6.837 Computer Graphics Hierarchical Modeling

Wojciech Matusik, MIT EECS

Some slides from BarbCutler & Jaakko Lehtinen

Recap

- Vectors can be expressed in a basis
  - Keep track of basis with left notation $\vec{v} = \vec{\mathbf{b}}^t \mathbf{c}$
  - Change basis $\vec{v} = \vec{\mathbf{a}}^t M^{-1} \mathbf{c}$
- Points can be expressed in a frame (origin+basis)
  - Keep track of frame with left notation
  - adds a dummy 4th coordinate always 1

$$\tilde{p} = \tilde{o} + \sum_i c_i \vec{b}_i = \begin{bmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 & \tilde{o} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ 1 \end{bmatrix} = \vec{\mathbf{f}}^t \mathbf{c}$$

Frames & transformations

- Transformation S wrt car frame f

$$\tilde{p} = \vec{\mathbf{f}}^t \mathbf{c} \Rightarrow \vec{\mathbf{f}}^t S \mathbf{c}$$

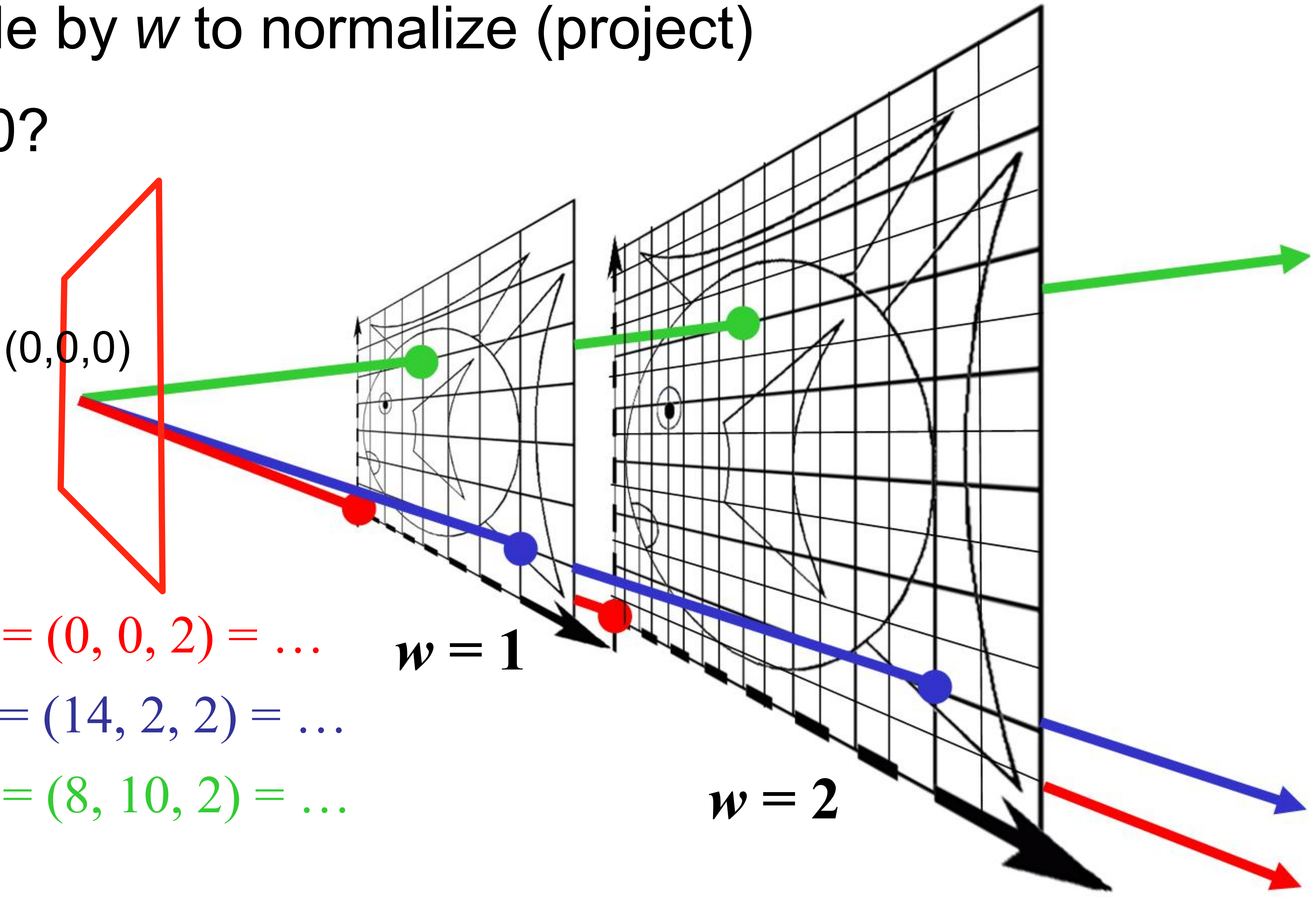
- how is the world frame a affected by this?
- *we have* $\vec{a}^t = \vec{f}^t A \quad \vec{\mathbf{f}}^t = \vec{\mathbf{a}}^t A^{-1}$
- *which gives* $\vec{a}^t A^{-1} \Rightarrow \vec{a}^t A^{-1} S$

$$\vec{a}^t \Rightarrow \vec{a}^t A^{-1} S A$$

- *i.e. the transformation in a is A-1SA*
- *i.e., from right to left, A takes us from a to f, then we apply S, then we go back to a with A-1*

Homogeneous Visualization

- Divide by *w* to normalize (project)
- *w* = 0?



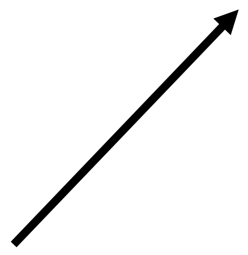
(0, 0, 1) = (0, 0, 2) = …

(7, 1, 1) = (14, 2, 2) = …

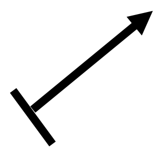
(4, 5, 1) = (8, 10, 2) = …

Different objects

- **Points**
  - represent locations



- **Vectors**
  - represent movement, force, displacement from A to B
- **Normals**
  - represent orientation, unit length

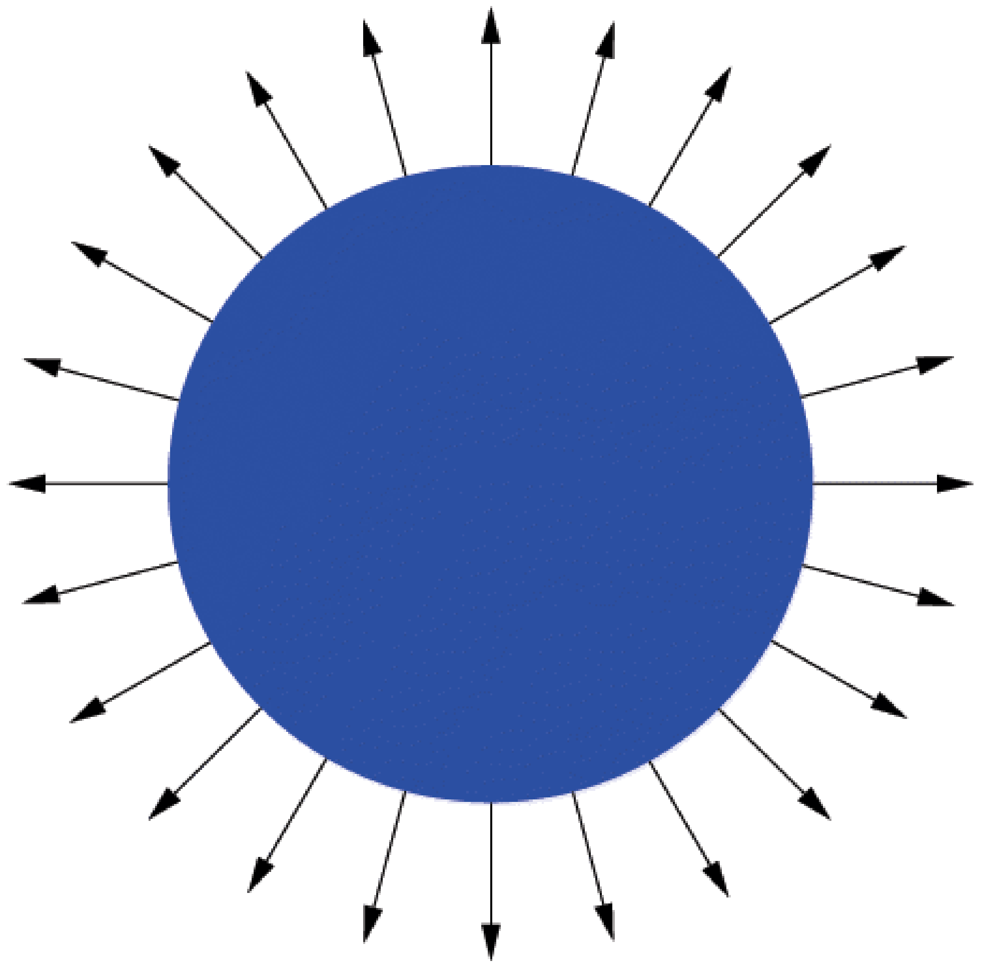
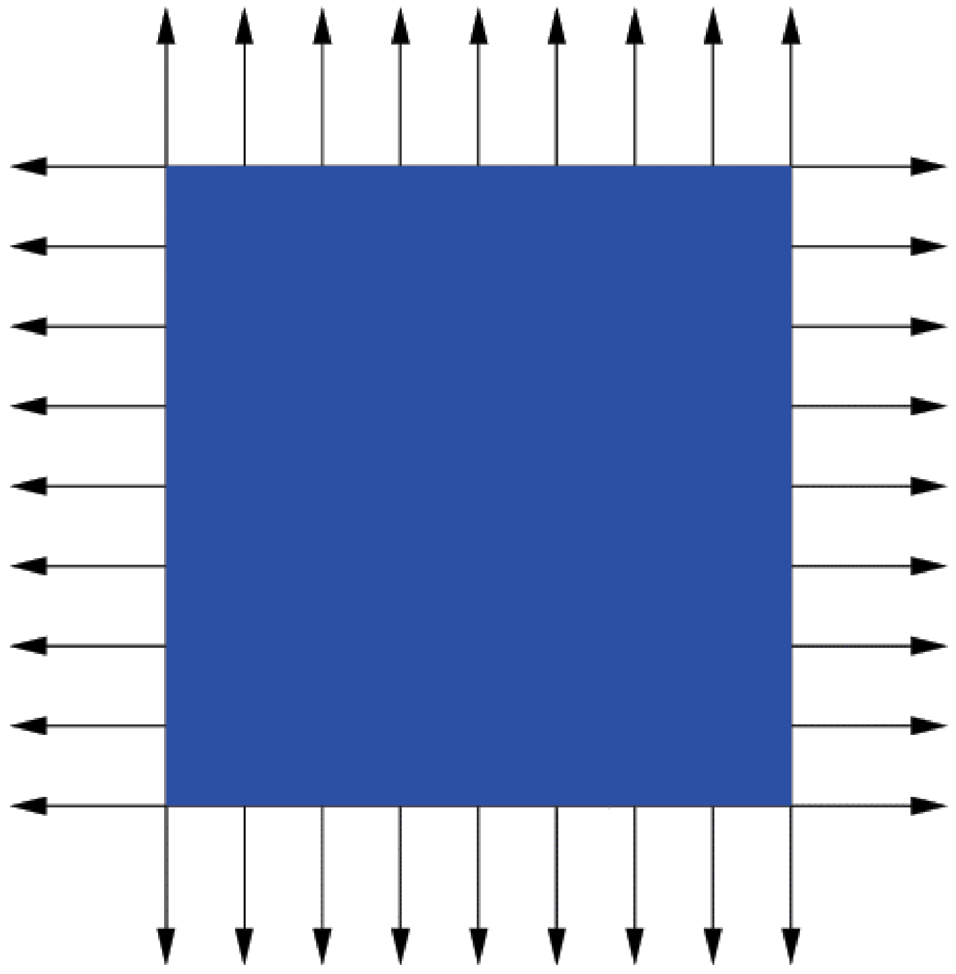


- **Coordinates**
  - numerical representation of the above objects **in a given coordinate system**

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

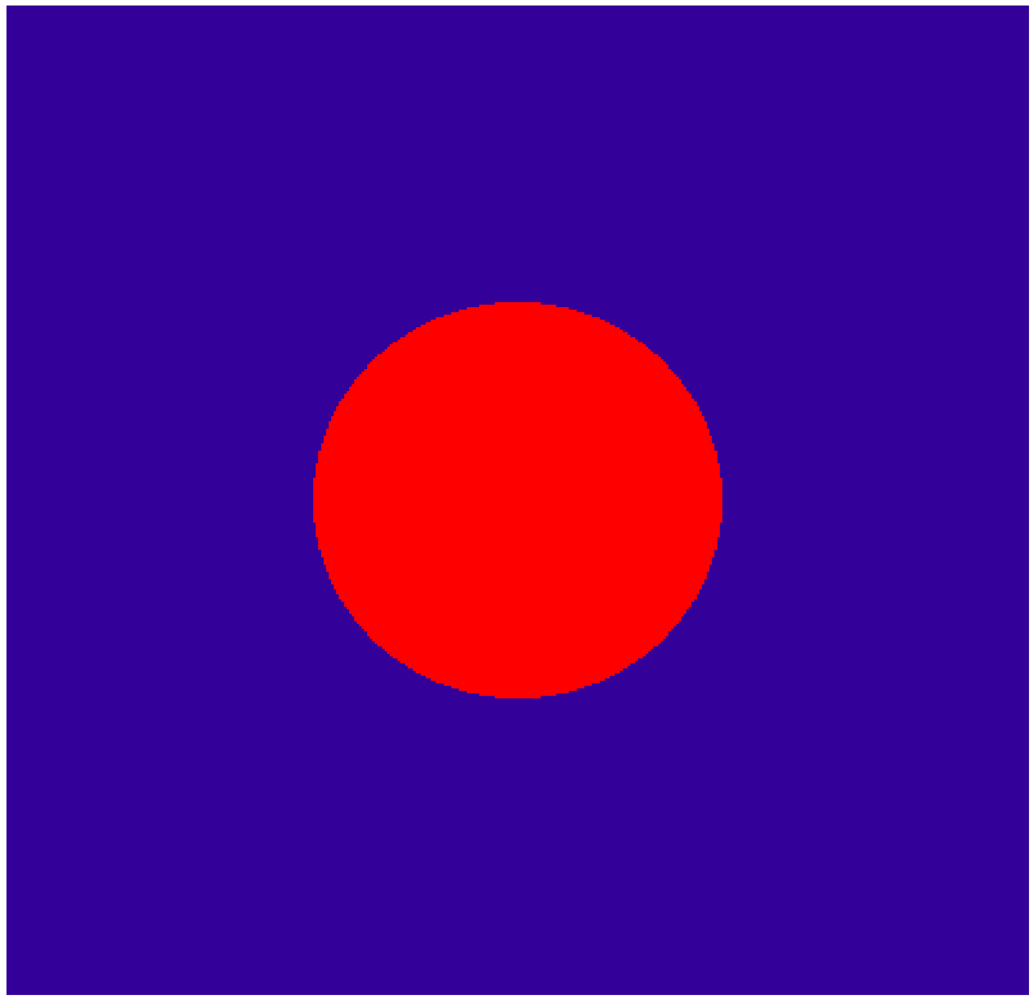
Normal

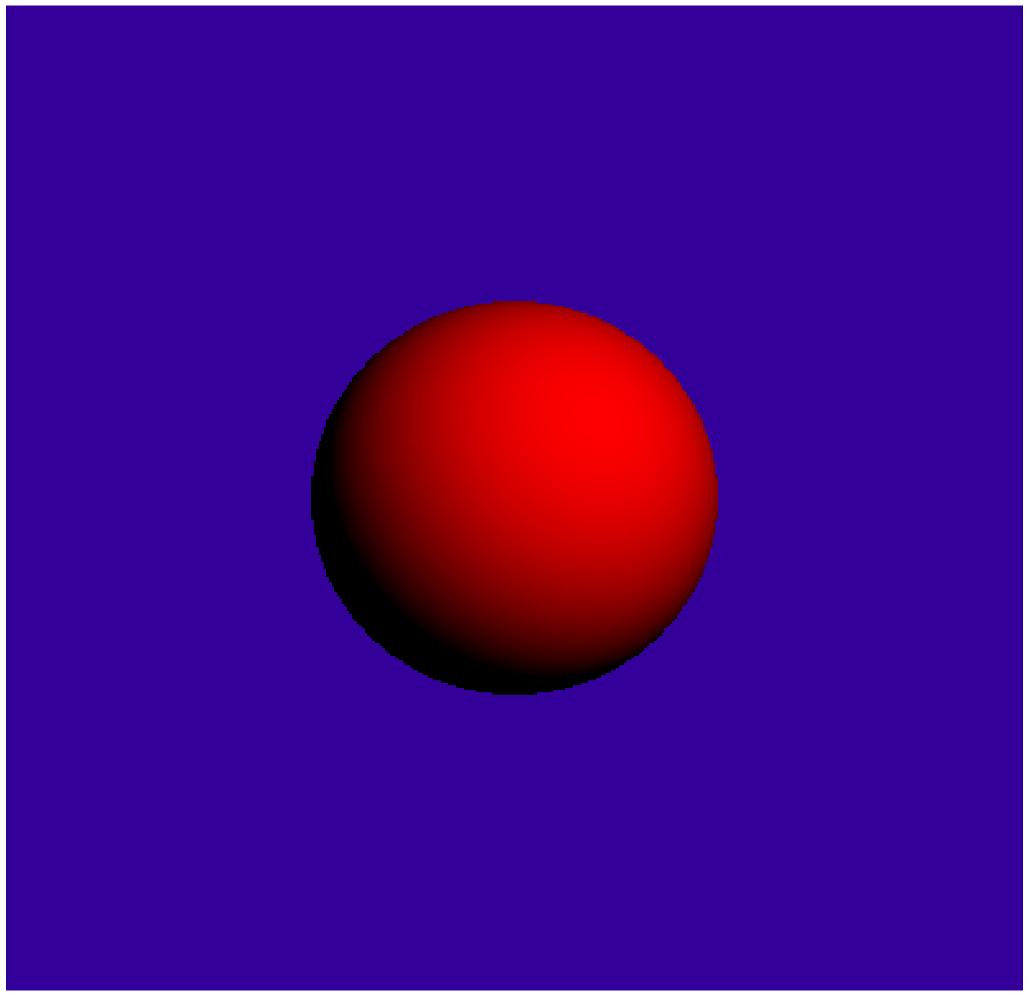
- Surface Normal: unit vector that is locally perpendicular to the surface

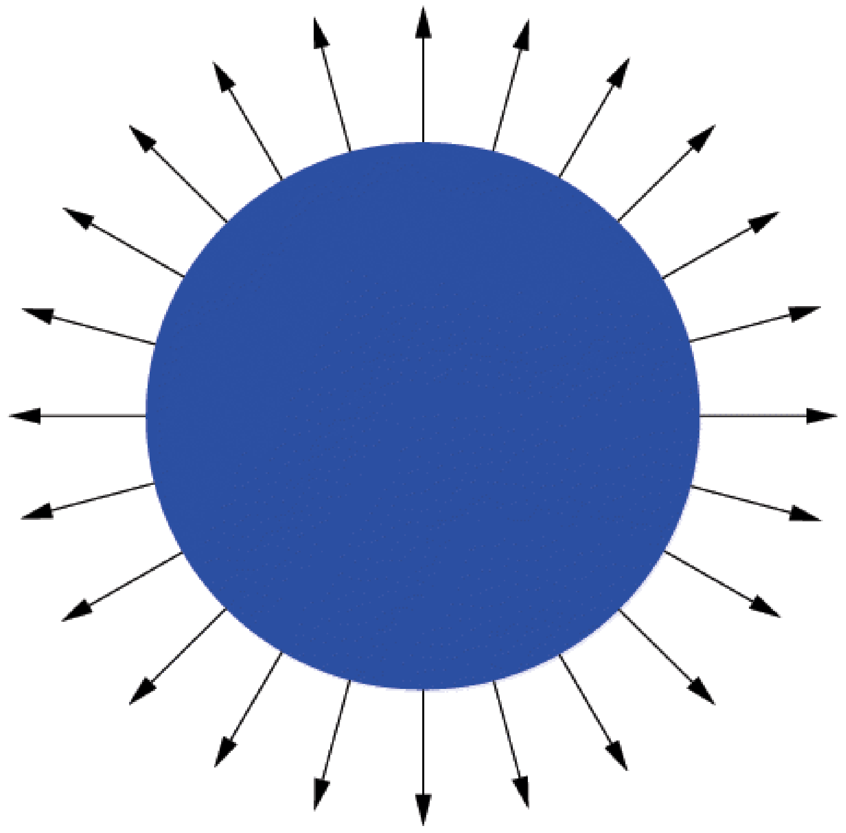
Why is the Normal important?

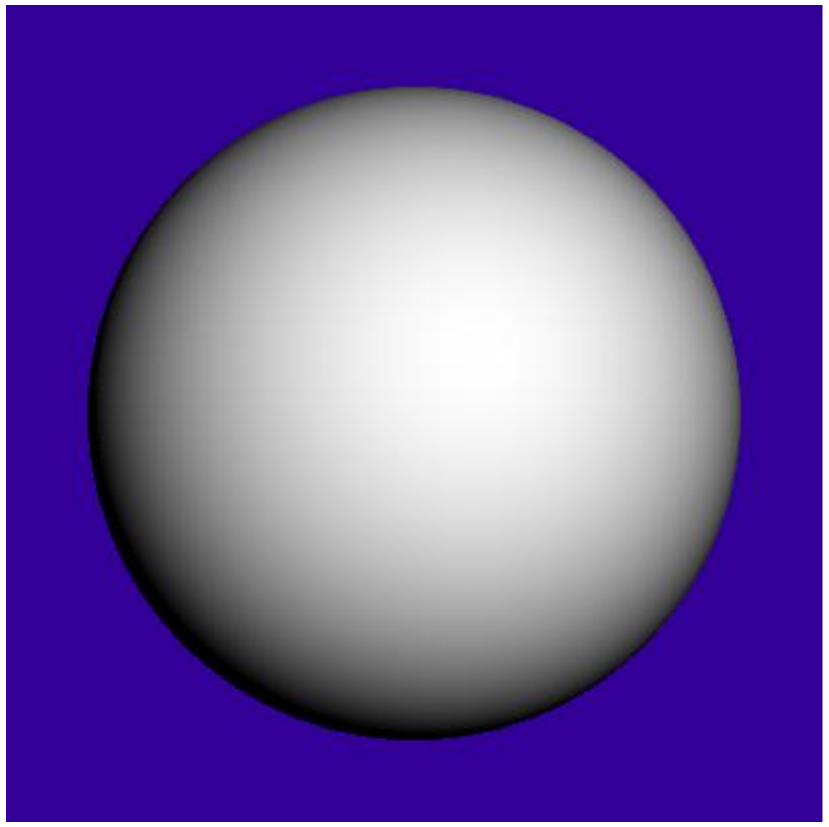
- It's used for shading — makes things look 3D!

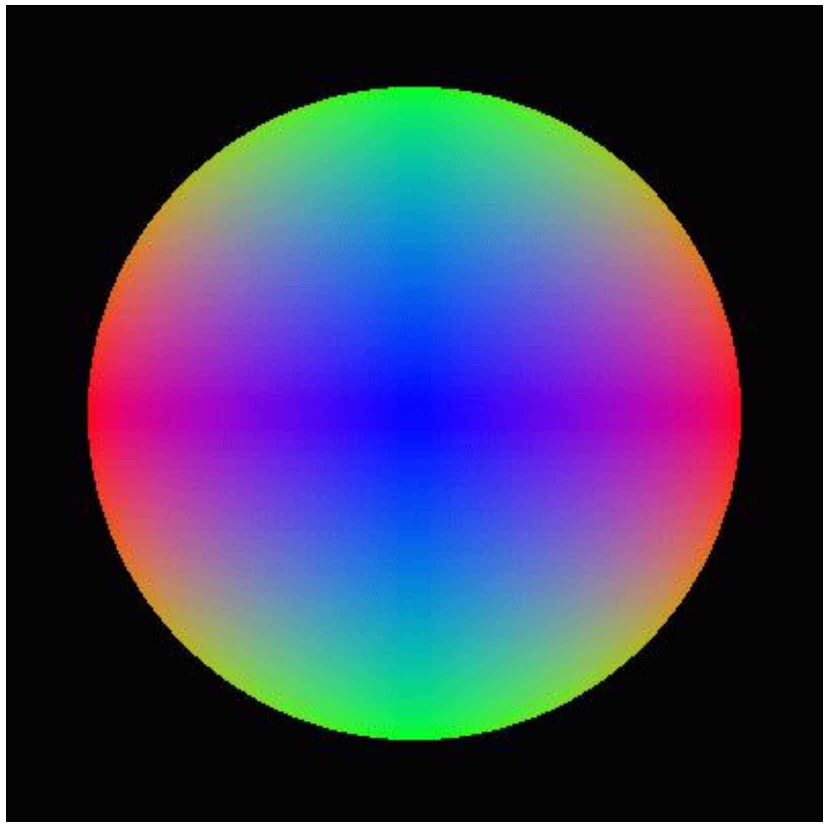

object color only


Diffuse Shading

Visualization of Surface Normal





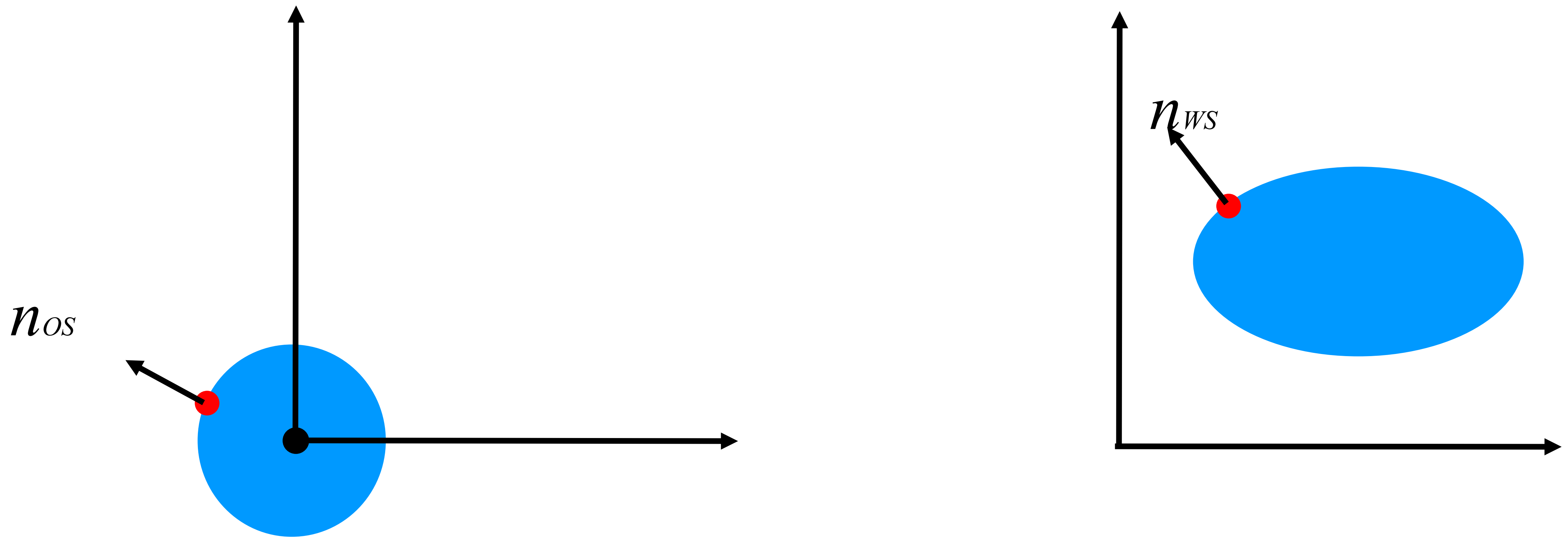


$\pm x$ = Red

$\pm y$ = Green

$\pm z$ = Blue

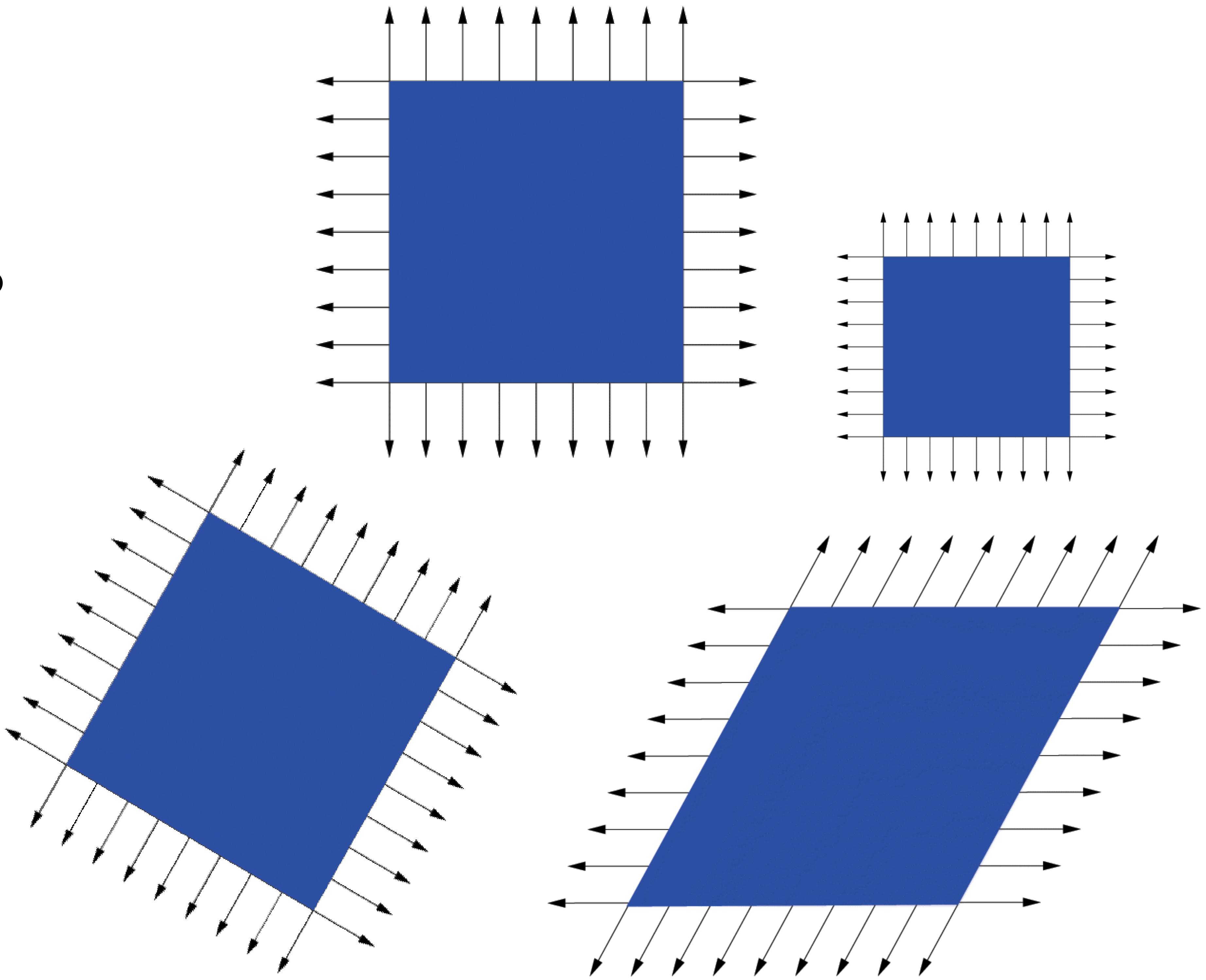
How do we transform normals?



