

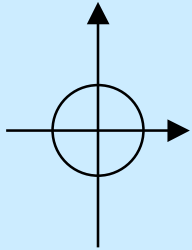
# **Computer Graphics: Modeling Techniques**

**Dr. Michael Laszlo  
Nova Southeastern University  
School of Computer and Information Sciences  
2001**

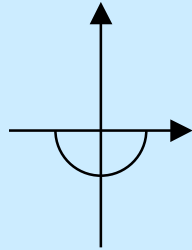
# Object Hierarchies

- An *object hierarchy* is a tree structure used to model and manipulate scenes in two and three dimensions.
- Each leaf node corresponds to a primitive object such as a circle, rectangle, or polygon in 2D, or a sphere, ruled cylinder, or Platonic solid in 3D. Each primitive is defined in its own local coordinate system.
- Each internal node defines a local coordinate system in which its child nodes are placed. The local coordinate system is described by a *local matrix* (LM) which maps the child from the child's coordinate system into its parent's coordinate system.

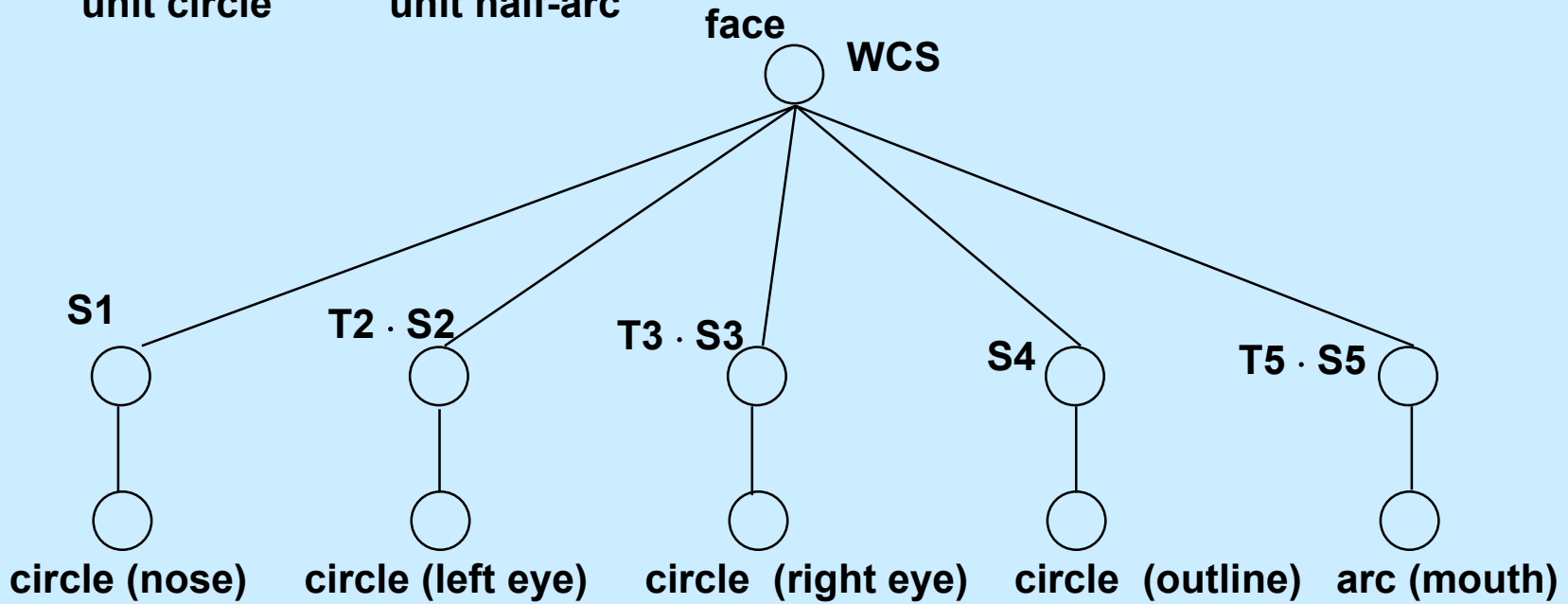
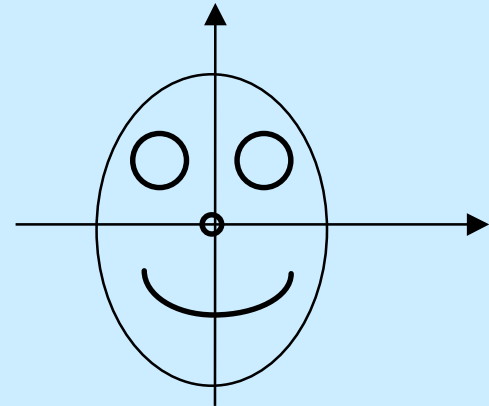
# Object Hierarchies



