《Fundamentals of Computer Graphics》

Lecture 8、Ray tracing

Part 6: CSG objects

Yong-Jin Liu

liuyongjin@tsinghua.edu.cn

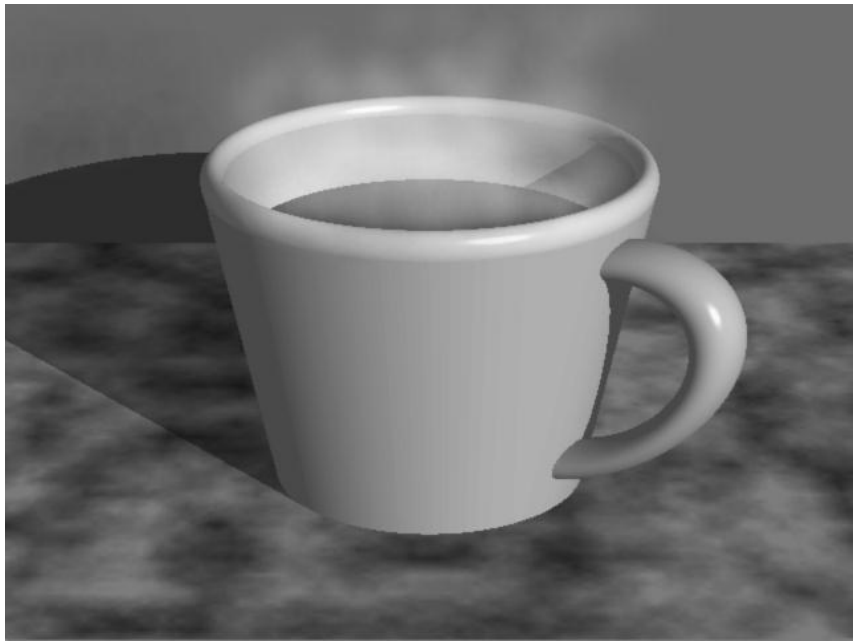
Material by S.M.Lea (UNC)

Compound Objects

- So far, we can ray trace only a limited variety of shapes: transformed spheres, planes, cones, etc.
- But if we can combine these simple shapes into more complex ones and develop a ray tracing method for them, much richer and more interesting scenes can be ray traced.
- The method, known as **constructive solid geometry** (CSG), provides such a method. Arbitrarily complex shapes are defined by set operations, also called **boolean operations**, on simpler shapes.
- Objects such as lenses and hollow fishbowls and objects with holes are easily formed by combining the generic shapes treated so far. Such objects are variously called **compound, Boolean**, or **CSG** objects.

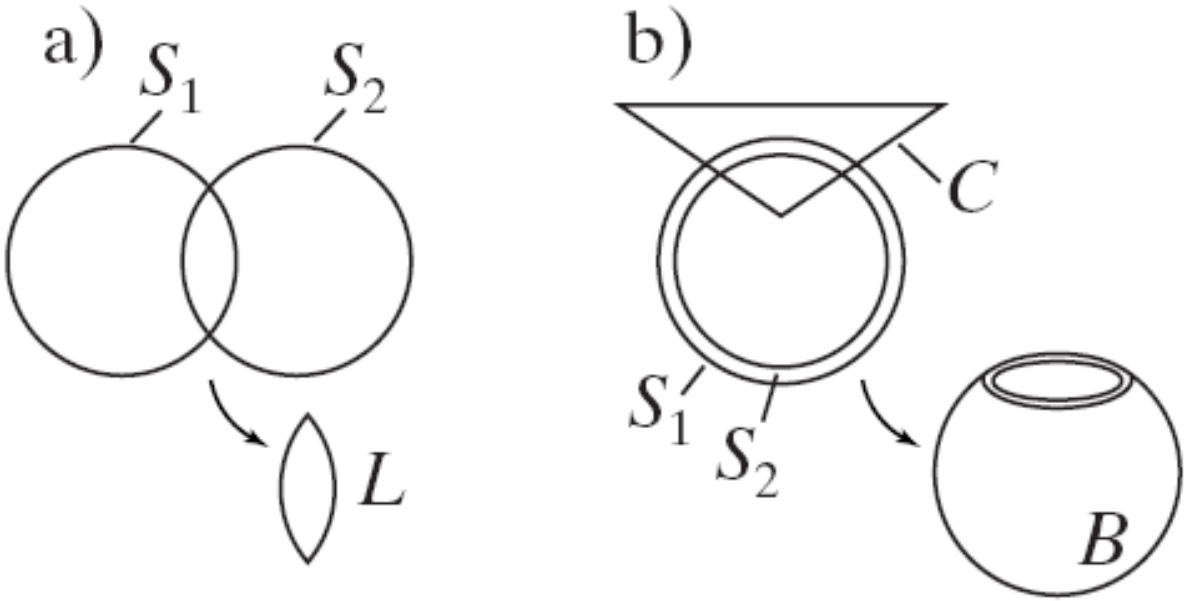
Compound Objects (2)

- The ray tracing method extends in a very organized way to compound objects; it is one of the great strengths of ray tracing that it fits so naturally with CSG models.



Compound Objects (3)

- We look at examples of the three Boolean operators: **union**, **intersection**, and **difference**.
- The figure shows two compound objects built from spheres. (a) is a lens shape constructed as the **intersection** of two spheres.



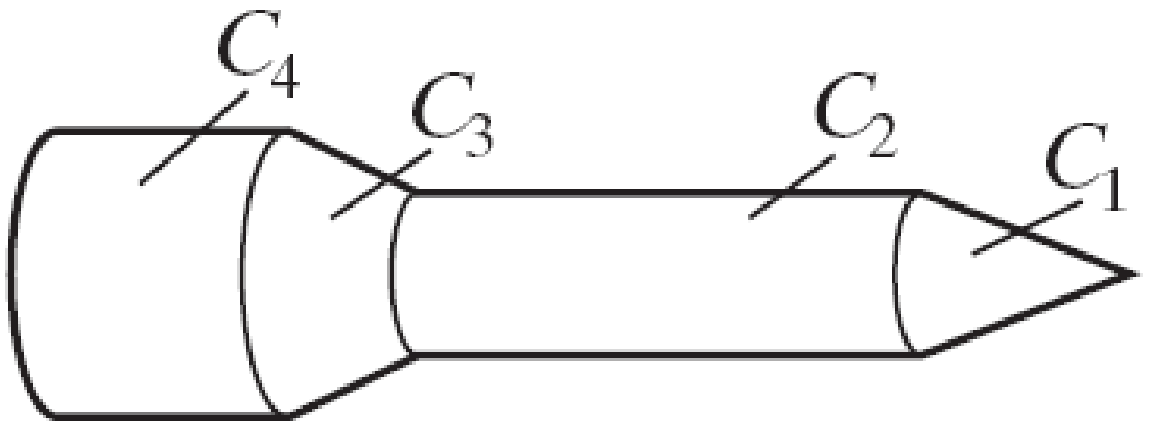
Compound Objects (4)