Object Space **World Space**

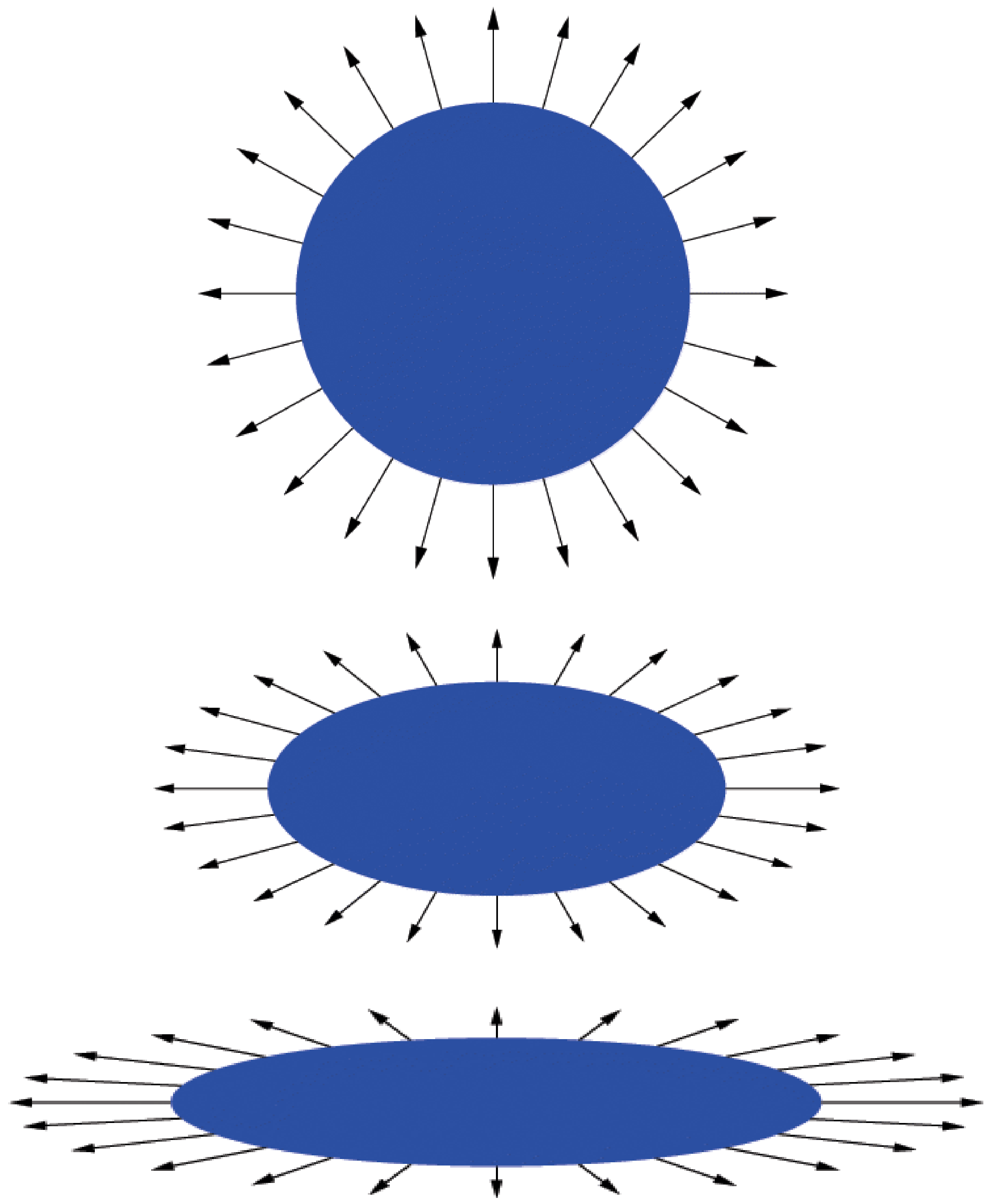
Transform Normal like Object?

- translation?
- rotation?
- isotropic scale?
- scale?
- reflection?
- shear?
- perspective?



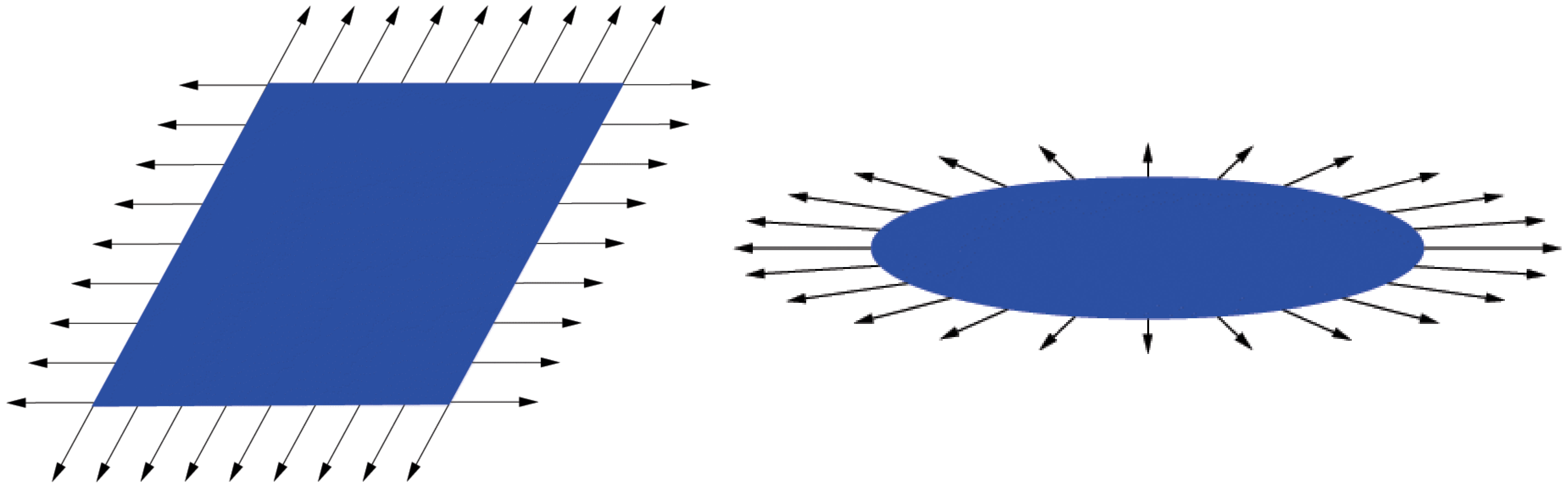
Transform Normal like Object?

- translation?
- rotation?
- isotropic scale?
- scale?
- reflection?
- shear?
- perspective?

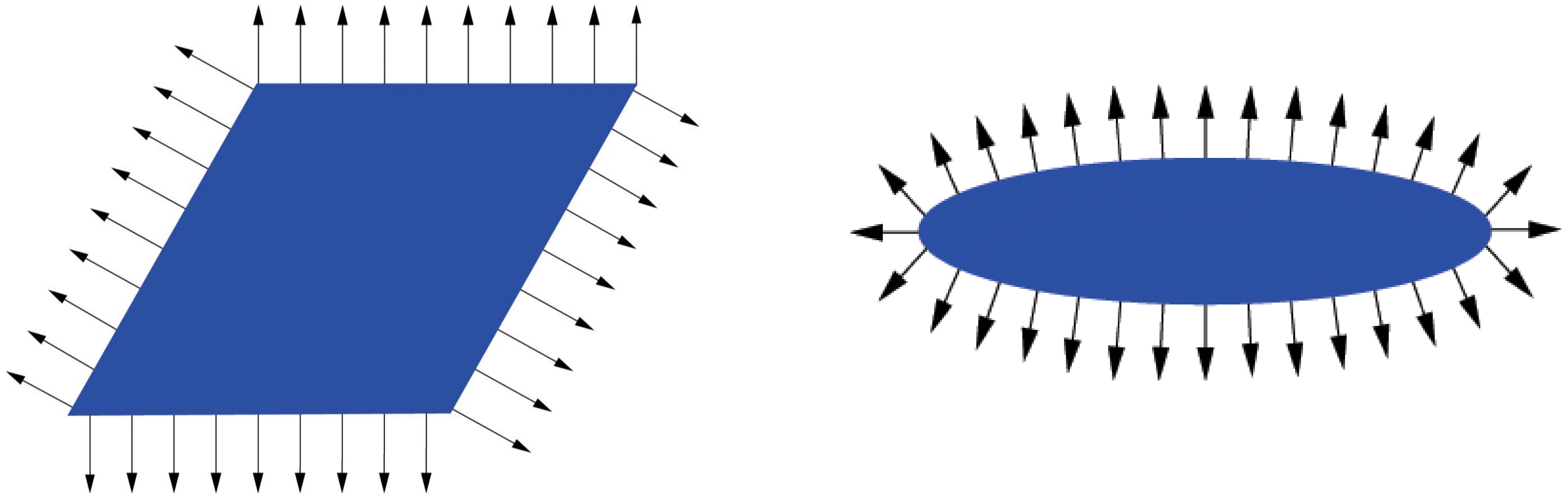


Transformation for shear and scale

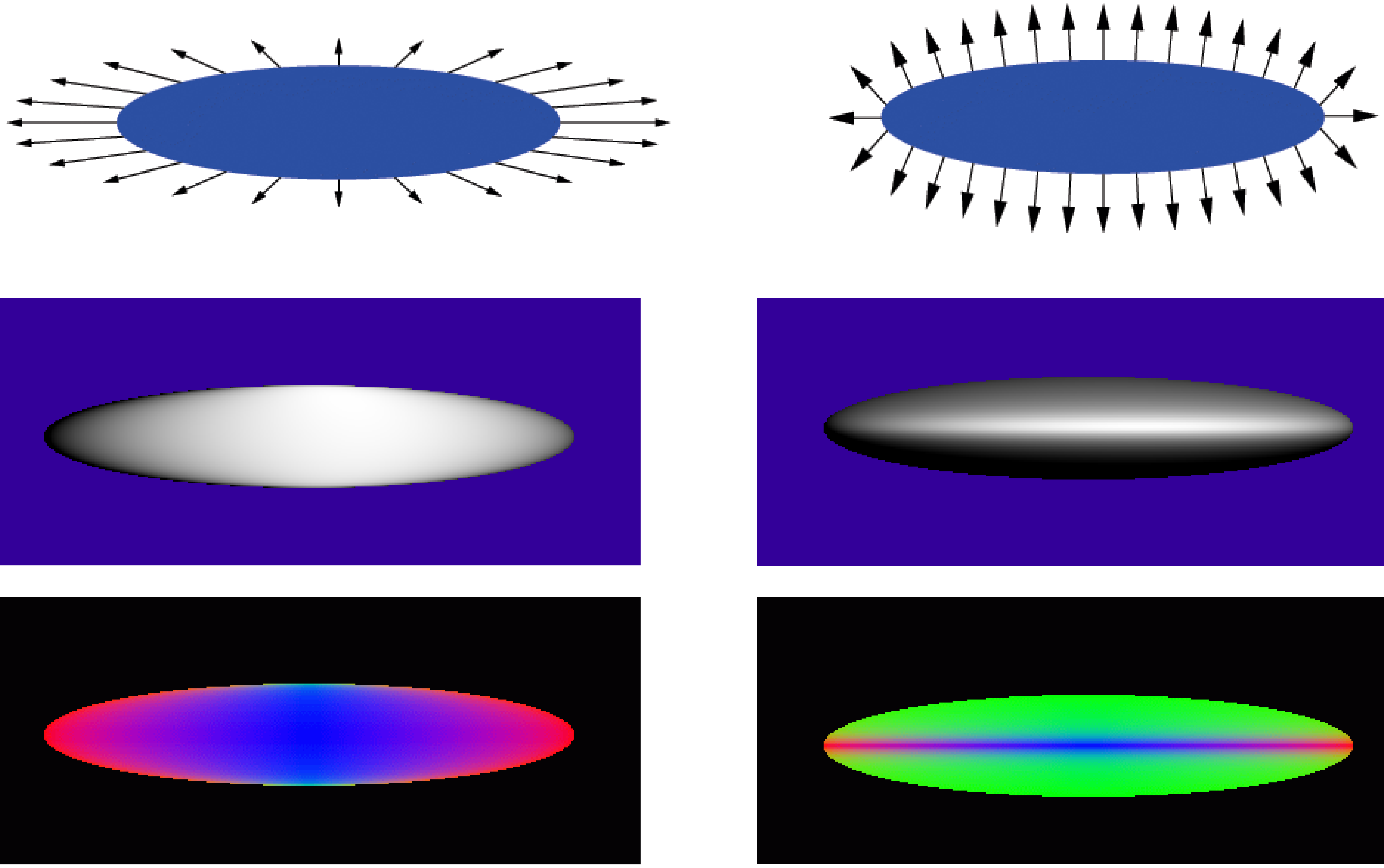
Incorrect
Normal
Transformation



Correct
Normal
Transformation



More Normal Visualizations

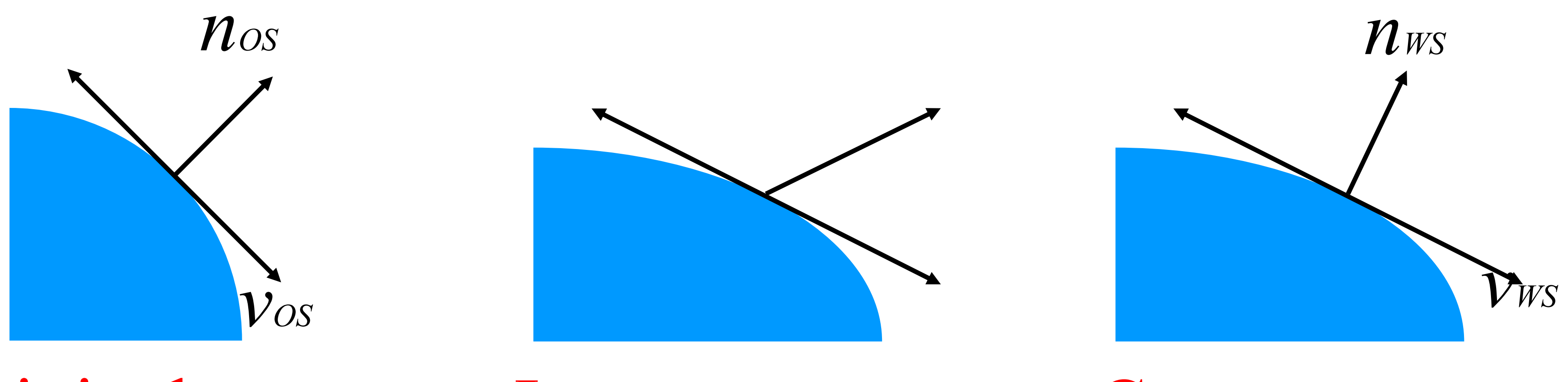


Incorrect Normal Transformation

Correct Normal Transformation

So how do we do it right?

- Think about transforming the *tangent plane* to the normal, not the normal *vector*



Original Incorrect Correct

Pick any vector *vOS* in the tangent plane, how is it transformed by matrix **M**?

$$v_{WS} = \mathbf{M} \, v_{OS}$$

Transform tangent vector *v*

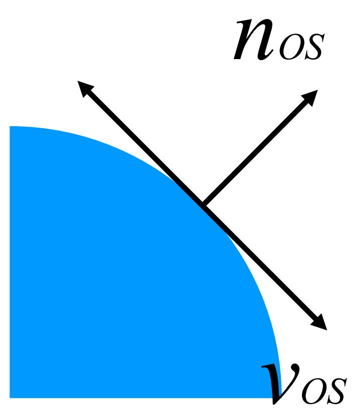
v is perpendicular to normal *n*:

Dot product $n_{os}^{\mathsf{T}}\, v_{os} = 0$

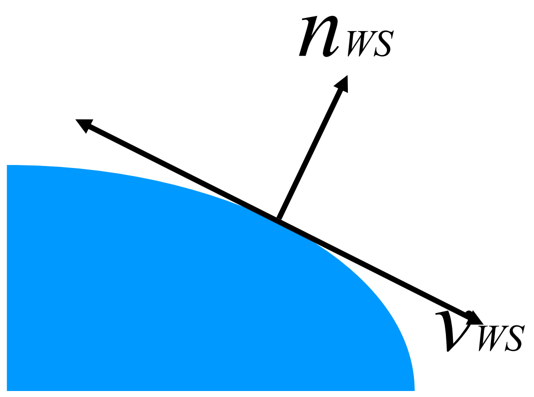
$$n_{os}^{\mathsf{T}}\ (\mathbf{M}^{-1}\ \mathbf{M})\ v_{os} = 0$$

$$(n_{os}^{\mathsf{T}}\ \mathbf{M}^{-1})\ (\mathbf{M}\ \ v_{os}) = 0$$

$$(n_{os}^{\mathsf{T}}\ \mathbf{M}^{-1})\ v_{ws} = 0$$



v_{ws} is perpendicular to normal n_{ws}:



$$n_{ws}^{\mathsf{T}} = n_{os}^{\mathsf{T}}\,(\mathbf{M}^{-1})$$

$$\boxed{n_{ws} = (\mathbf{M}^{-1})^{\mathsf{T}}\, n_{os}}$$

$$n_{ws}^{\mathsf{T}}\, v_{ws} = 0$$

Digression

$$\boxed{n_{WS} = (\mathbf{M}^{-1})^{\mathsf{T}} n_{OS}}$$

- The previous proof is not quite rigorous; first you’d need to prove that tangents indeed transform with **M**.
  - Turns out they do, but we’ll take it on faith here.
  - If you believe that, then the above formula follows.

Comment

- So the correct way to transform normals is:

$$n_{WS} = (\mathbf{M}^{-1})^{\mathsf{T}}\, n_{OS}$$ Sometimes denoted $\mathrm{M}^{-\mathrm{T}}$

- But why did $n_{WS} = \mathbf{M}\, n_{OS}$ work for similitudes?
- Because for similitude / similarity transforms,

$$(\mathbf{M}^{-1})^{\mathsf{T}} = \lambda\, \mathbf{M}$$

- e.g. for orthonormal basis:

$$\mathbf{M}^{-1} = \mathbf{M}^{\mathsf{T}} \quad \text{i.e.} \quad (\mathbf{M}^{-1})^{\mathsf{T}} = \mathbf{M}$$

Connections

- Not part of class, but cool
  - “Covariant”: transformed by the matrix
    - e.g., tangent
  - “Contravariant”: transformed by the inverse transpose
    - e.g., the normal
    - a normal is a “co-vector”

- Google “differential geometry” to find out more

- Further Reading
  - Buss, Chapter 2

- Other Cool Stuff
  - Algebraic Groups
  - http://phototour.cs.washington.edu/
  - http://phototour.cs.washington.edu/findingpaths/
  - Free-form deformation of solid objects
  - Harmonic coordinates for character articulation

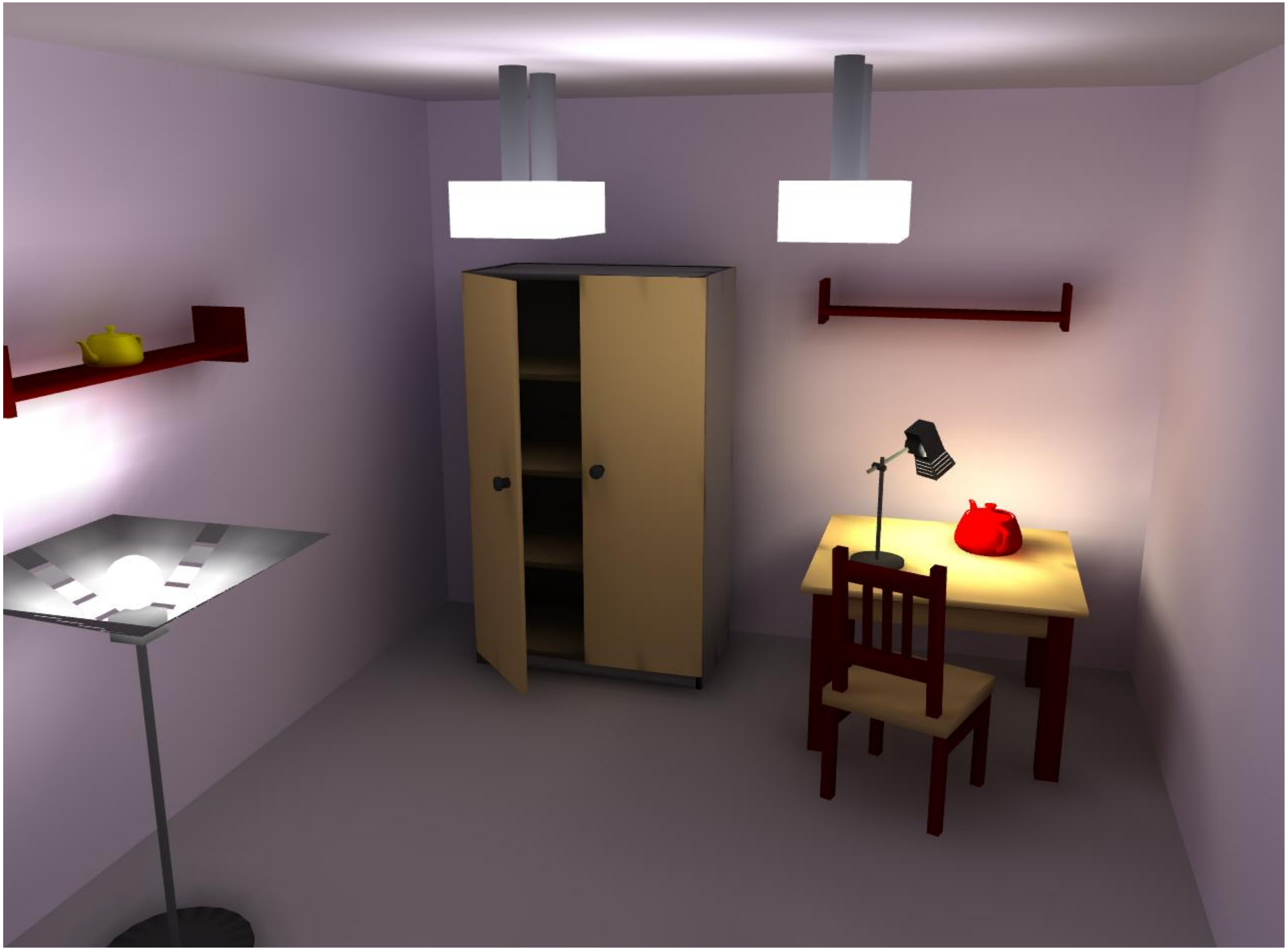
Question?

Hierarchical Modeling

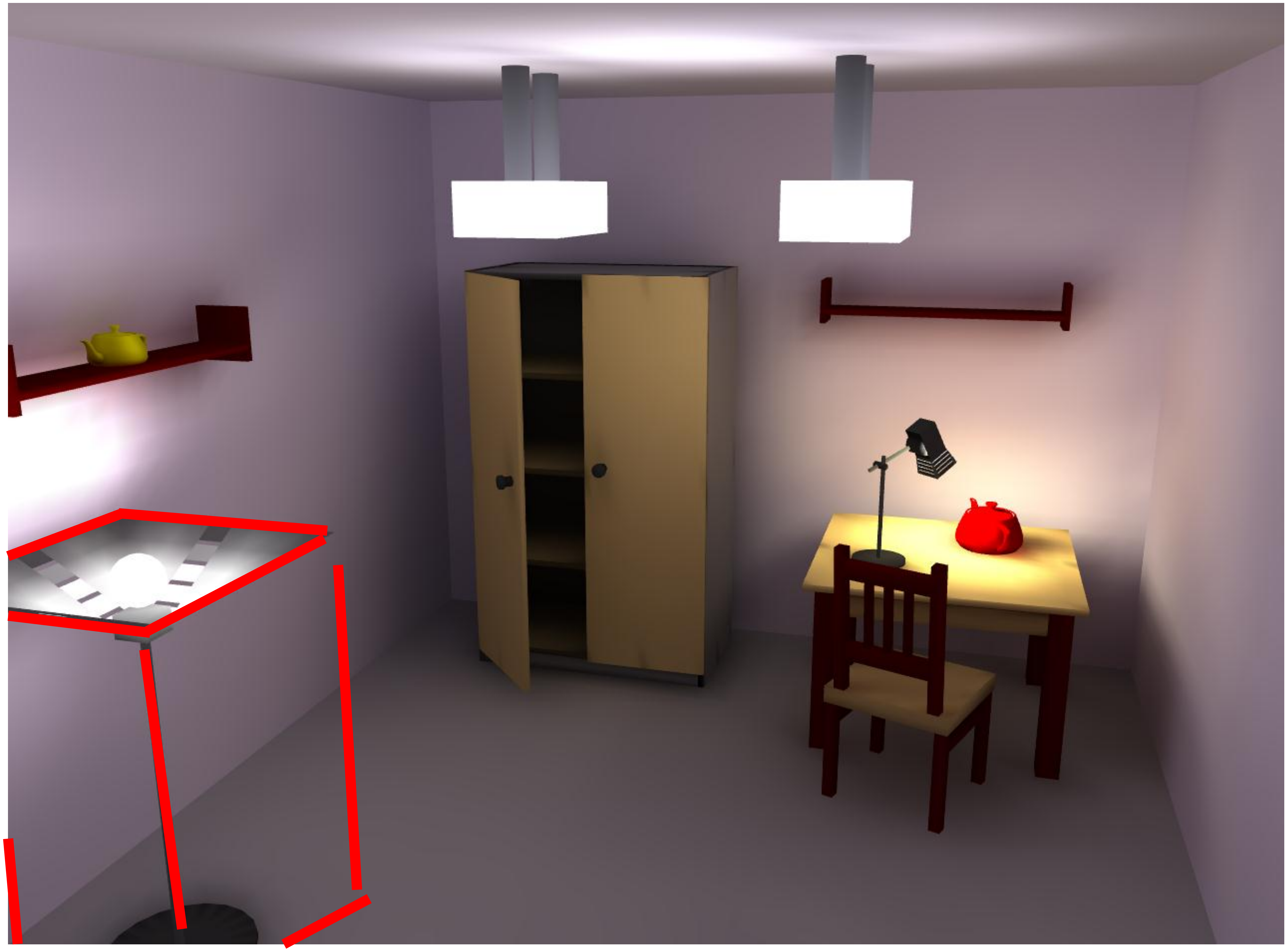
- Triangles, parametric curves and surfaces are the building blocks from which more complex real-world objects are modeled.