**primitive:  
unit circle**

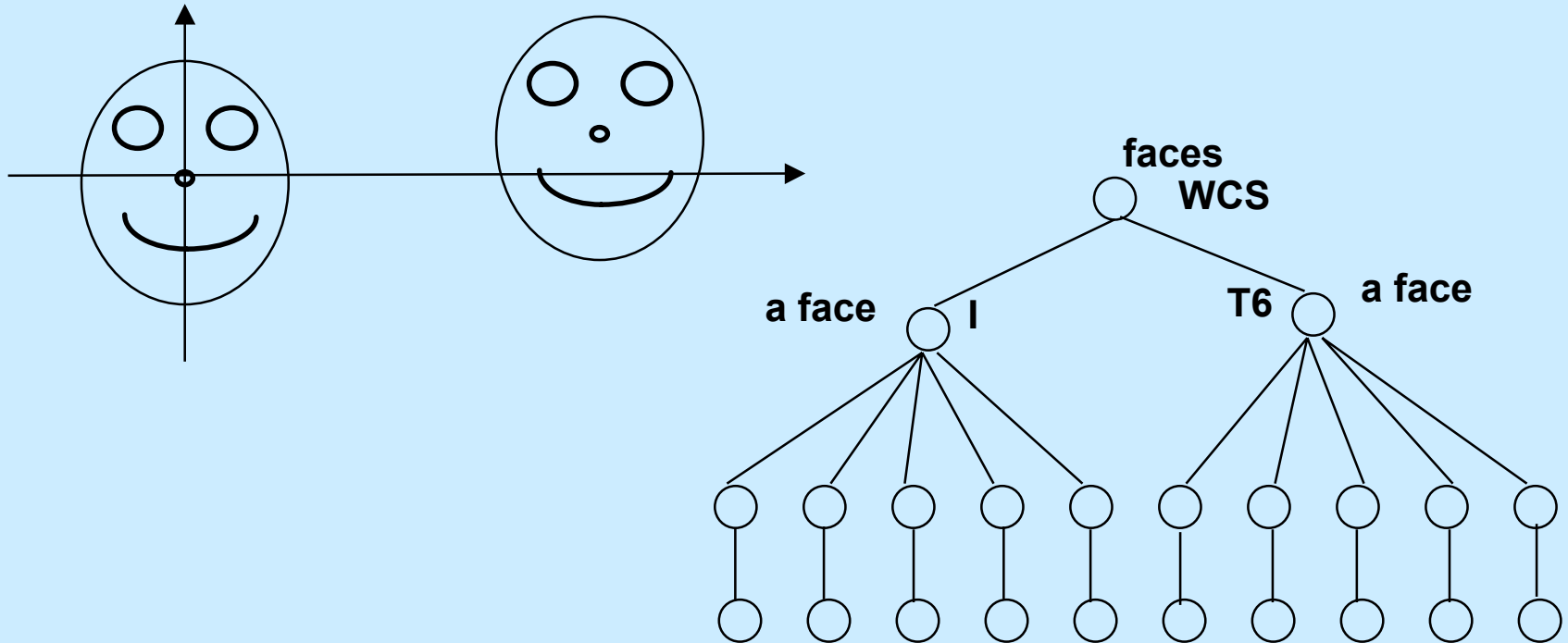


**primitive:  
unit half-arc**



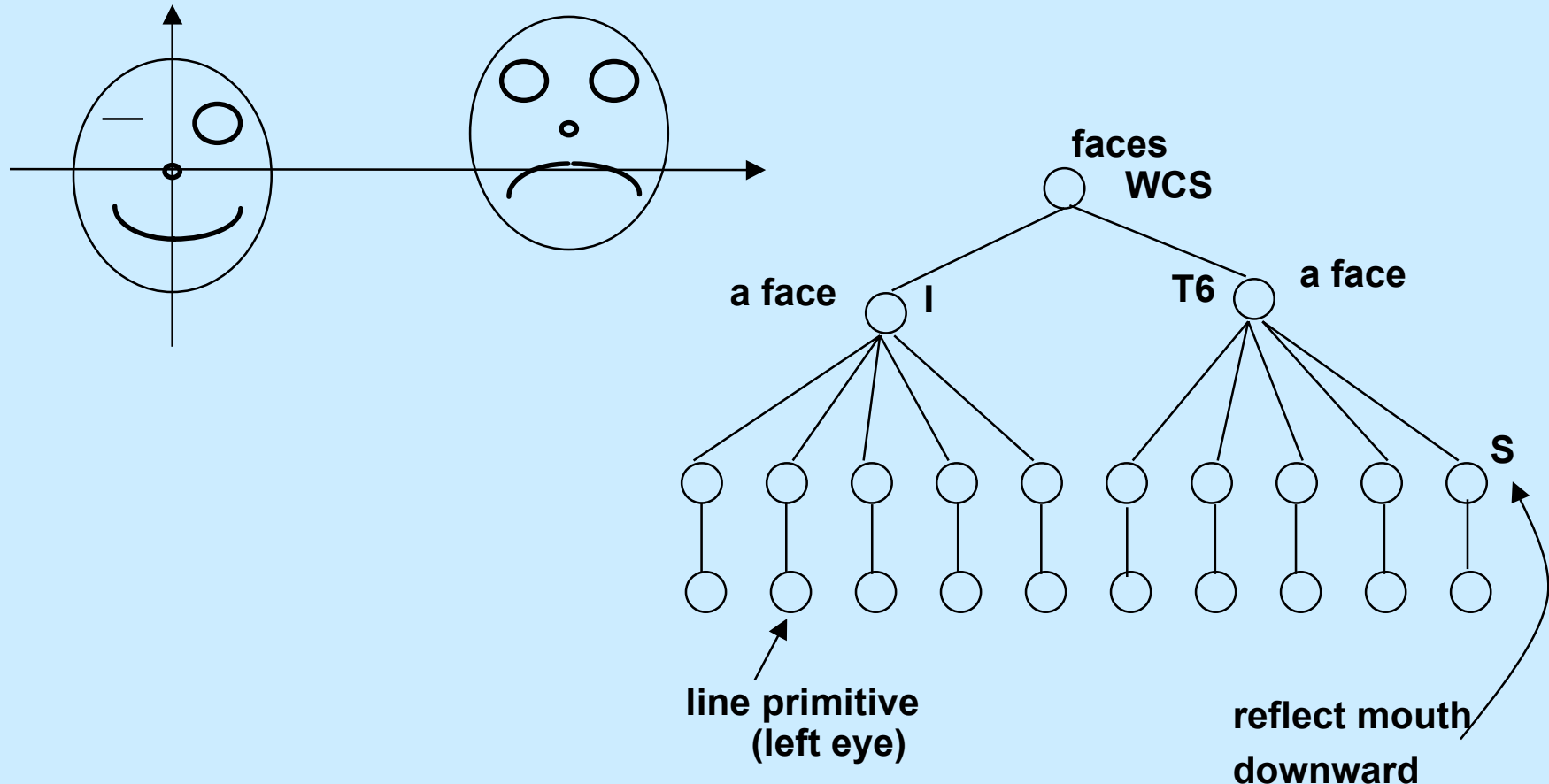
# Object Hierarchies

- Object hierarchies may extend to any depth. The following represents two faces by a 3-level hierarchy.



