CS-C3100 Computer Graphics

5.1 Hierarchical Modeling

Jaakko Lehtinen

with many slides from
Frédo Durand and Barb Cutler

In These Slides

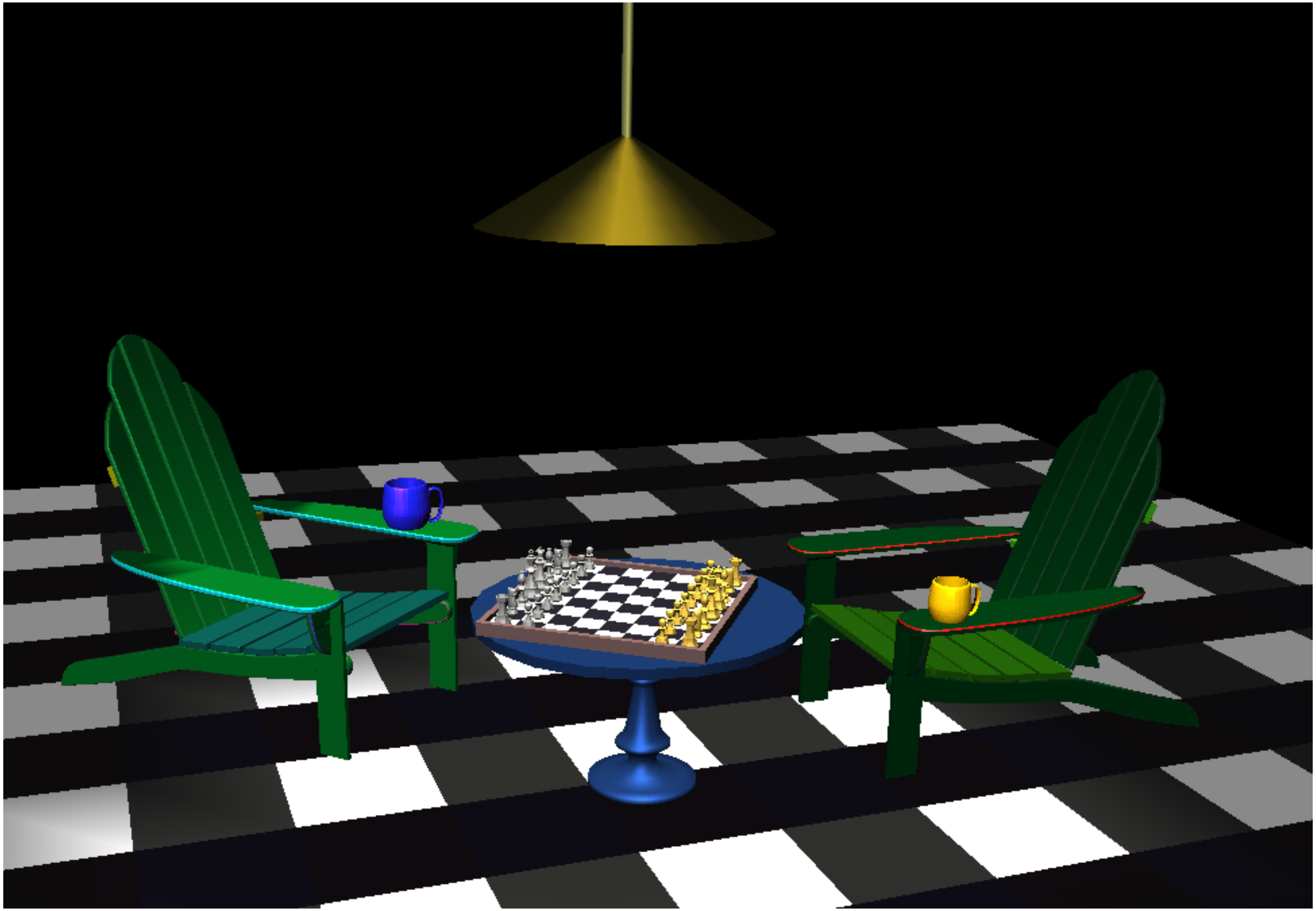
- Why object hierarchies are useful
- The Scene Graph
  - representing scenes by directed acyclic graphs (DAG)
  - traversing the scene graph

Hierarchical Modeling

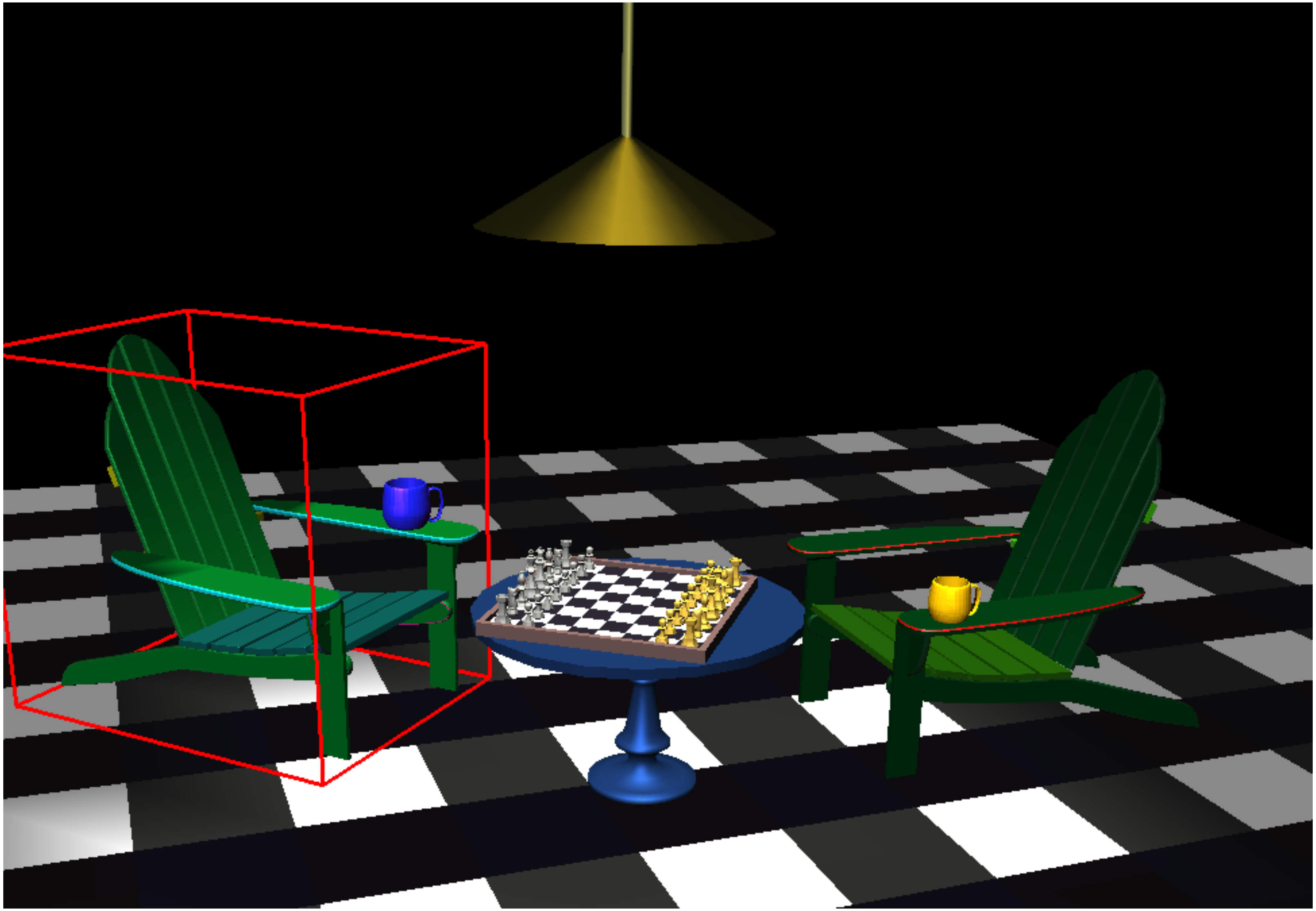
- Triangles, parametric curves and surfaces are the building blocks for more complex objects

- Hierarchical modeling creates complex real-world objects by combining simple primitive shapes into aggregate objects.