- A point is in the lens if and only if it lies in *both* spheres. Symbolically, one writes that "*L* is the intersection of the spheres *S*1 and *S*2", or $L = S_1 \cap S_2$
- (b) shows a bowl constructed using the difference operation. A point is in the **difference** of sets *A* and *B*, denoted *A*-*B*, if it is in *A* and not in *B*. Differencing is analogous to removing material, to cutting or carving.
- The bowl is specified by $B = (S_1 - S_2) - C$
- The solid globe, S_1, is hollowed out by removing all the points of the inner sphere, S_2. This forms a hollow spherical shell. The top is then opened by removing all points in the cone, *C*.

Compound Objects (5)

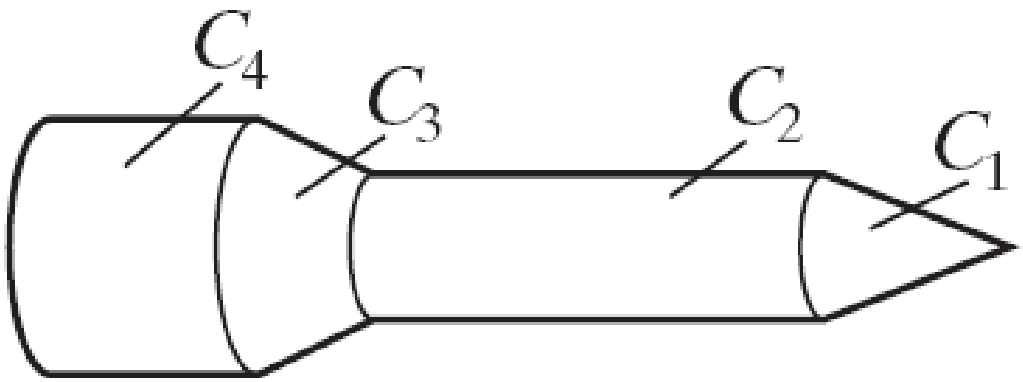
- A point is in the **union** of two sets *A* and *B*, $A \cup B$, if it is in *A* or in *B* or in both. Forming the union of two objects is analogous to gluing them together.
- The rocket is constructed as the union of two cones and two cylinders and is represented mathematically as

$$R = C_1 \cup C_2 \cup C_3 \cup C_4$$



Compound Objects (6)

- Cone C_1 rests on cylinder C_2. Cone C_3 is partially embedded in C_2 and rests on the fatter cylinder C_4.

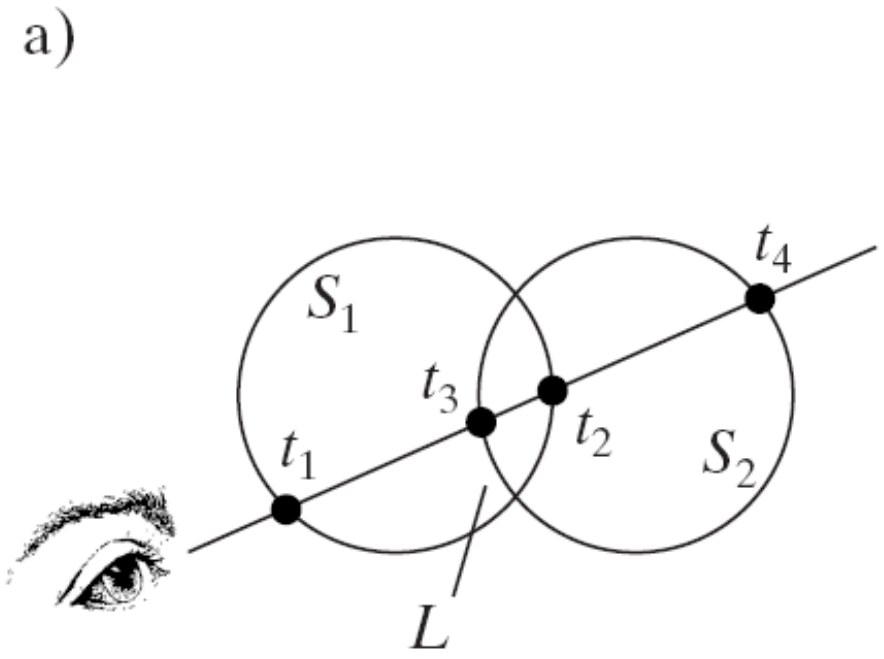


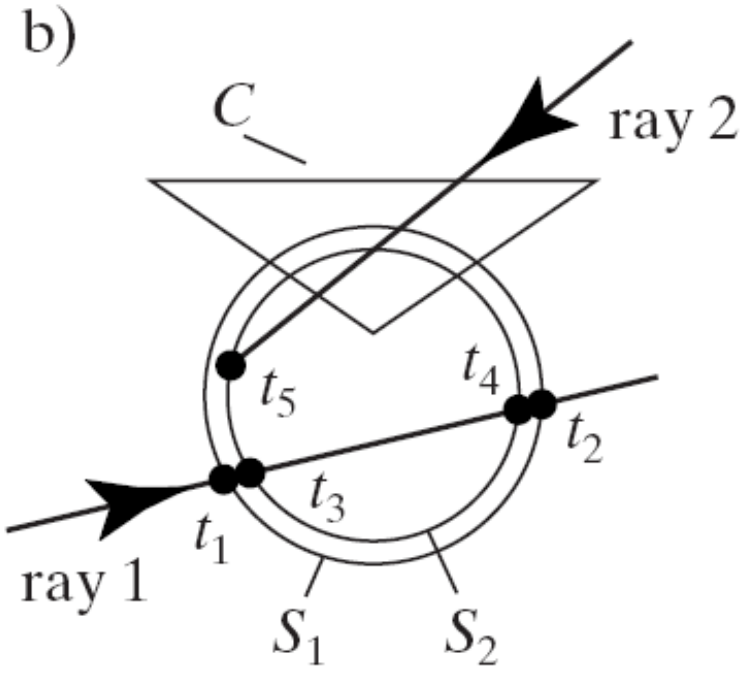
Ray Tracing CSG Objects

- How do we ray trace objects that are Boolean combinations of simpler objects?
- Due to the nature of the ray tracing process, it can be handled with surprising ease.
- In fact the intersection of a ray with a 3D CSG object is reduced to the intersection of an ordered list of one-dimensional *numbers* with one-dimensional *parts* of a CSG object!