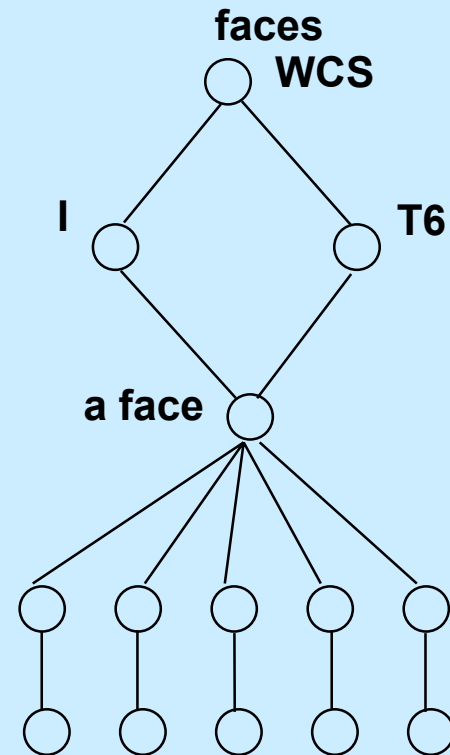
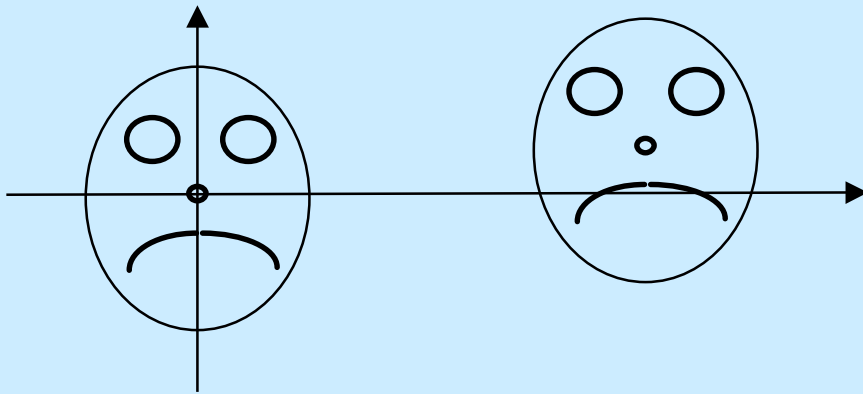
# Object Hierarchies

- If the hierarchy is maintained as a tree, we can modify one object group or instance without affecting any others.



# Object Hierarchies

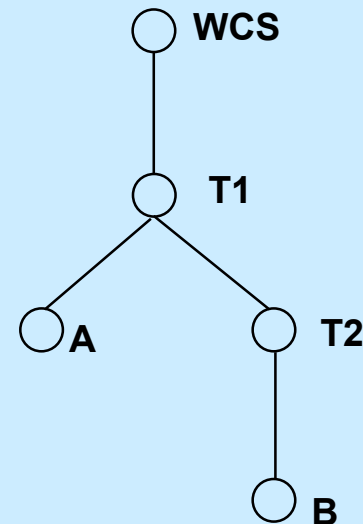
- If directed acyclic graphs (DAGs) are supported, the same object group can be referenced any number of times. Modifying the object group modifies all of its instances.



# Object Hierarchies in VRML

- Object hierarchies are built in VRML using nested *Transform* nodes. Each *Transform* node represents a local coordinate system defined in terms of its parent's coordinate system. A *Transform* node's children are located in the *Transform* node's coordinate system. The hierarchy's root represents the world coordinate system (WCS).

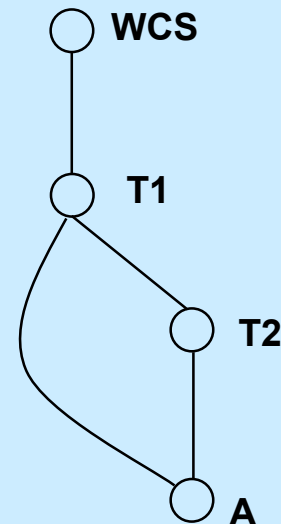
```
DEF T1 Transform {  
  translation 4 0 0  
  children [  
    DEF A Shape {  
      geometry Sphere {}  
    },  
    DEF T2 Transform {  
      rotation 1 0 0 0.78  
      children DEF B Shape {  
        geometry Box {}  
      }  
    }  
  ]  
}
```



# Object Hierarchies in VRML

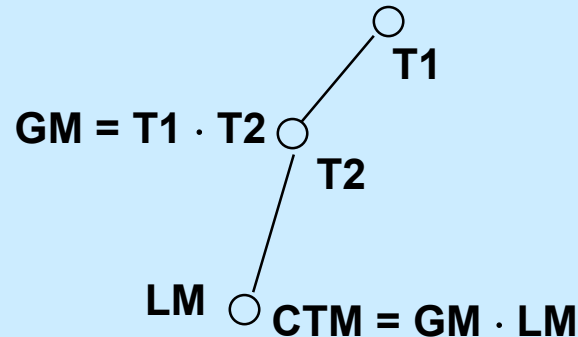
- When a node that belongs to the scene-graph is named (using *DEF*) and later referenced by name (using *USE*), multiple references to the same node are created. In this case, a directed acyclic graph (DAG) results.

```
• DEF T1 Transform {  
  translation 4 0 0  
  children [  
    DEF A Shape {  
      geometry Sphere {}  
    },  
    DEF T2 Transform {  
      rotation 1 0 0 1.57  
      children Shape USE A  
    }  
  ]  
}
```