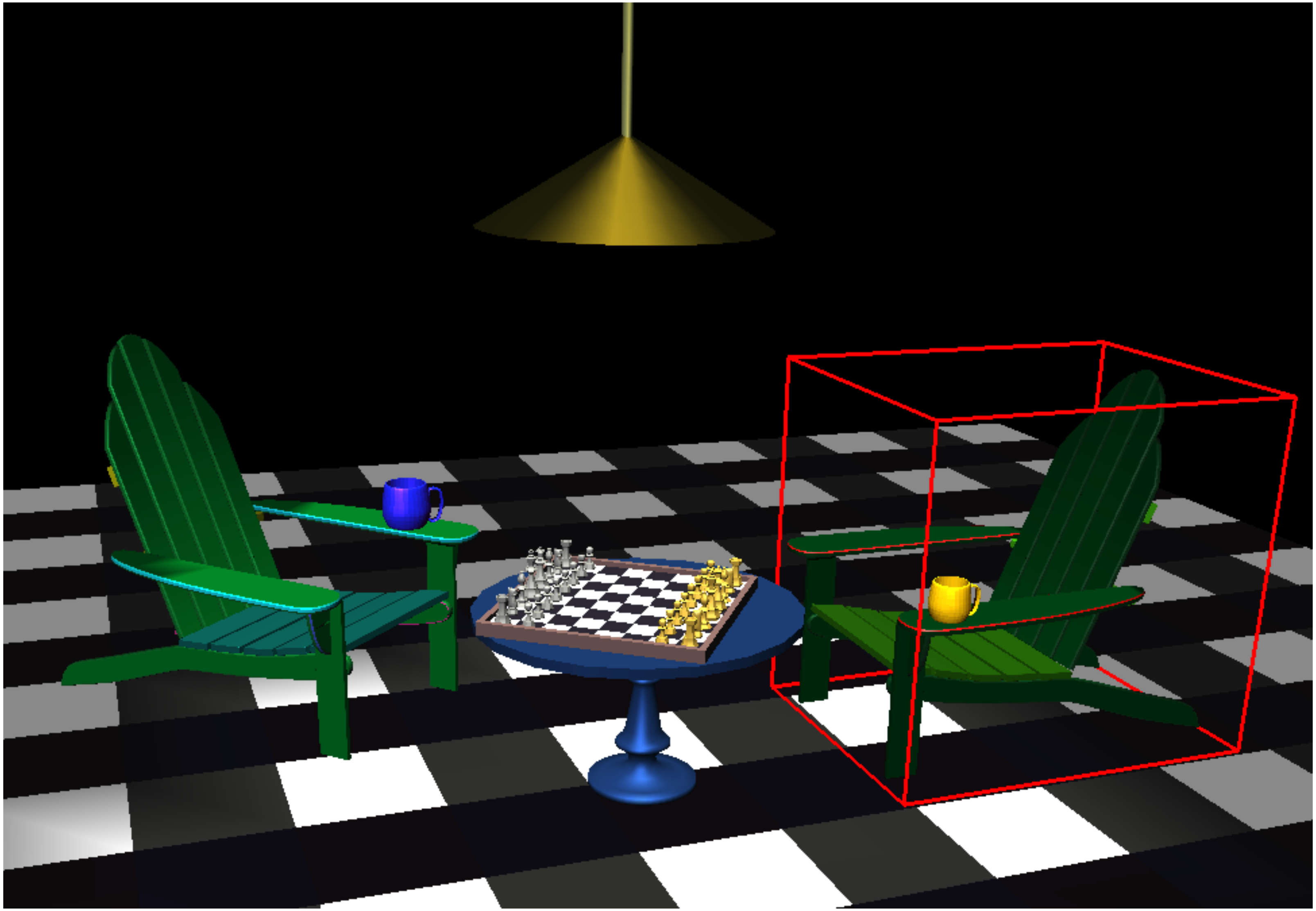
Hierarchical models



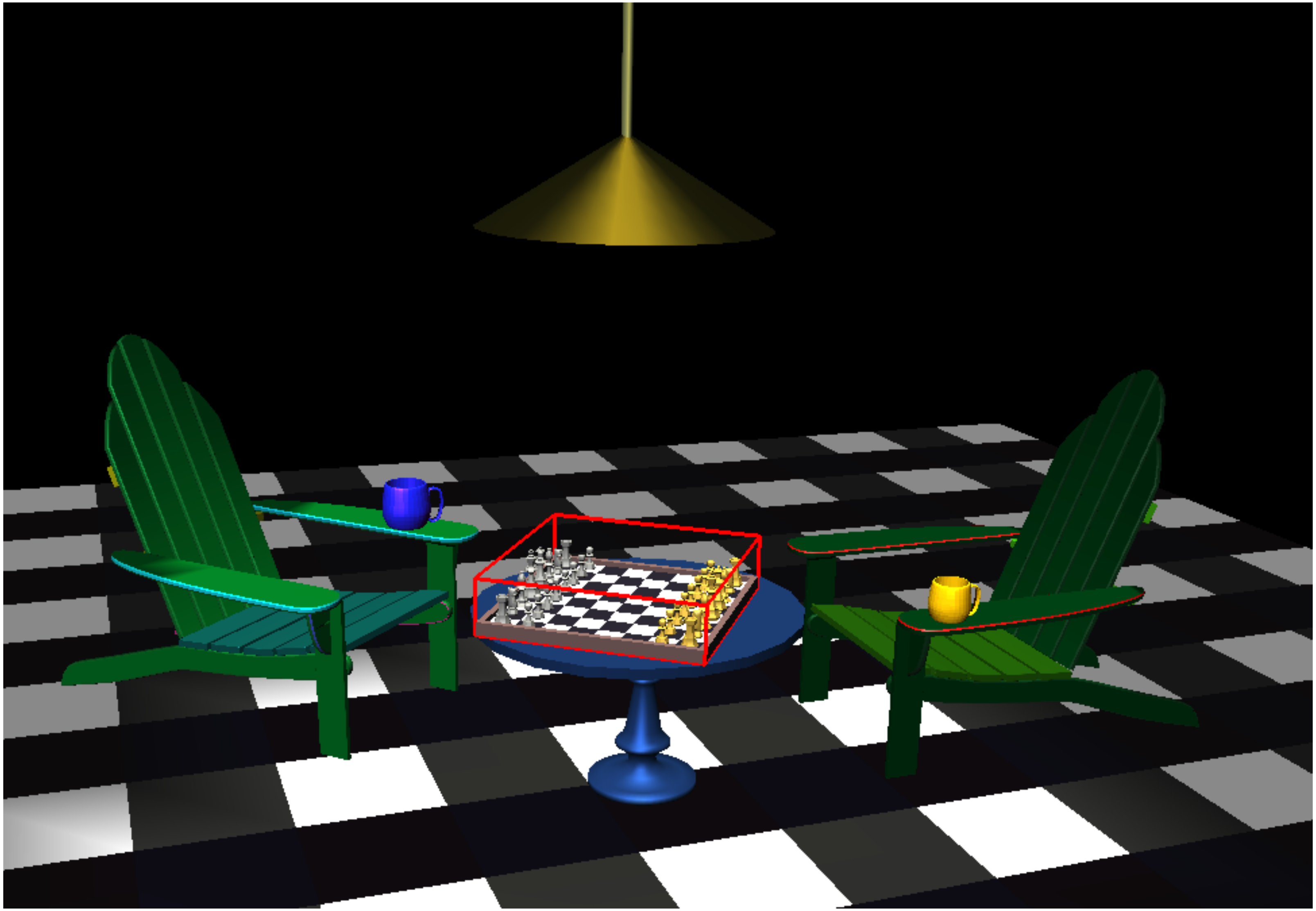
Hierarchical models



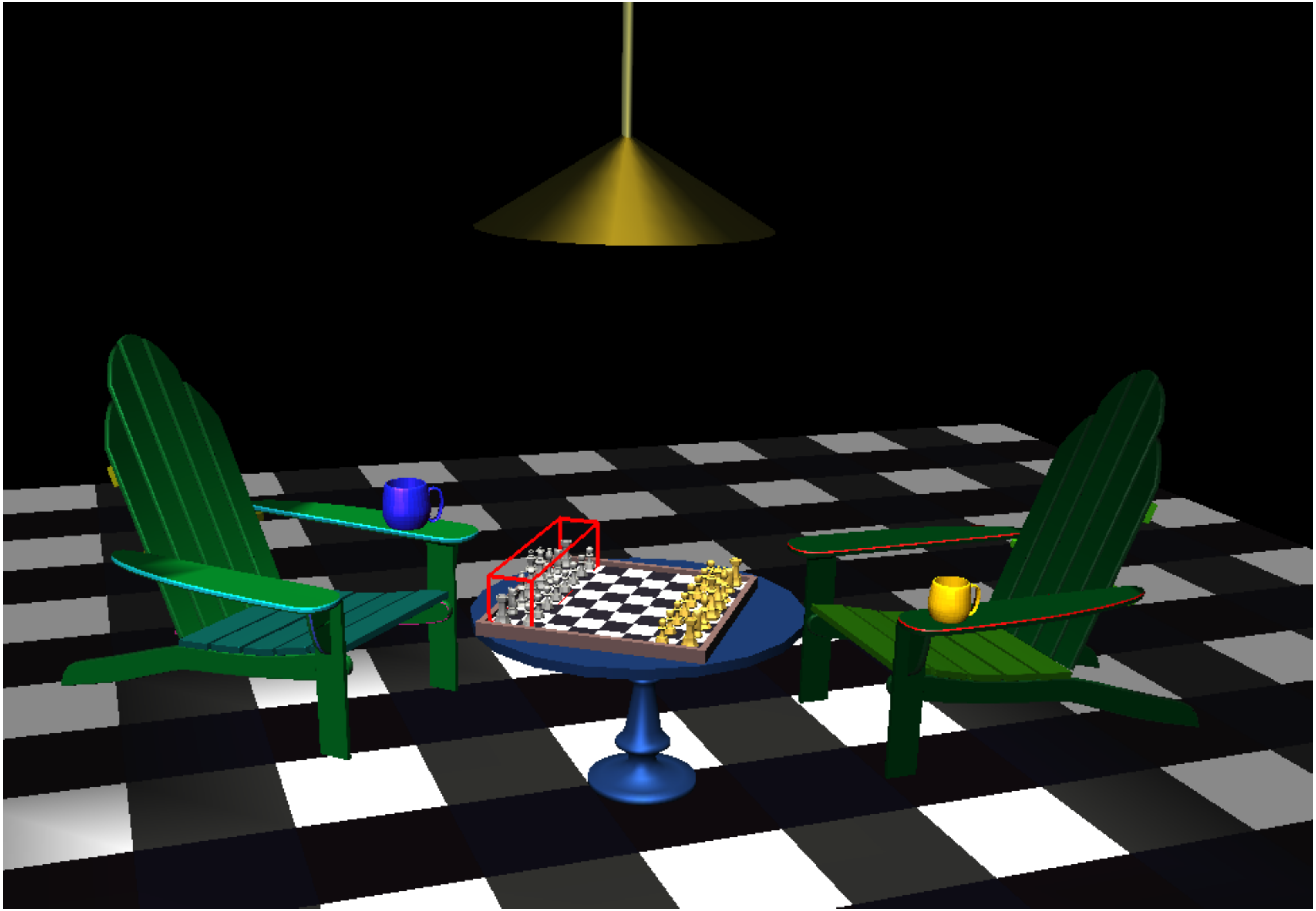
Hierarchical models



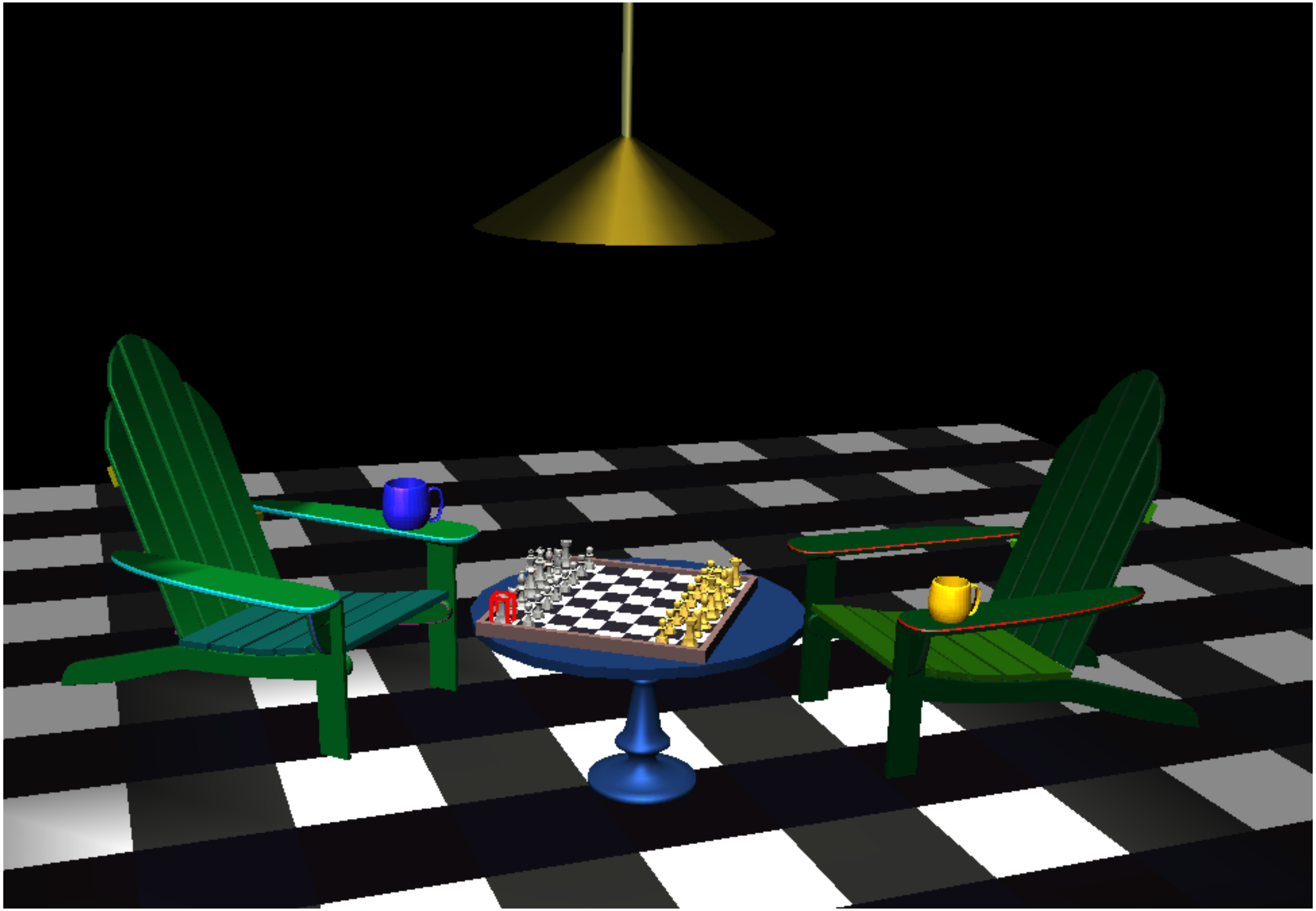
Hierarchical models



Hierarchical models

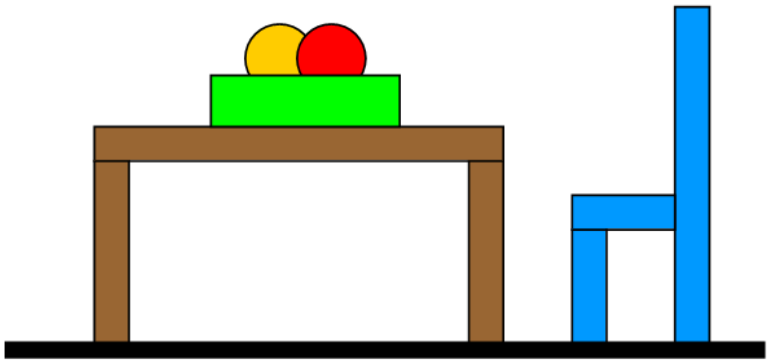


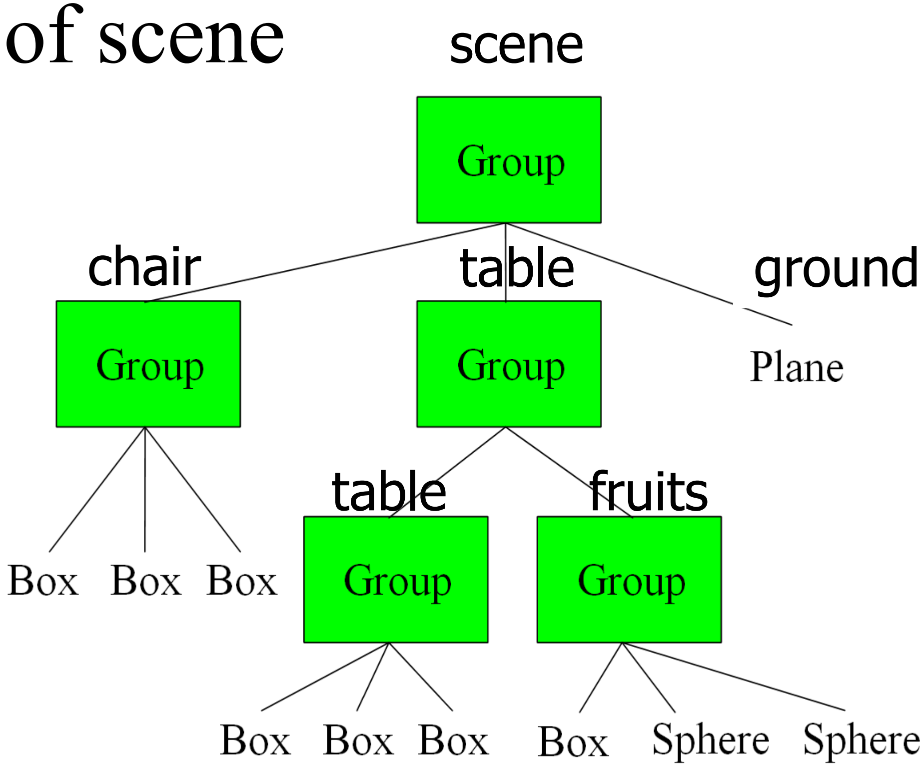
Hierarchical models



Hierarchical Grouping of Objects

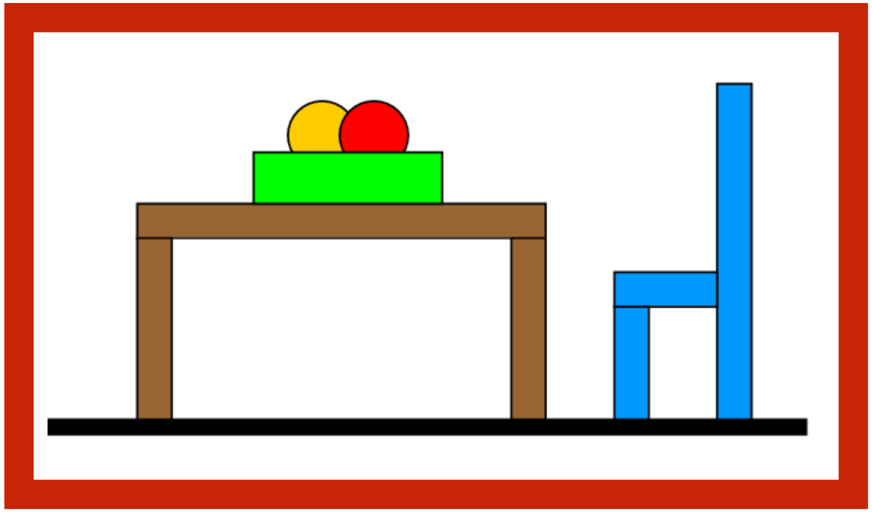
The "scene graph" represents
the logical organization of scene

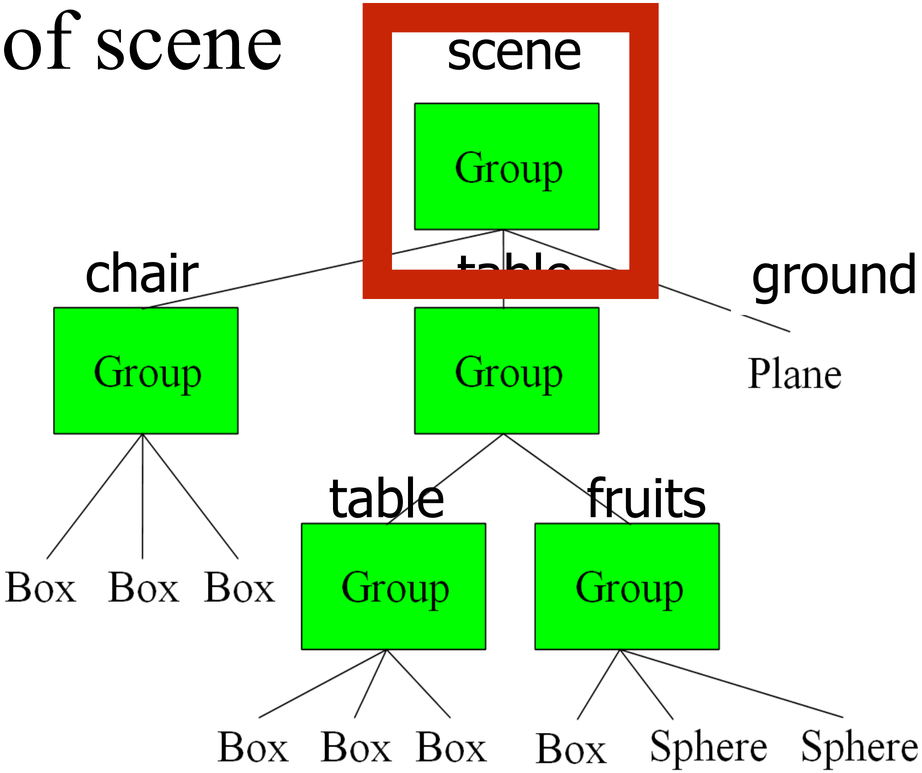




Hierarchical Grouping of Objects

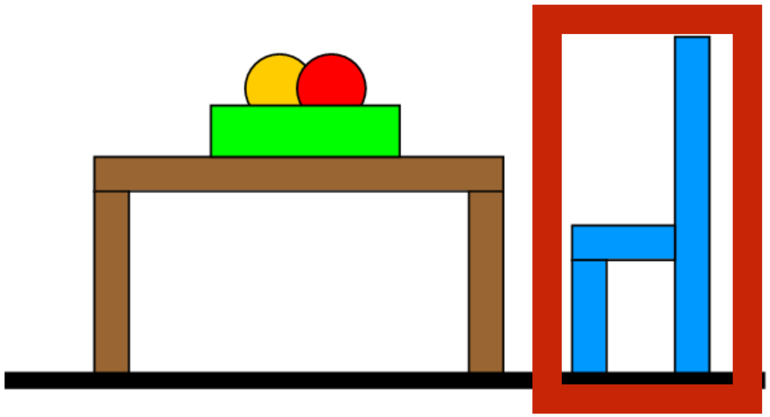
The "scene graph" represents
the logical organization of scene