Ray Tracing CSG Objects (2)

- First consider the proceeding examples. This figure shows a ray entering and exiting the spheres S_1 and S_2 at the times indicated.
- It is therefore inside lens L from t_3 to t_2, and the hit time is t_3.





Ray Tracing CSG Objects (3)

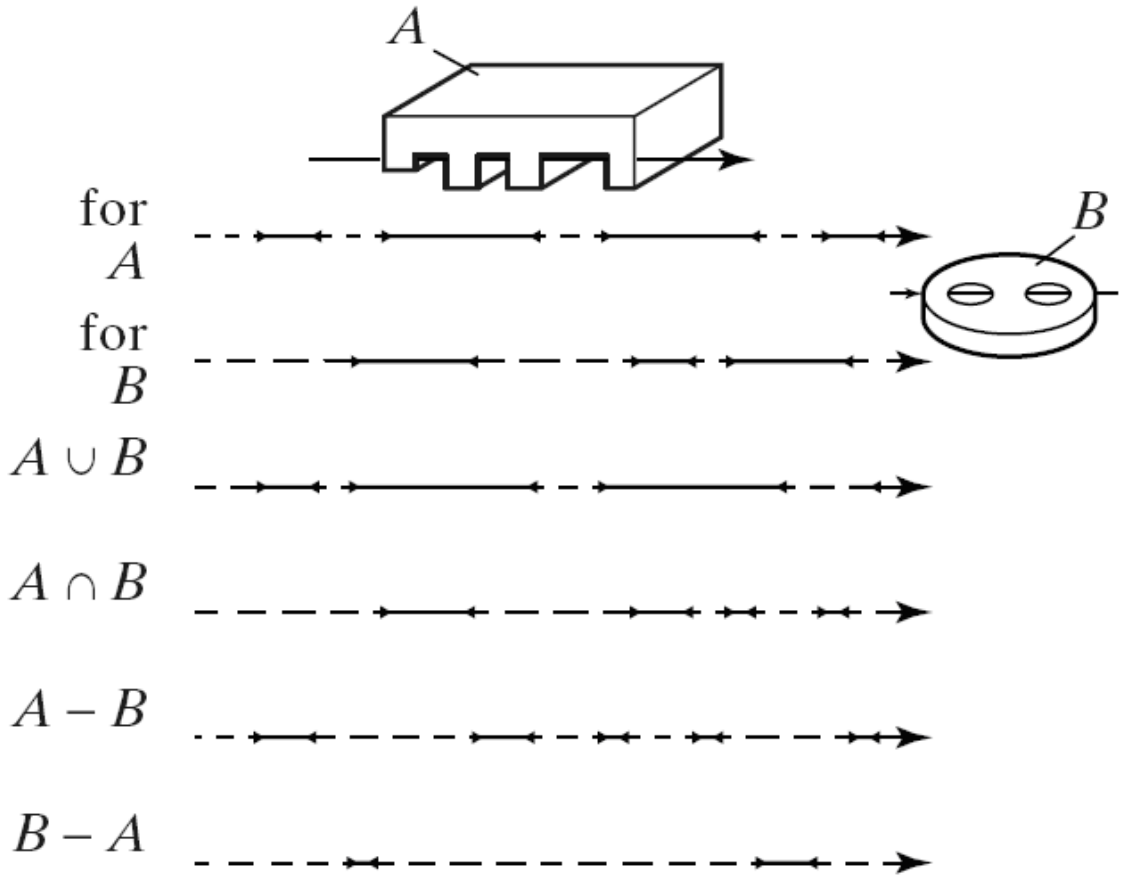
- If the lens is opaque, the familiar shading rules will be applied to find what color the lens has at the hit spot. If it is mirror-like or transparent, spawned rays are generated with the proper directions and are traced farther.
- The situation is similar for the bowl. Ray 1 first strikes the bowl at t_1, the smallest of the times for which it is in S_1 but not in either S_2 or C.
- Ray 2, on the other hand, first hits the bowl at t_5. Again, this is the smallest time for which the ray is in S_1, but in neither the other sphere nor the cone. The hits at earlier times are hits with component parts of the bowl, but not the bowl itself.

Ray Tracing CSG Objects (4)

- We organize these ideas into an algorithm to ray trace any compound object.
- Consider two objects, *A* and *B*, and a ray.
- Build a list of times at which the ray enters and exits from *A*, ordered so that the times are increasing. Because the object is solid, enter and exit times alternate.
- Build a similar list of enter and exit times for *B*.

Ray Tracing CSG Objects (6)

- Two such lists are shown in the figure.
- The interval between each enter time and the next exit time is shown solid; otherwise, it is shown dashed. The ray is inside the object for each solid interval.



Ray Tracing CSG Objects (7)

- Call the set of *t*-values for which the ray is inside the object its **inside set**.
- An inside set is specified by an ordered list that contains the alternating enter and exit times of the ray (t_1, t_2 , . . .), where t_1 is an enter time, t_2 is an exit time, and so forth.
  - This is sometimes called the ray's ***t*-list**.
- Given the inside sets for a ray with compound objects *A* and *B*, we want to find the inside set for the ray with a compound object built from *A* and *B*.

Ray Tracing CSG Objects (8)

- The four new objects we can build are
  $A \cup B, A \cap B, A - B, B - A$
- Consider $A \cup B$. The ray is inside the union of the objects if it is inside either of the objects. Thus the inside set for union is simply the union of the individual inside sets.
- The same thinking applies to the other three objects. The inside set for intersection is the intersection of the individual inside sets; the inside set for *A* - *B* is the difference of the inside set for *A* and that for *B*, and similarly for *B* - *A* (just reversing the roles of *A* and *B*).