# Display Traversal of Object Hierarchies

- At each node  $n$  of the tree or DAG, the *global matrix* ( $GM$ ) is the composition of transformation matrices extending from the root node down to  $n$ 's parent.
- The *local matrix* ( $LM$ ) at node  $n$  is the transformation matrix that defines node  $n$ 's coordinate system in terms of its parents.
- The *current transformation matrix* ( $CTM$ ) at node  $n$  is the composition of the  $LM$  and  $GM$ ; that is,  $CTM = GM \cdot LM$ .

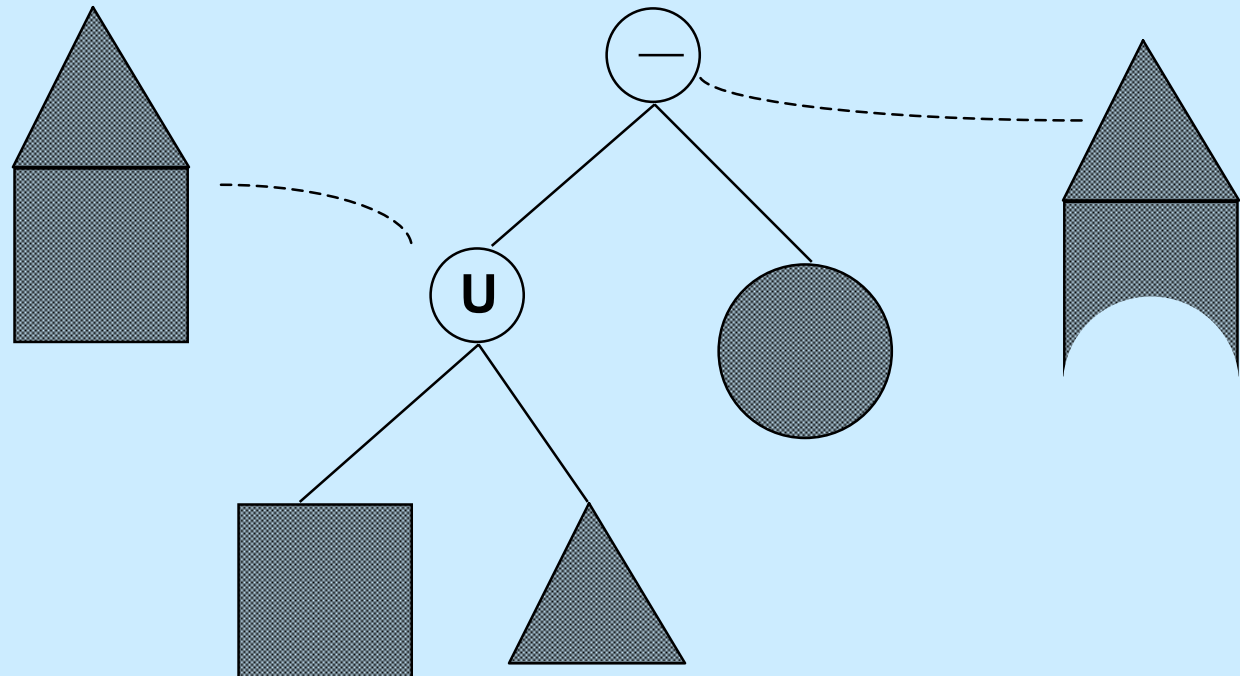


# Display Traversal of Object Hierarchies

- Display traversal is accomplished by a traversal of the tree or DAG. During the traversal, we maintain a stack  $S$  of the CTMs from the current node  $n$  up to the root of the tree. Node  $n$ 's CTM is at the top of stack  $S$ . Initially, stack  $S$  contains only the identity matrix.
- Consider what happens when we visit child  $p$  of node  $n$ . The matrix at the top of the stack is  $p$ 's GM. We compose this GM with  $p$ 's LM, thereby producing node  $p$ 's CTM. Then we push  $p$ 's CTM onto stack  $S$ .
  - If  $p$  is a leaf node, we draw the primitive it represents, transforming it by the CTM at the top of stack  $S$ .
  - If  $p$  is an internal node, we continue our recursive traversal down toward  $p$ 's children.
- When we return from node  $p$  up to its parent  $n$ , we pop the top matrix from stack  $S$ . Now node  $n$ 's CTM is at the top of the stack, as desired.