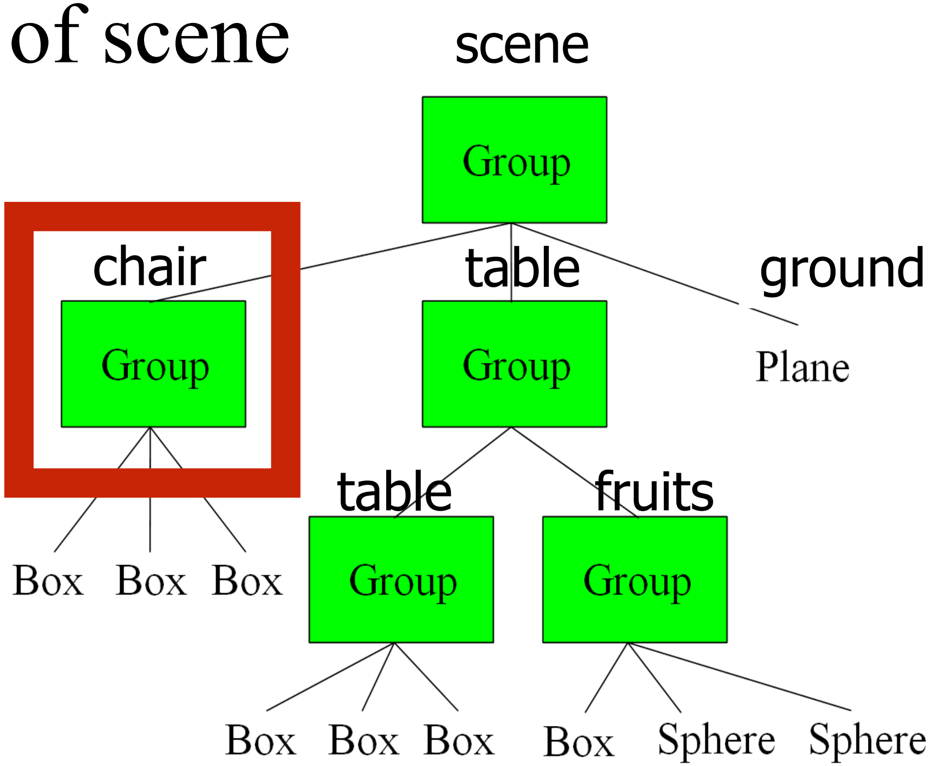




Hierarchical Grouping of Objects

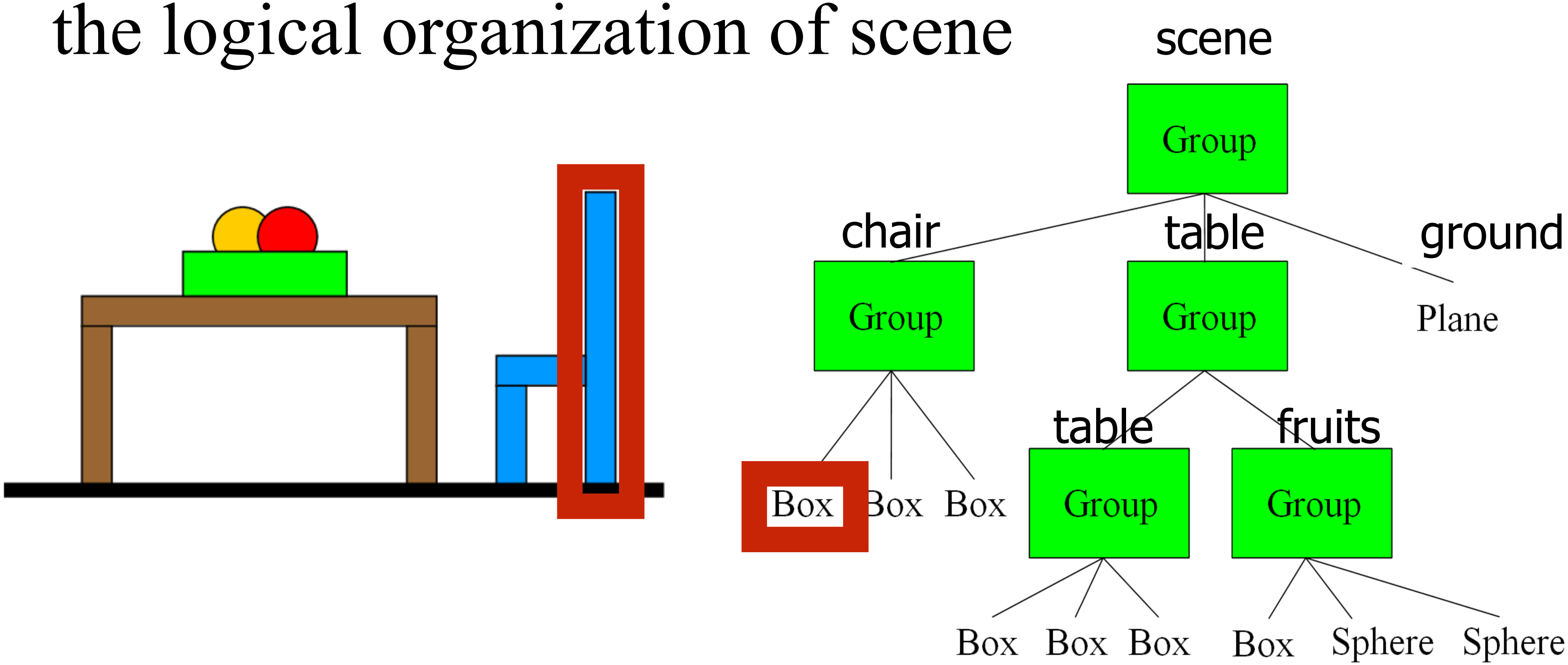
The "scene graph" represents
the logical organization of scene





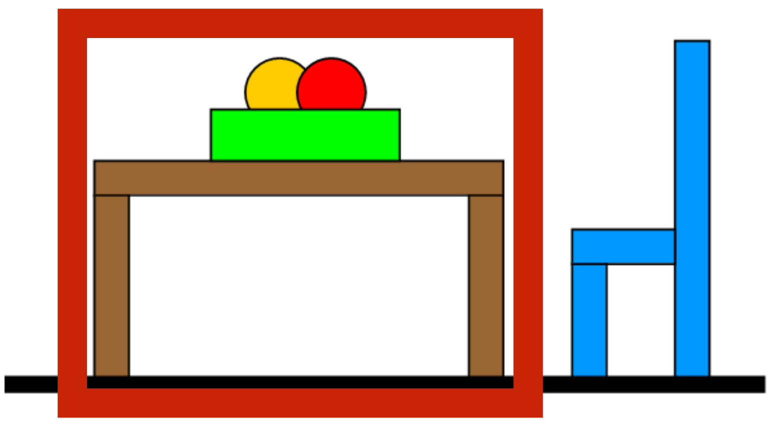
Hierarchical Grouping of Objects

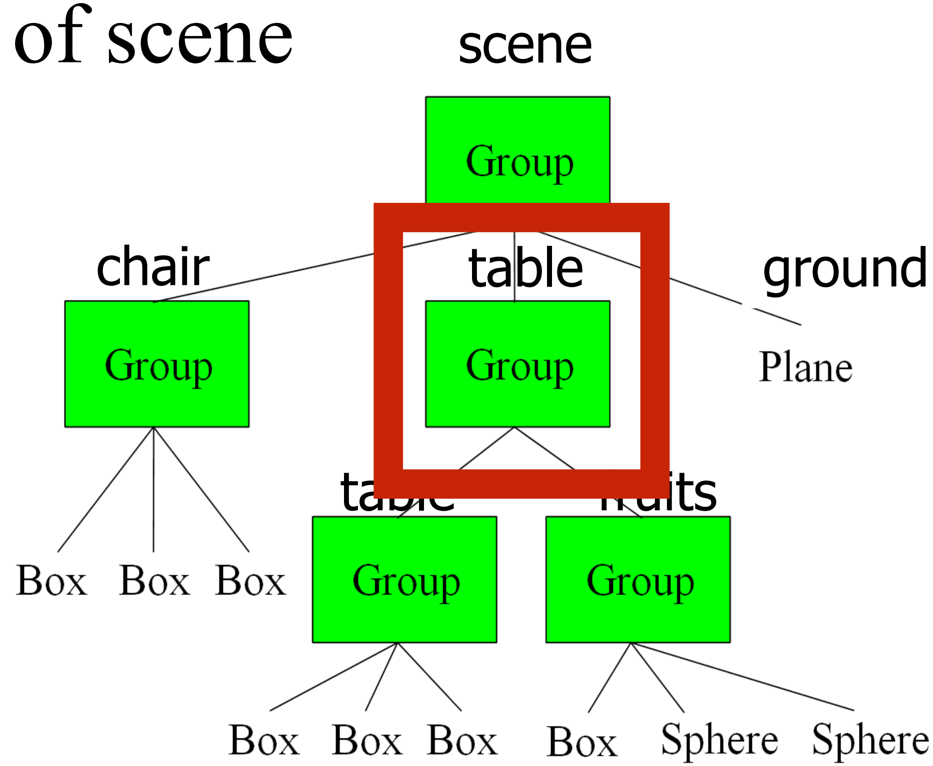
The "scene graph" represents
the logical organization of scene



Hierarchical Grouping of Objects

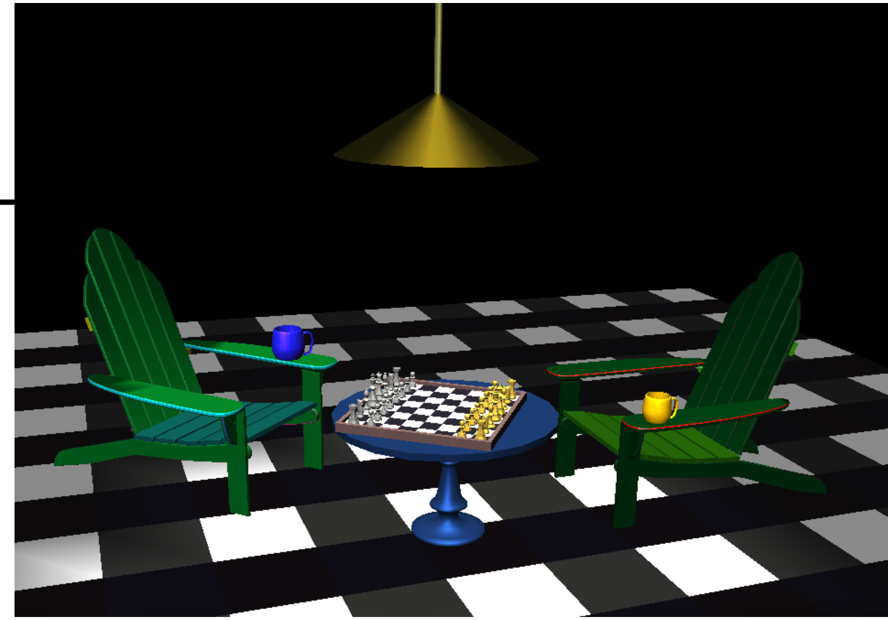
The "scene graph" represents
the logical organization of scene



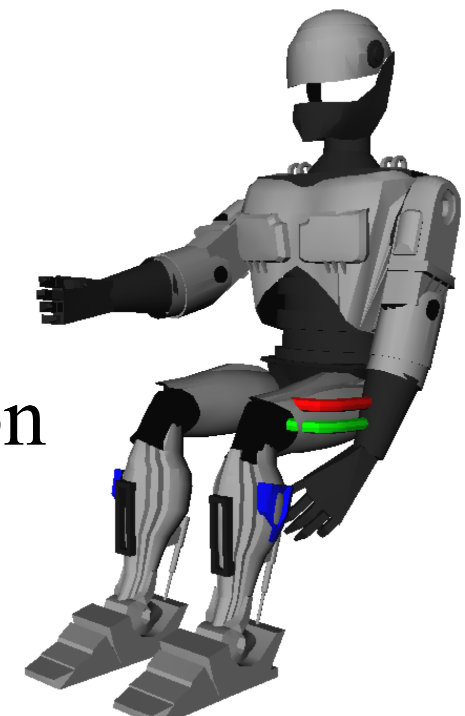


… and so on

Scene Graph

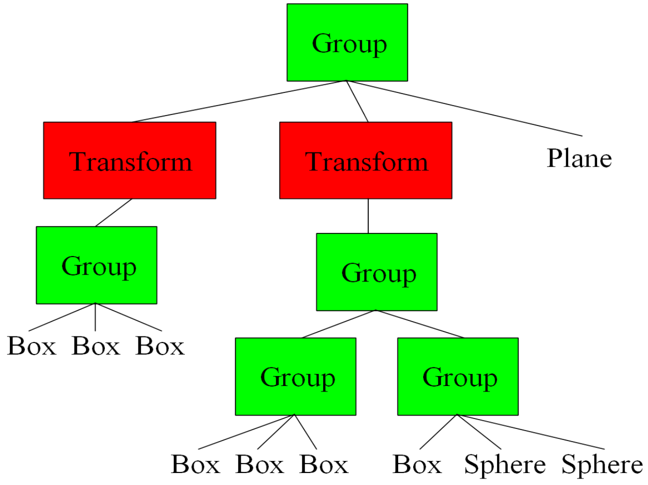


- Data structure
  for scene representation
  - **Geometry (meshes, etc.)**
  - **Transformations**
  - Materials, color
  - Multiple instances
- Basic idea: Hierarchical graph
- Useful for manipulation/animation
- And for rendering
  - Ray tracing acceleration,
    occlusion culling



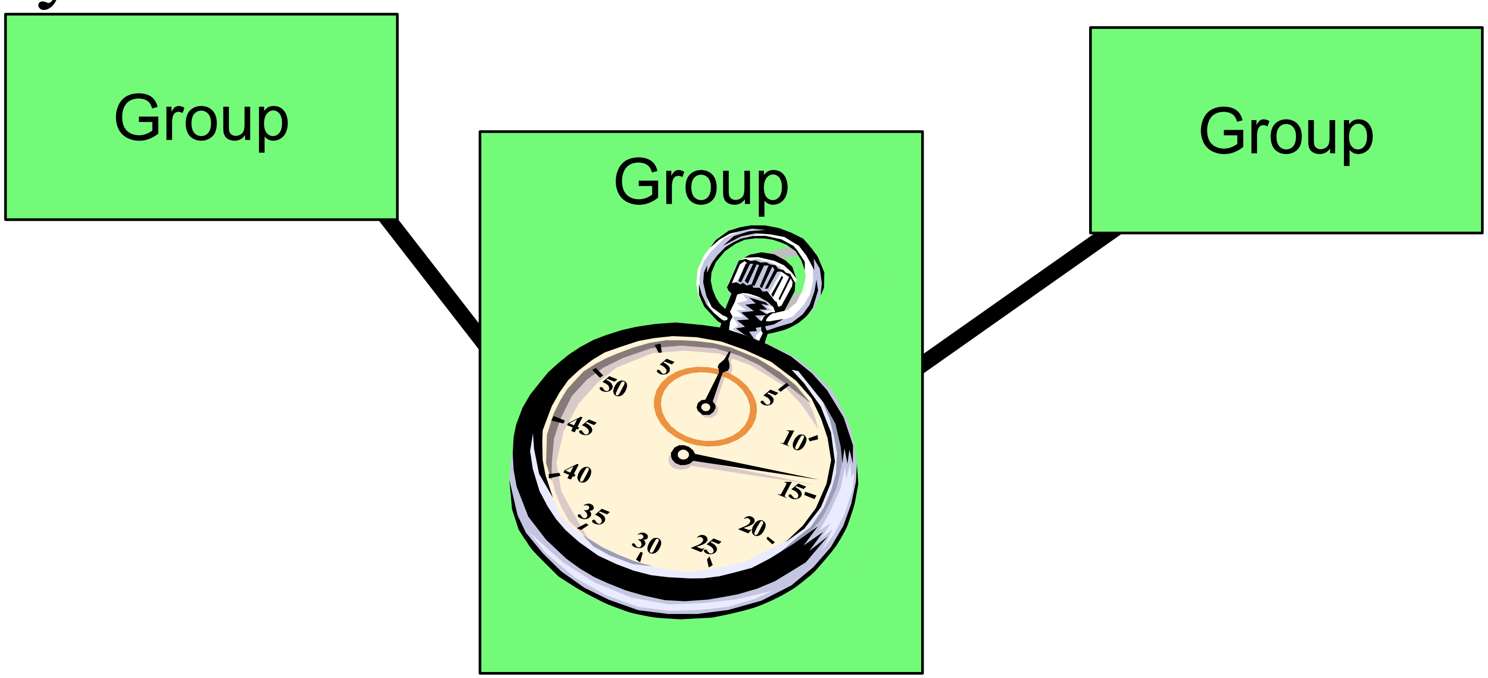
Scene Graph Representation

- Basic idea: Tree
- Comprised of several node types
  - Shape: 3D geometric objects
  - Transform: Affect current transformation
  - Property: Color, texture, transparency, etc.
  - Group: Collection of subgraphs
- C++ implementation
  - base class Object
    - child list (no parent!)
  - derived classes for each node type (group, geometry, etc.)