Ray Tracing CSG Objects (8)

- Although we are ultimately interested in only the first item in the inside set—the first hit time—we must retain *entire lists* for inside sets during the list building process, because the ultimate *first* hit time may lie deep within one of the intermediate lists.
- Happily, we already have mechanisms in place for creating inside sets in a ray tracer: the hit() method for each type of shape carefully collects all of the hits that a ray experiences with an object in the inter.hit[ ] array in time-sorted order.
- Up to this point, we have only made use of the first hit time in the list. As we begin to ray trace Boolean objects, however, we will utilize the entire list.

Example: Inside Sets

- A_list: 1.2 1.5 2.1 2.5 3.1 3.8
- B_list: 0.6 1.1 1.8 2.6 3.4 4.0
- Union: 0.6, 1.1, 1.2, 1.5, 1.8, 2.6, 3.1, 4.0
- Intersection: 2.1, 2.5, 3.4, 3.8
- A – B: 1.2, 1.5, 3.1, 3.4
- B – A: 0.6, 1.1, 1.8, 2.1, 2.5, 2.6, 3.8, 4.0
- To construct new inside sets, we need a function combineLists(), that takes as arguments an operator and two *t*-lists, and creates a new list of *t*-values according to one of the preceding four combining methods.

Ray Tracing CSG Objects (9)

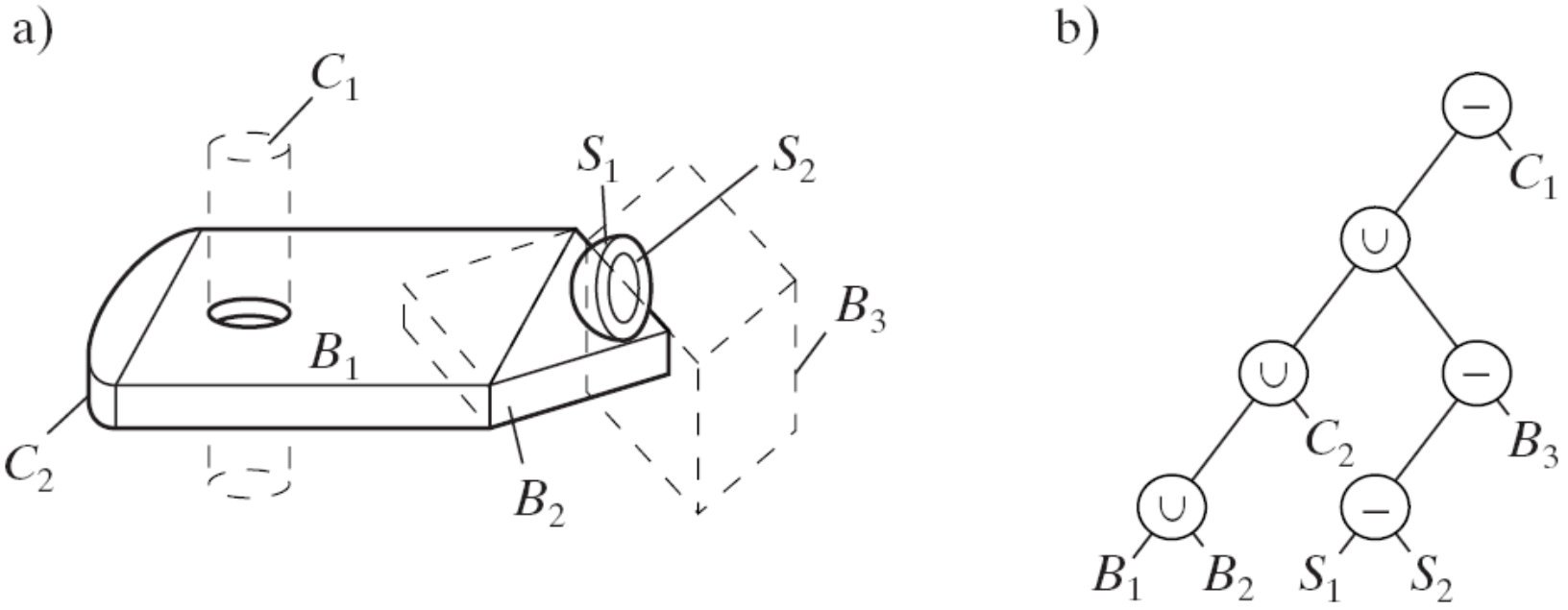
- The ray tracing process for a compound object boils down to ray tracing its component objects, building inside sets for each, and finally combining them.
- The first positive hit time on the combined list yields the point on the compound object that is hit first by the ray.
- The usual shading is then done, including the casting of secondary rays if the surface is shiny or transparent.

Data Structure for Compound Objects

- How do we represent a compound object such as a union in a program?
- Since a compound object is always the combination of two other (possibly compound) objects, Obj_1 *op* Obj_2, a binary tree structure provides a natural description.

Data Structure for Compound Objects (2)

- This figure shows the tree corresponding to the object $(((B_1 \cup B_2) \cup C_2) \cup ((S_1 - S_2) - B_3) - C_1$.
- Each internal node is an operator, and each leaf is a primitive shape.



Data Structure for Compound Objects (3)

- A tree is made up by combining subtrees, which are in turn primitive objects or subtrees.
- When a node contains the difference operator it is understood that the node produces the (left subtree) - (right subtree), as opposed to (right subtree) - (left subtree).
- The expression for a compound object can be rearranged to some degree without altering the ultimate shape of the object being represented. Each rearrangement gives rise to a different tree. Therefore, there is not a unique tree for a given compound object.

Data Structure for Compound Objects (4)

- Currently, all the object types like Sphere and Cube are derived from the Shape class, which is in turn derived from the basic GeomObj class.
- It seems natural to derive a new class Boolean from GeomObj, so that the object list (which is a list of GeomObj's) can hold a Boolean object.
- A Boolean object must be able to hold a binary tree, so we give it pointers left and right to point to its child subtrees. These must be pointers to GeomObj's, since in some places they point to geometric shapes, and in others to Boolean trees.

Data Structure for Compound Objects (5)

```
class Boolean: public GeomObj {
public:
   GeomObj *left, *right; //pointers to the children
   Boolean() {left = right = NULL;}// constructor
   virtual bool hit (Ray &r, Intersection &inter);
… other methods …
};
```

Data Structure for Compound Objects (6)

- Because GeomObj's have sphere and box extents, so do Boolean's, which is useful since Boolean's are expensive to ray trace, and anything that speeds this up is a blessing.
- We shall design a special hit() routine for Boolean's that understands how to intersect a ray with a Boolean, which capitalizes again on polymorphism to simplify code.
- Actually, since hit() must operate differently for unions, intersections, and differences, it will improve matters to derive separate classes for the three kinds of Booleans: UnionBool, IntersectionBool and DifferenceBool.