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

Hierarchical models



Hierarchical models



Hierarchical models

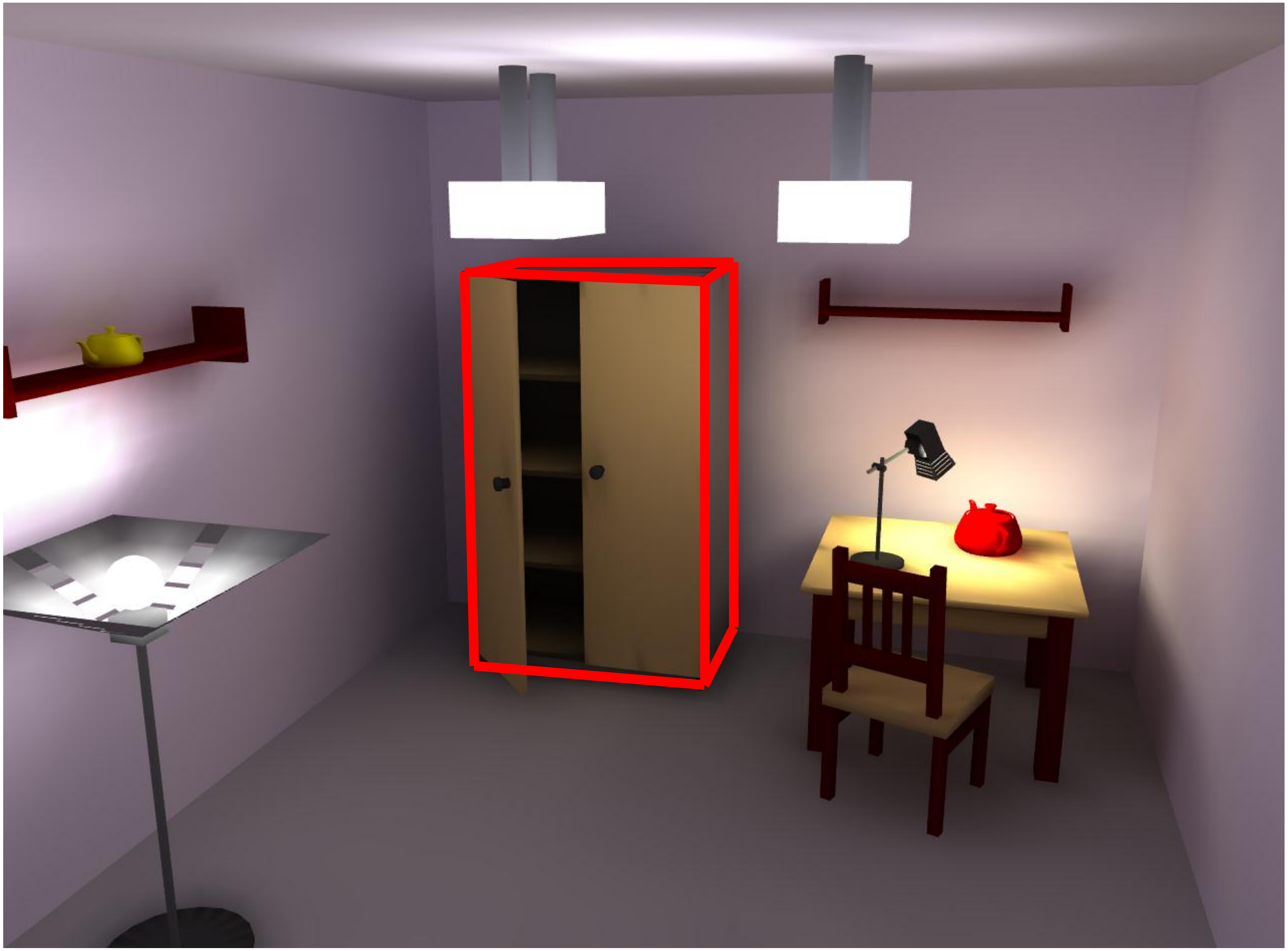
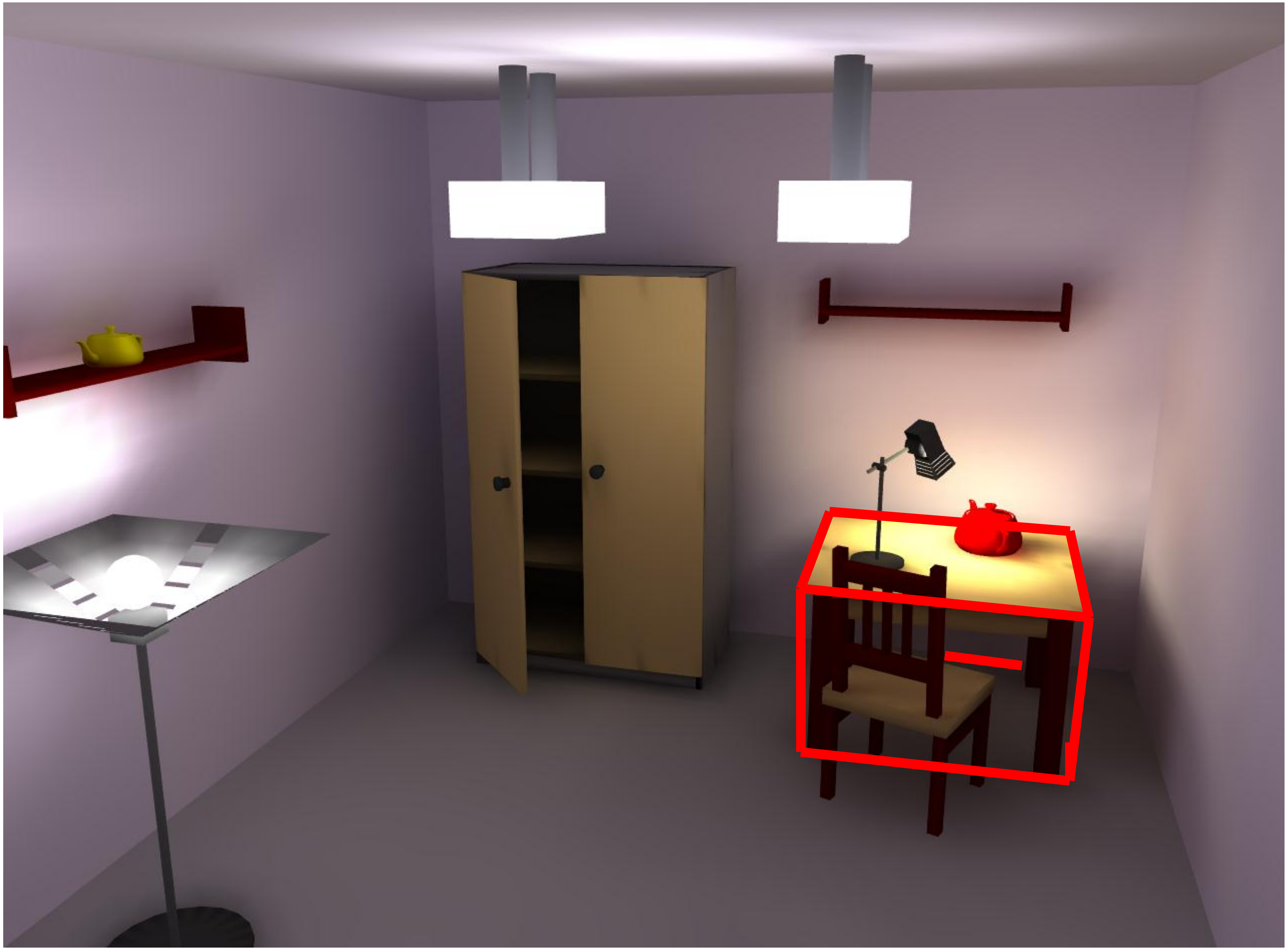
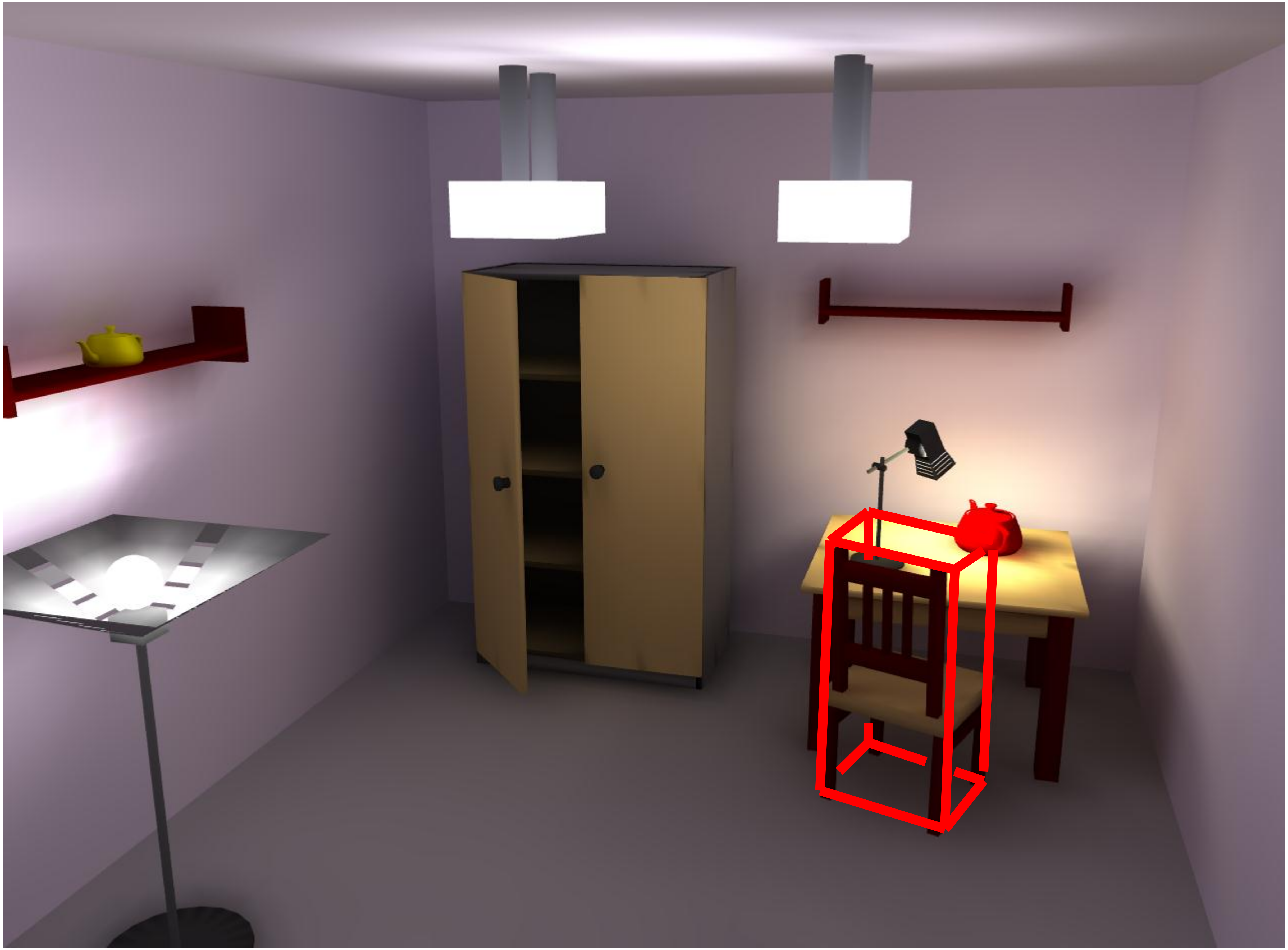


Image courtesy of David Bařina, Kamil Dudka, Jakub Filák, Lukáš Hefka on Wikimedia Commons. License: CC-BY-SA. This content is excluded from our Creative Commons license. For more information, see http://ocw.mit.edu/help/faq-fair-use/.

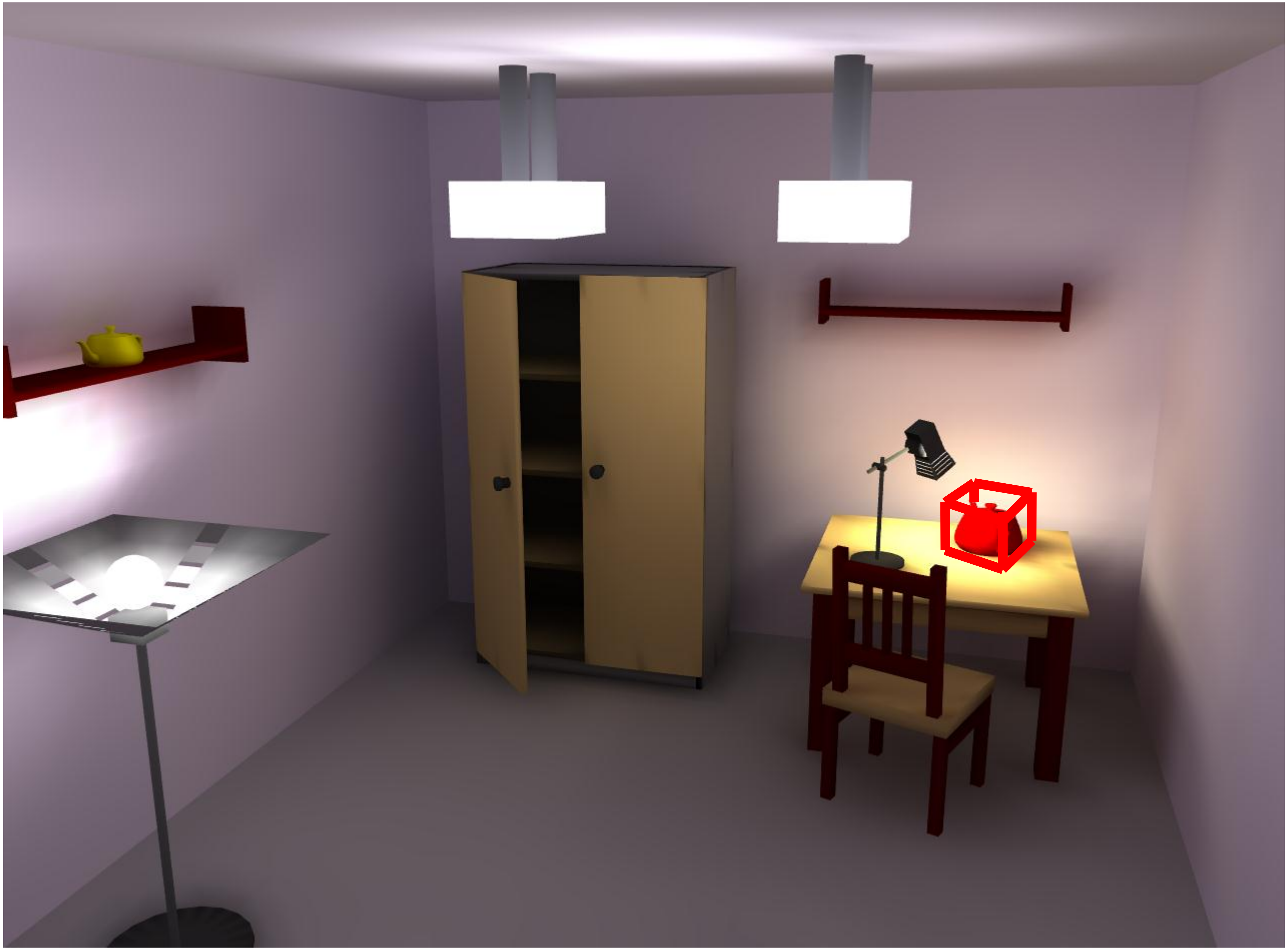
Hierarchical models



Hierarchical models

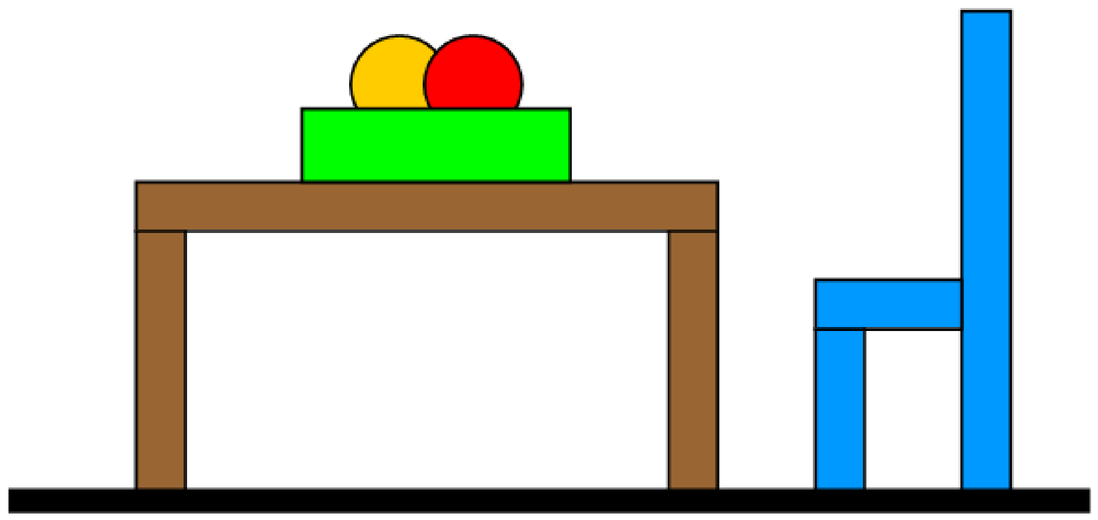


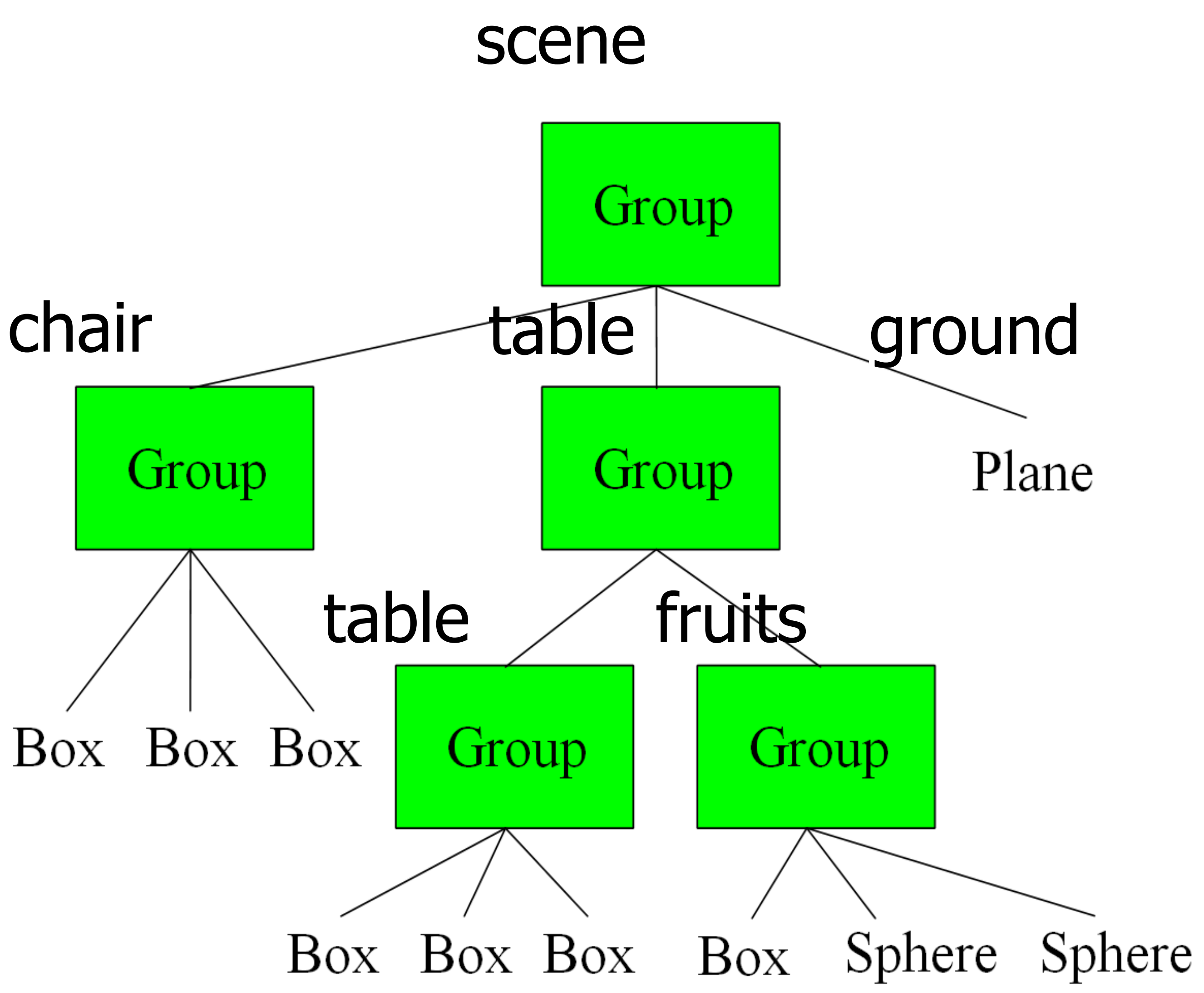
Hierarchical models



Hierarchical Grouping of Objects

- The "scene graph" represents the logical organization of scene

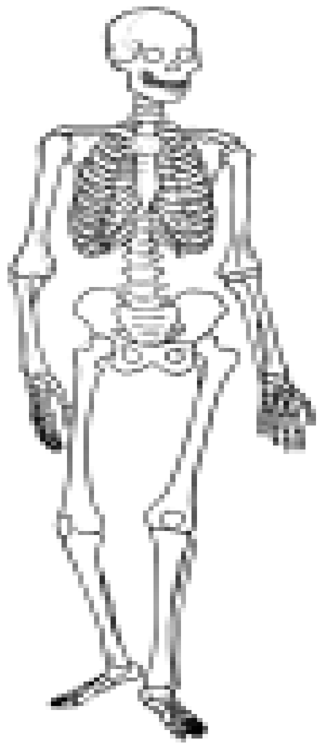




Scene Graph

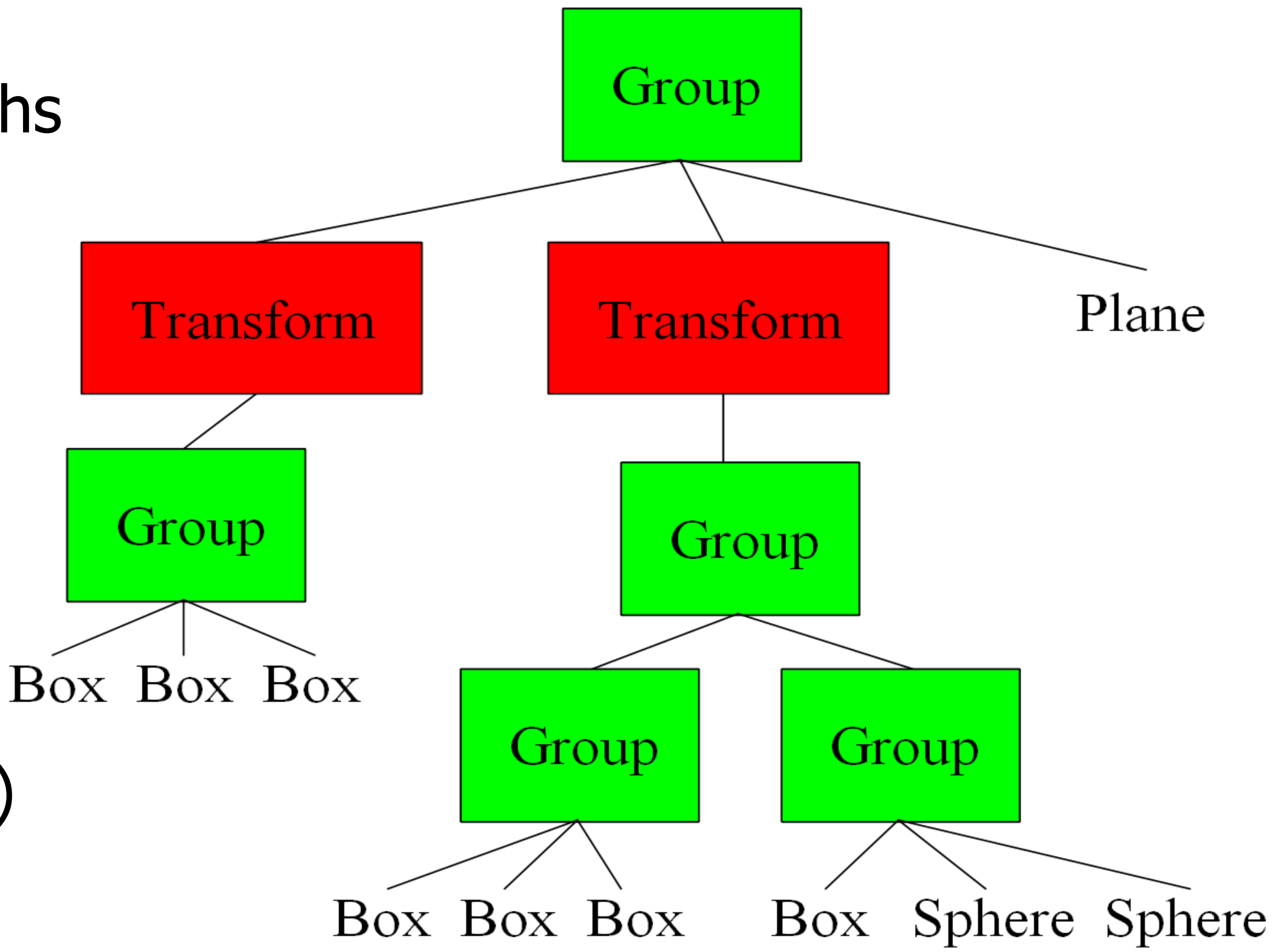
- Convenient Data structure for scene representation
  - Geometry (meshes, etc.)
  - Transformations
  - Materials, color
  - Multiple instances

- Basic idea: Hierarchical Tree
- Useful for manipulation/animation
  - Also for articulated figures
- Useful for rendering, too
  - Ray tracing acceleration, occlusion culling
  - But note that two things that are close to each other in the tree are NOT necessarily spatially near each other



Scene Graph Representation

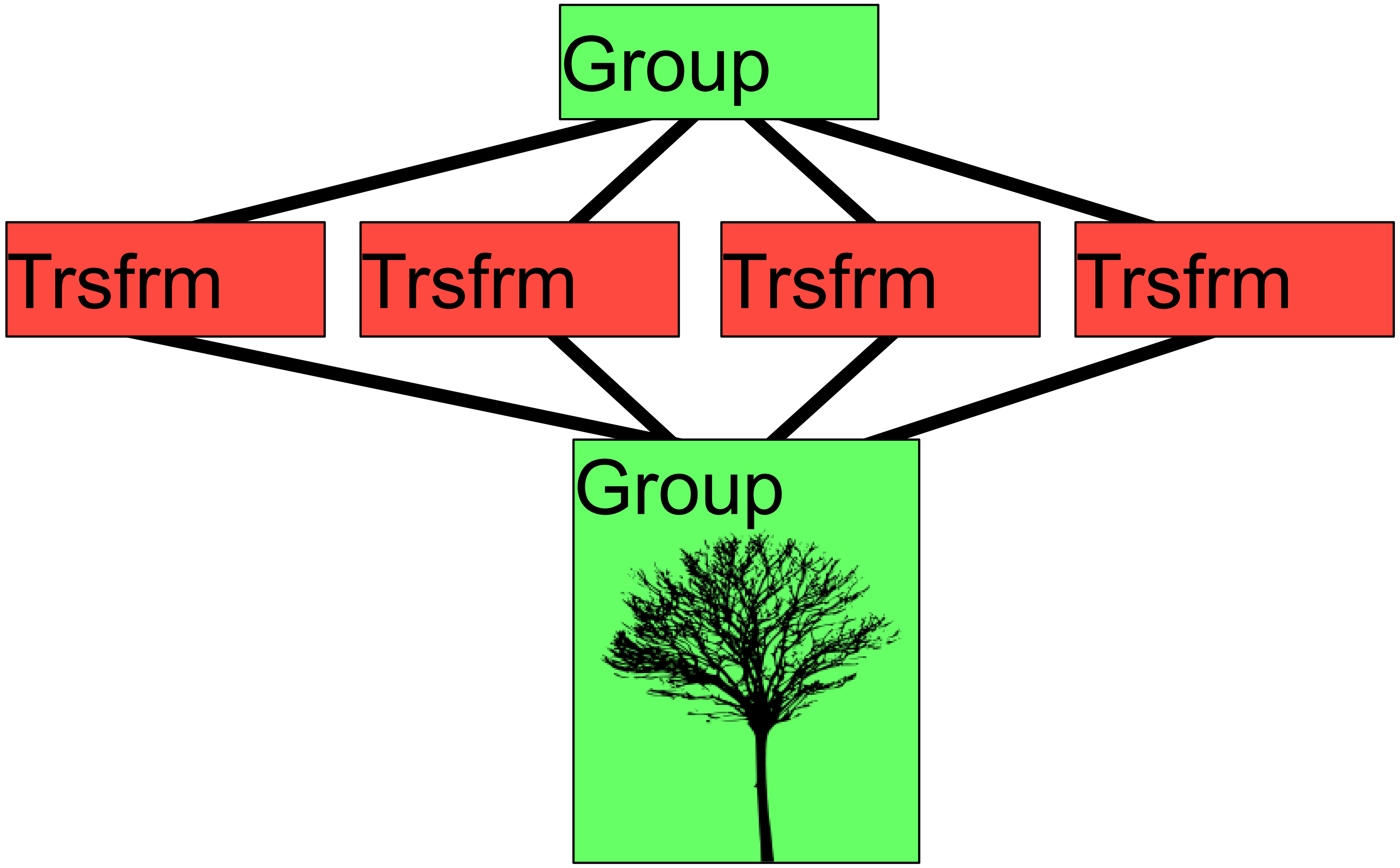
- Basic idea: Tree
- Comprised of several node types
  - Shape: 3D geometric objects
  - Transform: Affect current transformation
  - Property: Color, texture
  - Group: Collection of subgraphs