Scene Graph Representation

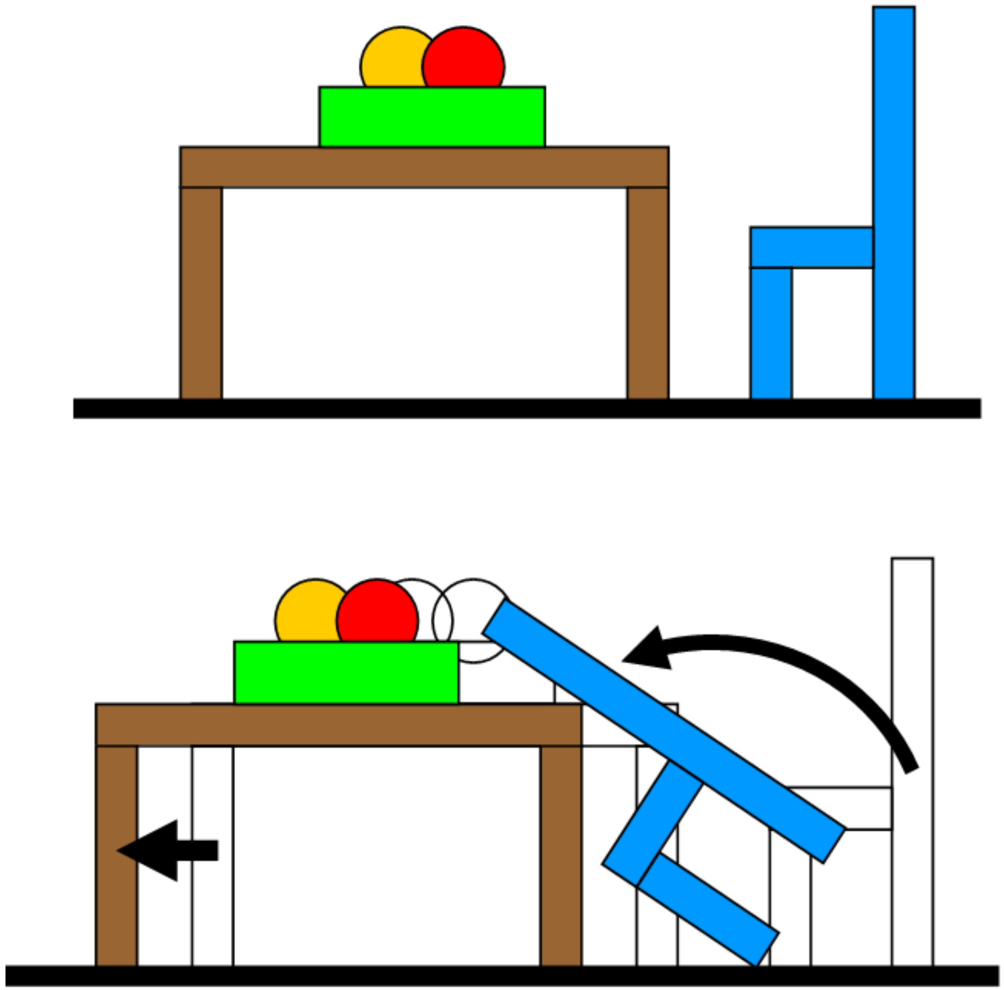
- In fact, generalization of a tree:
  Directed Acyclic Graph (DAG)
  – Means a node can have multiple parents, but cycles are not allowed
- Why?



Scene Graph Representation

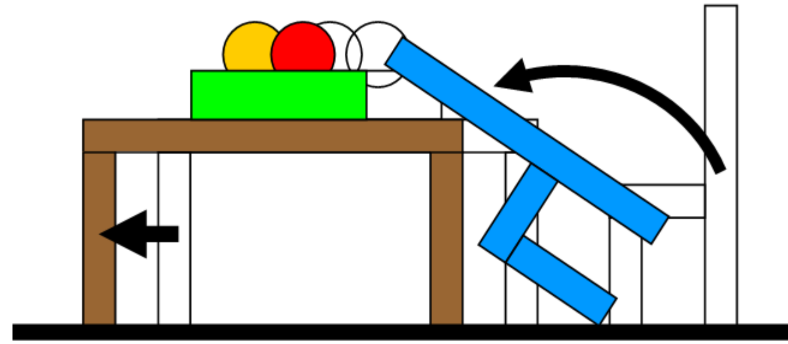
- In fact, generalization of a tree:
  Directed Acyclic Graph (DAG)
  - Means a node can have multiple parents, but cycles are not allowed
- Why? Allows *multiple instantiations*
  - "Several copies of the same object in different locations and orientations"
  - Reuse complex hierarchies many times in the scene using different transformations & other properties

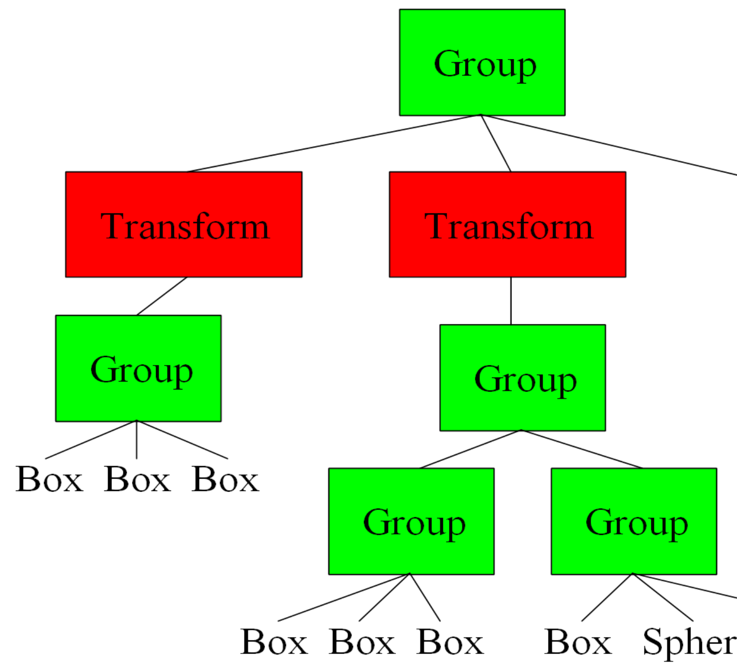
Adding Transformations



Hierarchical Transformation of

- A "transformation node" affects the whole subtree
- Each node has its local coordinate system
- Transformations are always specified relative to parent!
- Aggregate object-to-world transform is the concatenation of all transforms on the way from current node to root





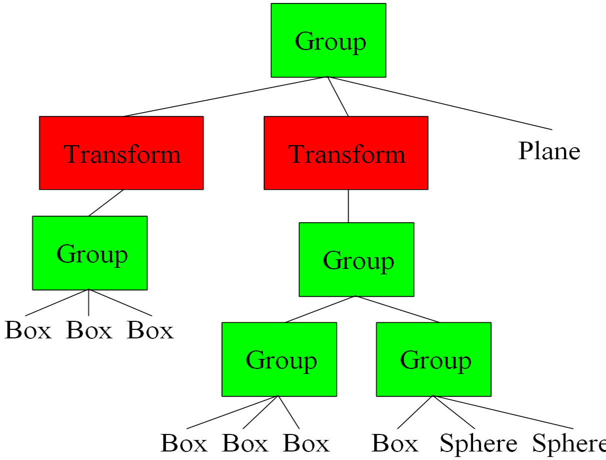
Sidenote

- **In practice**, we most often specify a transformation for **all nodes** and don’t explicitly use special “transformation nodes”
  - Concretely: the “Node” base class contains a Mat4f!
  - It’s the UI’s job to manage that so that it’s intuitive
  - But only do this once you’ve wrapped your head around the simpler concept

Scene Graph Traversal

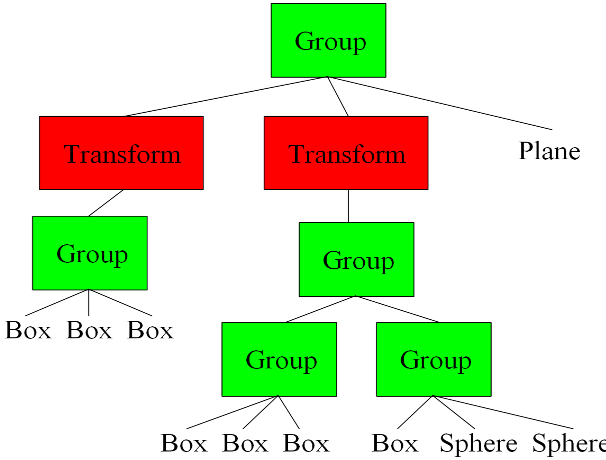
- Depth first recursion
  - Visit node, then subtrees (top to bottom, left to right)
  - When visiting a geometry node: Draw it!

- How to handle transformations?
  - Transformations always specified in coordinate system of the parent

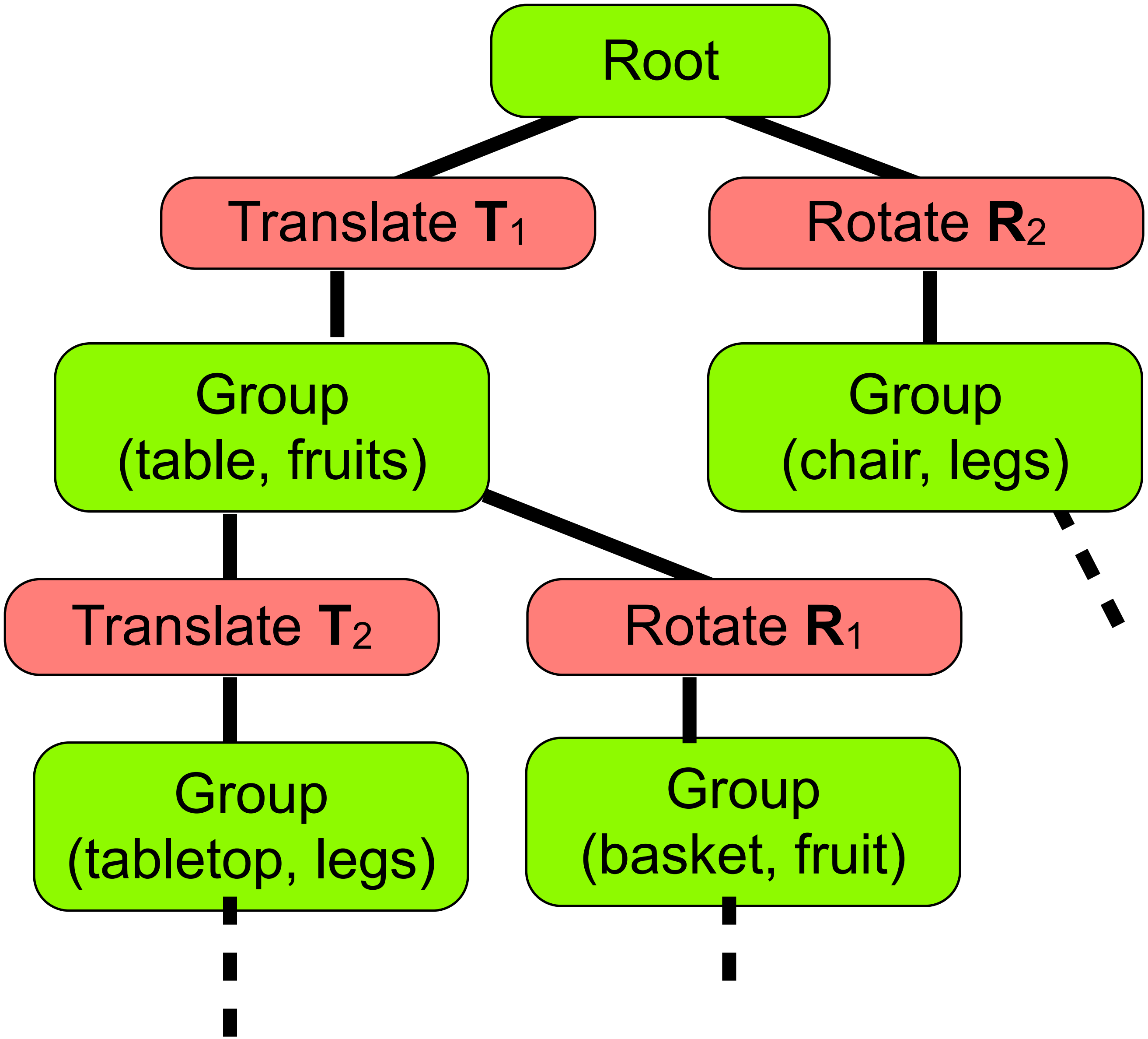


Scene Graph Traversal

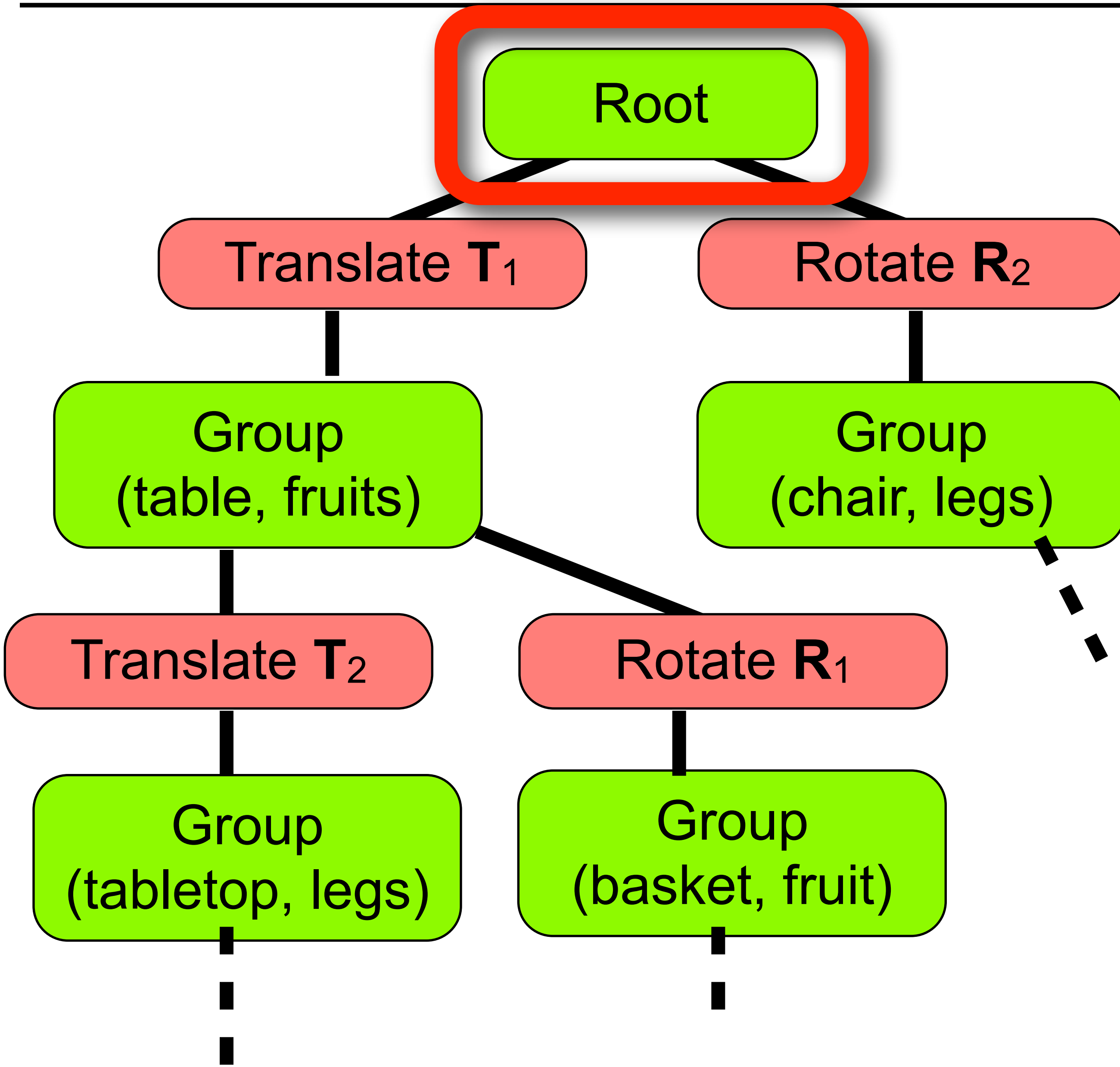
- How to handle transformations?
  - Traversal algorithm keeps a **transformation state S**
    - a 4x4 matrix initialized to identity **I** in the beginning
  - Geometry nodes always drawn using current **S**
  - When visiting a transformation node **T**: multiply current state **S** with **T**, then visit child nodes
    - Has the effect that nodes below will have new transformation
  - When all children have been visited, **undo the effect of T!**