Data Structure for Compound Objects (7)

- The definition of UnionBool is simply the following.

```
class UnionBool : public Boolean{
public:
   UnionBool(){Boolean();} //constructor
   virtual bool hit(Ray &r, Intersection &inter);
   … other methods …
};
```

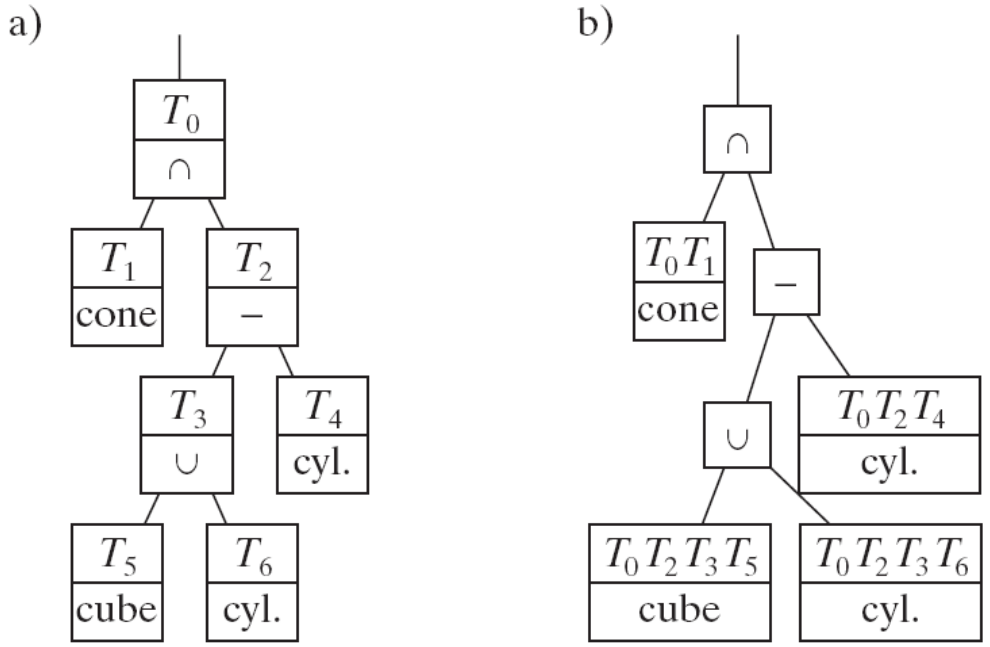
- The other definitions are nearly identical. Each type will have a hit() routine that is finely tuned to handle its specific nature.

Data Structure for Compound Objects (8)

- Boolean's do not have their own affine transformation; only Shape's do.
- This causes a Boolean tree to have transformations only at its leaves.
- This is a design decision that offers both advantages and disadvantages, and in fact some modeling systems and ray tracers operate differently.

Data Structure for Compound Objects (9)

- (a) tree for a Boolean object that *does* have a transformation at each internal node, as well as one at each leaf. The effect of each transformation is felt by all nodes beneath it in its subtrees.
- Tree (a) is equivalent geometrically to tree (b): both trees represent the same shape. In (b) the transformations have been combined to form a single transformation at each leaf.



Data Structure for Compound Objects (10)

- One advantage of keeping explicit transformations in internal nodes is that the designer can separately alter a particular transformation deep within a tree to adjust the shape of a CSG object after it has been created.
- It is also possible to have more control over the tightness of various extents for Boolean objects.
- A significant disadvantage, on the other hand, is the speed of ray tracing, since a ray must be inverse transformed at every node of the tree, not just at the leaves.

SDL and Booleans

- To specify a Boolean with *SDL*, use one of the key words union, intersection, or difference, followed by the specification of the left geometric object, then the specification of the right one (which can themselves be Boolean's).
- Example: the fishbowl shape, given by $(S_1 - S_2) – C$:

```
! make a fishbowl
rotate -90 1 0 0
difference
  difference
     sphere  ! outer sphere
     push scale 0.9 0.9 0.9 sphere pop !inner sphere
  push translate 0 0 1.3 rotate 180 1 0 0 cone pop
```

SDL and Booleans (2)

- The first transformation rotates the whole bowl about the *x*-axis.
- The difference is then formed between the left object (which is itself a difference of two spheres) and the cone.
- Because transformations given to *SDL* post multiply the current transformation, and the current transformation is placed in a shape object at the time it is created, transformations are automatically percolated down the tree.

Intersecting Rays with Booleans

- We need a hit() method to work with each type of Boolean object. It must form the inside set for the ray with the left subtree, the inside set for the ray with the right subtree, and then combine the two sets appropriately.
- Consider a skeleton of hit() for the case of intersection:
- Extent tests are first made to see if there is an early out.
- Then the proper hit() is called for the left subtree, and unless the ray misses this subtree, the hit list lftInter is formed.
- If there is a miss, hit() returns false immediately, producing an early out because the ray must hit both subtrees in order to hit their intersection.
- Else the hit list rtInter is formed and finally the two hit lists are combined using the intersection operation.

Intersecting Rays with Booleans (2)