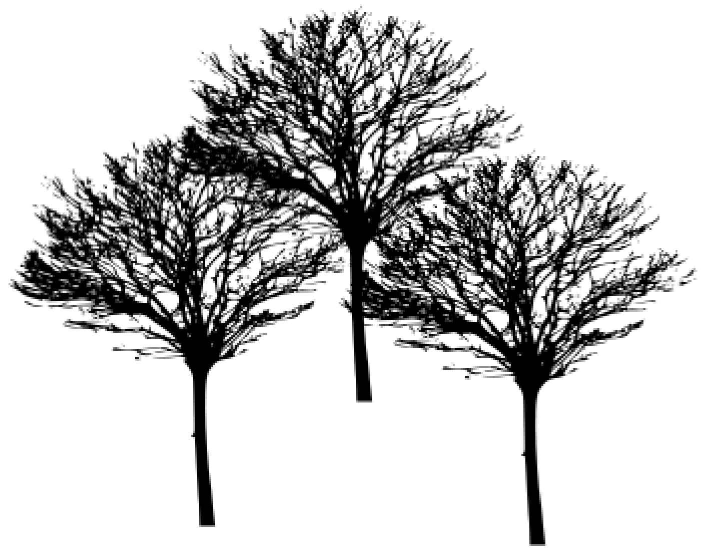


- C++ implementation
  - base class Object
    - children, parent
  - derived classes for each node type (group, transform)

Scene Graph Representation

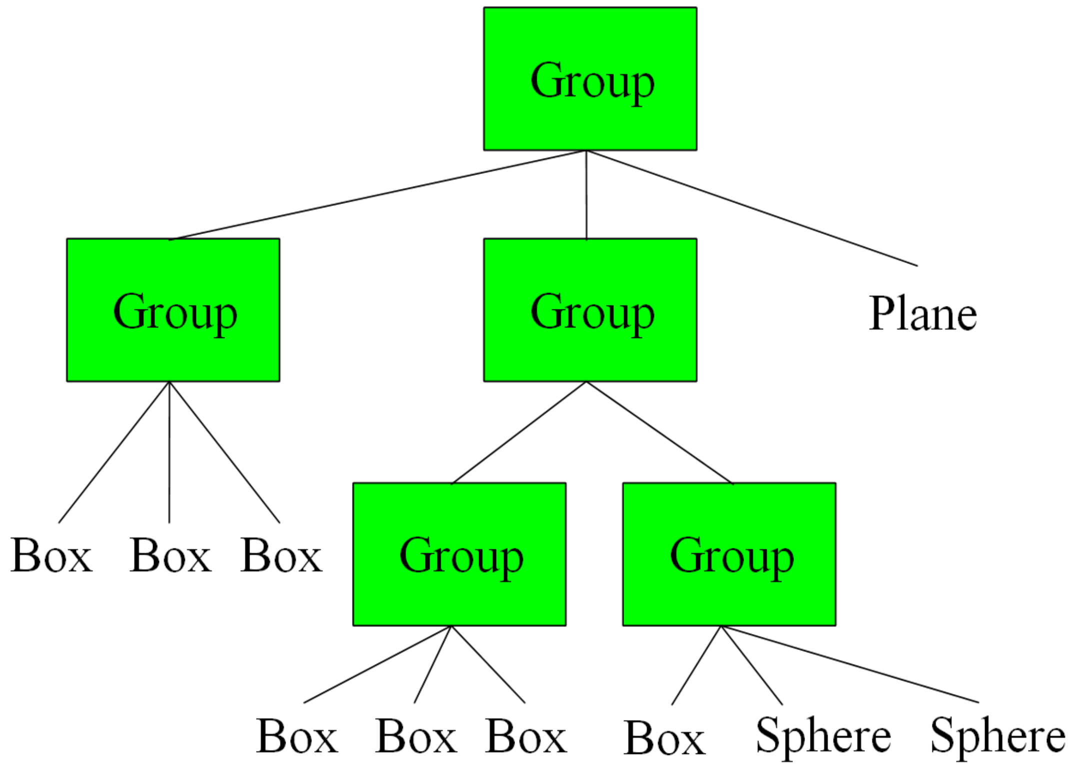
- In fact, generalization of a tree: Directed Acyclic Graph (DAG)
  - Means a node can have multiple parents, but cycles are not allowed
- Why? Allows multiple instantiations
  - Reuse complex hierarchies many times in the scene using different transformations (example: a tree)
    - Of course, if you only want to reuse meshes, just load the mesh once and make several geometry nodes point to the same data

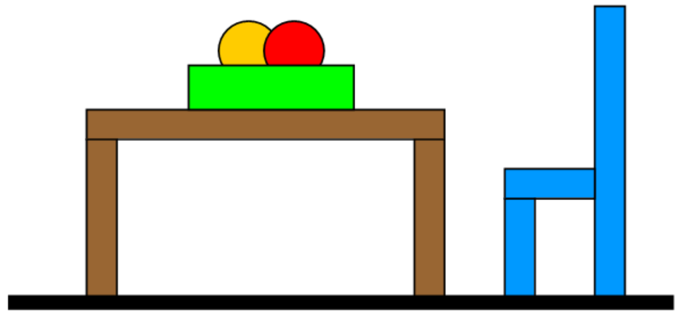




Simple Example with Groups

```
Group {
    numObjects 3
    Group {
        numObjects 3
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> } }
    Group {
        numObjects 2
        Group {
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> } }
        Group {
            Box { <BOX PARAMS> }
            Sphere { <SPHERE PARAMS> }
            Sphere { <SPHERE PARAMS> } } }
    Plane { <PLANE PARAMS> } }
```

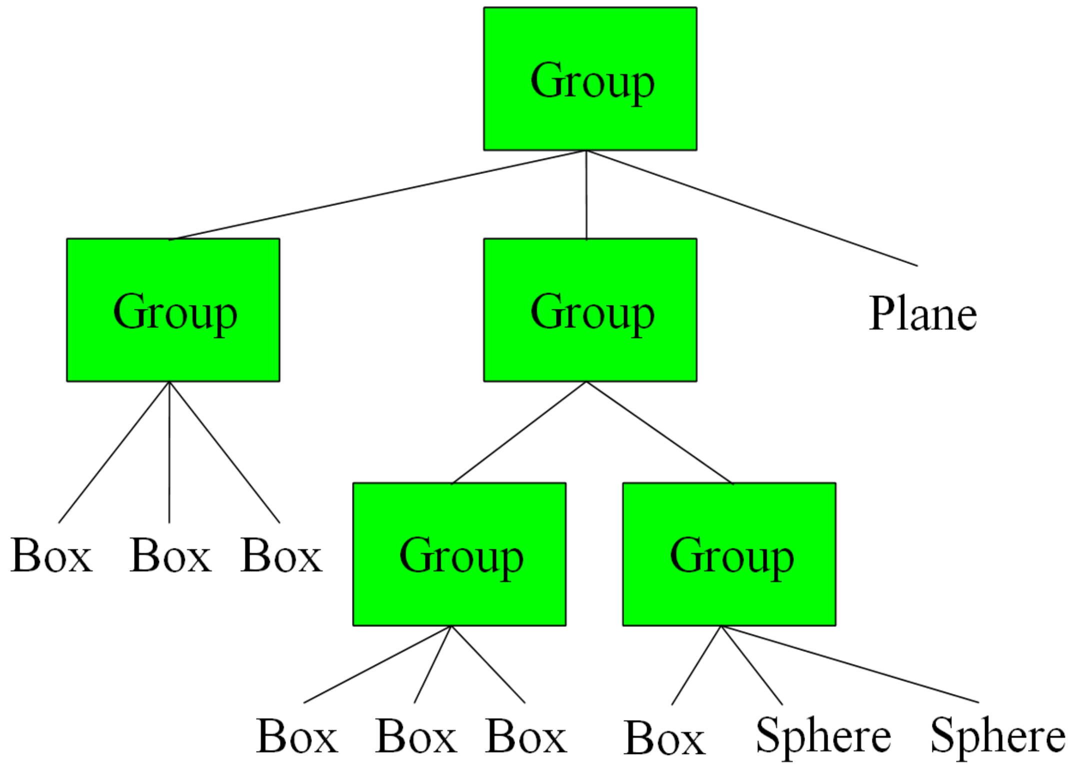


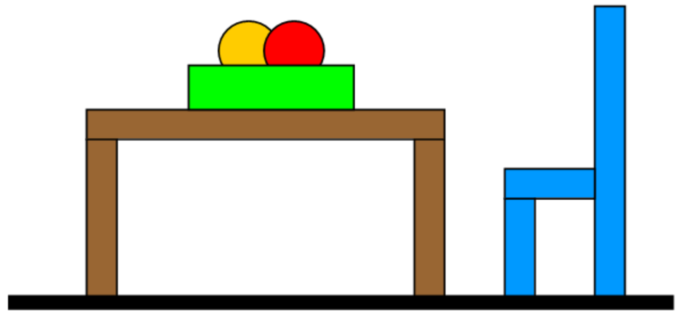


Text format is fictitious, better to use XML in real applications

Simple Example with Groups

```
Group {
    numObjects 3
    Group {
        numObjects 3
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> } }
    Group {
        numObjects 2
        Group {
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> } }
        Group {
            Box { <BOX PARAMS> }
            Sphere { <SPHERE PARAMS> }
            Sphere { <SPHERE PARAMS> } } }
    Plane { <PLANE PARAMS> } }
```

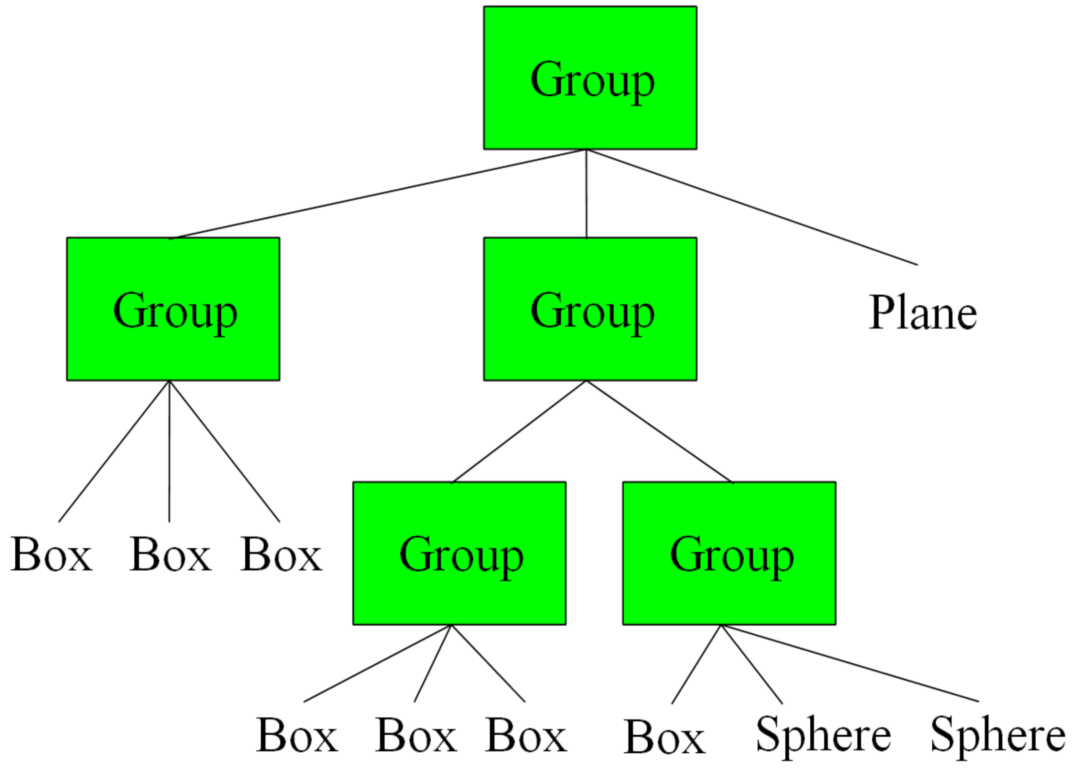


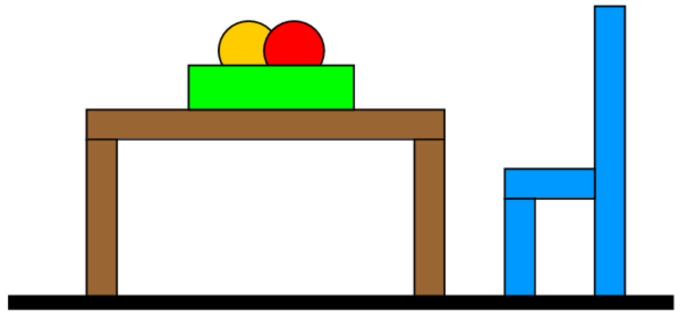


Text format is fictitious, better to use XML in real applications

Simple Example with Groups

```
Group {
    numObjects 3
    Group {
        numObjects 3
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> } }
    Group {
        numObjects 2
        Group {
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> } }
        Group {
            Box { <BOX PARAMS> }
            Sphere { <SPHERE PARAMS> }
            Sphere { <SPHERE PARAMS> } } }
    Plane { <PLANE PARAMS> } }
```



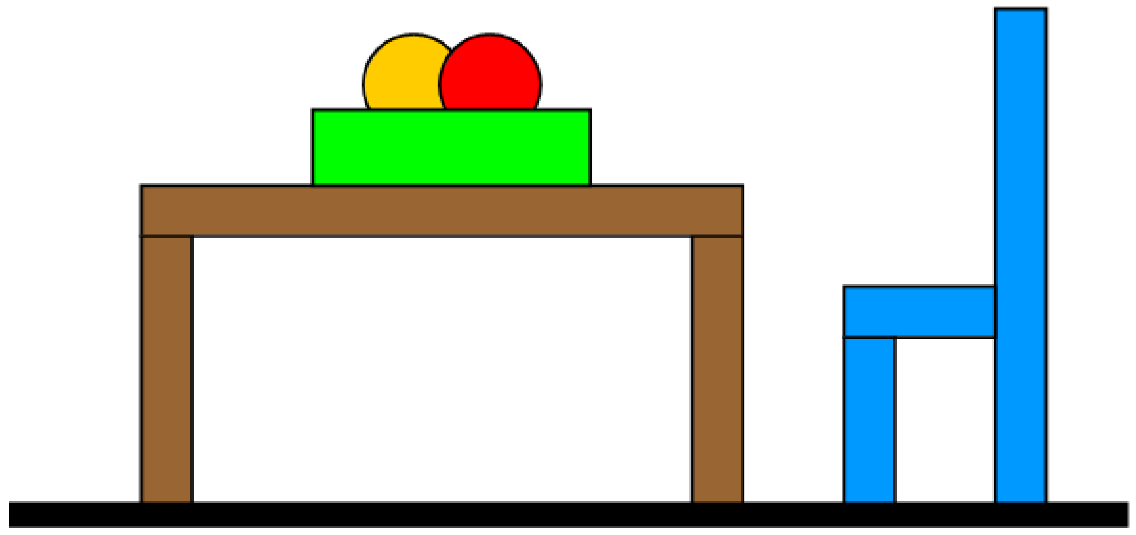


Here we have only simple shapes, but easy to add a “Mesh” node whose parameters specify an .OBJ to load (say)

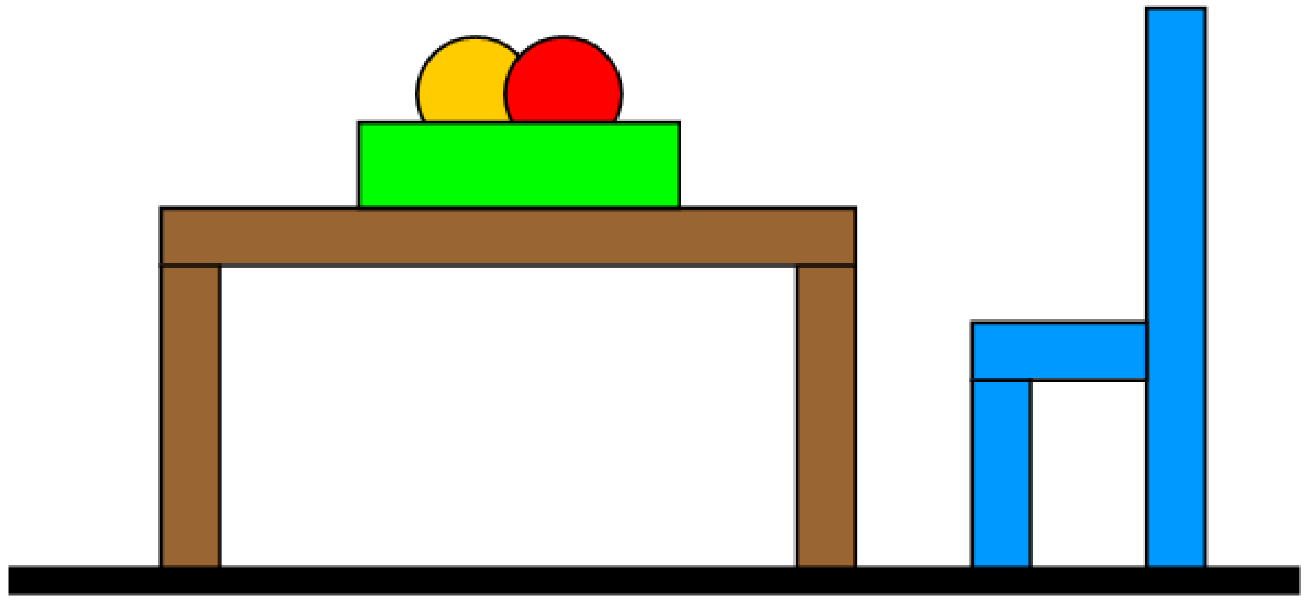
Adding Attributes (Material, etc.)

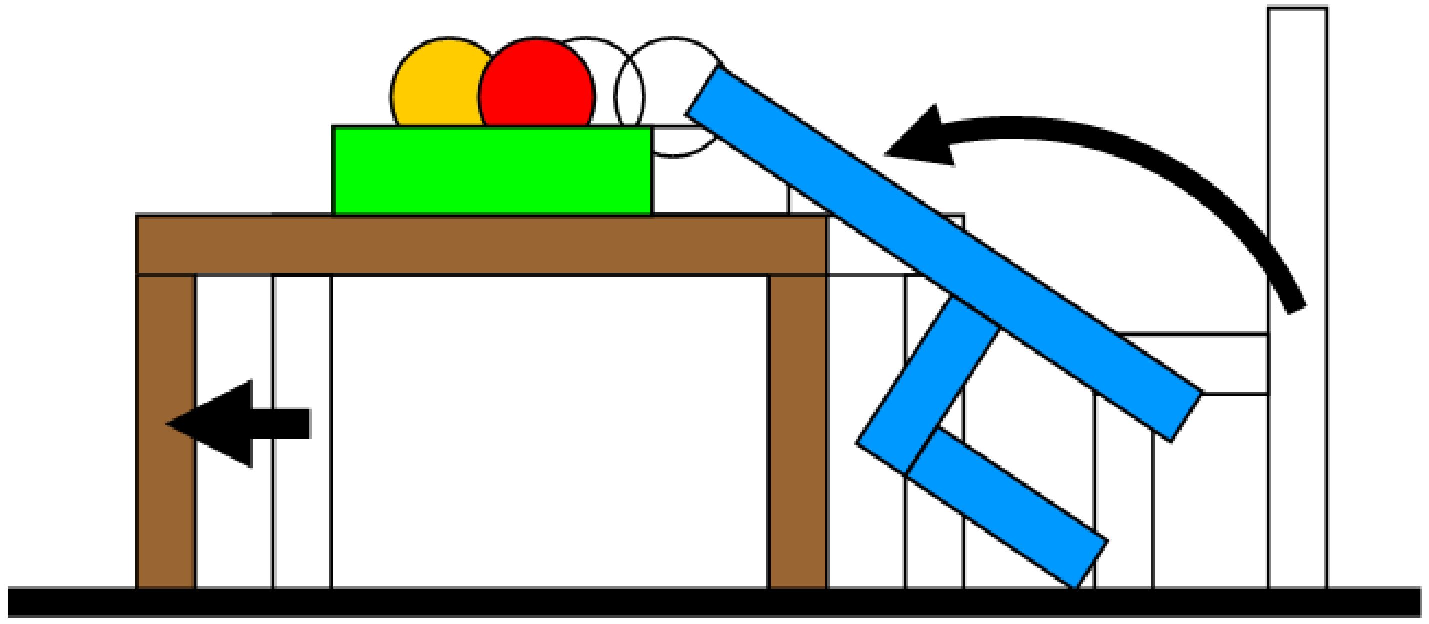
```
Group {
    numObjects 3
    Material { <BLUE> }
    Group {
        numObjects 3
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> }
        Box { <BOX PARAMS> } }
    Group {
        numObjects 2
        Material { <BROWN> }
        Group {
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> }
            Box { <BOX PARAMS> } }
        Group {
            Material { <GREEN> }
            Box { <BOX PARAMS> }
            Material { <RED> }
            Sphere { <SPHERE PARAMS> }
            Material { <ORANGE> }
            Sphere { <SPHERE PARAMS> } } }

    Plane { <PLANE PARAMS> } }
```



Adding Transformations



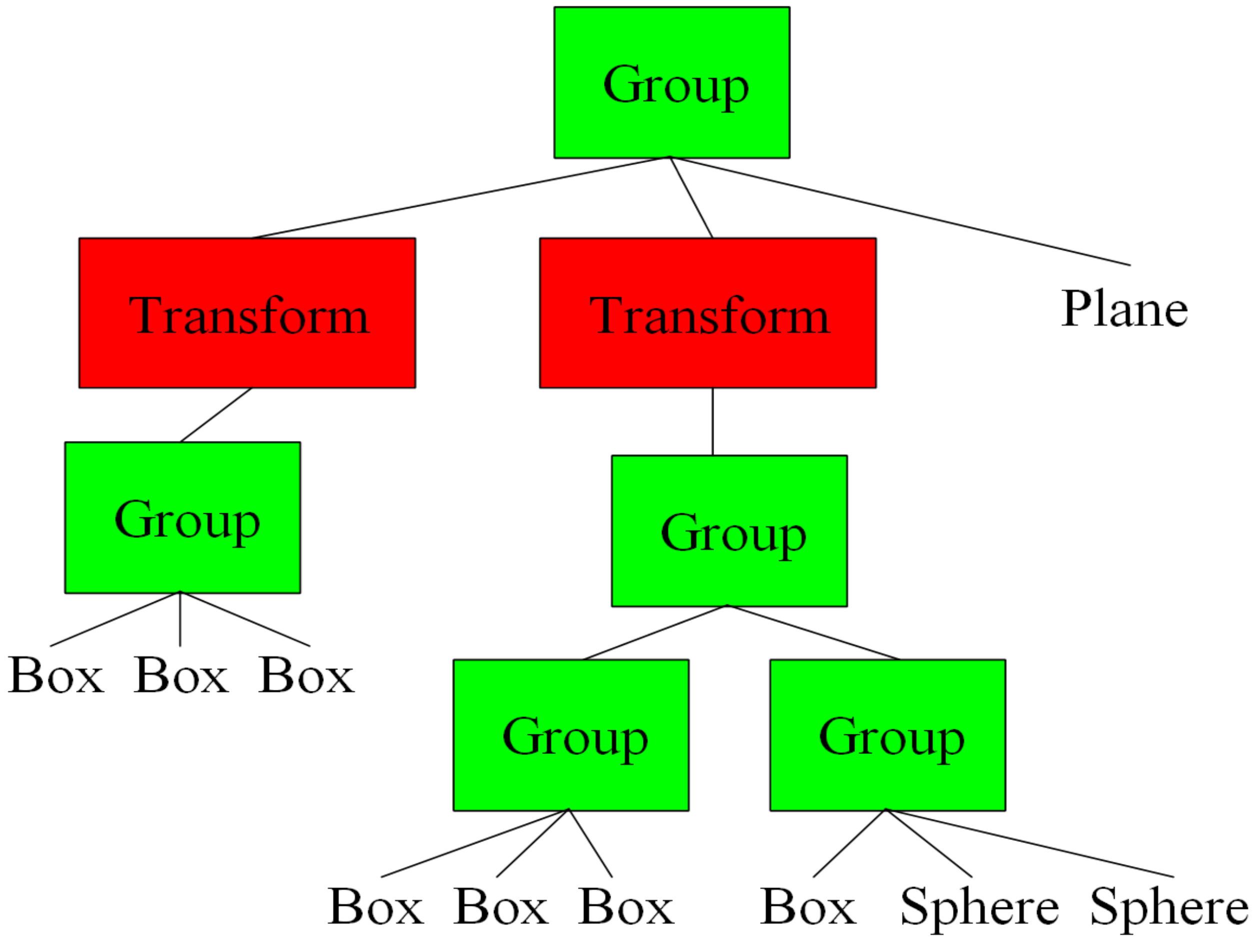


Questions?

Scene Graph Traversal

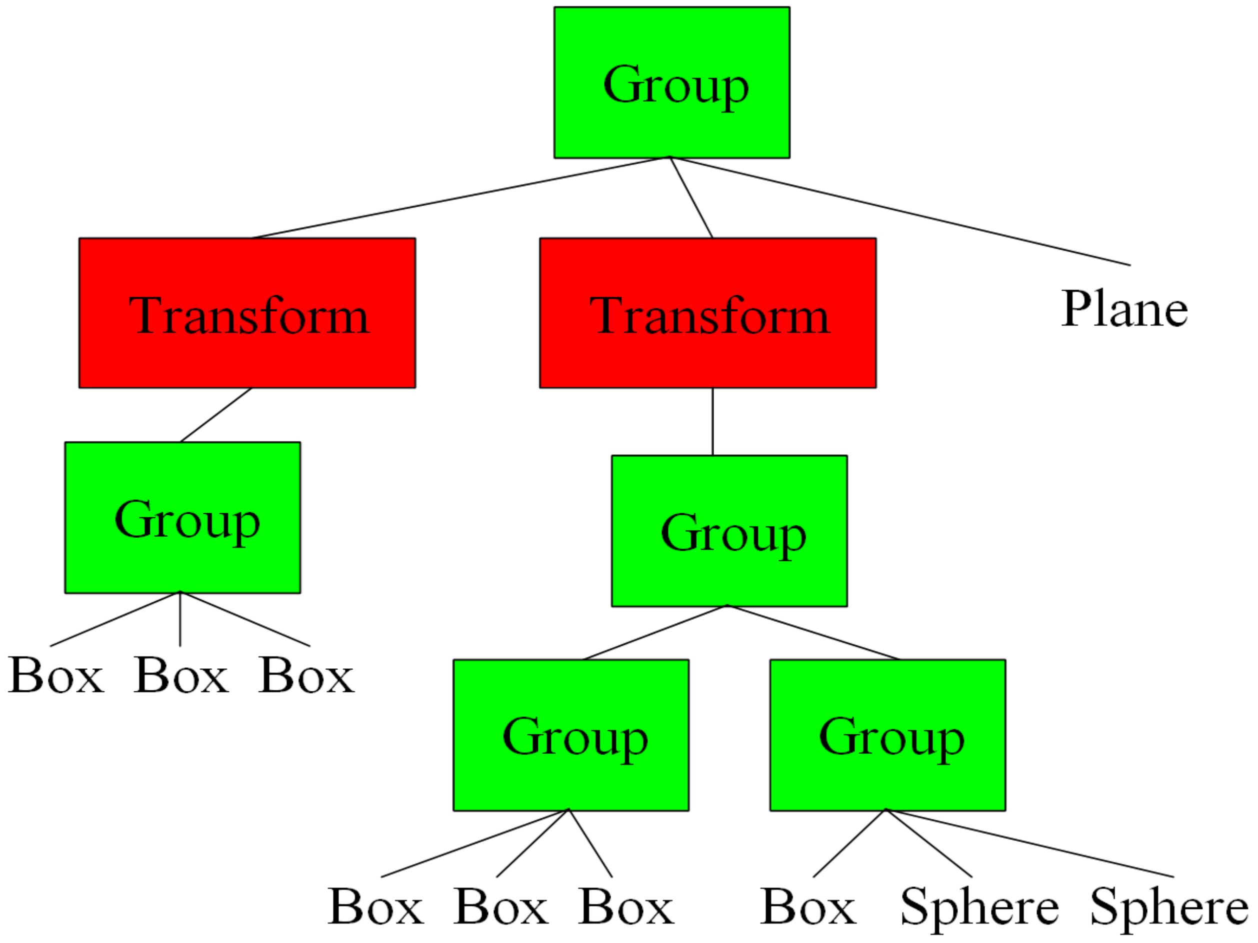
- Depth first recursion
  - Visit node, then visit subtrees (top to bottom, left to right)
  - When visiting a geometry node: Draw it!