Traversal Example

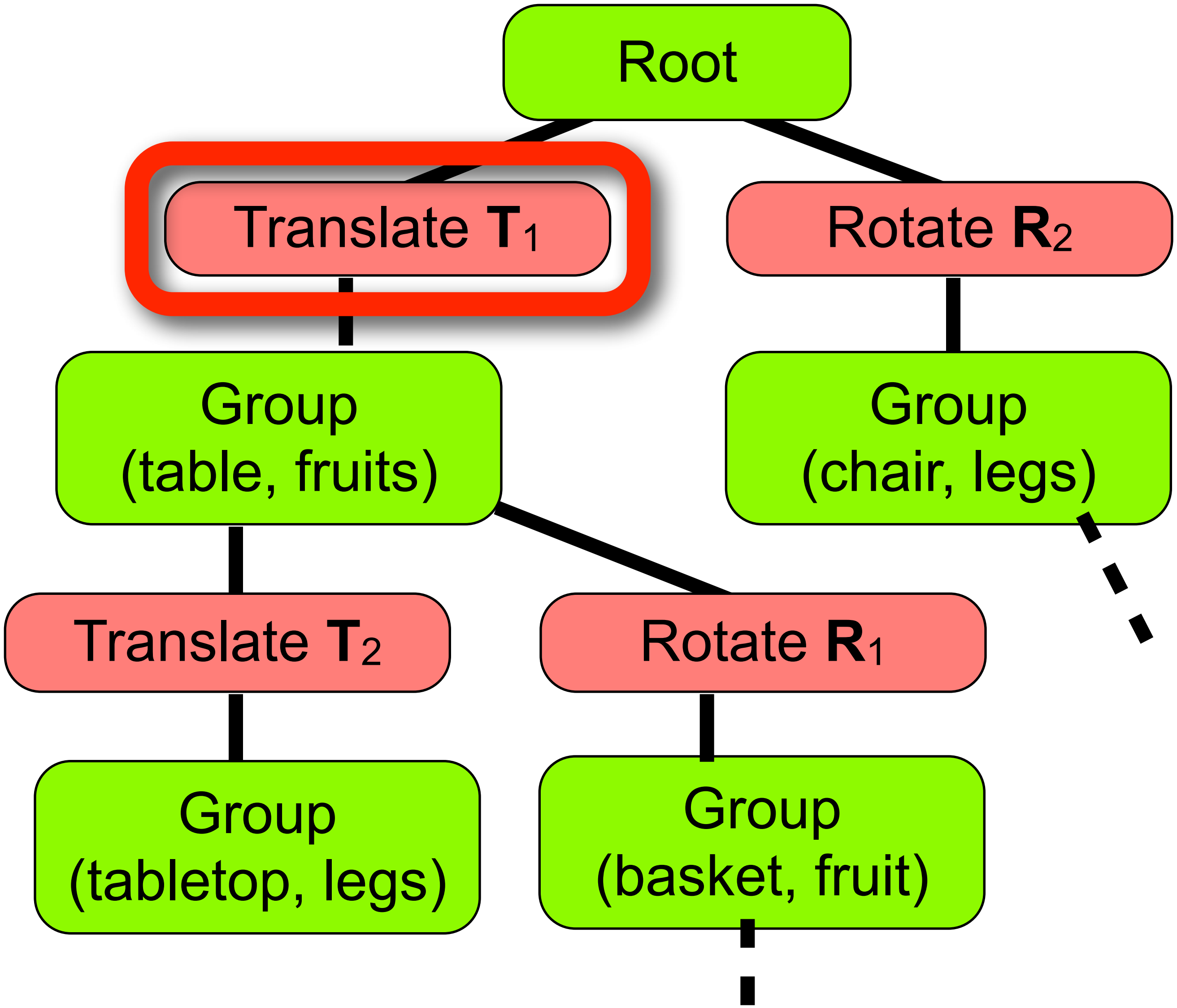


Traversal Example



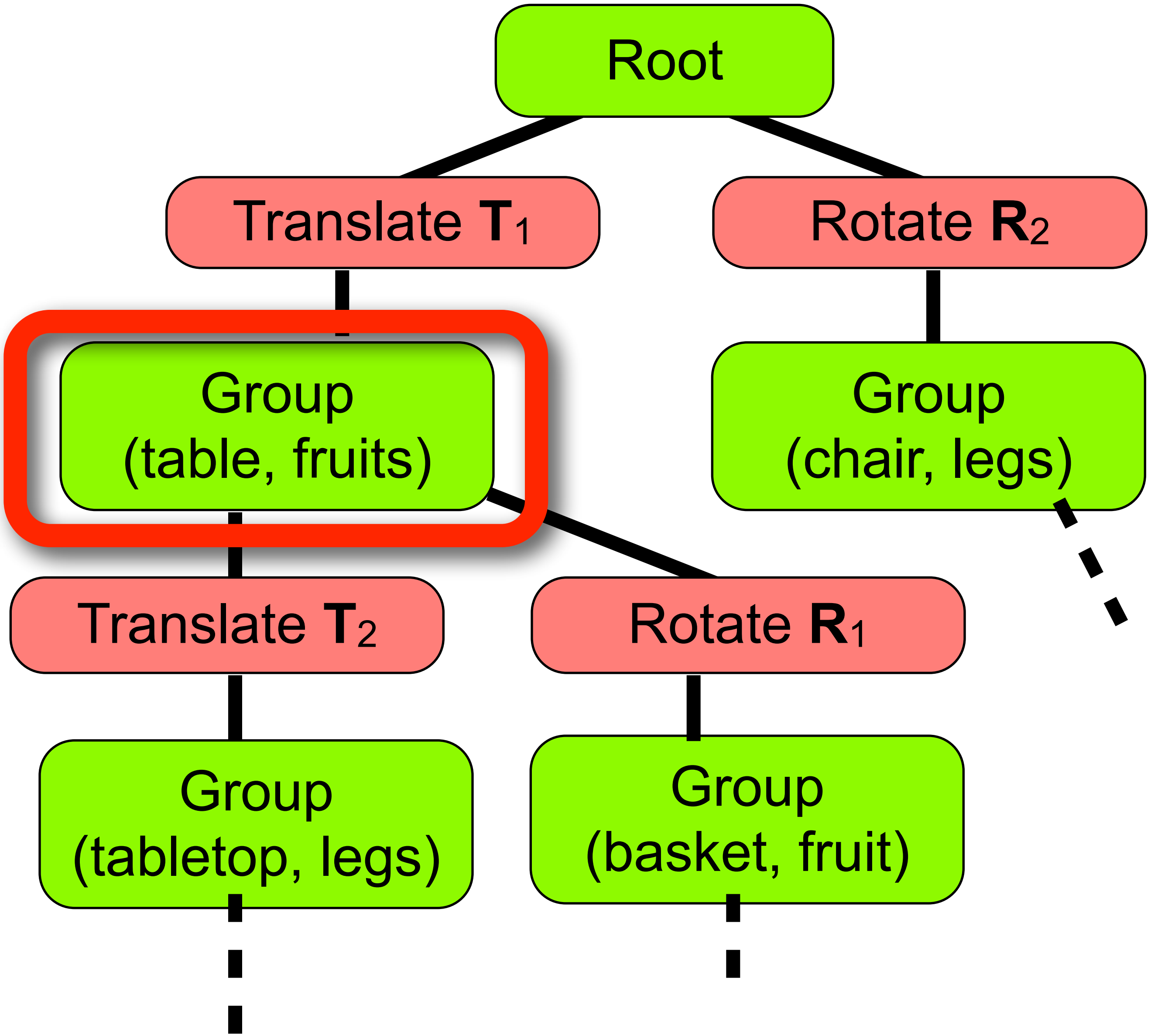
S = **I**

Traversal Example



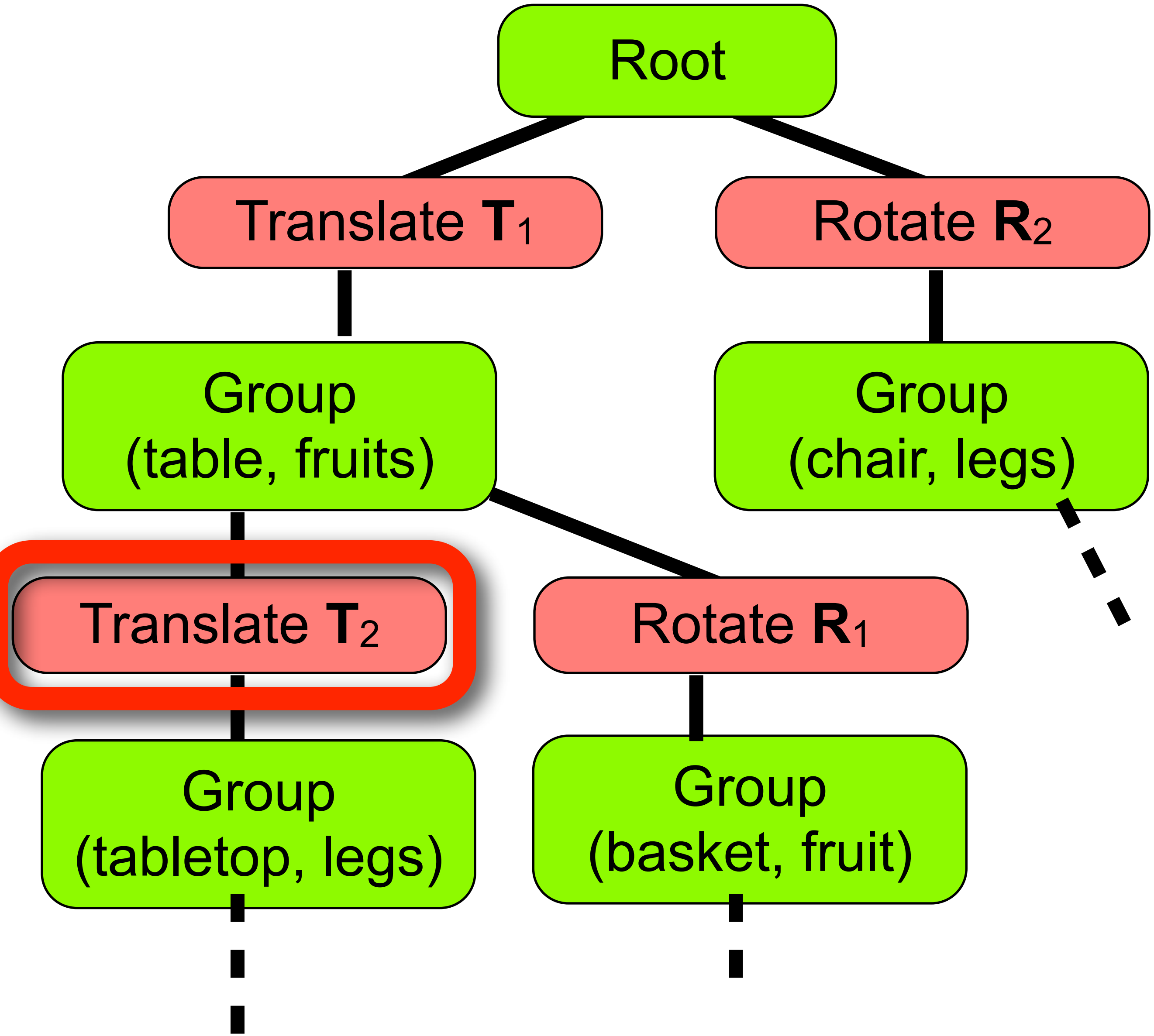
$\mathbf{S} = \mathbf{T}_1$

Traversal Example



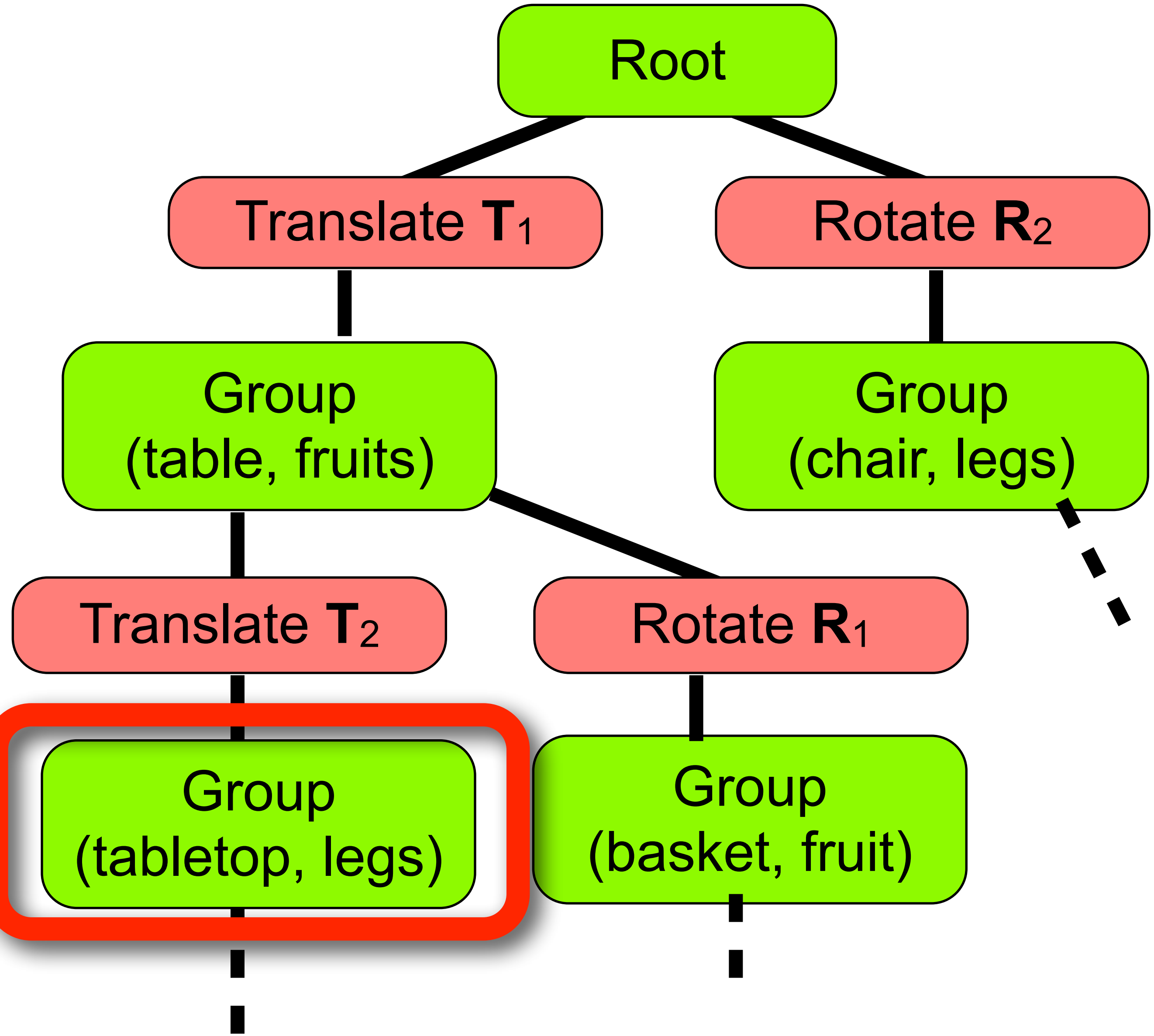
$\mathbf{S} = \mathbf{T}_1$

Traversal Example



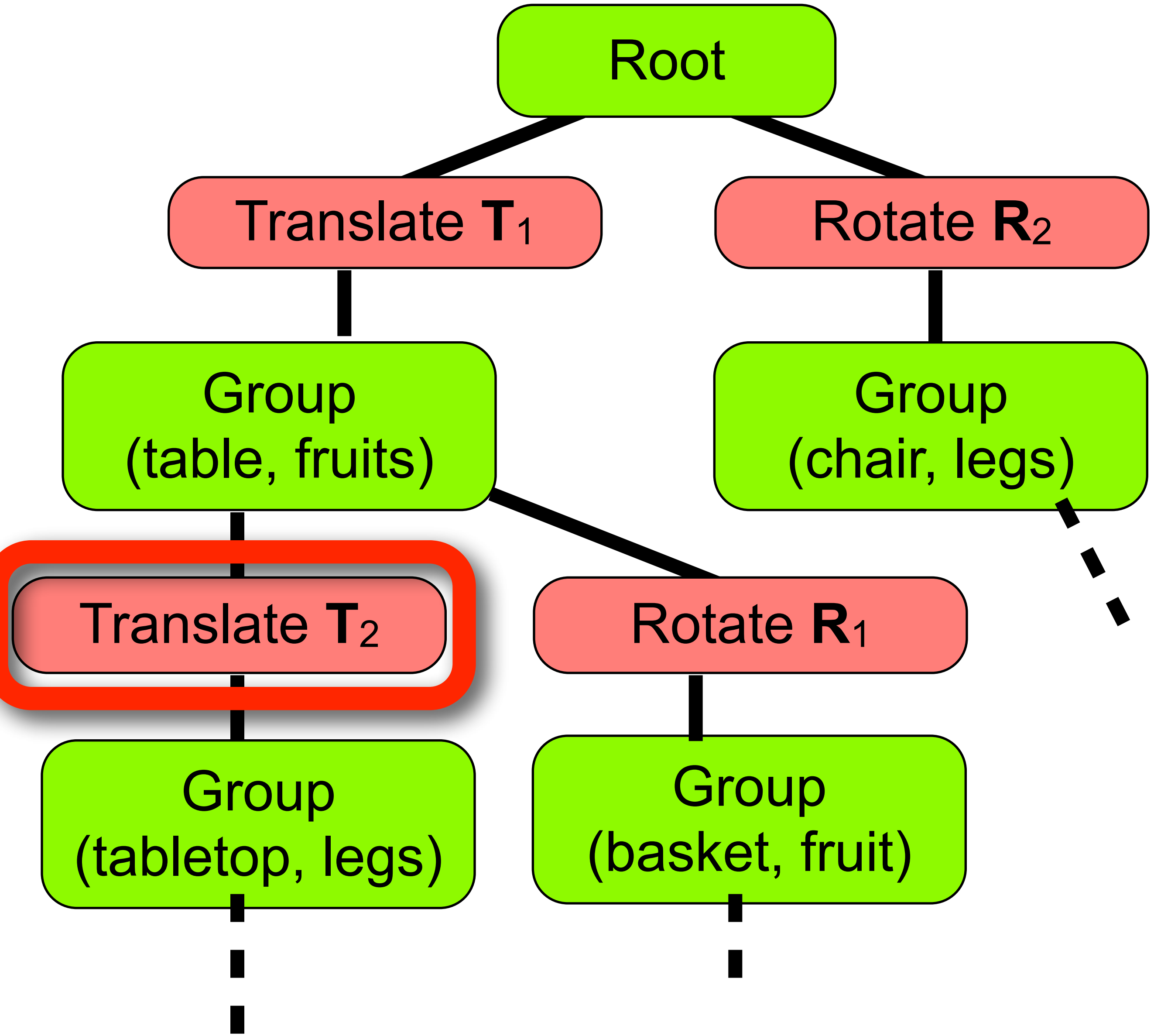
$$S = T_1\ T_2$$

