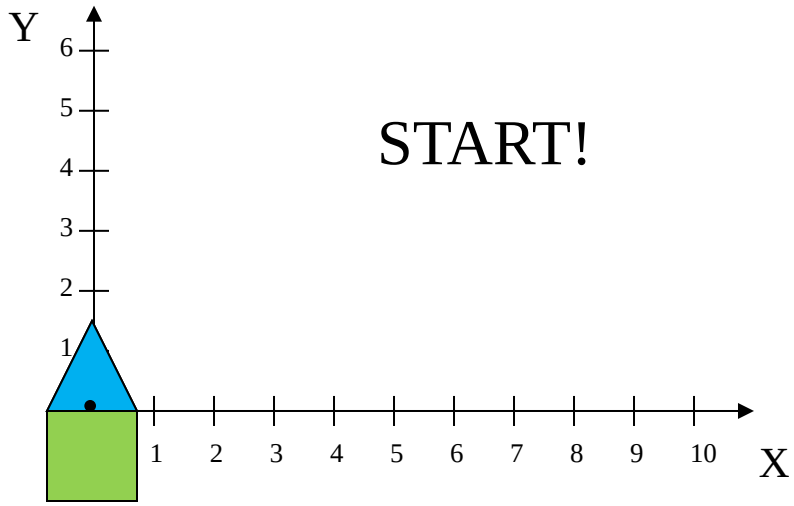
# Matrix Duality
# +
# Scene Graphs:
# How to Build Trees of Transformations

Jack Tumblin

used a few re-worked slides
by Andries van Dam

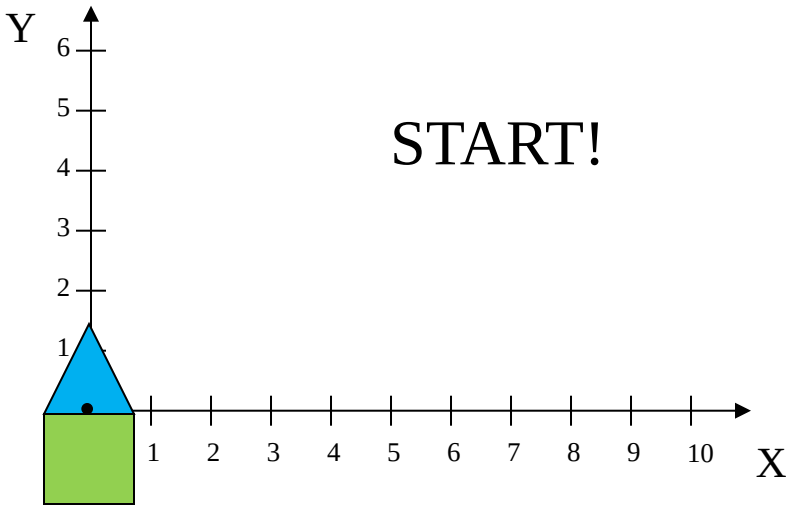
# Transformations are NOT Commutative!

Translate() by x=6, y=0 **then** Rotate() by 45º




Translation → Rotation

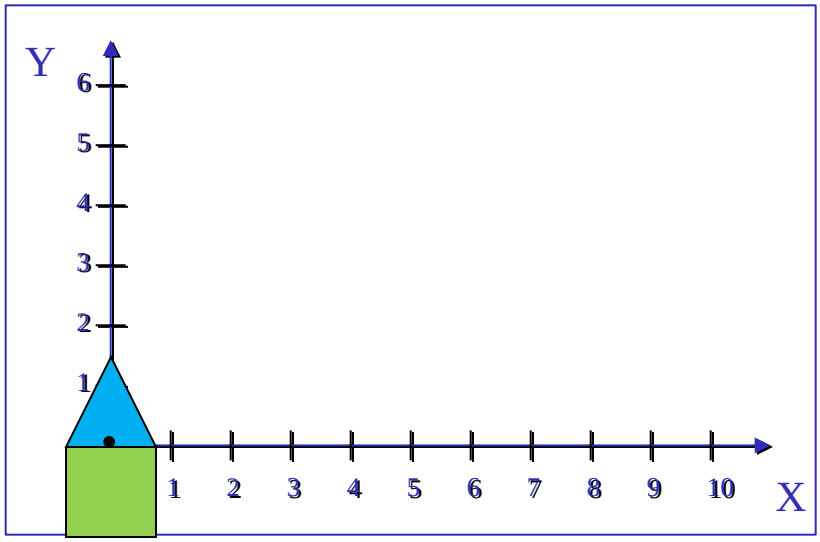
Rotate() by 45º **then** Translate() by x=6, y=0