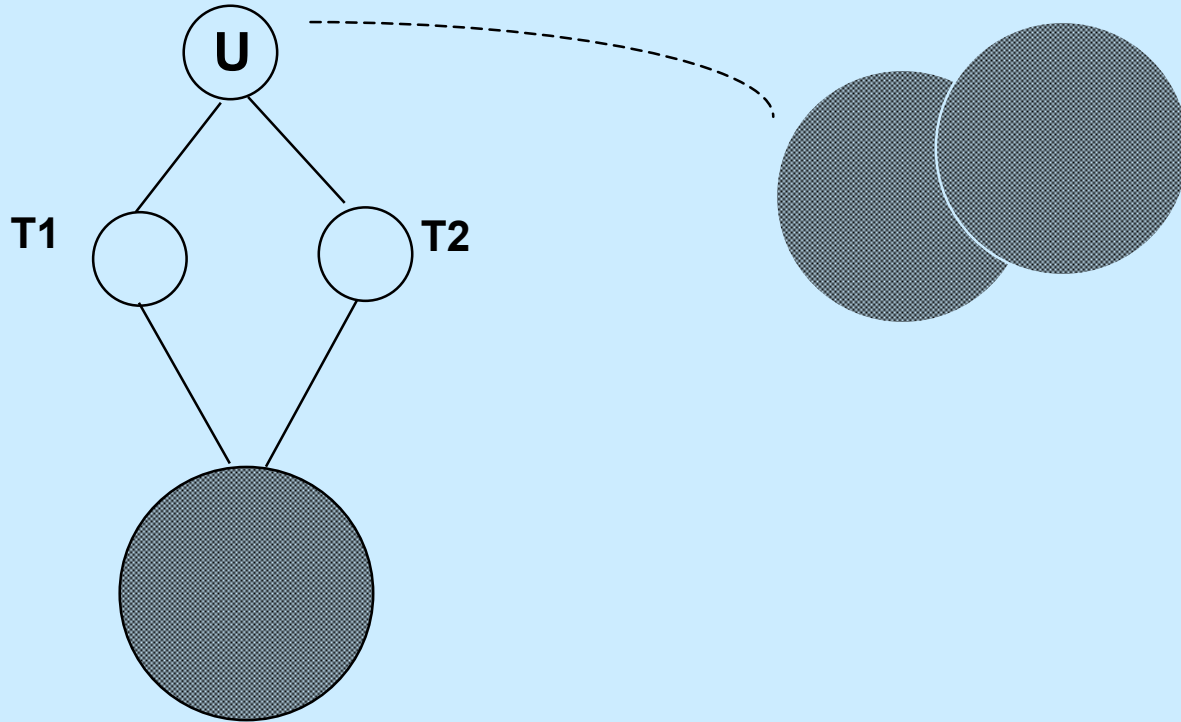
# Constructive Solid Geometry

- *Constructive solid geometry (CSG)* combines simple (usually convex) primitives by Boolean set operations. The result is a *CSG tree*.
- Each external node of the *CSG tree* corresponds to a primitive.
- Each internal node represents a Boolean set operation: union, Intersection, or difference. The node corresponds to the object formed by applying the set operation to the objects represented by its left and right subtree.