Traversal Example



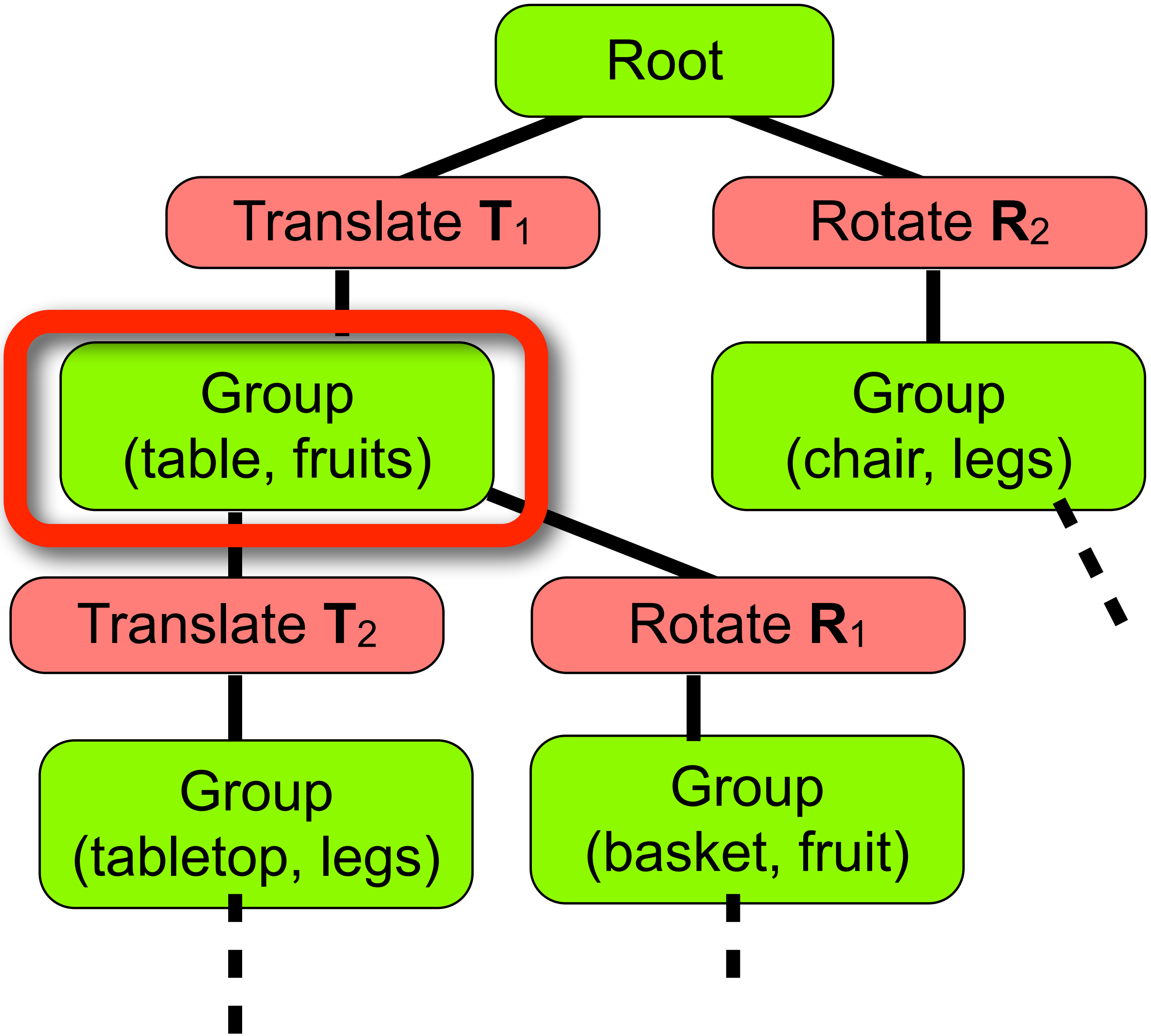
$$\mathbf{S} = \mathbf{T}_1\ \mathbf{T}_2$$

Traversal Example



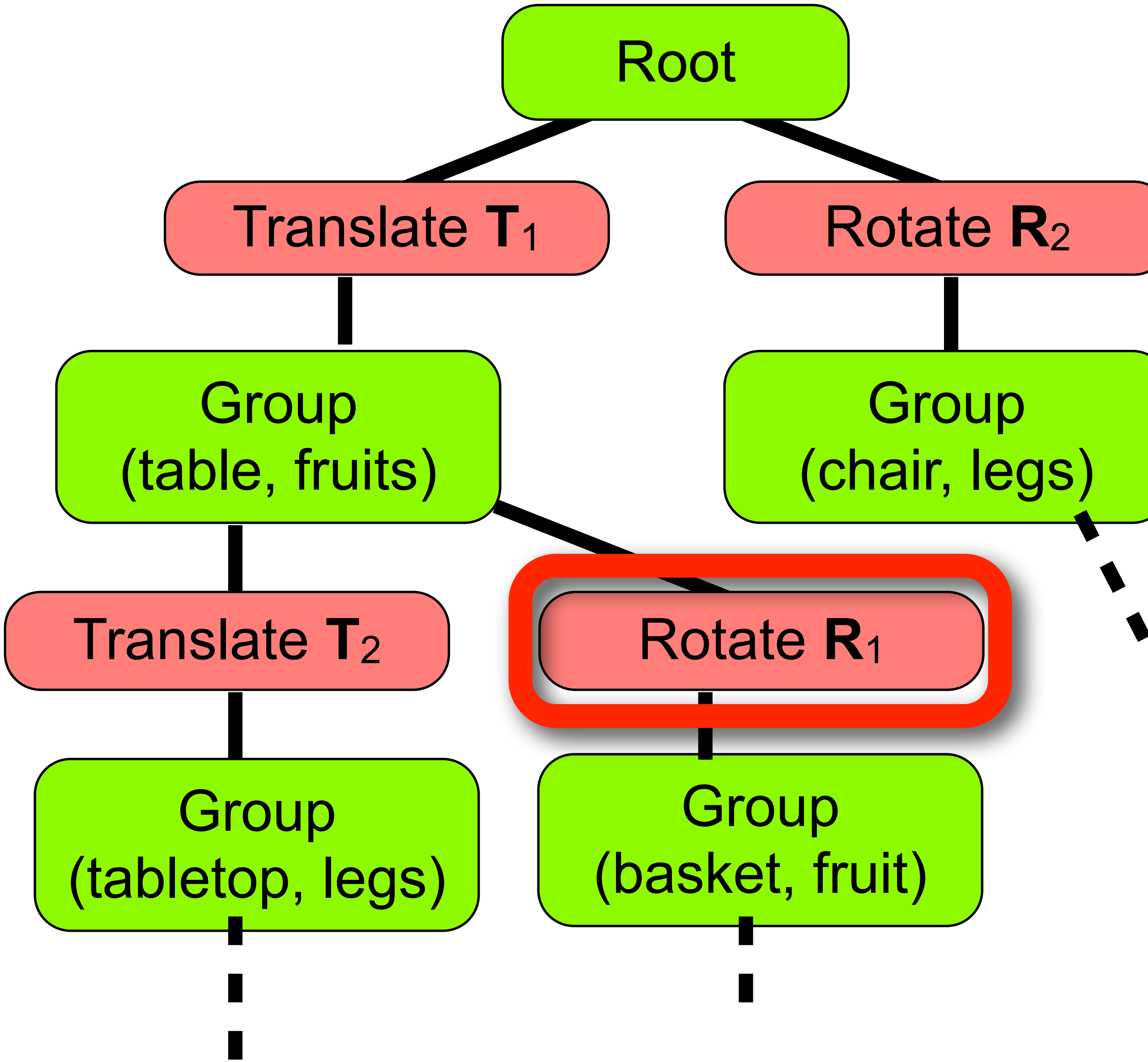
$$S = T_1\ T_2$$

Traversal Example



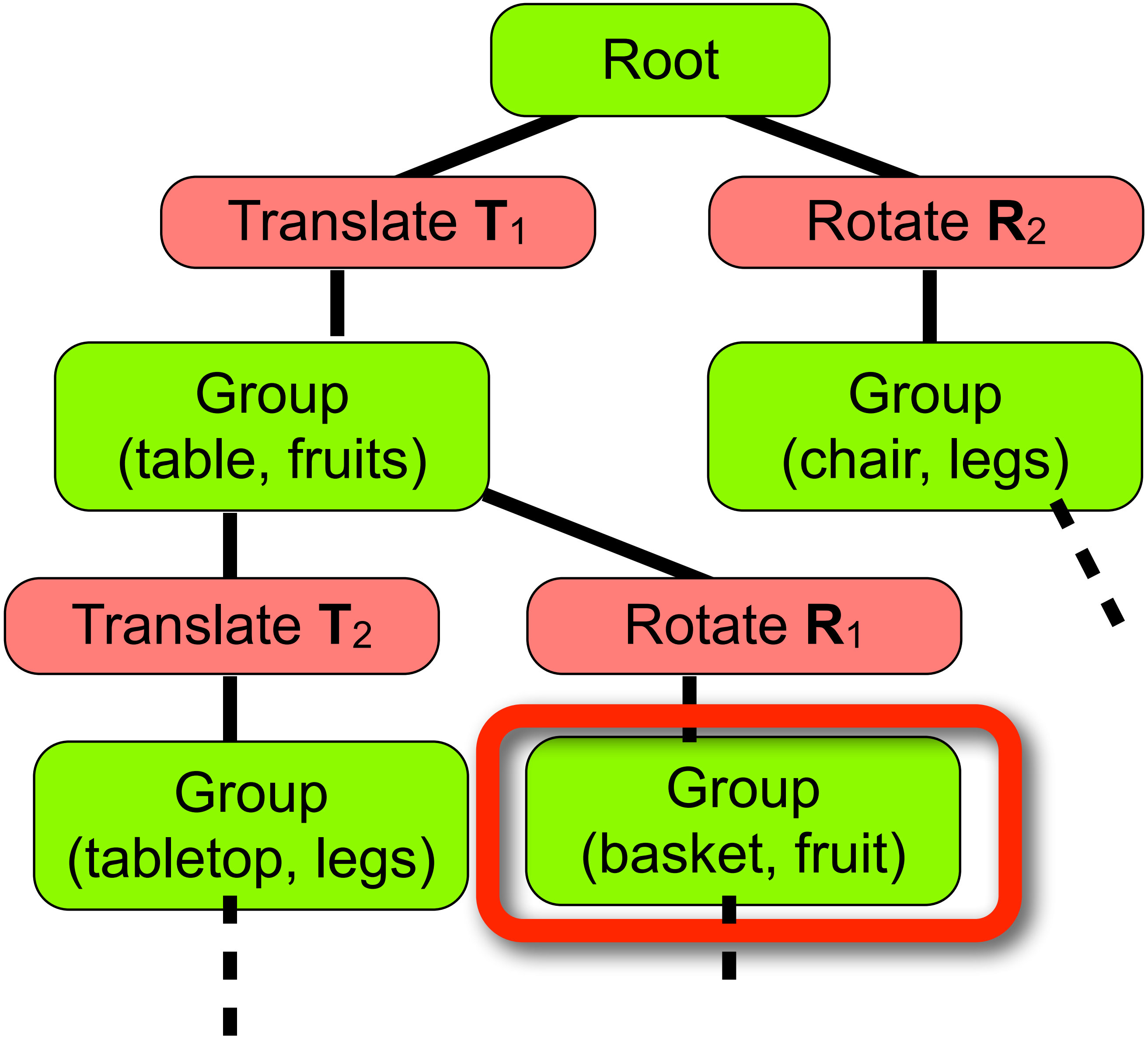
$$\mathbf{S} = \mathbf{T}_1$$

Traversal Example



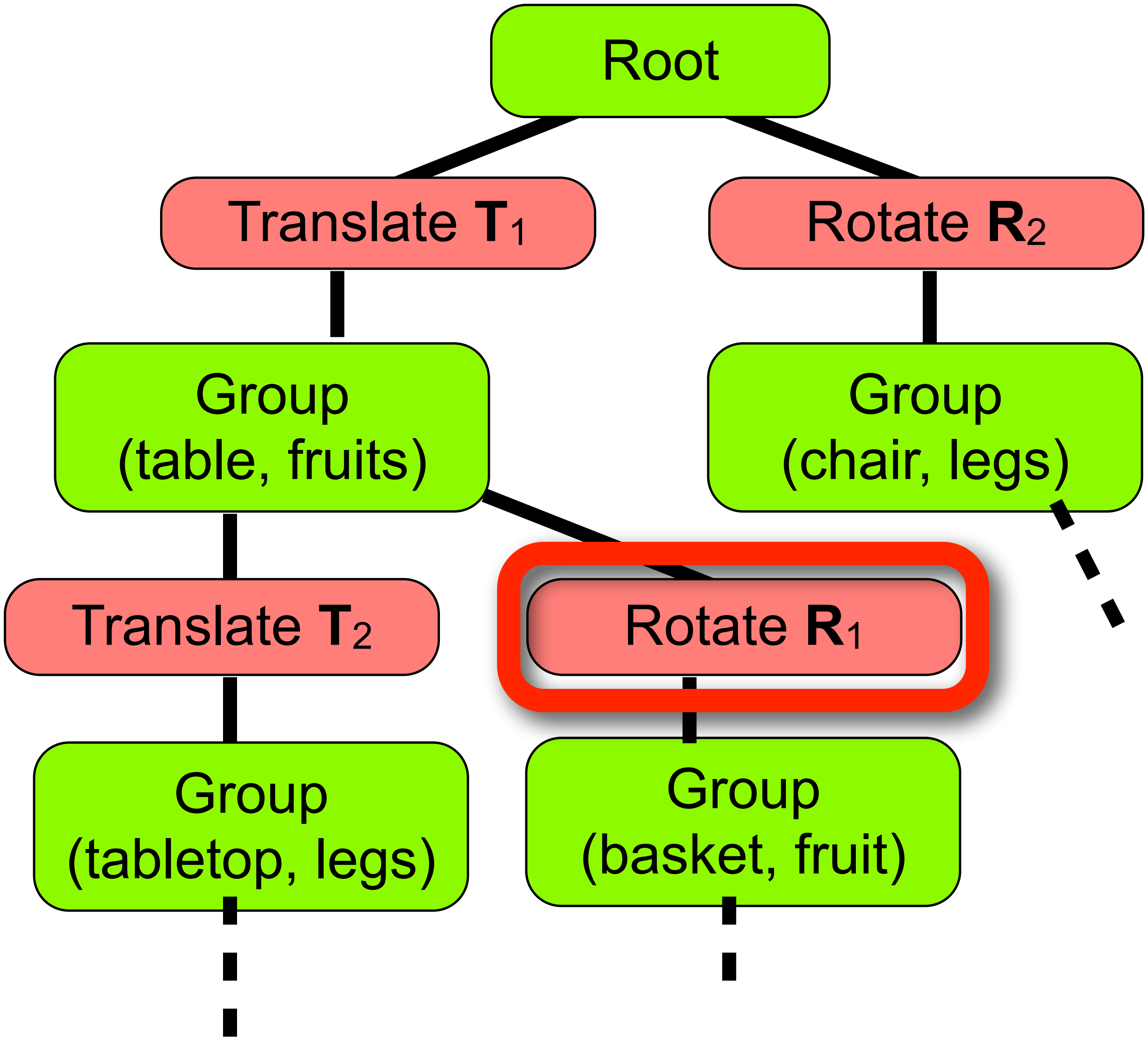
$$\mathbf{S} = \mathbf{T}_1\ \mathbf{R}_1$$