- How to handle transformations?
  - Remember, transformations are 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)
    - from world coordinates
    - Initialized to identity 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!**



Recall frames

- An object frame has coordinates O in the world
  (of course O is also our 4x4 matrix)

$$\vec{\mathbf{o}}^t = \vec{\mathbf{w}}^t O$$

- Then we are given coordinates c in the object frame

$$\vec{\mathbf{o}}^t \mathbf{c} = \vec{\mathbf{w}}^t O \mathbf{c}$$

- Indeed we need to apply matrix O to all objects

Frames and hierarchy

- Matrix *M1* to go from world to torso $\vec{\mathbf{t}}^t = \vec{\mathbf{w}}^t M_1$
- Matrix *M2* to go from torso to arm $\vec{\mathbf{a}}^t = \vec{\mathbf{t}}^t M_2$

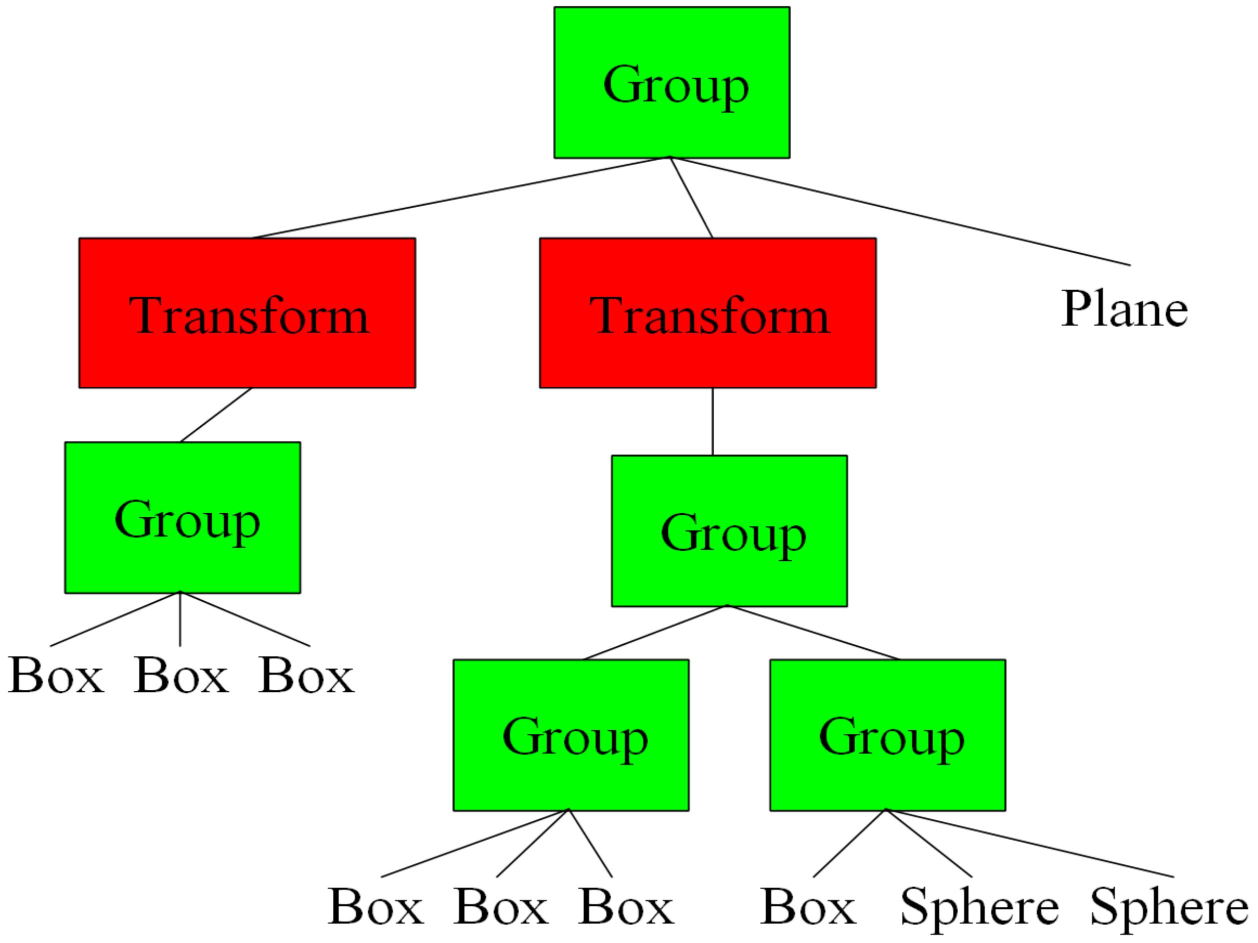
- How do you go from arm coordinates to world?

$$\vec{\mathbf{a}}^t \mathbf{c} = \vec{\mathbf{t}}^t M_2 \mathbf{c} = \vec{\mathbf{w}}^t M_1 M_2 \mathbf{c}$$

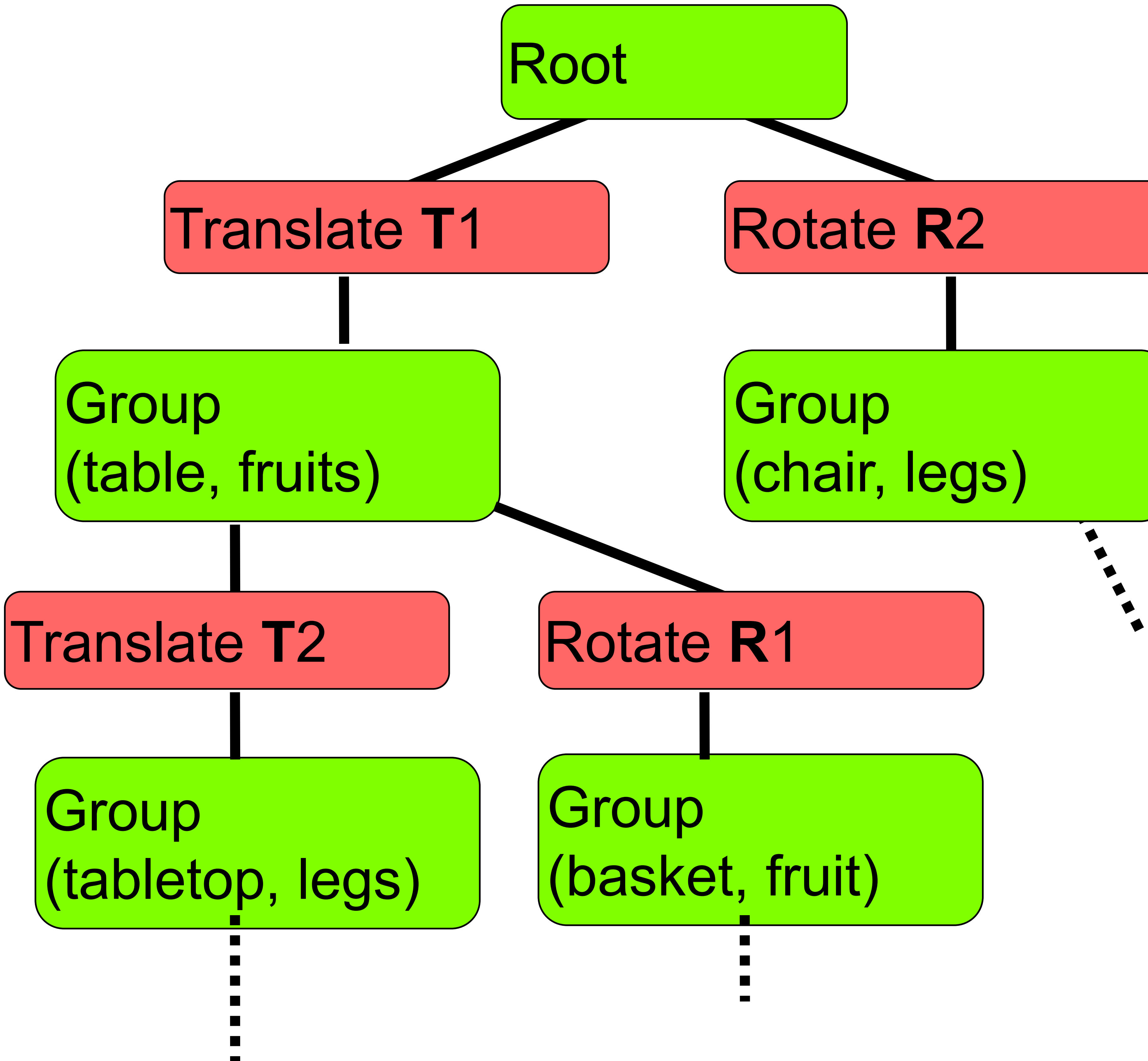
- We can concatenate the matrices
- Matrices for the lower hierarchy nodes go to the right

Recap: Scene Graph Traversal

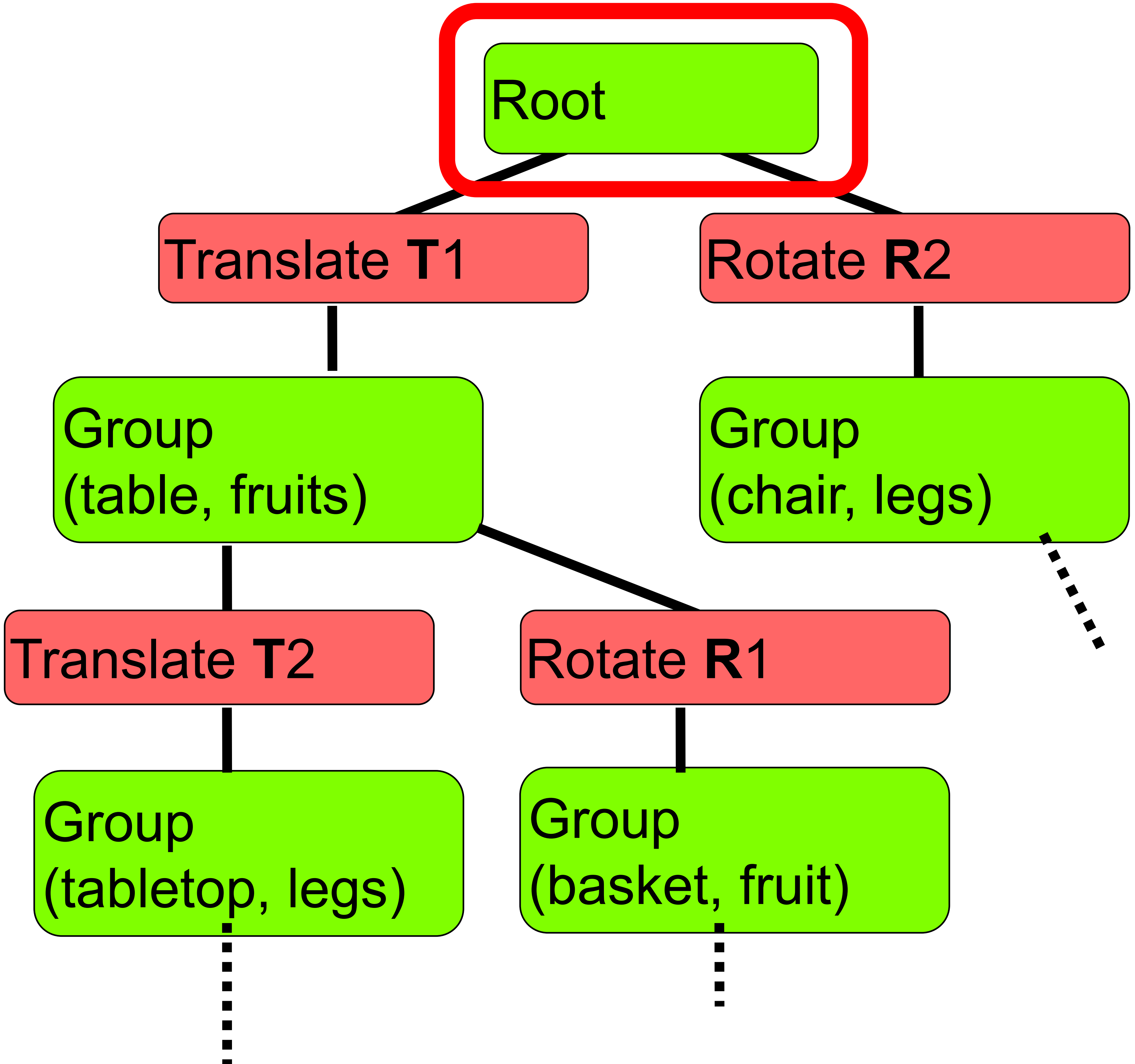
- How to handle transformations?
  - Traversal algorithm keeps a **transformation state S** (a 4x4 matrix)
    - from world coordinates
    - Initialized to identity in the beginning
  - Geometry nodes always drawn using current **S**
  - When visiting a transformation node **T**: multiply current state **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