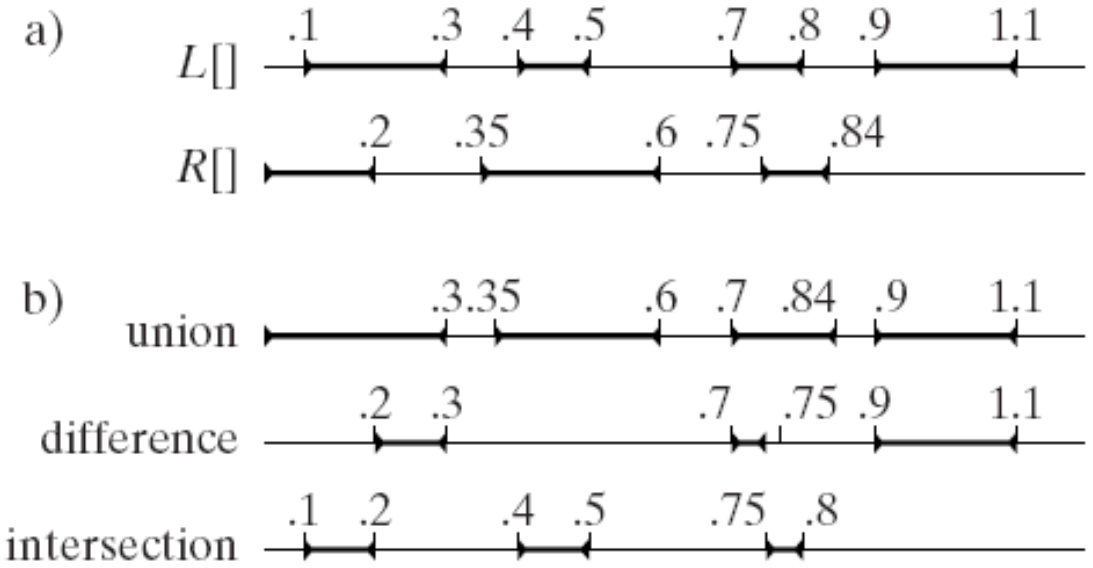
- **Skeleton of hit() for an intersection boolean object.**

```
bool IntersectionBool:: hit(Ray &r, Intersection &inter)
{
    Intersection lftInter, rtInter;
    if (ray misses the extents) return false;
    if ((!left->hit(r,lftInter))||(!right->hit(r,rtInter)))
            return false; // early out
    make the combined list: place it in inter
   return (inter.numHits > 0); // true if inter is not empty
}
```

- The code is similar for the UnionBool and DifferenceBool classes.

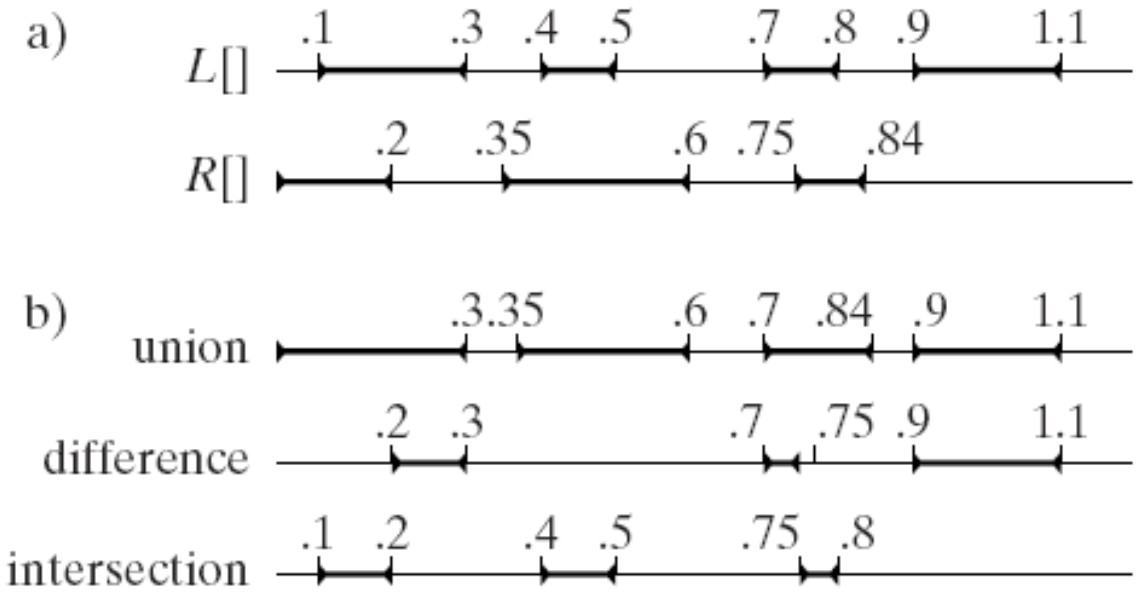
Combining t-lists

- Combining *t*-lists is intricate but logical. Consider the two example hit lists, *L* and *R*, (standing for left and right) shown in the figure.
- For brevity we call the left list simply *L*[ ] rather than leftInter.theHit[ ], and similarly for the right list. In addition, the hitObject and surface fields are not shown.



Combining t-lists (2)

- *L* shows 8 hits, and *R* shows 5 hits. Each list contains positive hit times in increasing order, and the *entering* field of the 0th element shows whether the first positive-time hit occurs with the ray entering or exiting the object.
- Since all objects are solid the entering values alternate after the 0th.