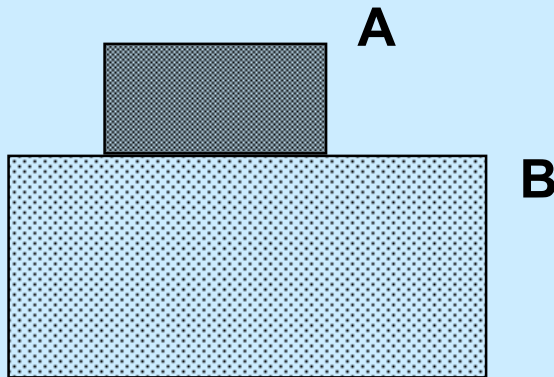
# Constructive Solid Geometry

- We can also include nodes representing coordinate systems within the *CSG* tree. *CSG* trees can also be directed acyclic graphs. Here *T1* and *T2* are translations



# Constructive Solid Geometry

- *CSG* schemes often use *regularized operators* to eliminate irregular results. This ensures that every result is of pure dimension: either the highest dimension or null.
  - Given set  $S$ , we define  $r(S) = \text{closure}(\text{interior}(S))$ .
  - Given sets  $A$  and  $B$ , we define  $A \text{ op}^* B = r(A \text{ op} B)$ .
  - The Boolean set operations are then  $\cup^*$ ,  $\cap^*$ , and  $-^*$ .



$$A \cap B = \text{—————}$$

$$A \cap^* B = \text{poof (null)}$$

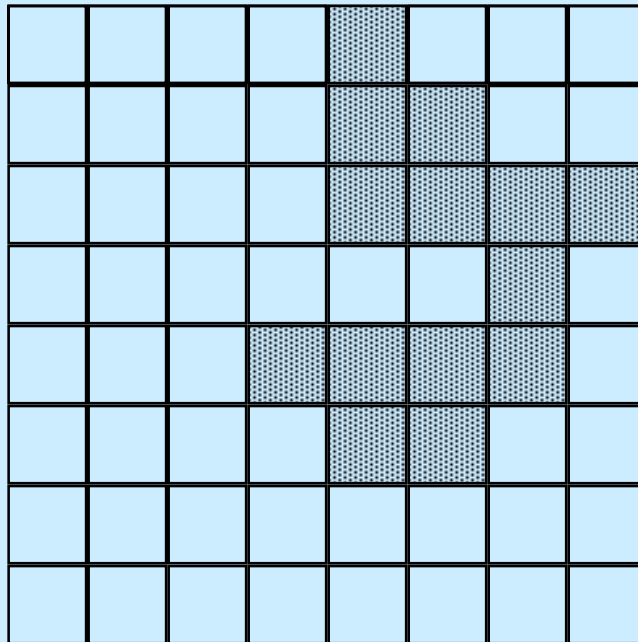
# Constructive Solid Geometry

- There are a number of operations CSG trees support. In particular, they efficiently support ray intersection methods, one reason CSG trees are so popular for ray tracing. We will look at this in detail later in the course.
- Here is a recursive algorithm for *point classification*: given a point  $P$ , decide whether  $P$  lies in the object represented by a CSG tree. Here  $\text{Object}(T)$  returns the primitive represented by  $T$  where  $T$  is a leaf node, and  $\text{Operator}(T)$  returns the Boolean set operator associated with  $T$  where  $T$  is an internal node.

```
// returns TRUE just if P lies in object given by T
int ClassifyPoint(Point P, CSGTree T)
{
    if (isLeaf(T))
        return PrimitiveClassify(P, Object(T));
    else
        return apply(operator(T), ClassifyPoint(P, lchild(T)),
                    ClassifyPoint(P, rchild(T)));
}
```

# Spatial Enumeration

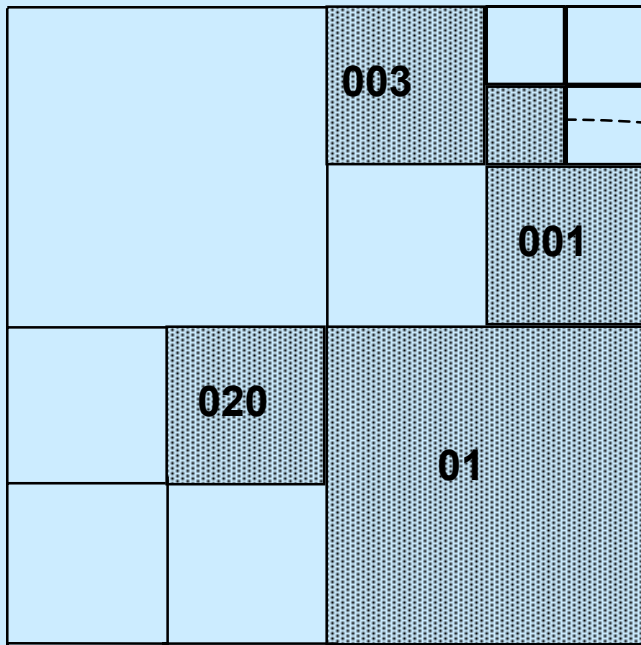
- This method represents an object in a 2D (or 3D) Boolean array each element of which corresponds to a cell in space. A bit in each element indicates whether the cell lies in the object. If the array holds integers or records instead of bits, it is possible to distinguish between multiple objects.