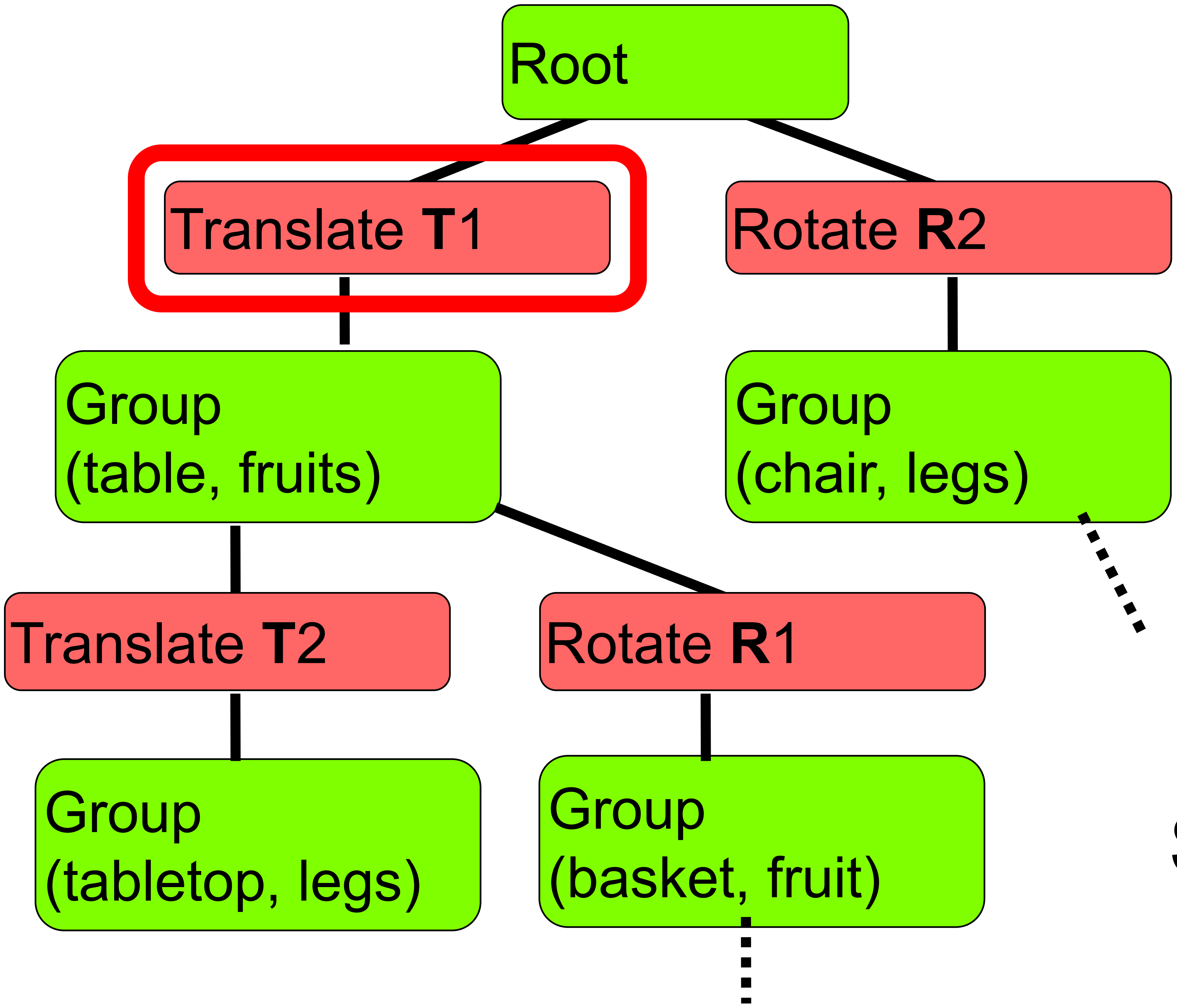


Traversal Example



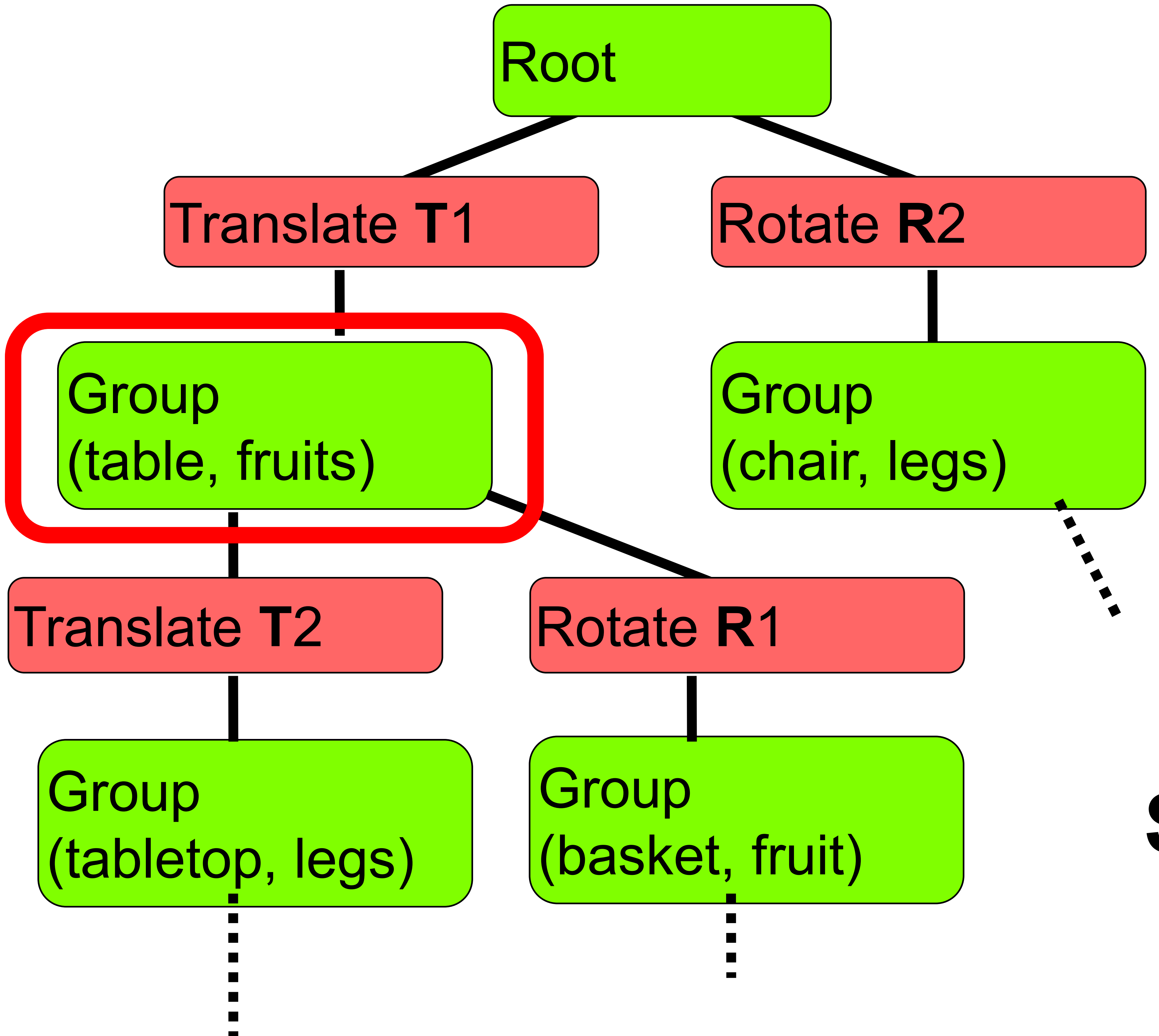
$\mathbf{S} = \mathbf{I}$

Traversal Example



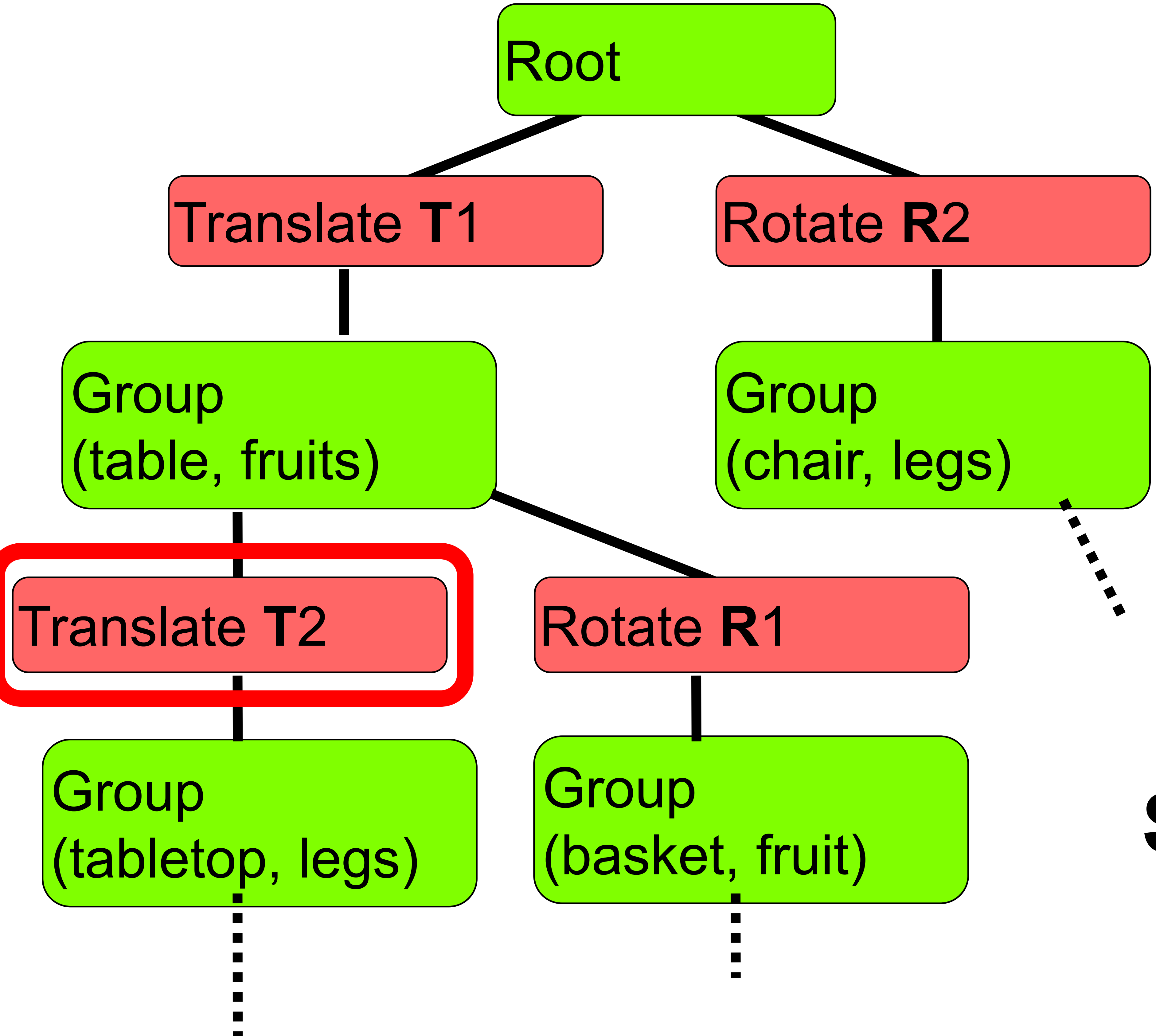
S = **T1**

Traversal Example



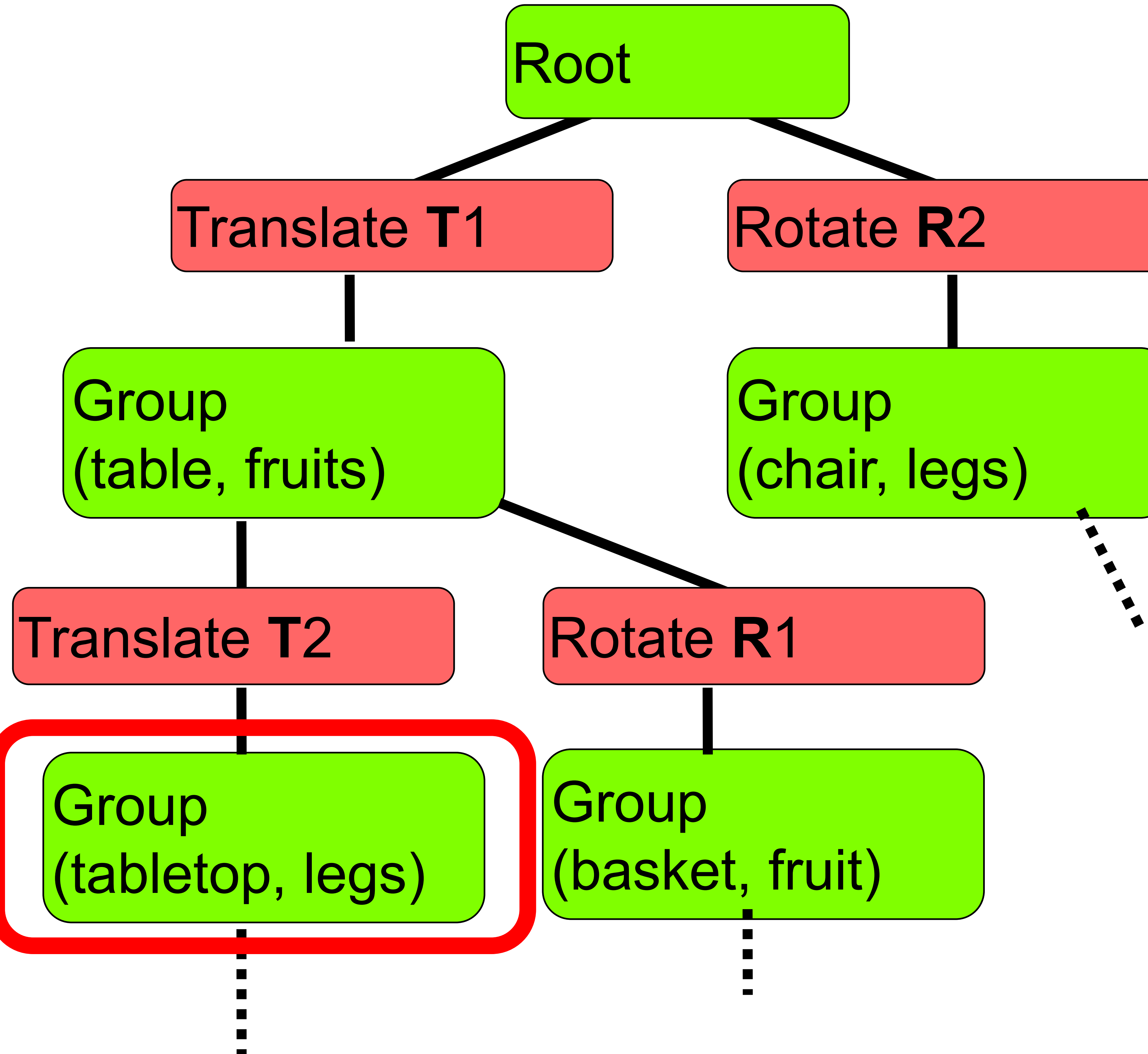
S = **T1**

Traversal Example



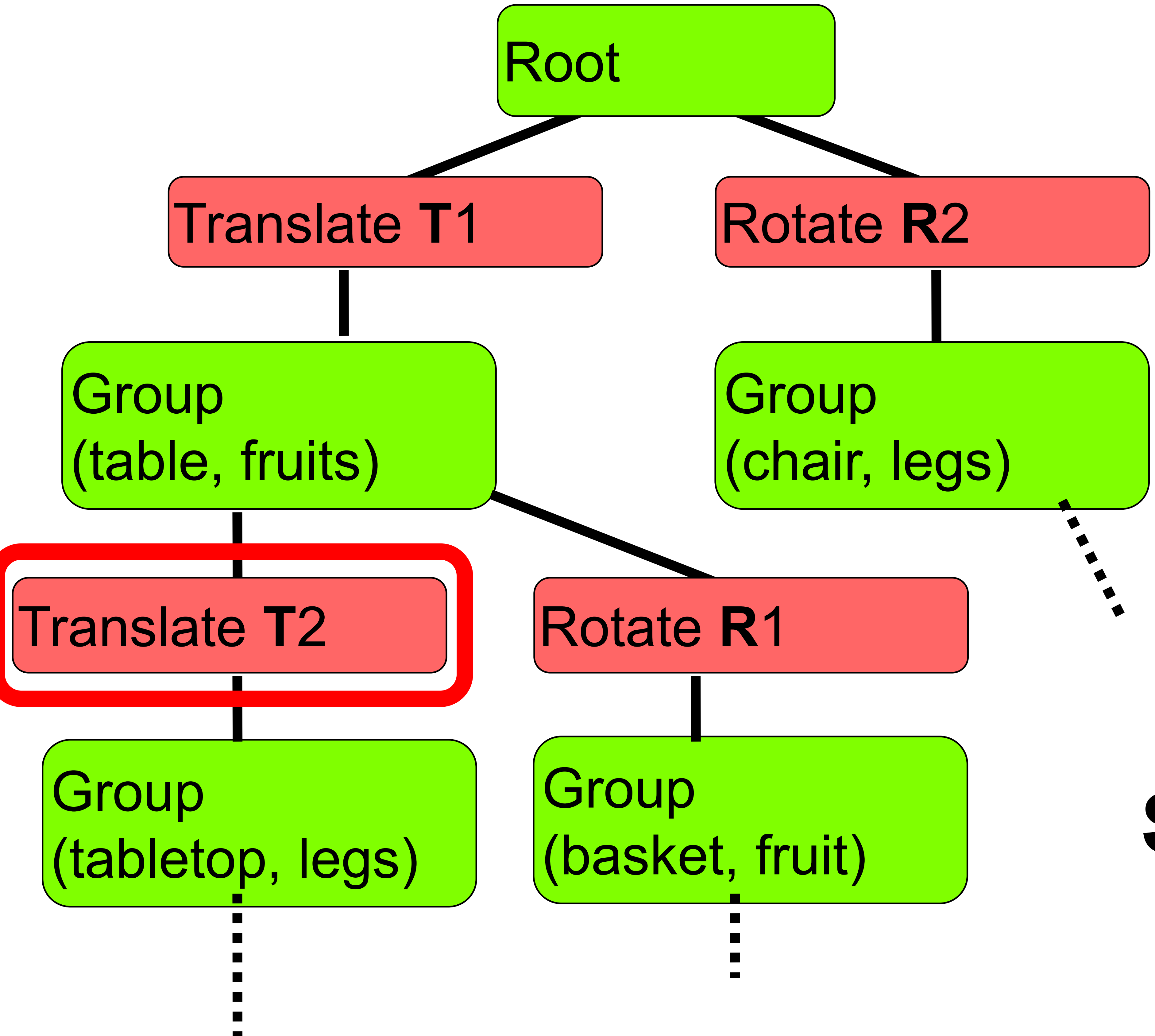
S = **T1 T2**

Traversal Example



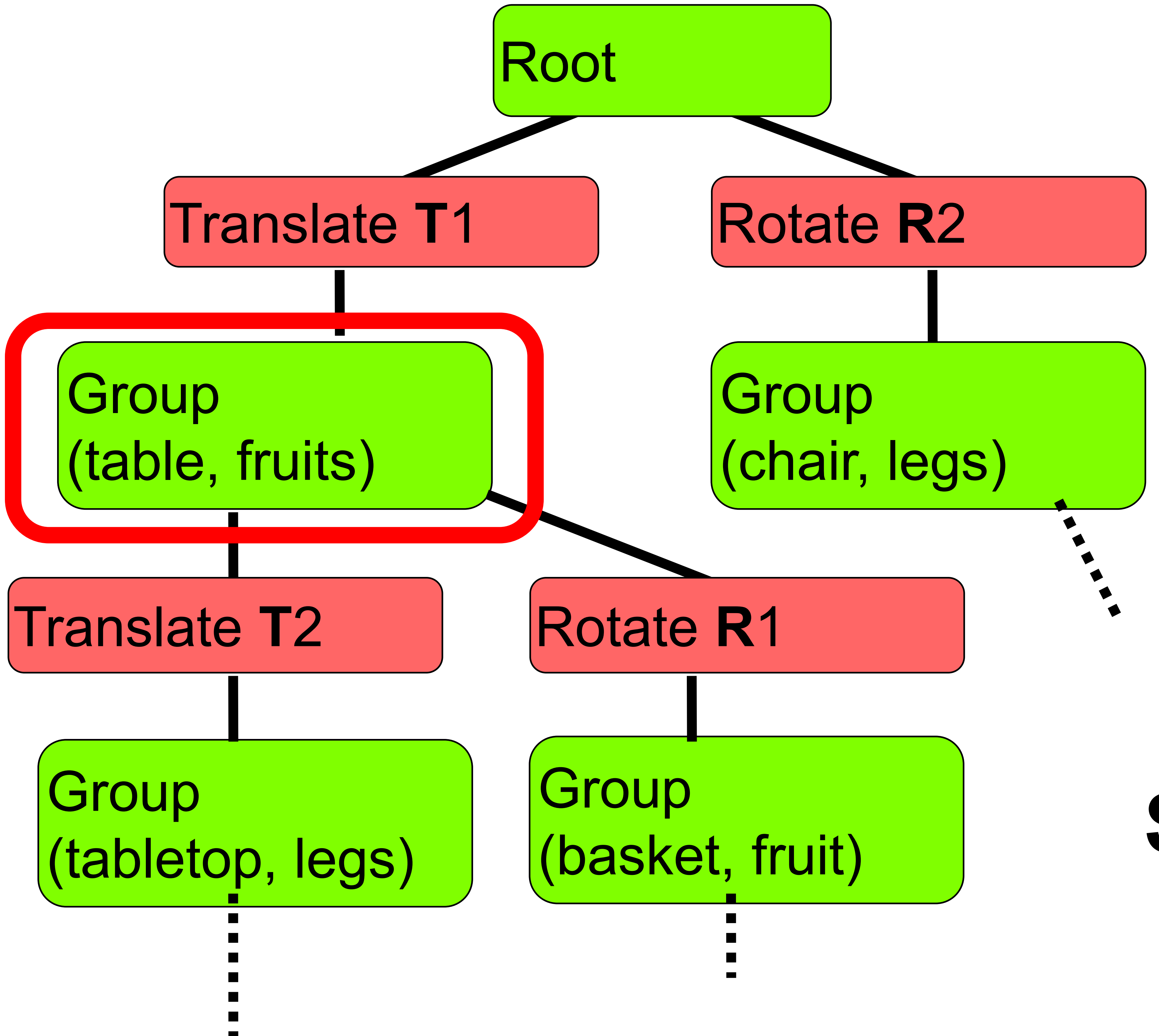
S = T1 T2

Traversal Example



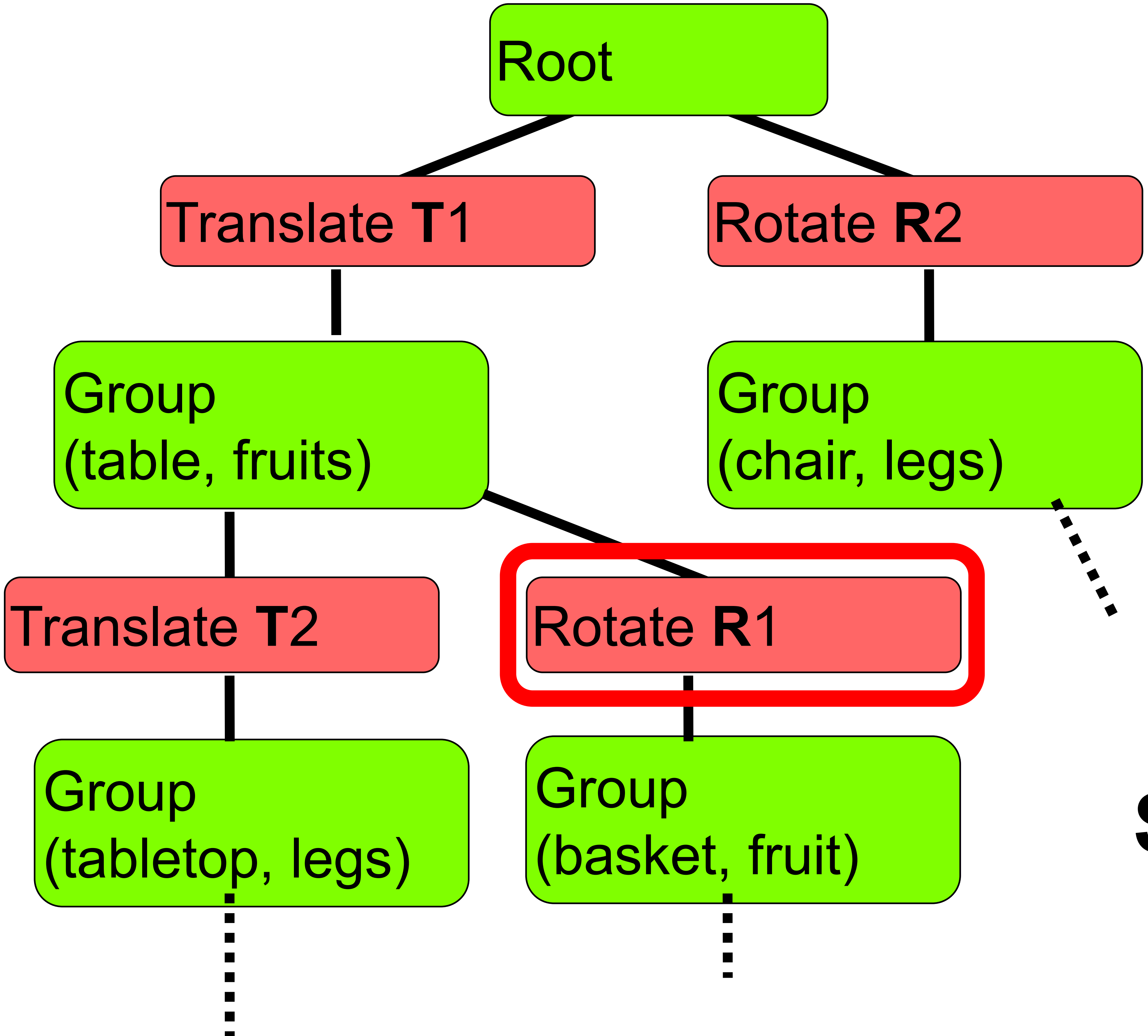
S = **T1 T2**

Traversal Example



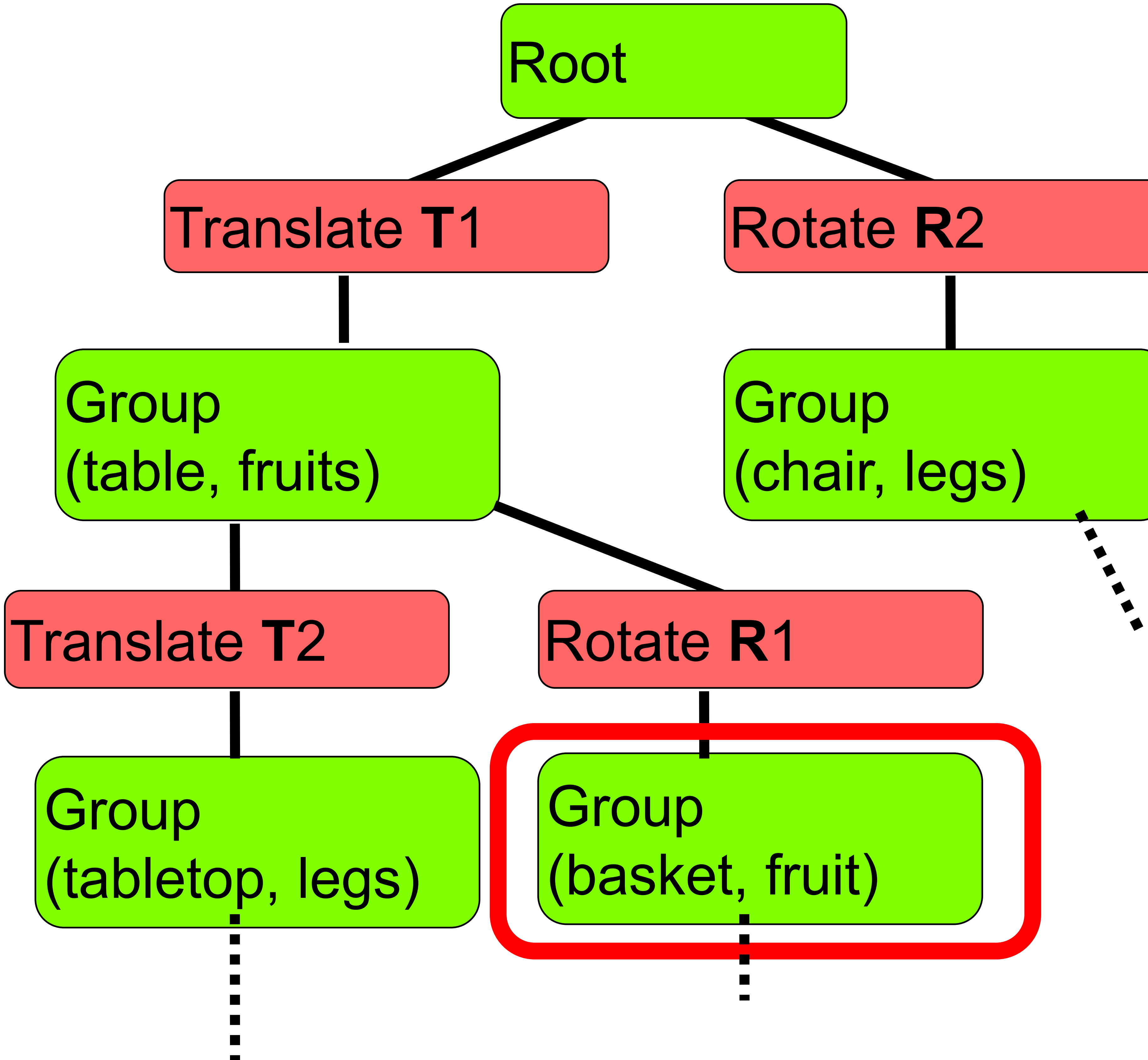
S = **T1**

Traversal Example



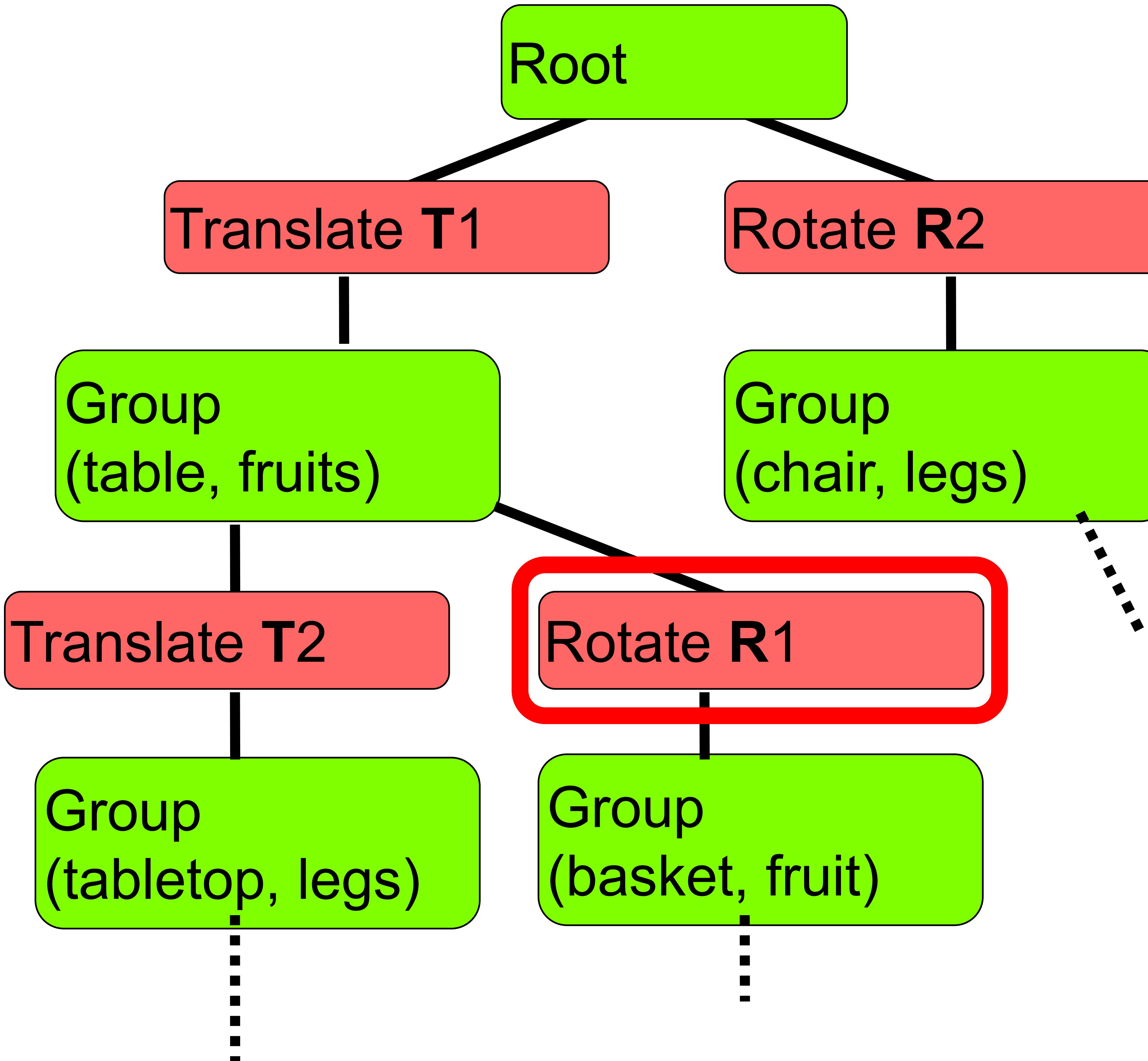
S = T1 R1

Traversal Example



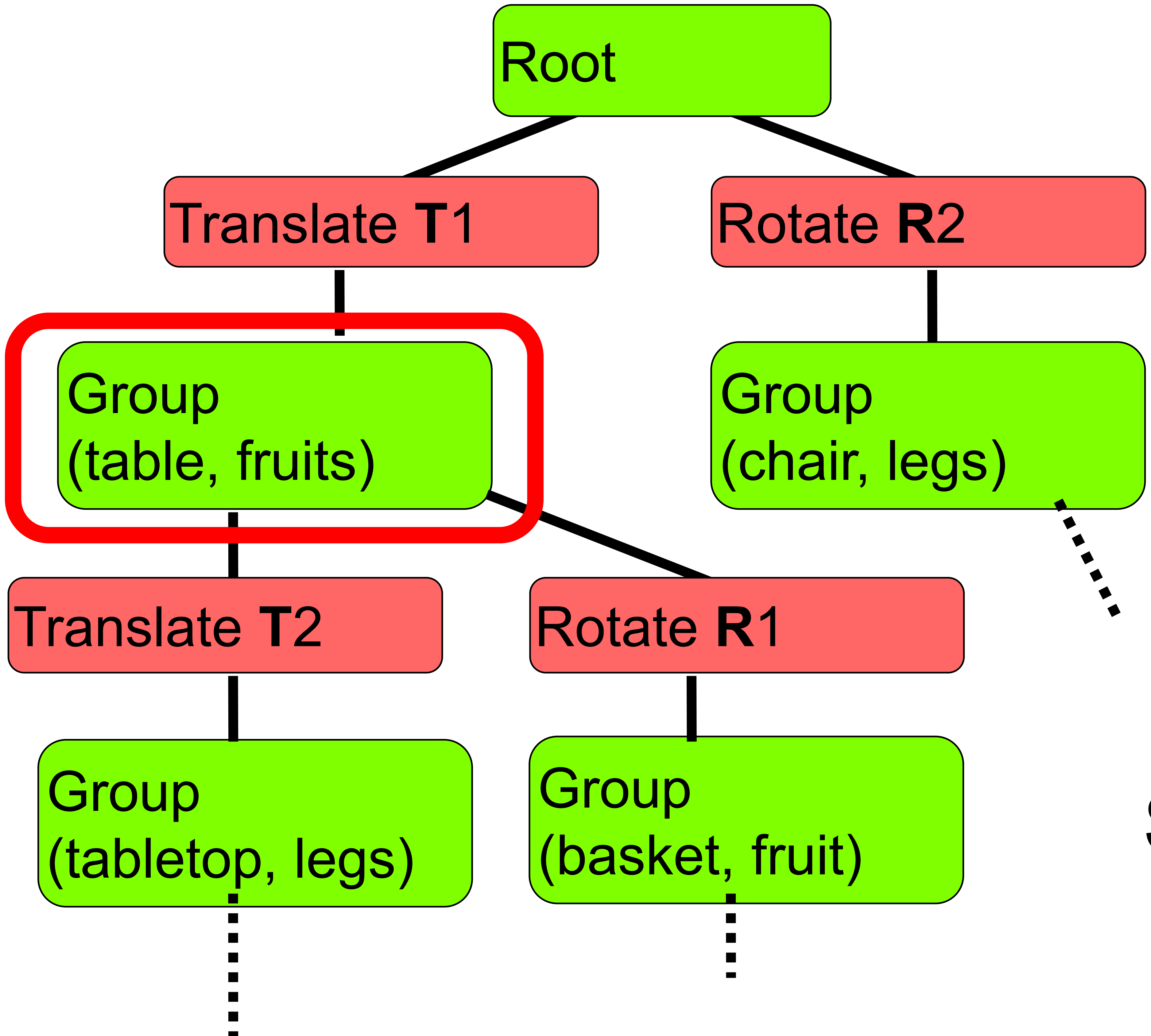
S = T1 R1

Traversal Example



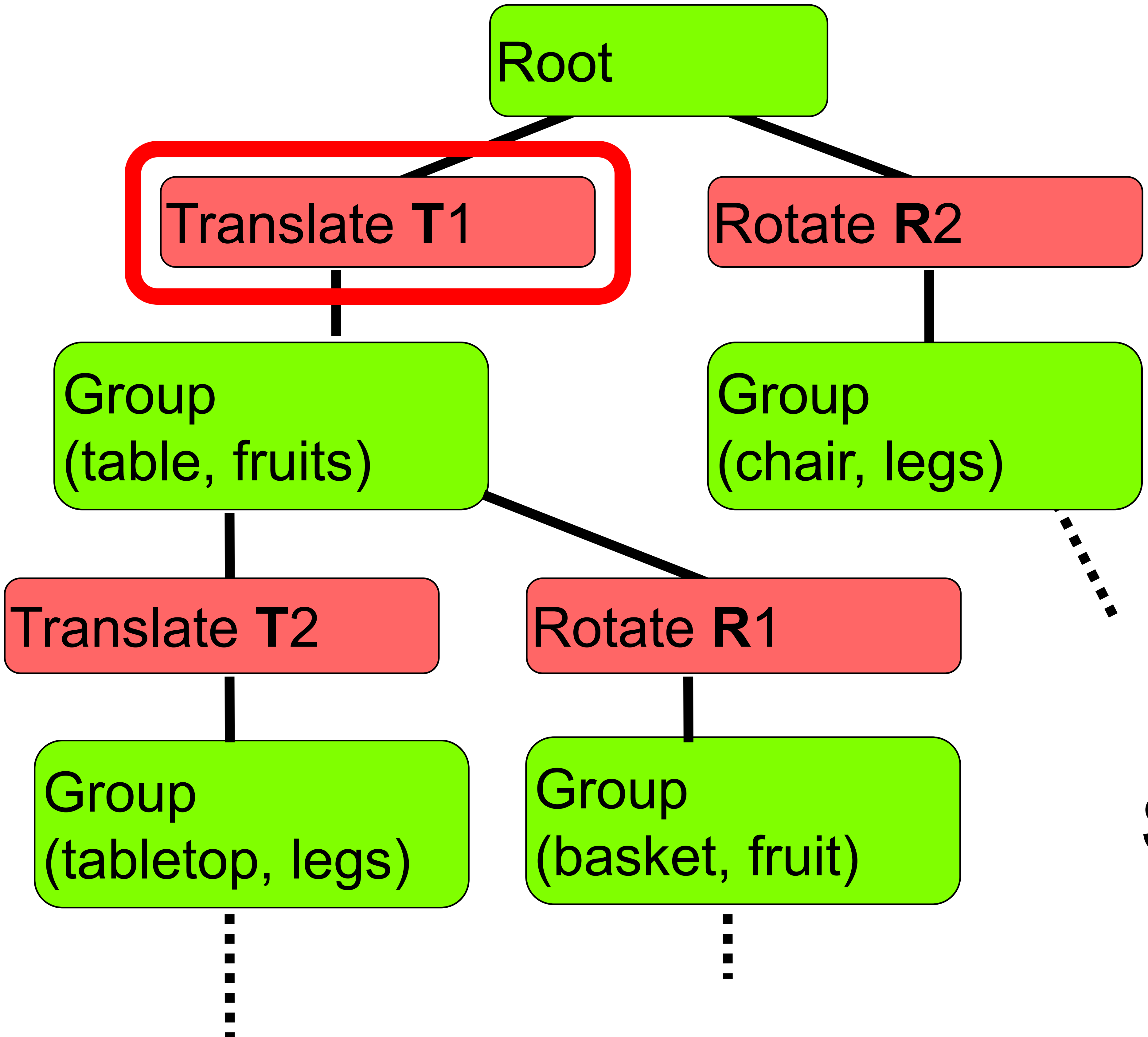
S = T1 R1

Traversal Example



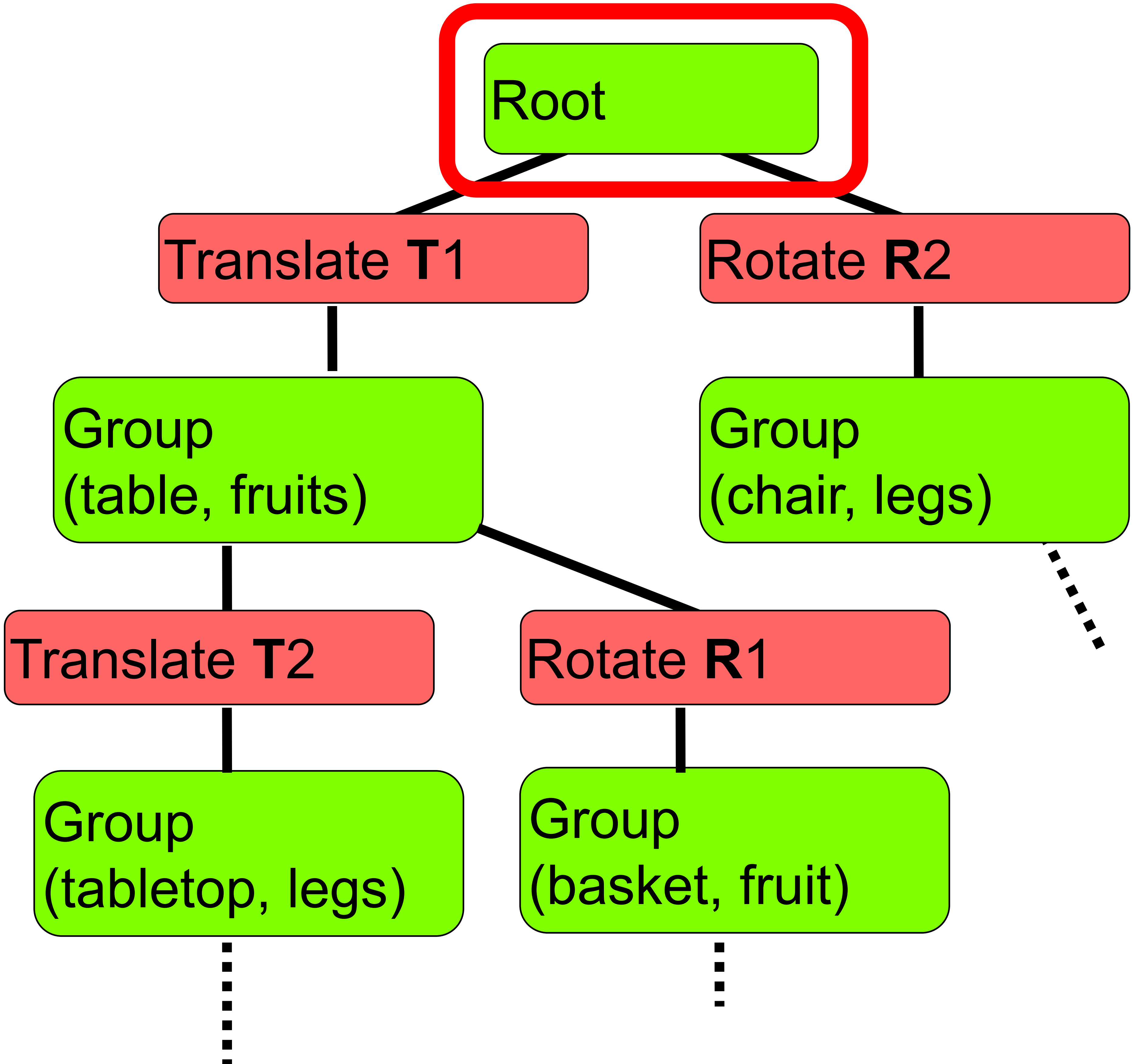
S = **T1**

Traversal Example



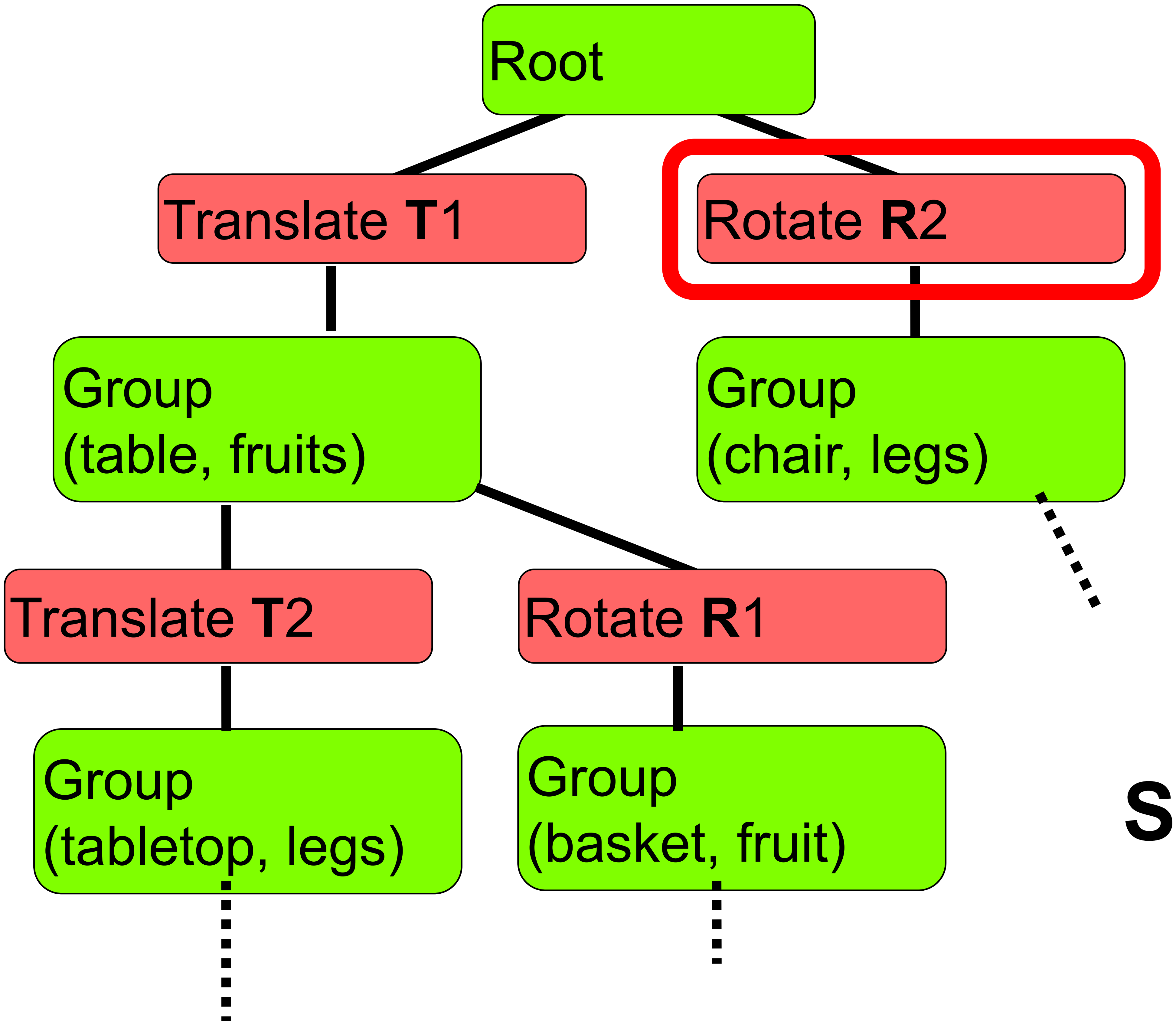
S = **T1**

Traversal Example



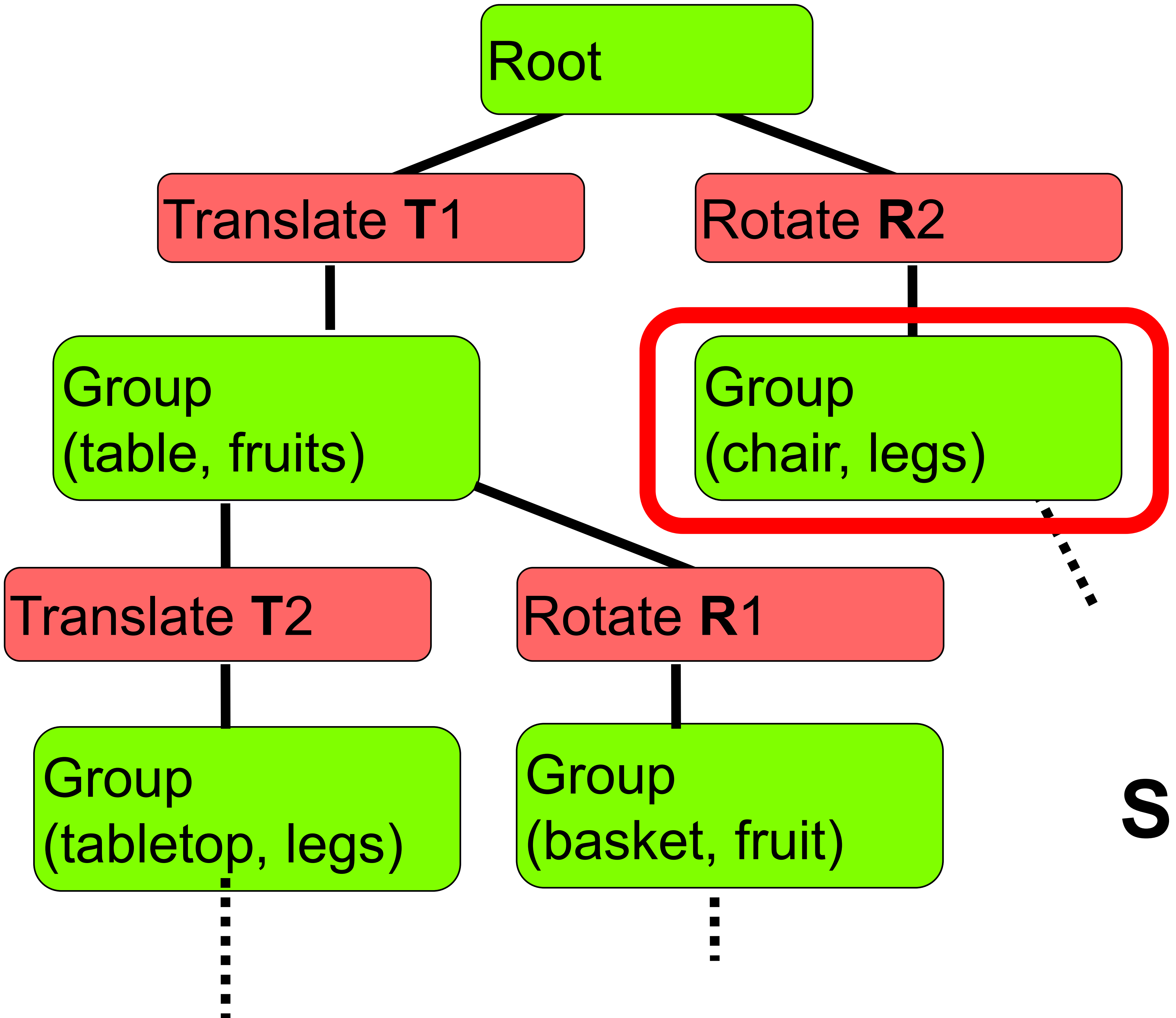
S = **I**

Traversal Example



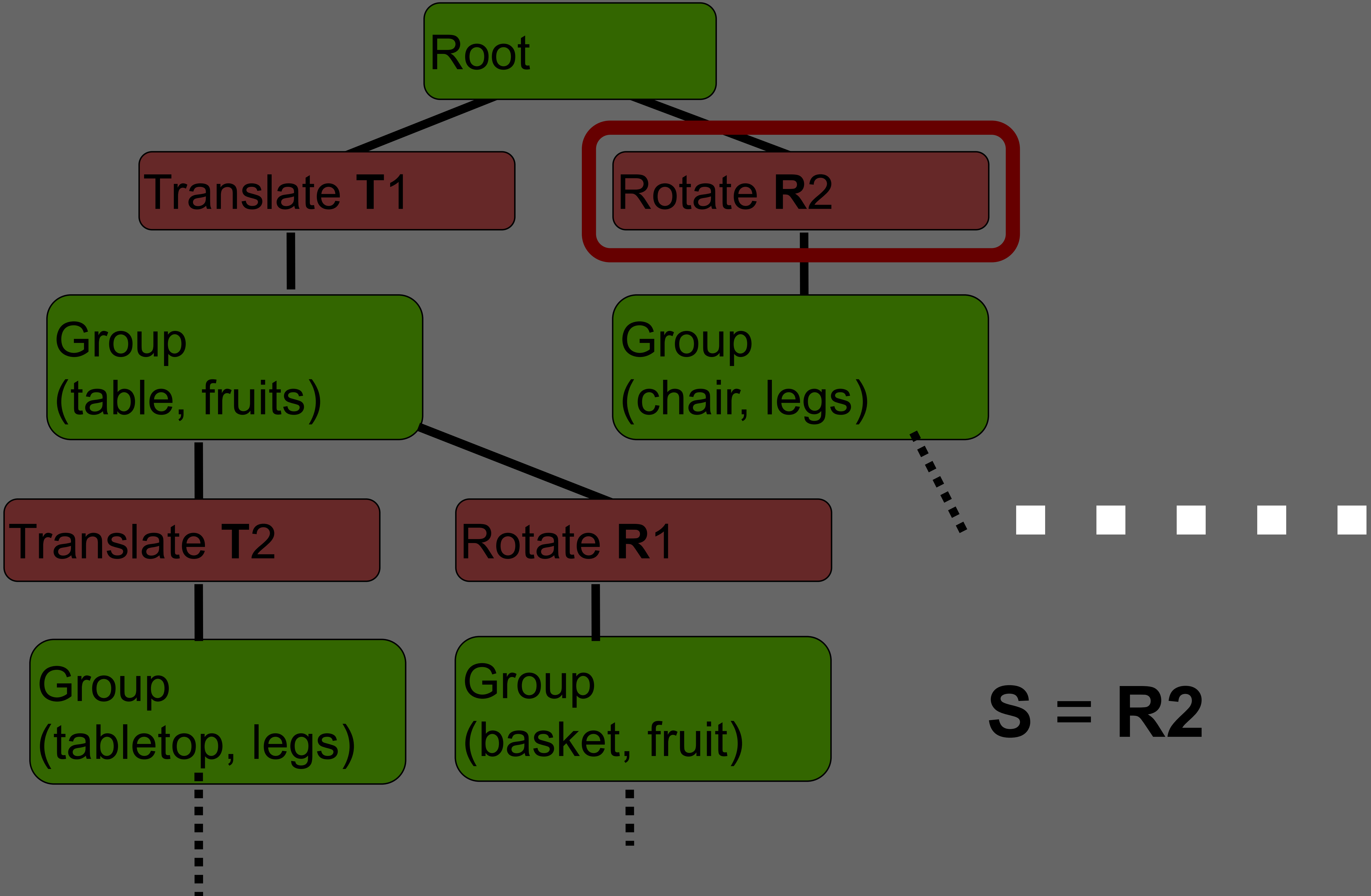
S = **R2**

Traversal Example



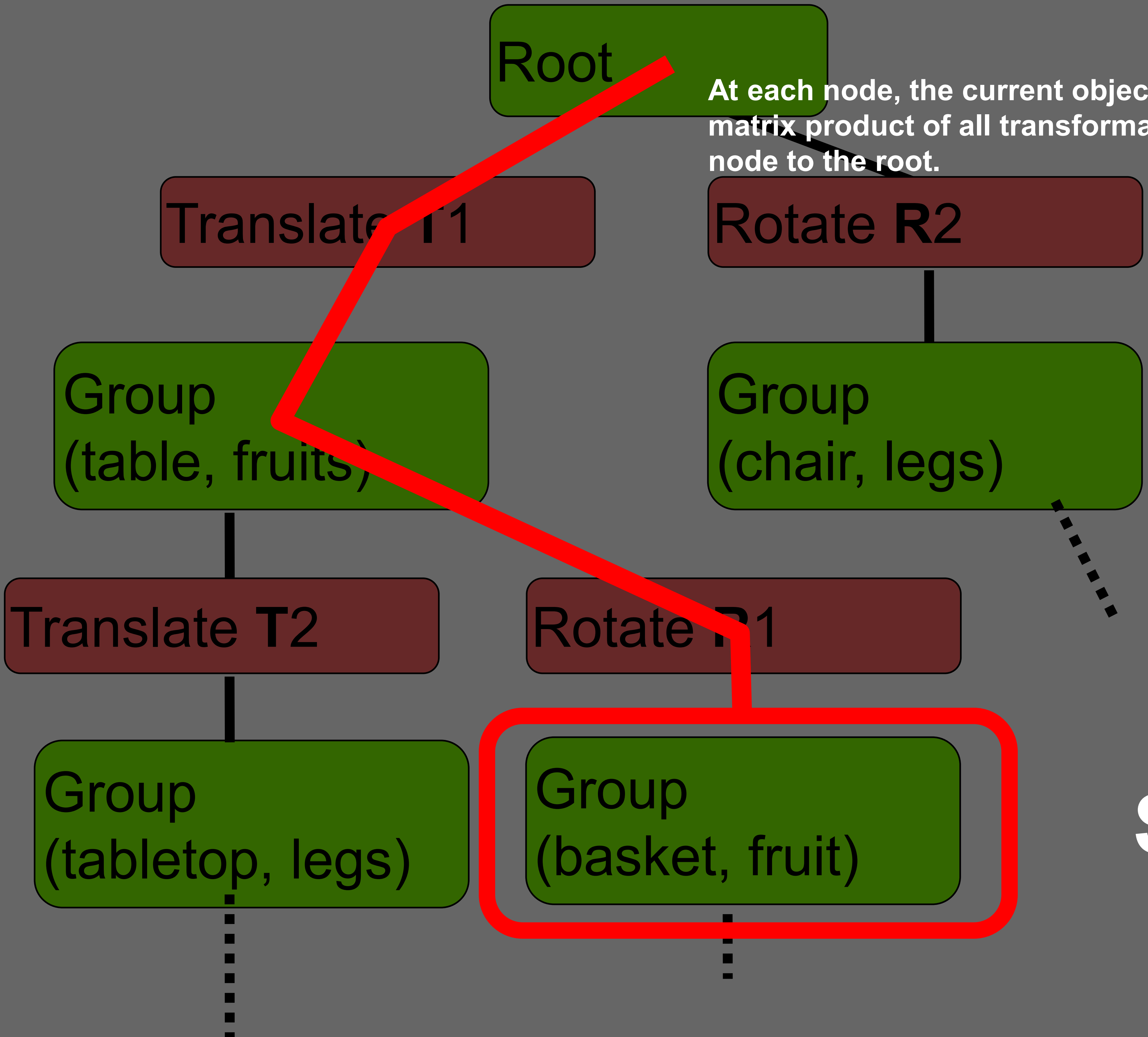
S = **R2**

Traversal Example



S = R2

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 = T1R1

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? **T***$\mathbf{T}$-1 = **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 object onto a plane – no way to undo, inverse doesn’t exist!

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? **T*****T**-1 = **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 object onto a plane – no way to undo, inverse doesn’t exist!

Can you think of a data structure suited for this?

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? **T*****T**-1 = **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 object onto a plane – no way to undo, inverse doesn't exist!
- **Solution:** Keep state variables in a **stack**
  - Push current state when entering node, update current state
  - Pop stack when leaving state-changing node
  - See what the stack looks like in the previous example!

Questions?

Plan

- Hierarchical Modeling, Scene Graph
- OpenGL matrix stack
- Hierarchical modeling and animation of characters
  - Forward and inverse kinematics

Hierarchical Modeling in OpenGL

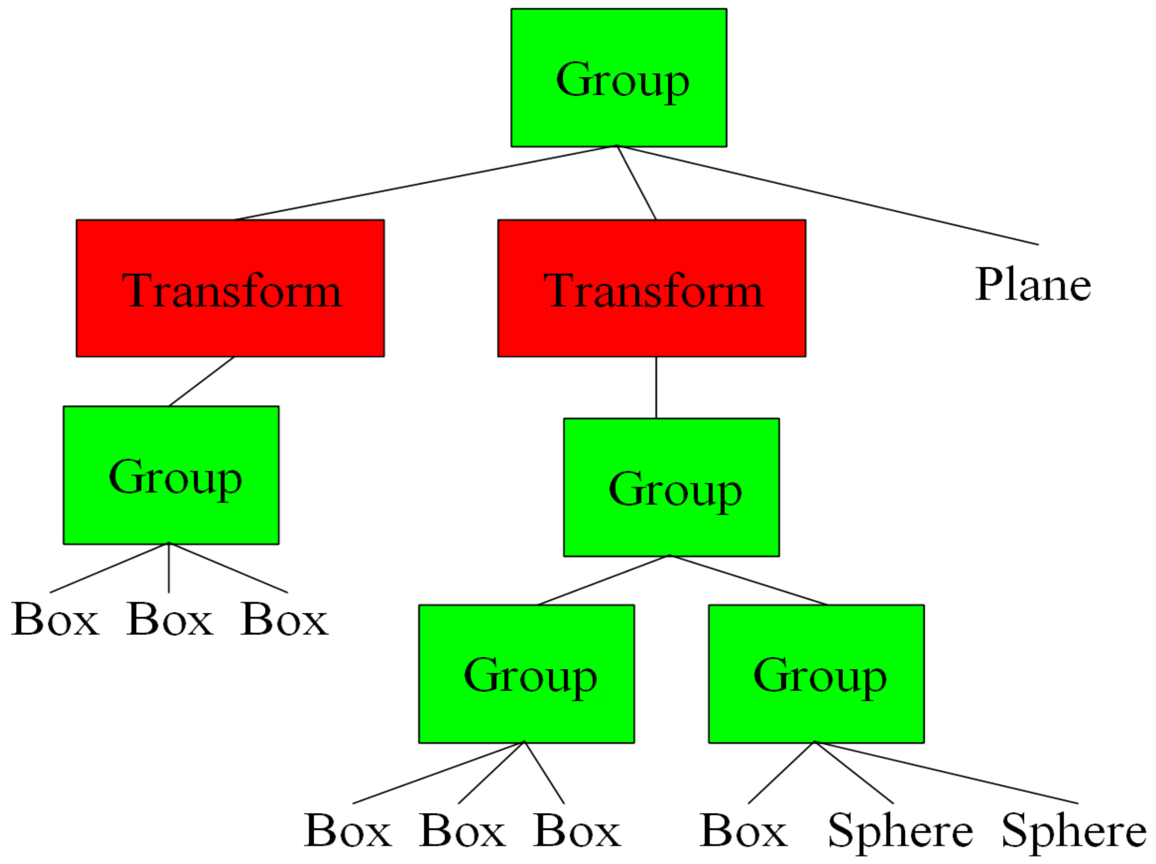
- The OpenGL Matrix Stack implements what we just did!

- Commands to change current transformation
  - glTranslate, glScale, etc.
- Current transformation is part of the OpenGL state, i.e., all following draw calls will undergo the new transformation
  - Remember, a transform affects the whole subtree
- Functions to maintain a matrix stack
  - glPushMatrix, glPopMatrix
- Separate stacks for modelview (object-to-view) and projection matrices

When You Encounter a Transform Node

- Push the current transform using glPushMatrix()
- Multiply current transform by node's transformation
  - Use glMultMatrix(), glTranslate(), glRotate(), glScale(), etc.