Traversal Example



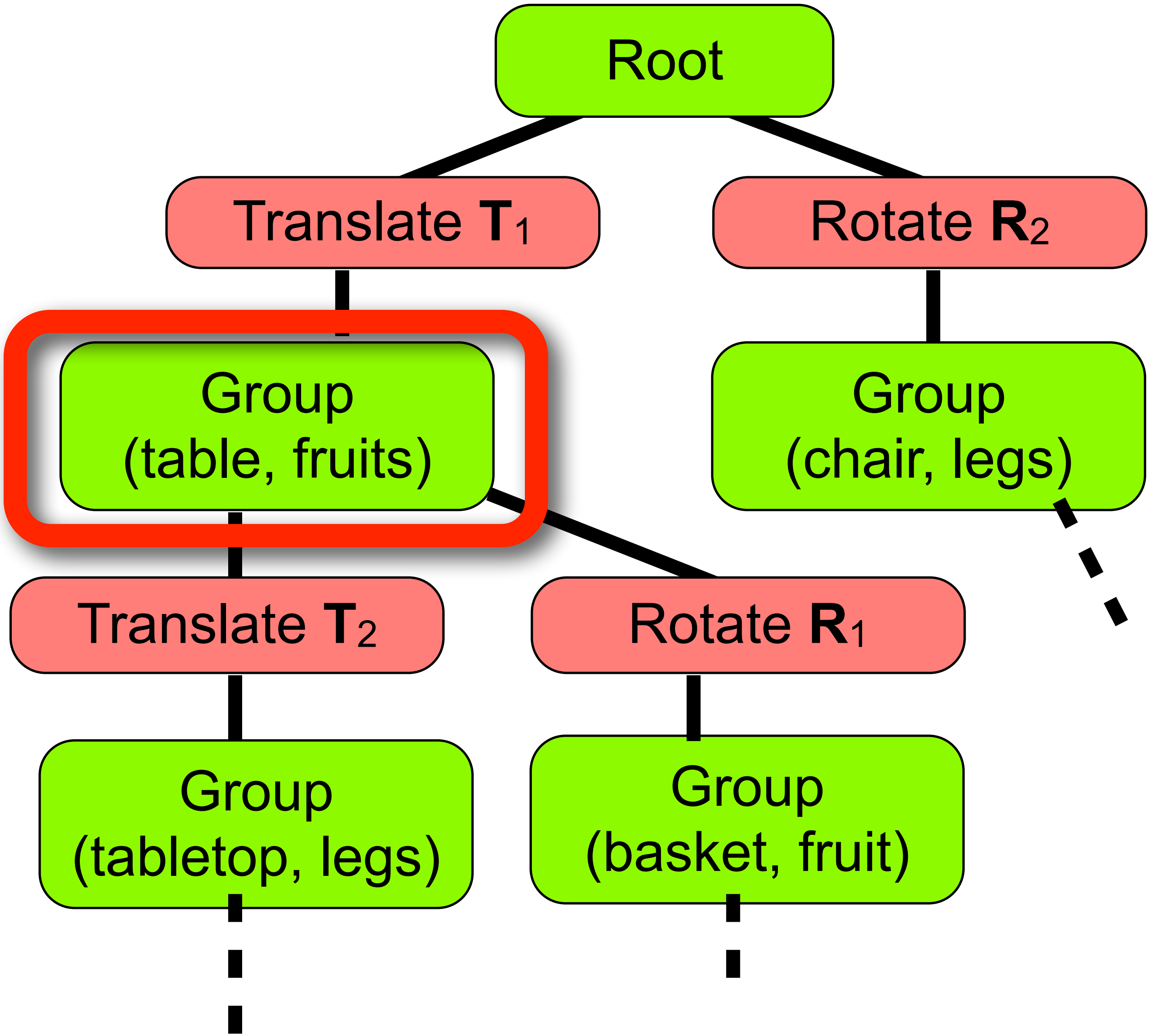
$$\mathbf{S} = \mathbf{T}_1\ \mathbf{R}_1$$