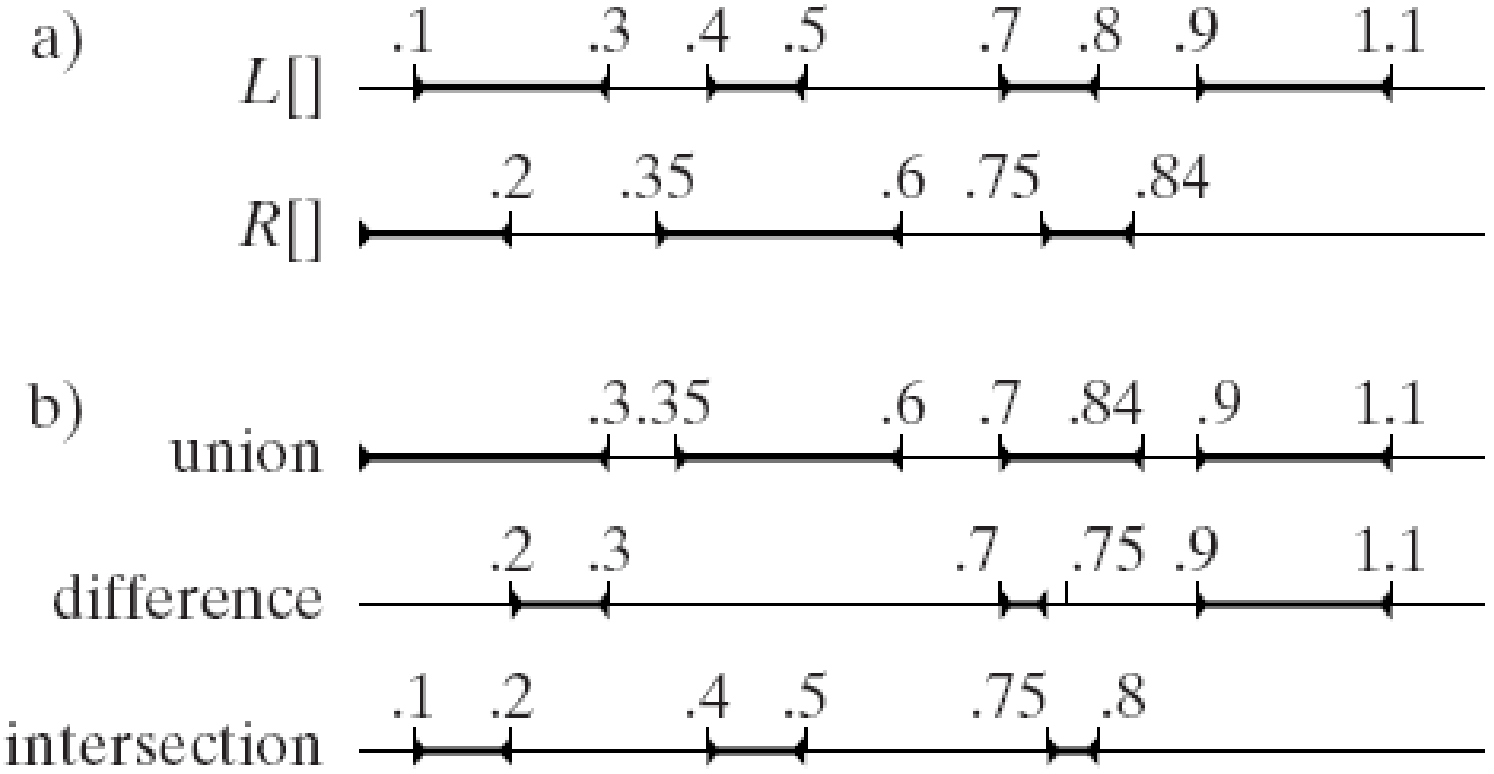


Combining t-lists (3)

- The lists are combined by scanning through both lists, noting at each step which next item in the two lists has the smaller hit time, and keeping track of whether the ray is inside or outside the object just before this next hit.
- These pieces of information are combined in a manner appropriate to the operator involved: *union*, *intersection*, or *difference*.

Combining t-lists (3)

- The figure (b) shows the resulting *t*-list after combination, for each of the Boolean operators.



Combining t-lists: Example

- Suppose that the operator is *difference*, that just before the next hit time the ray is inside the left object and outside the right object, so it must be inside the difference.
- The next hit time on each list is examined; suppose the right one is found to be smaller. This means that just after this next hit time the ray is inside the right object, so it must now be outside the difference.
- At this instant the state of the ray in the combined object changes from inside to outside.

Combining t-lists (4)

- The state of the ray (whether it is inside or outside the object at the current time) as it progresses through the subtrees is described by the following variables.
- bool lftInside; // true if ray is inside left object;
- bool rtInside; // true if ray is inside right object;
- bool combInside; // true if ray is inside combined object;
- These are the states of the ray *between* hits: that is, just before the next hit to be considered.
- lftInside is initialized according to whether the ray enters or exits the left object at the first hit; that is, according to L[0].isEntering. Specifically, lftInside is false if the first hit is entering (because before the hit the ray must be outside), and is true if the first hit is exiting. So we can initialize lftInside to!L[0].isEntering, and similarly for rtInside.

Combining t-lists (5)

- Then combInside is found as a logical combination of lftInside and rtInside; if the operator is *union,* combInside is true if lftInside or rtInside or both are true. This is nicely captured using the logical operators: combInside = lftInside || rtInside.
- Similarly if the operator is *intersection* we use combInside = lftInside && rtInside, and if it is *difference* we use combInside = lftInside && !rtInside.
- The algorithm proceeds by incrementing through both the *L*[ ] and *R*[ ] lists, at each step noting the smaller of the hit times, and using it to update lftInside, rtInside, and combInside. If combInside *changes* at any point the latest hit event is added to the *C*[ ] list.

Combining t-lists (6)

- When either *L*[ ] or *R*[ ] has been consumed (when its final hit time has been examined) the other unconsumed list may or may not need to be considered:
- for **intersection**: ignore the unconsumed list;
- for **union**: increment through the unconsumed list, adding hit events as appropriate;
- for **difference**: if the left list is the unconsumed one, increment through it adding hit events as appropriate. If the right list is unconsumed, ignore it.

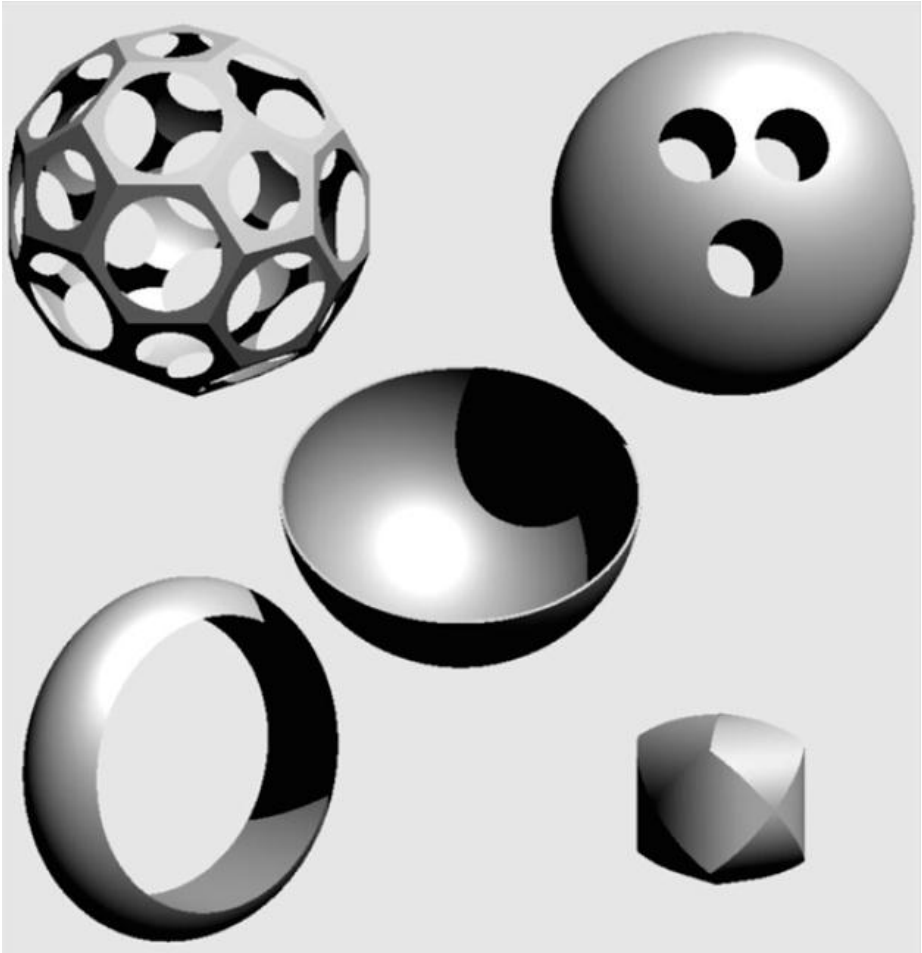
Combining t-lists (7)