- Traverse the subtree
  - Issue draw calls for geometry node
- Use glPopMatrix() when done.

- Simple as that!



More Specifically…

- An OpenGL transformation call corresponds to a matrix **T**
- The call multiplies current modelview matrix **C** by **T** from the right, i.e. **C**′ = **C** * **T**.
  - This also works for projection, but you often set it up only once.

- This means that the transformation for the subsequent vertices will be **p**′ = **C** * **T** * **p**
  - Vertices are column vectors on the right in OpenGL
  - This implements hierarchical transformation directly!

More Specifically…

- An OpenGL transformation call corresponds to a matrix **T**
- The call multiplies current modelview matrix **C** by **T** from the right, i.e. **C**′ = **C** * **T**.
  - This also works for projection, but you often set it up only once.

- This means that the transformation for the subsequent vertices will be **p**′ = **C** * **T** * **p**
  - Vertices are column vectors on the right in OpenGL
  - This implements hierarchical transformation directly!

- At the beginning of the frame, initialize the current matrix by the viewing transform that maps from world space to view space.
  - For instance, glLoadIdentity() followed by gluLookAt()

Questions?

- Further reading on OpenGL
  Matrix Stack and hierarchical model/view transforms
  - http://www.glprogramming.com/red/chapter03.html

- It can be a little confusing if you don't think the previous through, but it's really quite simple in the end.
  - I know very capable people who after 15 years of experience still resort to brute force (trying all the combinations) for getting their transformations right, but it's such a waste :)

Plan

- Hierarchical Modeling, Scene Graph
- OpenGL matrix stack
- Hierarchical modeling and animation of characters
  - Forward and inverse kinematics

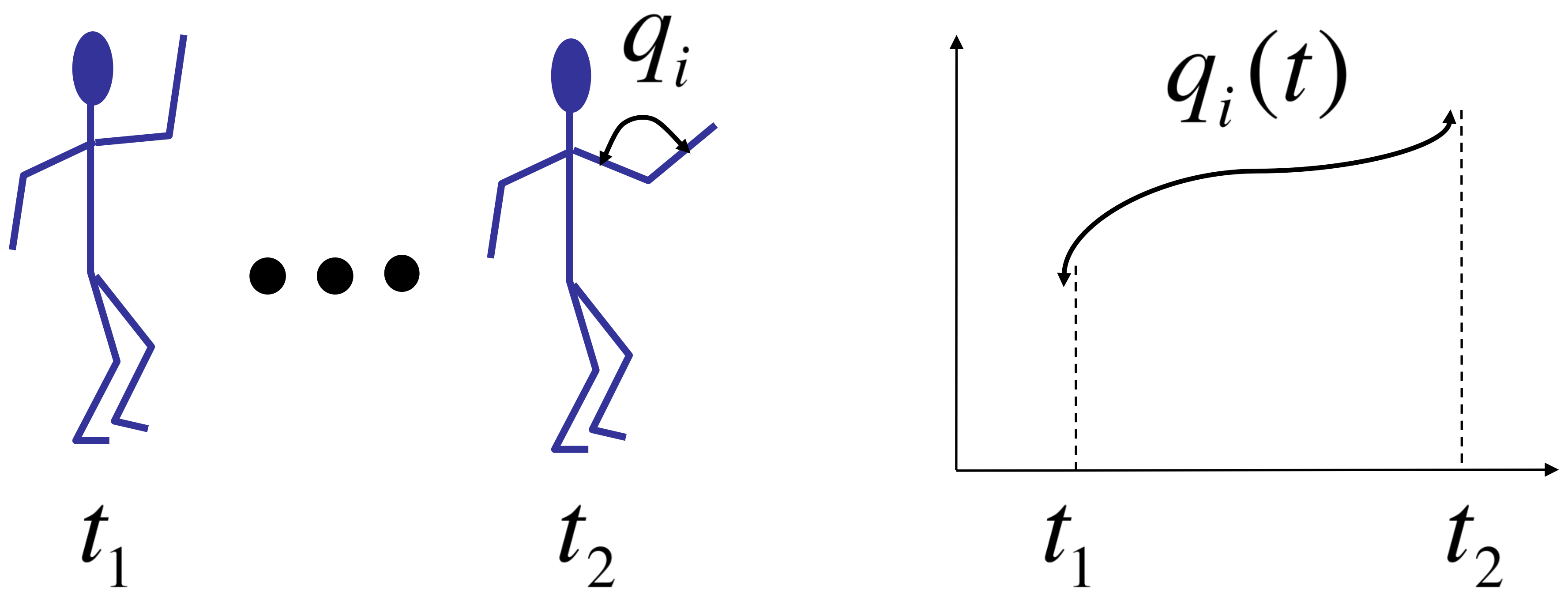
Animation

- Hierarchical structure is essential for animation
  - Eyes move with head
  - Hands move with arms
  - Feet move with legs
  - …
- Without such structure the model falls apart.

Articulated Models

- **Articulated models** are rigid parts connected by joints
  - each joint has some angular degrees of freedom

- Articulated models can be animated by specifying the joint angles as functions of time.

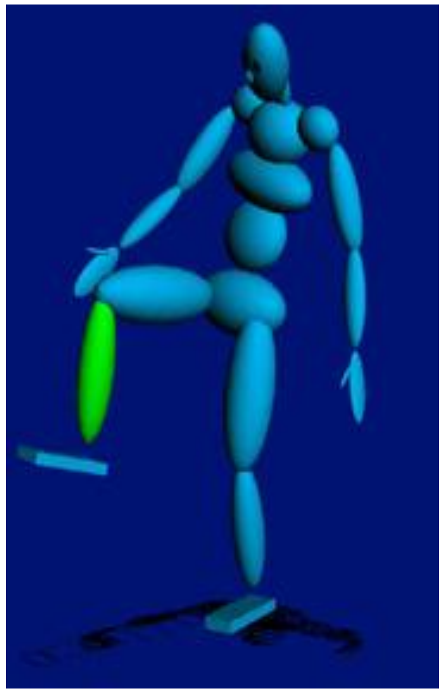


Joints and bones

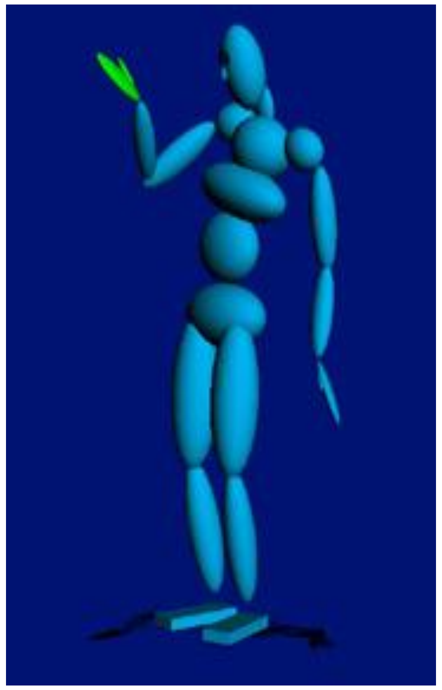
- Describes the positions of the body parts as a function of joint angles.
  - Body parts are usually called "bones"

- Each joint is characterized by its degrees of freedom (dof)
  - Usually rotation for articulated bodies

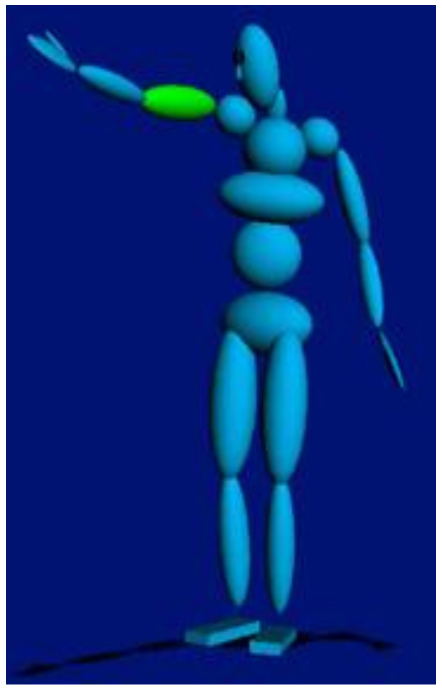
1 DOF: knee



2 DOF: wrist



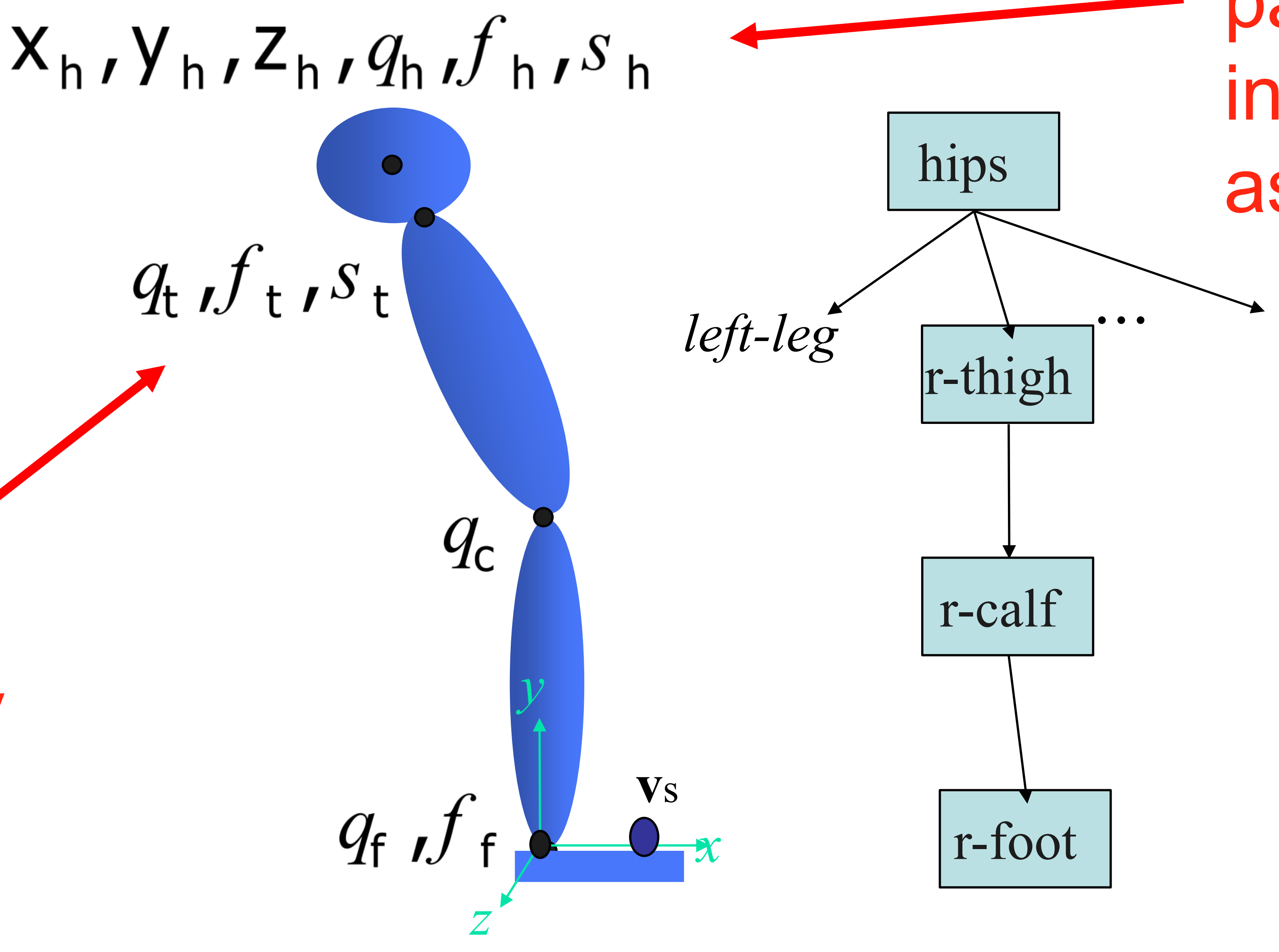
3 DOF: arm



Skeleton Hierarchy

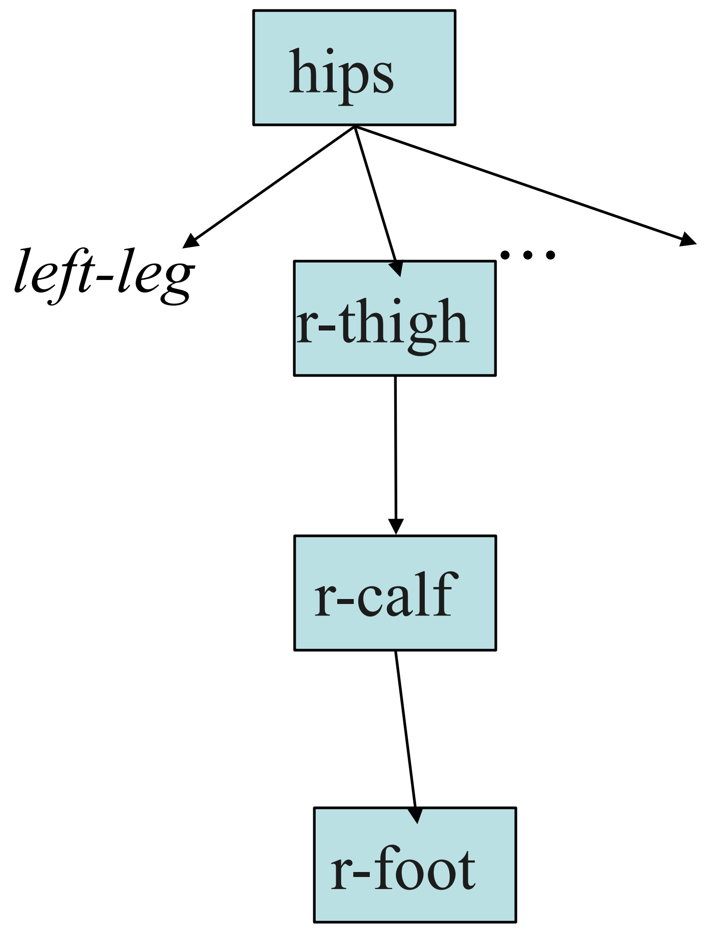
- Each bone position/orientation described relative to the parent in the hierarchy:

For the root, the parameters include a position as well



Joints are specified by angles.

