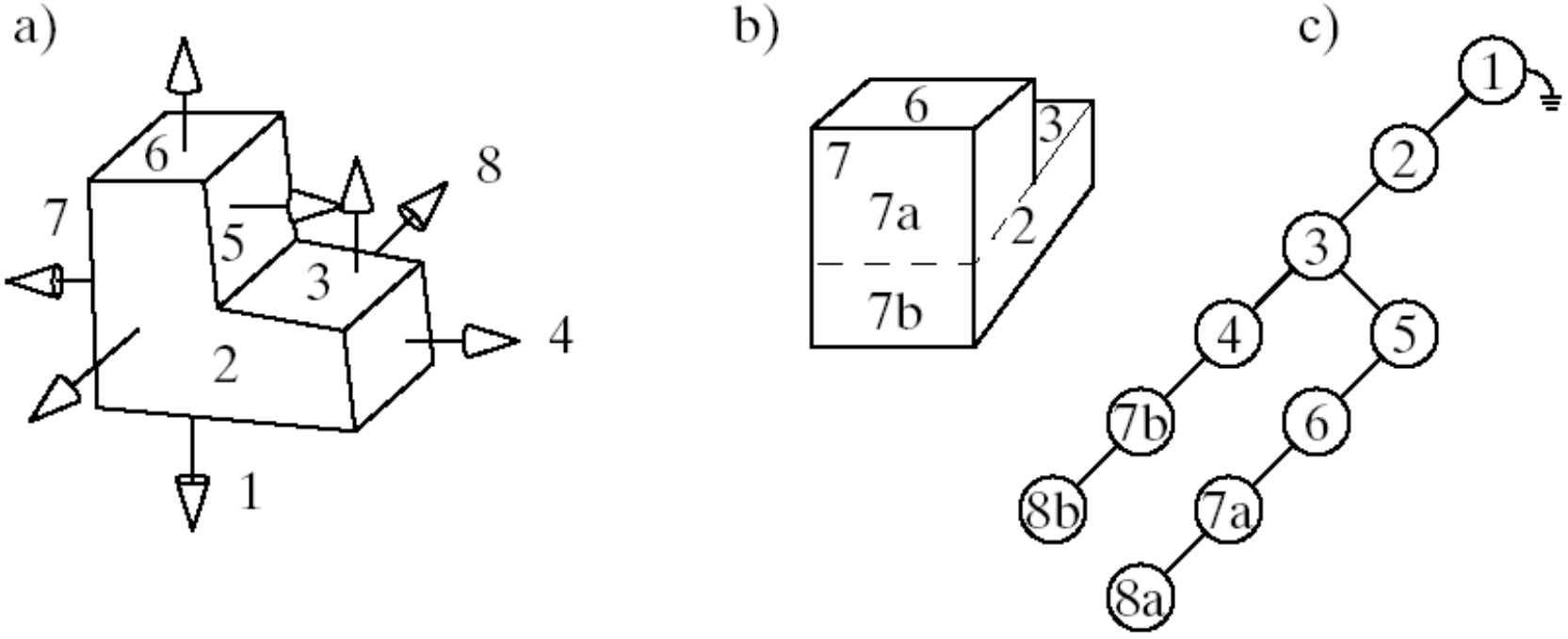
1 *7b* *8b* 4 3 *7a* 8a 6 5 2

c)

1 *4* *8b* 7b 3 *5* *8a* 7a 6 2

Traversing the Tree (2)

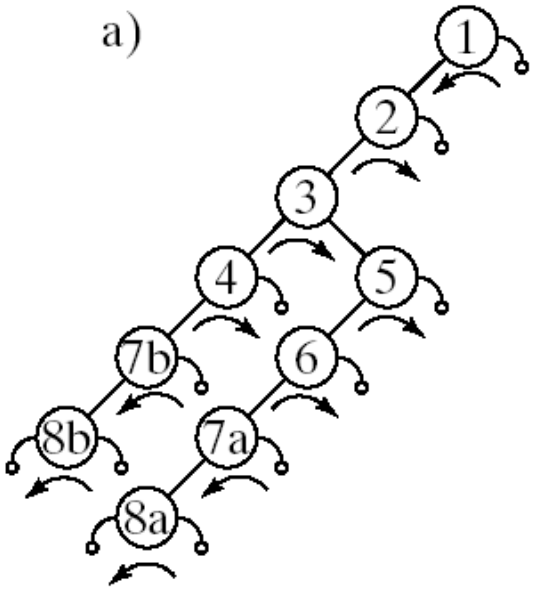
- For the 3D scene in (a), the tree in part c) can be traversed using this scheme, and the block will be drawn properly.



Traversing the Tree (3)

- The arrows indicate the order in which nodes of the BSP tree for the block are visited for the particular view of the block shown.
- The order of increasing hit times is shown in part b). Those faces which are underlined are never drawn.

a)



b)

1 *7b* *8b* 4 3 *7a* 8a 6 5 2

c)

1 *4* *8b* 7b 3 5 *8a* 7a 6 2