Traversal Example



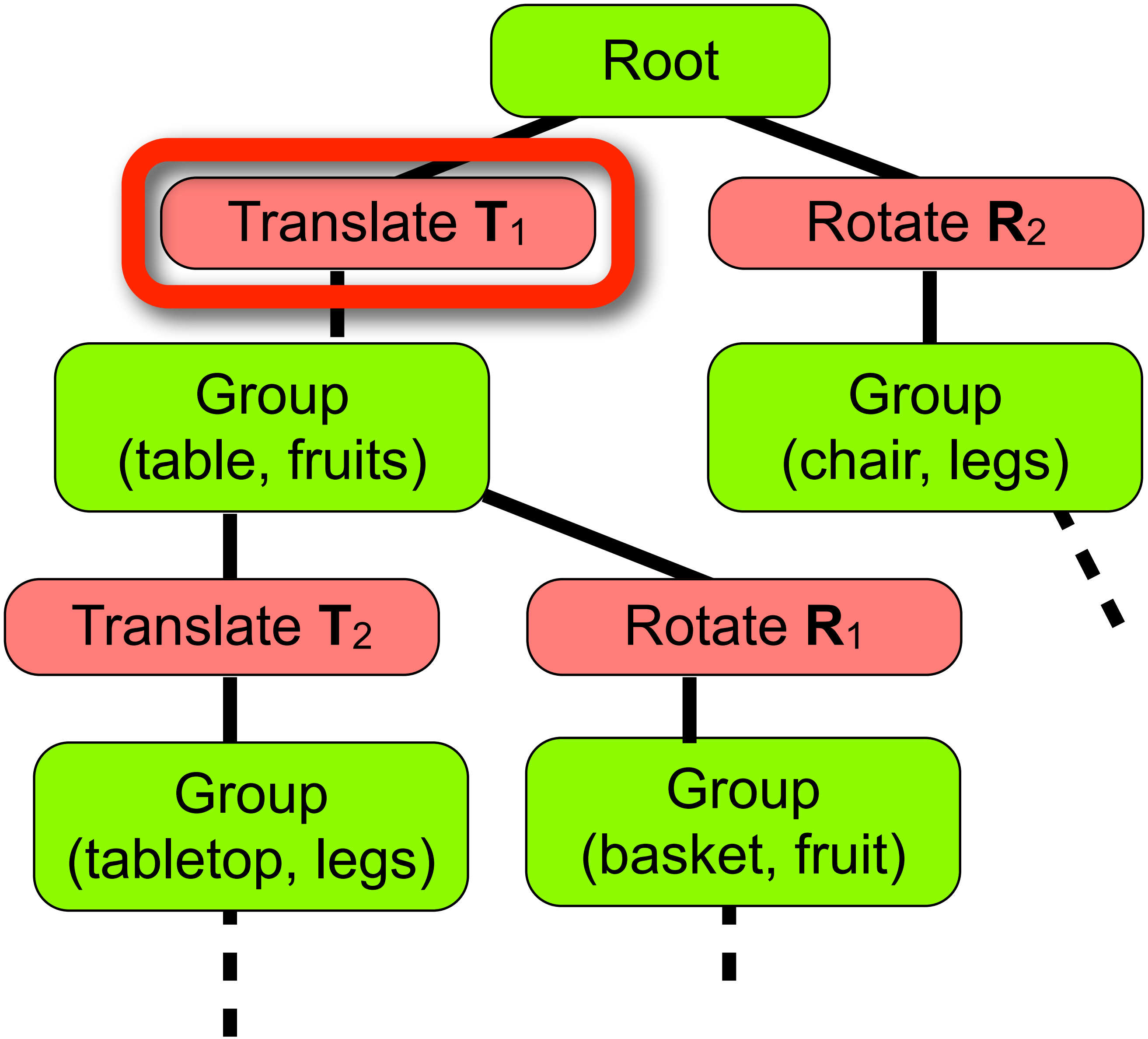
$$S = T_1 R_1$$

Traversal Example



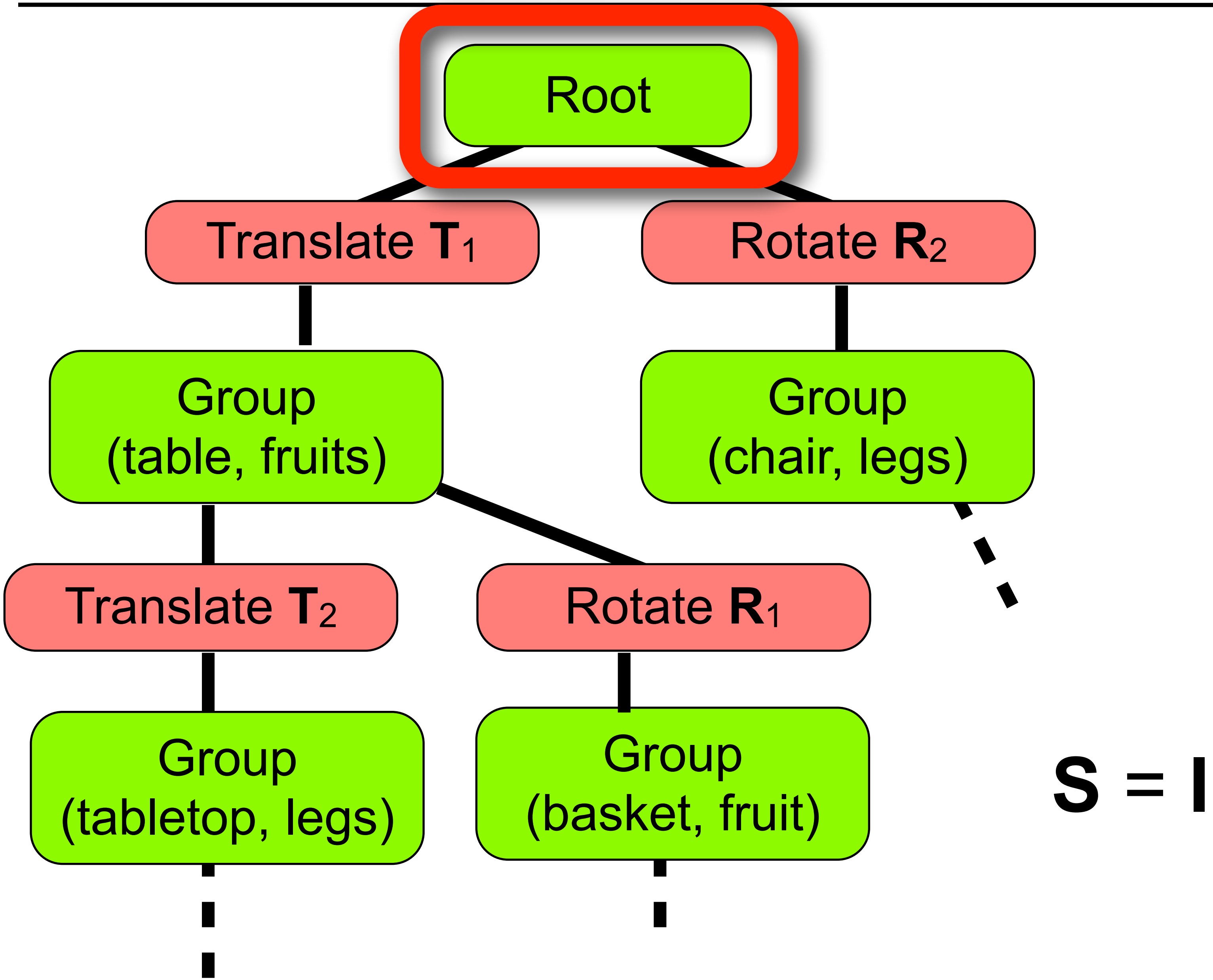
$S = T_1$

Traversal Example

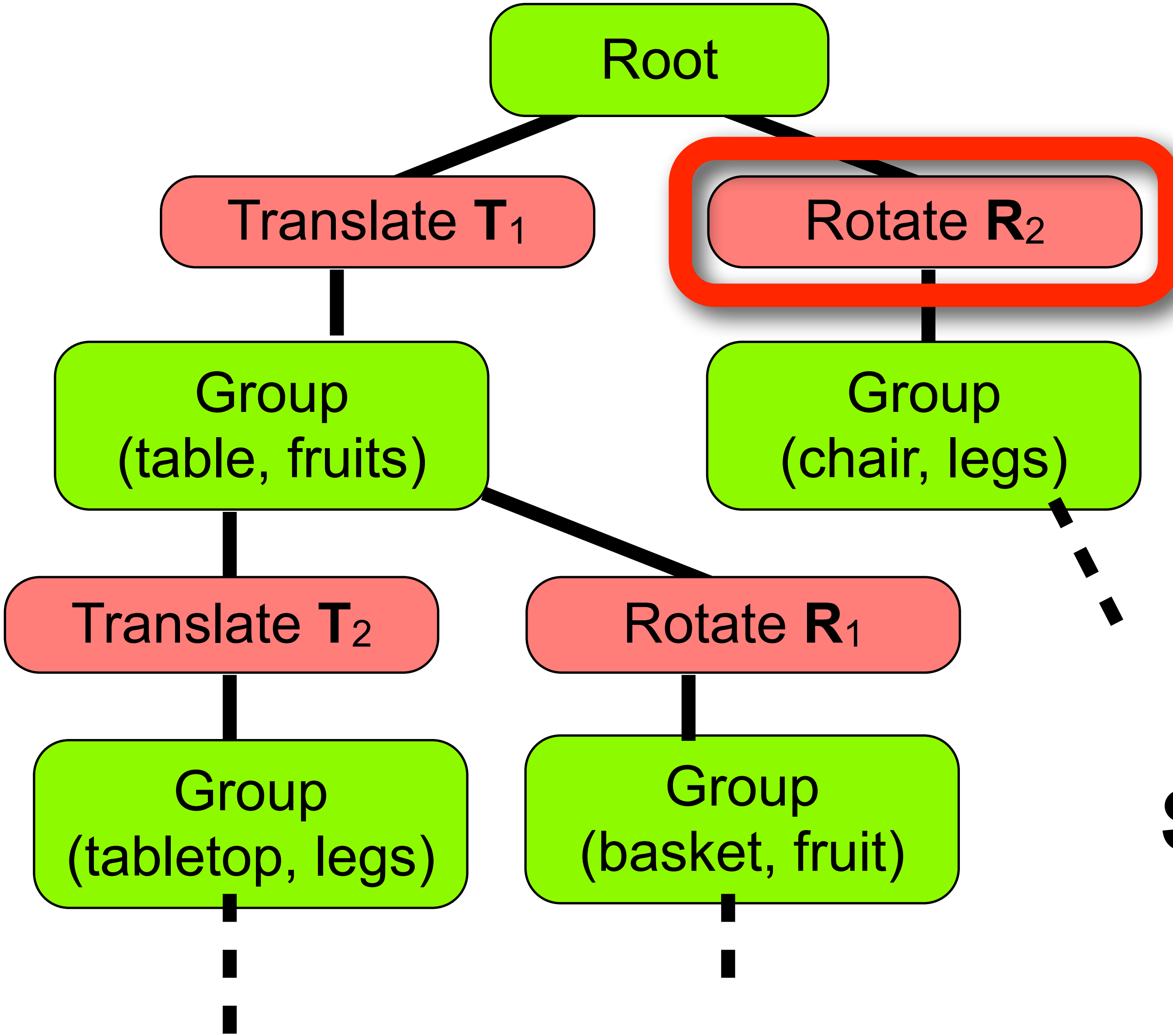


$S = T_1$

Traversal Example

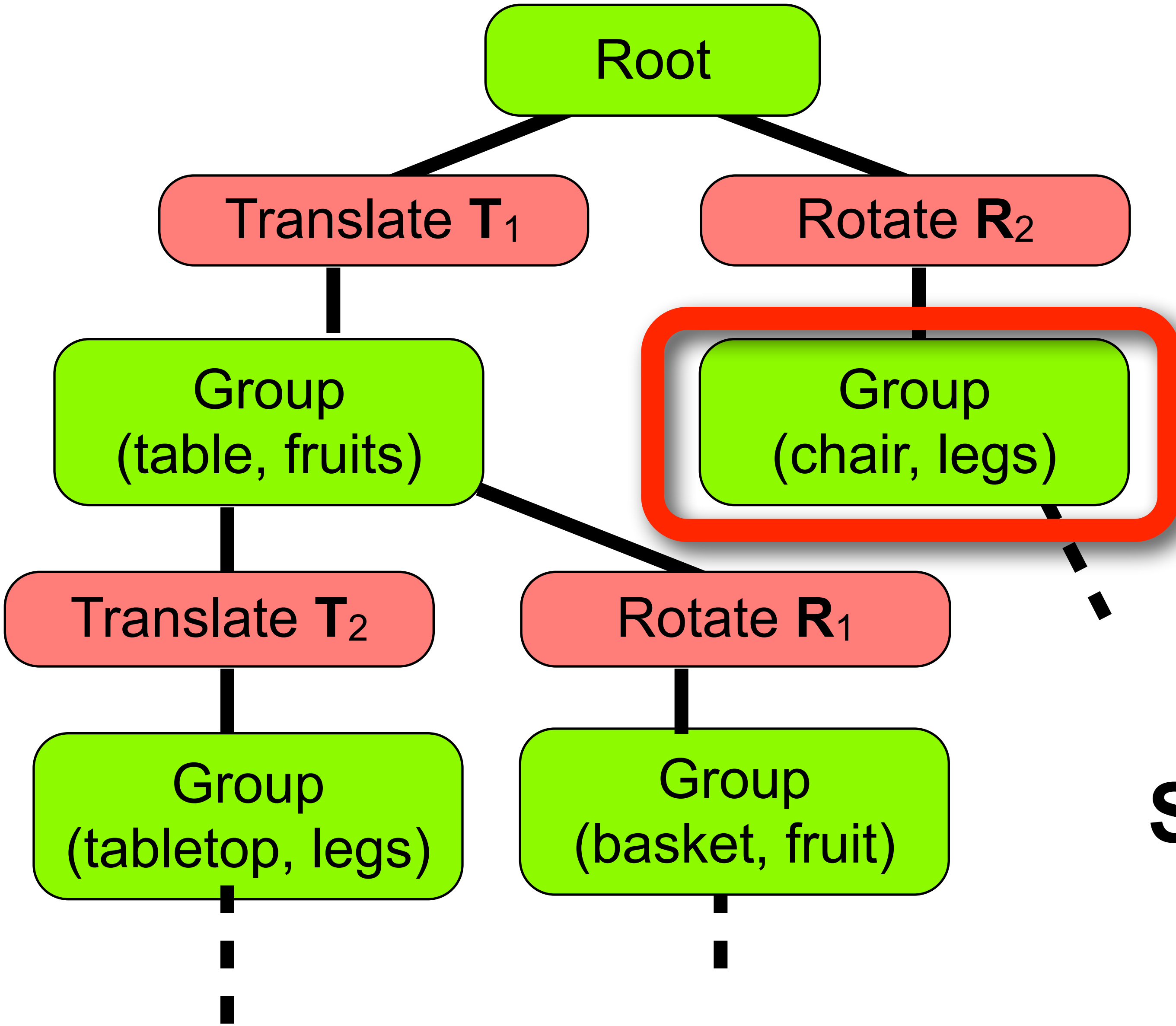


Traversal Example



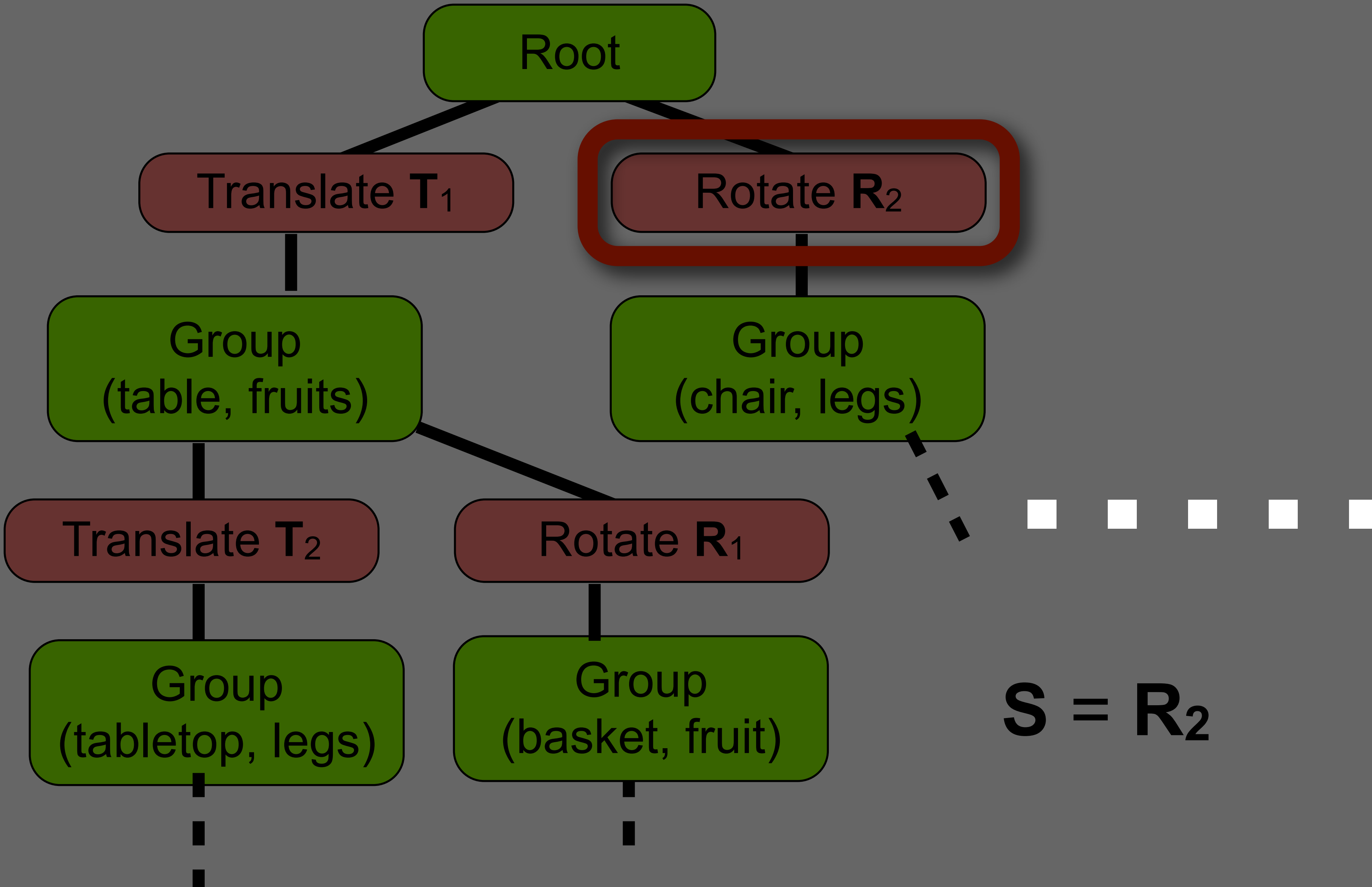
$\mathbf{S} = \mathbf{R}_2$

Traversal Example



$S = R_2$

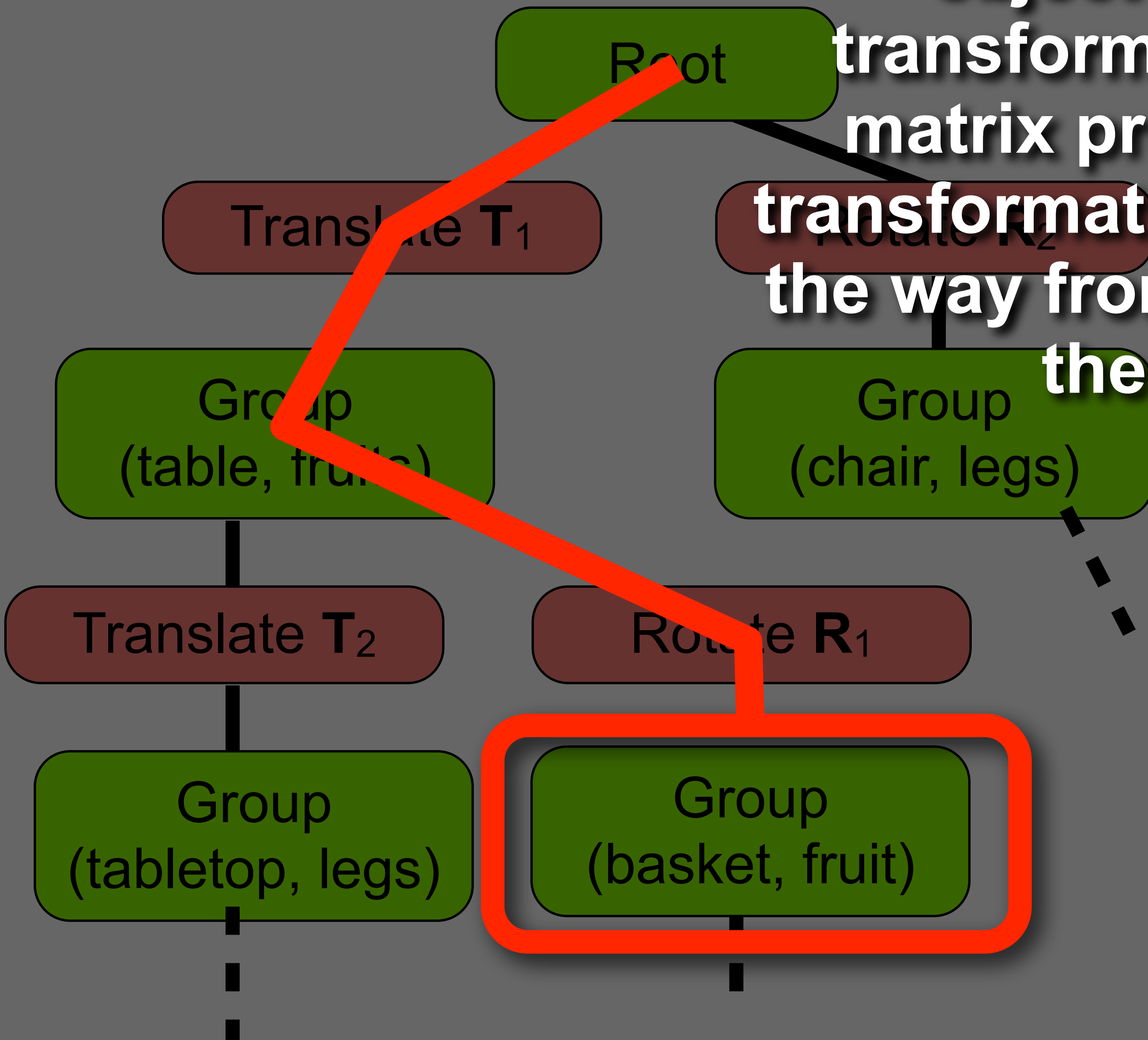
Traversal Example



$$\mathbf{S} = \mathbf{R_2}$$

Traversal Example

At each node, the current object-to-world transformation is the matrix product of all transformations found on the way from the node to the root.



$$S = T_1R_1$$

Traversal State

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, “undo” when leaving**

Traversal State

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, "undo" when leaving**

- How to implement?
  - Bad idea to undo transformation by inverse matrix **(Why?)**

Traversal State

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, “undo” when leaving**

- How to implement?
  - Bad idea to undo transformation by inverse matrix
  - Why I? $\mathbf{T}^*\mathbf{T}^{-1} = \boldsymbol{I}$ does not necessarily hold in floating point even when T is an invertible matrix – you accumulate error
  - Why II? **T** might be singular, e.g., could flatten a 3D

Traversal State

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, "undo" when leaving**

Can you think of a data structure suited for this?

- How to implement?
  - Bad idea to undo transformation by inverse matrix
  - Why I? $\mathbf{T}*\mathbf{T}^{-1} = \boldsymbol{I}$ does not necessarily hold in floating point even when T is an invertible matrix – you accumulate error
  - Why II? **T** might be singular, e.g., could flatten a 3D

Traversal State – Stack

- The state is updated during traversal
  - Transformations
  - But also other properties (color, etc.)
  - **Apply when entering node, "undo" when leaving**

- How to implement?
  - Bad idea to undo transformation by inverse matrix
  - Why I? $\mathbf{T}^*\mathbf{T}^{-1} = \boldsymbol{I}$ does not necessarily hold in floating point even when T is an invertible matrix – you accumulate error
  - Why II? **T** might be singular, e.g., could flatten a 3D

Barebones Traversal Example

```
class NodeBase
{
    std::vector<NodeBase*> children; // note: no parent pointer, just children!
    Mat4f transform;
    // function to call when traversal reaches this node
    virtual void visit( Mat4f S );
};
// derive classes for geometry, etc., from NodeBase

void traverse( NodeBase* pNode, Mat4f S )
{
    // update current transform
    Mat4f newS = S * pNode->transform;
    // visit node (for geometry, this means draw it, etc.)
    pNode->visit( newS );
    // recursive call to children, using new transformation
    for ( int i = 0; i < pNode->children.getSize(); ++i )
        traverse( pNode->children[i], newS )
}

void drawScene( NodeBase* pRoot )
{
    // first set things set up
    // ...
    // then call traverse for root with identity transformation
    traverse( pRoot, Mat4f::identity() );
}
```

Barebones Traversal Example

```
class NodeBase
{
    std::vector<NodeBase*> children; // note: no parent pointer, just children!
    Mat4f transform;
    // function to call when traversal reaches this node
    virtual void visit( Mat4f S );
};
// derive classes for geometry, etc., from NodeBase

void traverse( NodeBase* pNode, Mat4f S )
{
    // update current transform
    Mat4f newS = S * pNode->transform;
    // visit node (for geometry, this means draw it, etc.)
    pNode->visit( newS );
    // recursive call to children, using new transformation
    for ( int i = 0; i < pNode->children.getSize(); ++i )
        traverse( pNode->children[i], newS )
}

void drawScene( NodeBase* pRoot )
{
    // first set things set up
    // ...
    // then call traverse for root with identity transformation! done!
    traverse( pRoot, Mat4f::identity() );
}
```

Note 1: This example is using the built-in stack for pushing and popping the transform (that's what happens in recursive function calls, remember CS101!), but you could just as well maintain a stack yourself. This also works out-of-the-box for DAGs, i.e., shared subtrees.

Note 2: Other state (e.g. materials) would also need to be carried along if needed

Note 3: I cut corners and made the transformation part of the base node class to save space (a perfectly ok thing to do in practice)

That’s All!