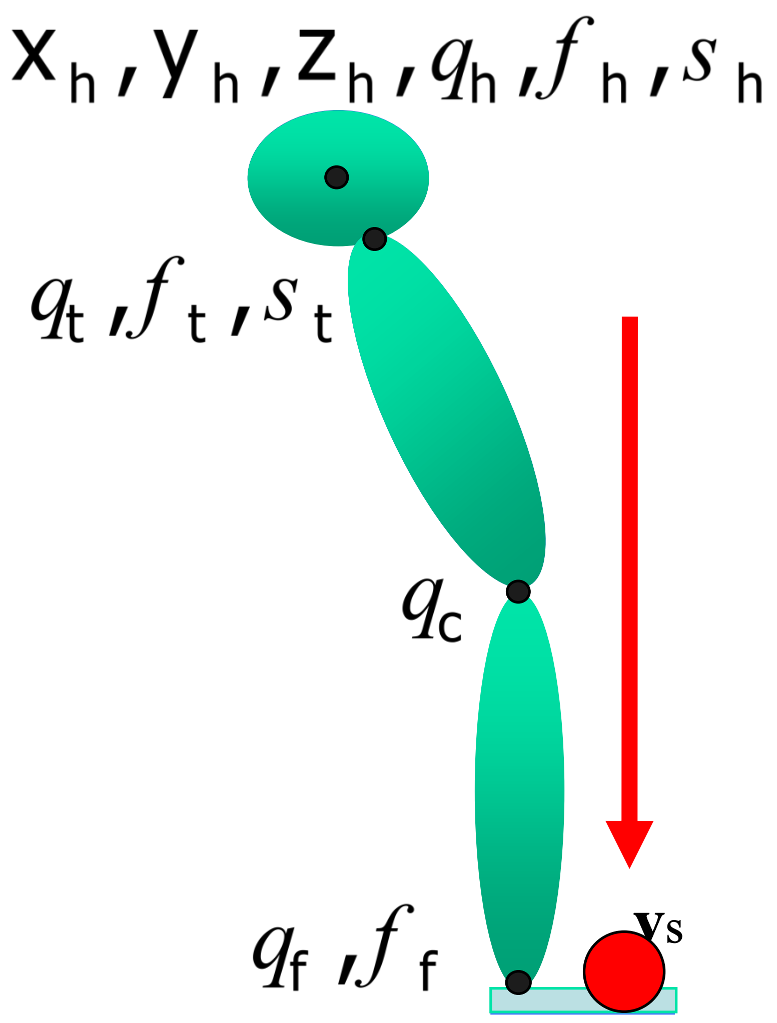
Draw by Traversing a Tree



```
glLoadIdentity();
glPushMatrix();
  glTranslatef(…);
  glRotate(…);
  drawHips();
  glPushMatrix();
    glTranslate(…);
   glRotate(…);
   drawThigh();
   glTranslate(…);
   glRotate(…);
   drawCalf();
   glTranslate(…);
   glRotate(…);
   drawFoot();
  glPopMatrix();
  left-leg
```

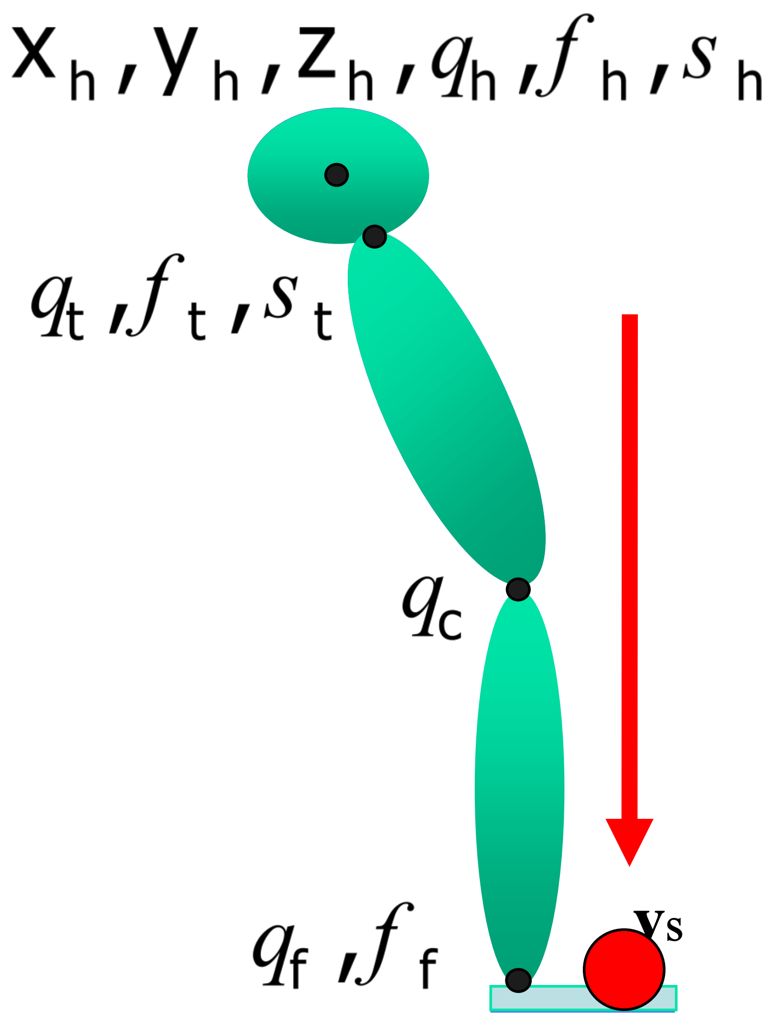
- Assumes drawing procedures for thigh, calf, and foot use joint positions as the origin for a drawing coordinate frame

Forward Kinematics


$x_h, y_h, z_h, \theta_h, \phi_h, \sigma_h$
$\theta_t, \phi_t, \sigma_t$
θ_c
θ_f, ϕ_f
vs

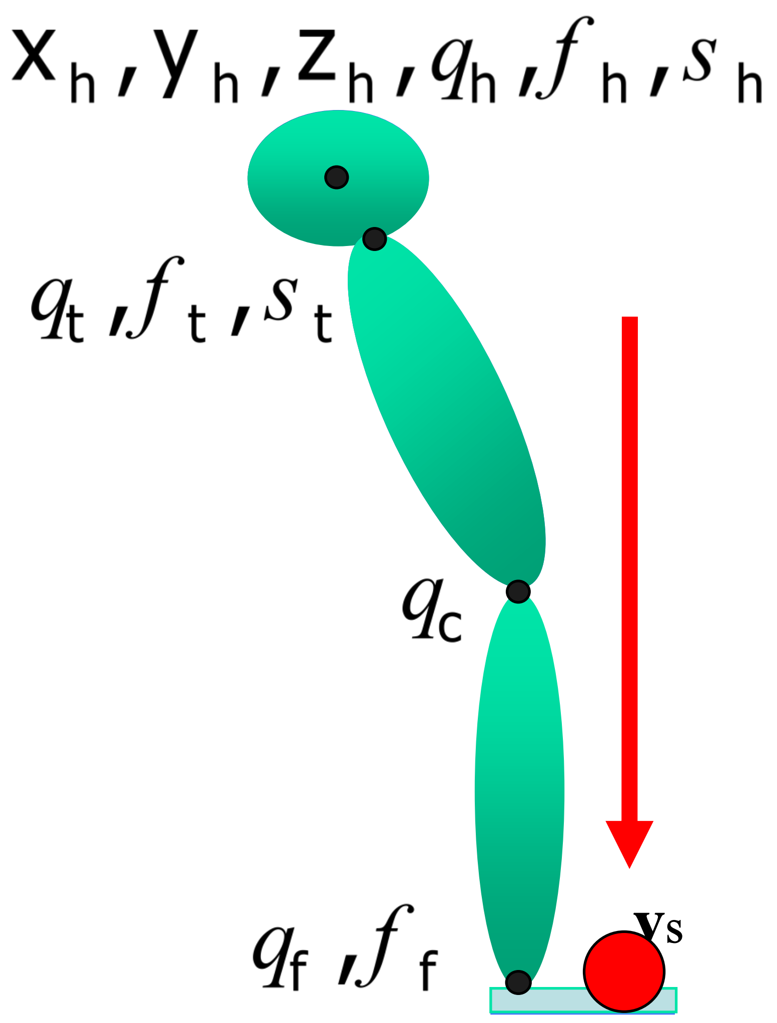
How to determine the world-space position for point **vs**?

Forward Kinematics



Transformation matrix **S** for a point **v**s is a matrix composition of all joint transformations between the point and the root of the hierarchy. **S** is a function of all the joint angles between here and root.

Forward Kinematics



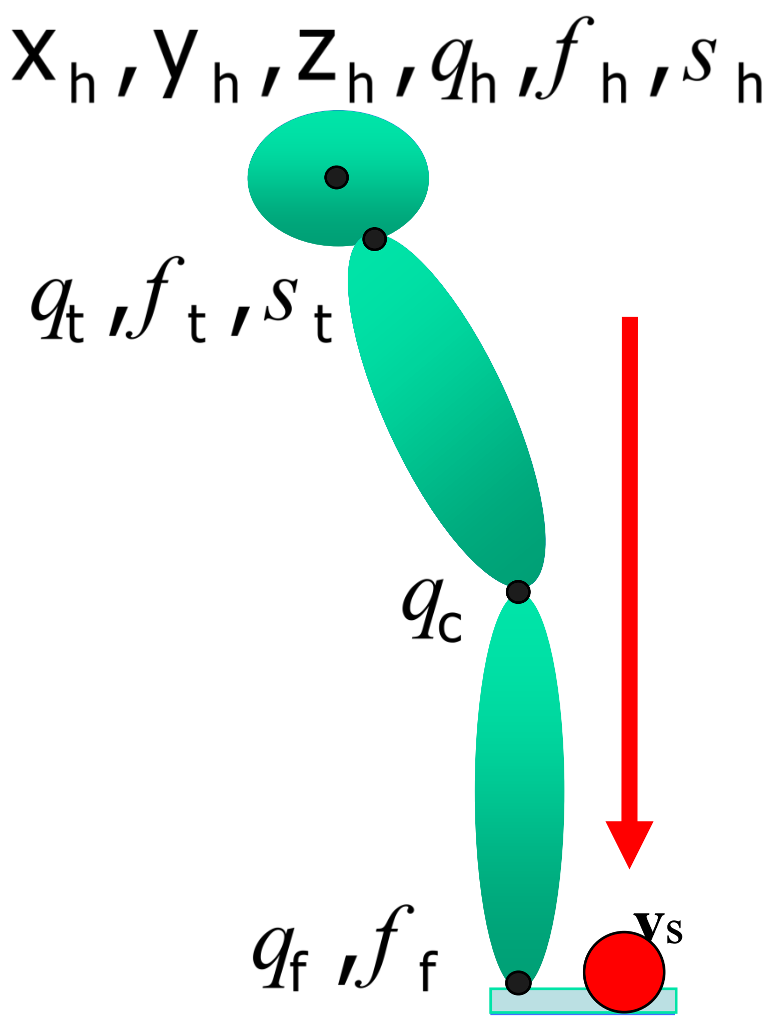
Transformation matrix **S** for a point **v**s is a matrix composition of all joint transformations between the point and the root of the hierarchy. **S** is a function of all the joint angles between here and root.

Note that the angles have a non-linear effect.

This product is **S**

$$\mathbf{v}_w = \boxed{\mathbf{T}(x_h,y_h,z_h)\mathbf{R}(q_h,f_h,s_h)\ \mathbf{TR}(q_t,f_t,s_t)\ \mathbf{TR}(q_c)\ \mathbf{TR}(q_f,f_f)}\ \mathbf{v}_s$$

Forward Kinematics



Transformation matrix **S** for a point **v**s is a matrix composition of all joint transformations between the point and the root of the hierarchy. **S** is a function of all the angles between here and root.

Note that the angles have a non-linear effect.

This product is **S**

$$\mathbf{v}_w = \boxed{\mathbf{T}(x_h, y_h, z_h)\mathbf{R}(q_h, f_h, s_h)\ \mathbf{TR}(q_t, f_t, s_t)\ \mathbf{TR}(q_c)\ \mathbf{TR}(q_f, f_f)}\ \mathbf{v}_s$$

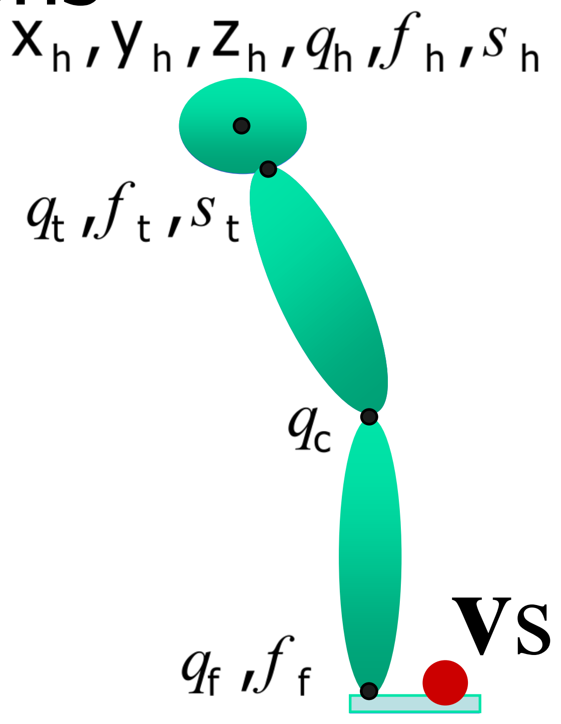
$$v_w = S\left(\underbrace{x_h, y_h, z_h, \theta_h, \phi_h, \sigma_h, \theta_t, \phi_t, \sigma_t, \theta_c, \theta_f, \phi_f}_{p}\right) v_s = S(p) v_s$$

parameter vector **p**

Questions?

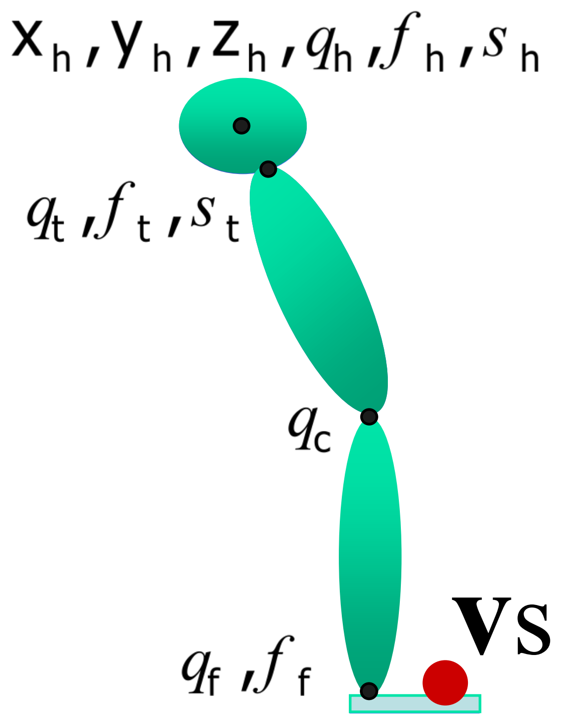
Inverse Kinematics

- Context: an animator wants to "pose" a character
- Specifying every single angle is tedious and not intuitive
- Simpler interface:
  directly manipulate position of e.g. hands and feet
- That is, specify vw, infer joint transformations



Inverse Kinematics

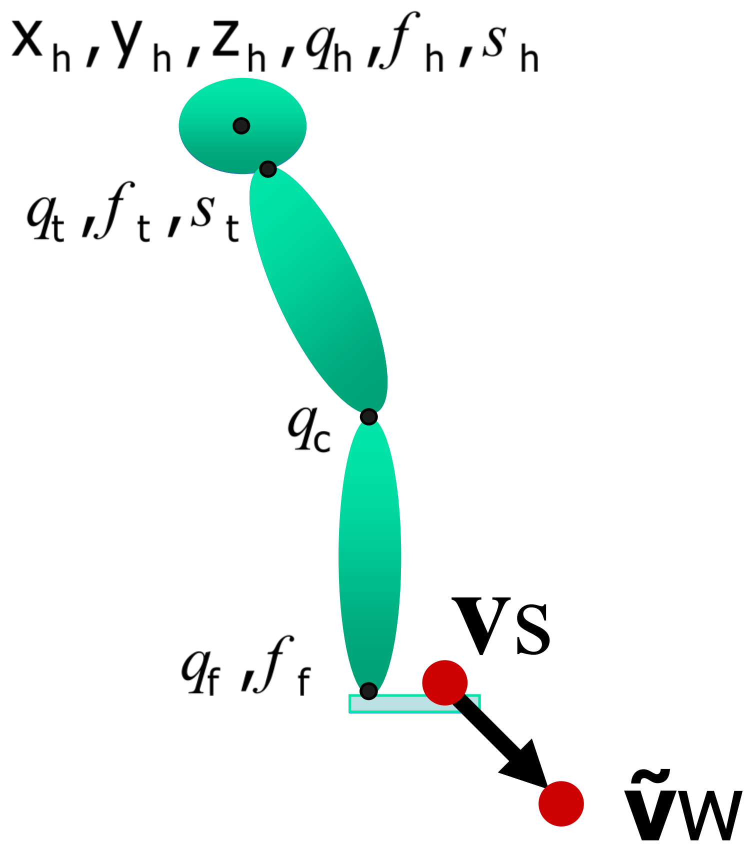
- Forward Kinematics
  - Given the skeleton parameters **p** (position of the root and the joint angles) and the position of the point in local coordinates **v**s, what is the position of the point in the world coordinates **v**w?
  - Not too hard, just apply transform accumulated from the root.



Inverse Kinematics

- Forward Kinematics
  - Given the skeleton parameters **p** (position of the root and the joint angles) and the position of the point in local coordinates **v**s, what is the position of the point in the world coordinates **v**w?
  - Not too hard, just apply transform accumulated from the root.

- Inverse Kinematics
  - Given the current position of the point and the desired new position $\tilde{\mathbf{v}}_w$ in world coordinates, what are the skeleton parameters **p** that take the point to the desired position?



Inverse Kinematics

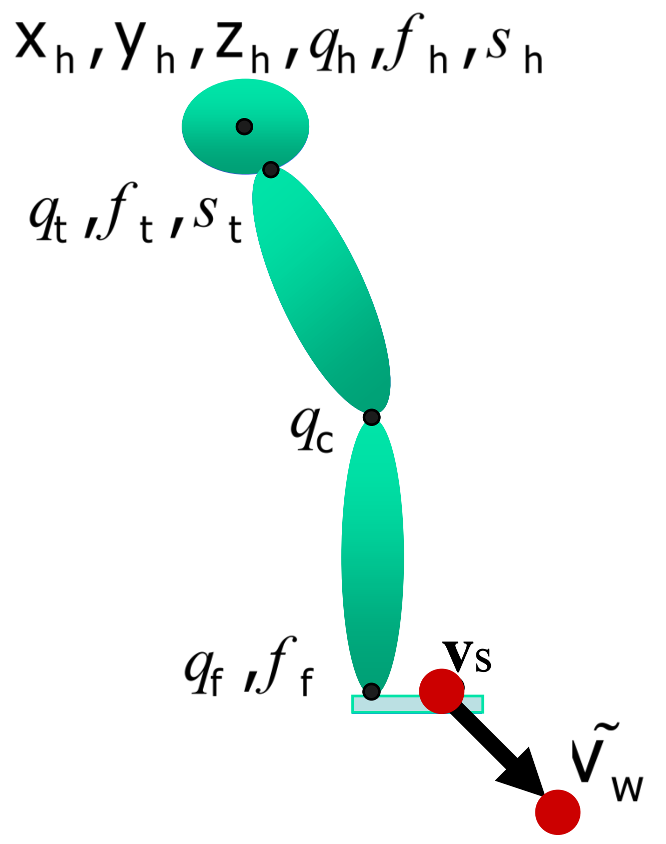
- Given the position of the point in local coordinates **v**s and the desired position $\tilde{\mathbf{v}}_w$ in world coordinates, what are the skeleton parameters **p**?

$$\tilde{v}_w = S\left(\underbrace{x_h, y_h, z_h, \theta_h, \phi_h, \sigma_h, \theta_t, \phi_t, \sigma_t, \theta_c, \theta_f, \phi_f}_{p}\right) v_s = S(p) v_s$$

skeleton parameter vector **p**

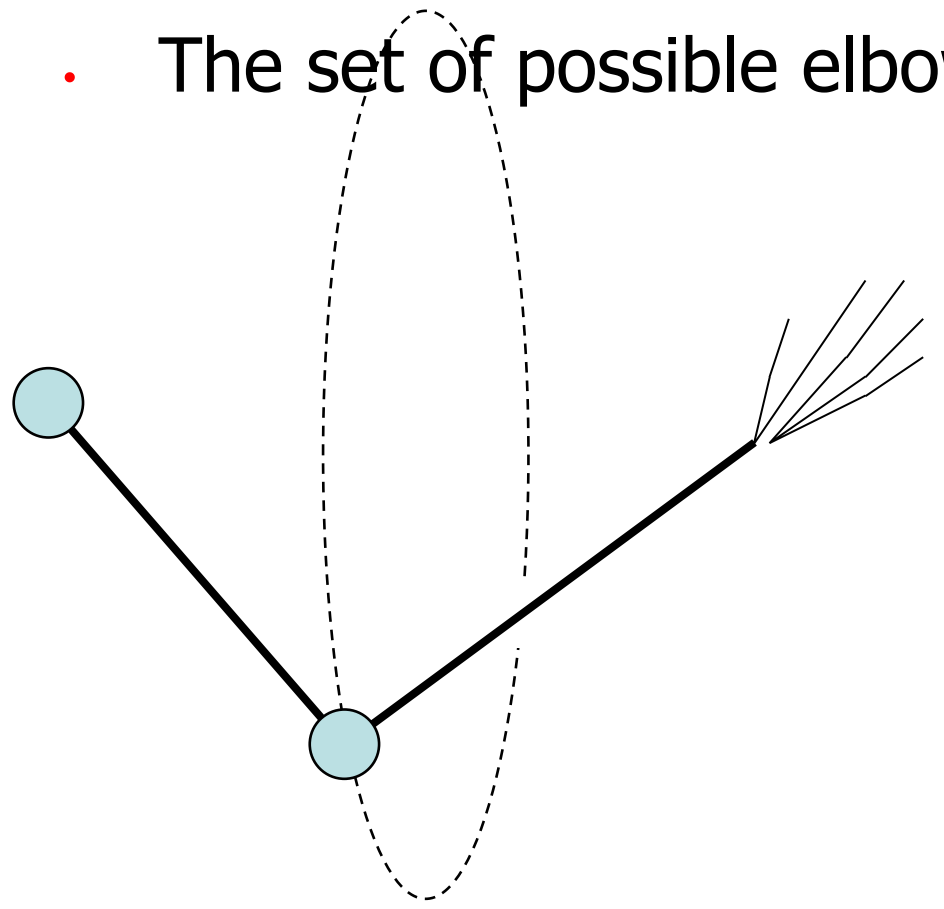
- Requires solving for **p**, given **v**s and $\tilde{\mathbf{v}}_w$
  - Non-linear and …

It’s Underconstrained



- Count degrees of freedom:
  - We specify one 3D point (3 equations)
  - We usually need more than 3 angles
  - **p** usually has tens of dimensions

- Simple geometric example (in 3D): specify hand position, need elbow & shoulder
  - The set of possible elbow location is a circle in 3D



How to tackle these problems?

$$\boldsymbol{v}_{\mathrm{WS}} = \boldsymbol{S}(\boldsymbol{p})\,\boldsymbol{v}_{\mathrm{s}}$$

- Deal with non-linearity:
  Iterative solution (steepest descent)
  - Compute Jacobian matrix of world position w.r.t. angles
    - Jacobian: "If the parameters **p** change by tiny amounts, what is the resulting change in the world position **v**WS?"
  - Then invert Jacobian.
    - This says "if **v**WS changes by a tiny amount, what is the change in the parameters **p**?"
  - But wait! The Jacobian is non-invertible (3xN)
  - Deal with ill-posedness: Pseudo-inverse
    - Solution that displaces things the least
    - See http://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse
  - Deal with ill-posedness: Prior on "good pose" (more advanced)

$$\left[\frac{\partial(\boldsymbol{v}_{\mathrm{WS}})_i}{\partial p_j}\right]$$

- Additional potential issues: bounds on joint angles, etc.
  - Do not want elbows to bend past 90 degrees, etc.

Example: Style-Based IK

- Video

- Prior on “good pose”

- Link to paper: Grochow, Martin, Hertzmann, Popovic: Style-Based Inverse Kinematics, ACM SIGGRAPH 2004

Mesh-Based Inverse Kinematics

- Video

- Doesn’t even need a hierarchy or skeleton: Figure proper transformations out based on a few example deformations!

- Link to paper:
  Sumner, Zwicker, Gotsman, Popovic: Mesh-Based Inverse Kinematics, ACM SIGGRAPH 2005

That's All for Today!

Further reading

- OpenGL Matrix Stack and hierarchical model/view transforms
  - http://www.glprogramming.com/red/chapter03.html