- When combining lists, we must put all necessary information into each new hit[ ] record.
- Specifically, the isEntering and hitObject fields must be properly filled because if a hit ever becomes the first hit of the ray we will be gleaning properties about the surface hit from the hitObject itself.
- When an exiting hit on the right object causes the ray to be entering the object itself, we must be sure the hit information points to the left object, so that what we see in the hole are the properties of the left object.

Combining t-lists (8)

- We also must make sure that the logic of hit() for CSG objects works properly when the object is transparent and the ray is traveling inside the Boolean (as when the ray passes through a martini glass).

Examples of Compound Objects



Extents for Compound Objects

- During a preprocessing step, the tree for the CSG object is scanned, and extents are built for each node and stored within the node itself.
- Later, during ray tracing, the ray can be tested against each extent encountered with the potential benefit of an early out in the intersection process if it becomes clear the ray cannot hit the object. This can save the cost of combining lists further up the CSG tree.
- Extents are tested as always inside the hit() method of the object's class.
- If the ray hits the extent for this node of the tree, it is intersected with the left and right subtrees in the usual manner.

Box Extents

- How do we define a (world) box extent for a CSG node?
  - A Boolean object doesn't have a generic version.
- Suppose the node represents the object *A op B* where *A* and *B* are shapes and *op* is one of the Boolean operators.
- The box extent for this object must enclose the object, so that we can rest assured the ray misses the object if it misses the box. The shape of the object might be quite complicated, so it will be very difficult to find the tightest aligned box automatically without an inordinate amount of processing (e.g. the intersection of a rotated torus with a skewed cube).

Box Extents (2)

- We take the simplest approach and create the box extent of *L op R* out of the box extent, *E*(*L*), of *L* and the box extent, *E*(*R*), of *R*. We define the box extent differently for the different operators.
- a) **union:** Take as the box extent the aligned box that fits about both *E*(*L*) and *E*(*R*) simultaneously. This is equivalent to $E(E(L) \cup E(R))$
- b) **intersection**: Take as the box extent the intersection of *E*(*L*) and *E*(*R*). The intersection of two aligned boxes is always an aligned box, and it is easy to compute its (*left*, *top*, *right*, *bottom*, *front*, *back*) components. This extent will most likely be reasonably tight if the extents of *L* and *R* are tight.

Box Extents (3)

- c) **difference**: Take as the box extent simply $E(L)$.
- This is very conservative, as we are failing to take advantage of the possibility that ($E(L)$-$E(R)$) may be considerably smaller than $E(L)$-$E(R)$. We may have chopped out a lot of space from $E(L)$.
- But to do a more thorough analysis would be very costly.

Box Extents (4)

- Compound objects have box extents formed recursively because the box extent for an internal node of a Boolean tree would be constructed out of the box extents of its children nodes.
- For shape objects, the box extent would be constructed non-recursively as we discussed above: finding the cloud of points for the object in world coordinates and building an aligned box around it.
- Each type of shape would have its own makeBoxExtent() method. Each method would build the box extent, store it within the object, and return it for use by other objects.

Example Code: UnionBool

```
Cuboid UnionBool :: makeBoxExtent()
{
    Cuboid lft = left->makeBoxExtent();
    Cuboid rt = right->makeBoxExtent();
    Cuboid tmp;
    tmp.left = min(lft.left,rt.left);//form the union
    tmp.top = max(lft.top,rt.top);
  //etc. for the other four values
    worldBoxExtent = tmp; //store it in the object
    return tmp;
}
```

Box Extents (5)

- The preprocessing step would call makeBoxExtent() for each object on the object list.
- When this method is called for a Boolean object, it creates and stores a box extent at the root node of the tree, as well as at each internal node.

Sphere Extents

- World sphere extents would be built in a similar way to box extents. As discussed earlier, for shapes, the cloud of points is found in world coordinates, and the sphere extent is the closest fitting sphere about this cloud.
- For Boolean objects we must decide how to form the sphere extent for a union, difference, and intersection of two objects.

Sphere Extents (2)

- The difference is easy; use the sphere extent for the left object itself. It won't be too tight, but it is serviceable.
- The same choice could be made for intersections, as it is very difficult to compute the true sphere extent for an intersection.
- For unions it is probably simplest to combine the clouds of points for the left and right objects, and then to form a single sphere about it.

Projection Extents

- There is strong motivation to use projection extents for CSG objects, due to the cost of ray tracing them.
- A projection extent can be built for each node of a CSG object in much the same way as we built box extents:
  - The projection extent of a node is formed by a combination of the projection extents of the left and right sub-objects.

Projection Extents (2)

- Using *P*(*object*) to mean the projection extent of *object* we have, according to the rules above,

$$P(L \cup R) = P(P(L) \cup P(R))$$

$$P(L \cap R) = P(L) \cap P(R)$$

$$P(L - R) = P(L)$$

- As before, each class has its own makeProjectionExtent() method (for eye rays only).

Ray Tracing vs. Ray Casting

- Although ray tracing is a method that produces very high quality images of any scene desired, it is rather time consuming and therefore not usually suitable for real-time applications, such as games or simulations.
- Ray-casting is a much faster technique, but it suffers from severe geometric constraints on the scene of interest and the resulting images are blocky and non-realistic.

Ray Tracing vs. Ray Casting (2)

- The figure shows a screenshot from Wolfenstein 3D that illustrates the type of scene which can be ray cast. Notice the blocky and coarse quality of the image and its texture.

Ray Tracing vs. Ray Casting (3)

- Ray-casting has largely fallen out of favor in recent years as innovations in graphics hardware have made some of the limitations of ray-tracing unimportant.
- The companion web site links to a fine tutorial on ray-casting by F. Permadi.