# Quadtrees

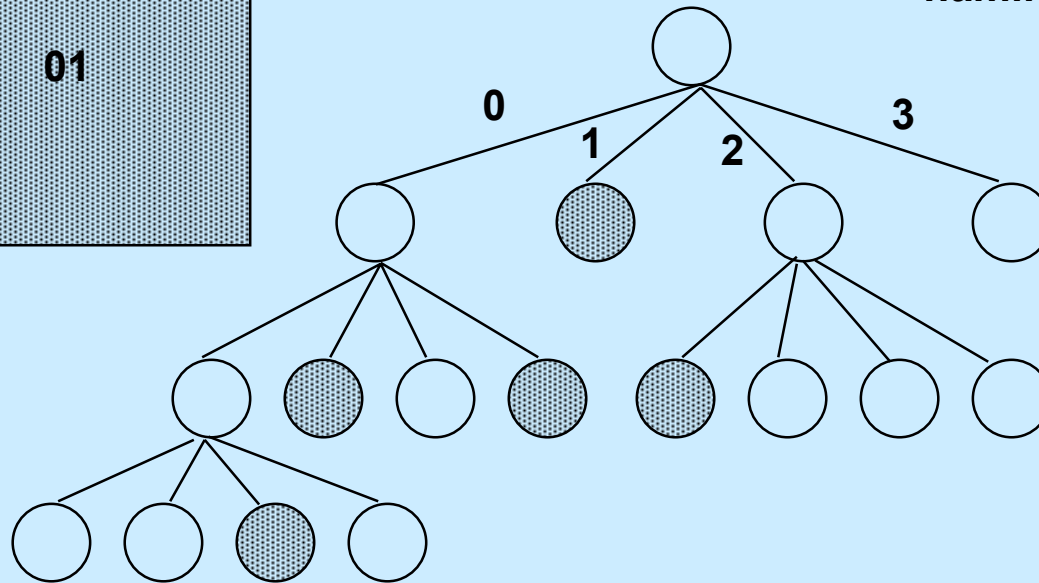
- A *quadtree subdivision* is a recursive subdivision of a square. At each division, two *cutlines*, one vertical and the other horizontal, cross at the center of the square, subdividing it into four congruent quadrants. The quadtree subdivision is represented by a four-way branching tree.
- Every node of the quadtree corresponds to a quadrant of the subdivision. Each leaf node contains a bit indicating whether its quadrant lies in the object. Each internal node corresponds to a quadrant that is *heterogeneous* (partly belonging to the object and partly not).
- The advantage of a quadtree over spatial enumeration is that large homogeneous areas (which lie wholly inside or outside the object of interest) can be represented compactly. Later in the course we will see other uses for the quadtree, stemming from the fact that its shape is sensitive to the distribution of objects in space.
- The 3D version of a quadtree is called an *octree*.

# Quadtrees



3	0
2	1

naming scheme



To name a quadrant, start with 0, then concatenate the edge labels leading from the root node down to the quadrant's node.

# Quadtrees

- **As an example of the power of this recursive data structure, let us again consider *point classification*: given a point  $P$ , decide whether  $P$  lies in the object represented by a given quadtree  $T$ . In the following code, `Contains(P, Q)` returns `TRUE` just if point  $P$  lies in quadrant  $Q$ .**

```
// returns TRUE iff P lies in object represented by T
int ClassifyPoint(Point P, Quadtree T)
{
    if (not Contains(P, Quadrant(T))
        return 0; // false
    else if (isLeaf(T))
        return 1; // true
    else
        return Or(ClassifyPoint(P, child(T, 0)),
                  ClassifyPoint(P, child(T, 1)),
                  ClassifyPoint(P, child(T, 2)),
                  ClassifyPoint(P, child(T, 3)));
}
```