Rotation → Translation

# METHOD 1:
# 1a) Draw ‘house’ at origin

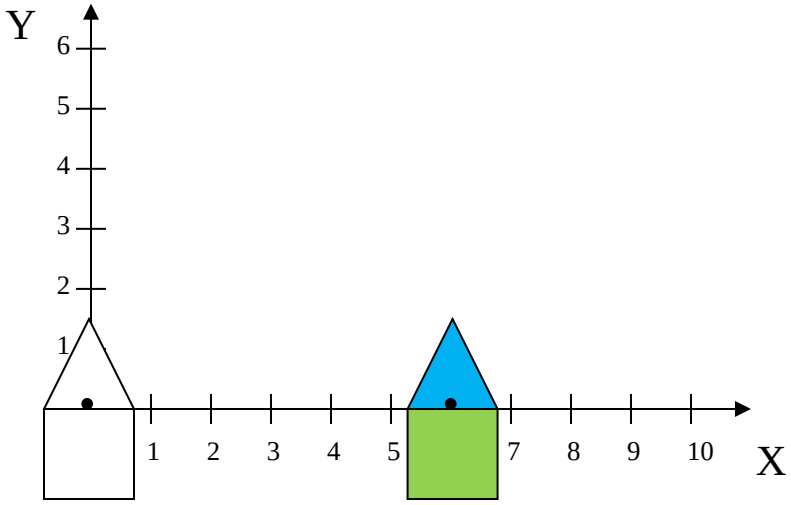
Translate()
by x=6, y=0
**then**
Rotate() by 45°




Translation → Rotation

# METHOD 1:
# 1b) Add 6 to all its x coords

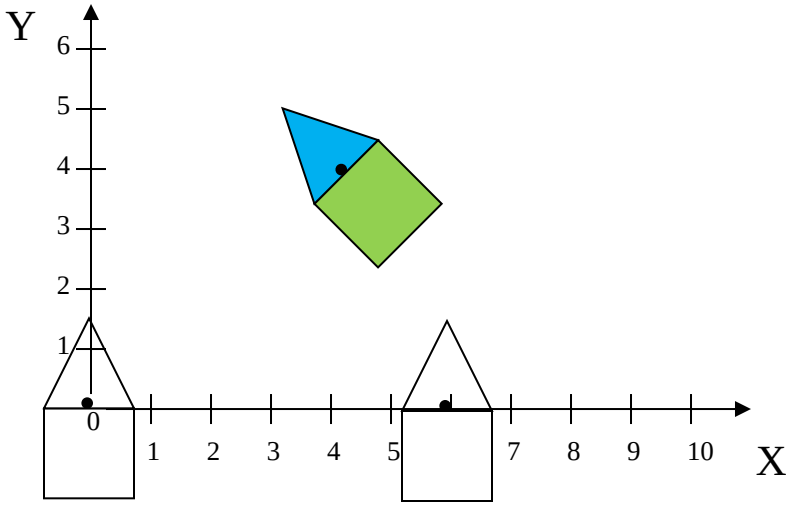
**Translate()**
by x=6, y=0
**then**
**Rotate()** by 45°




Translation → Rotation

# METHOD 1:
# 1c) All new x,y coords again

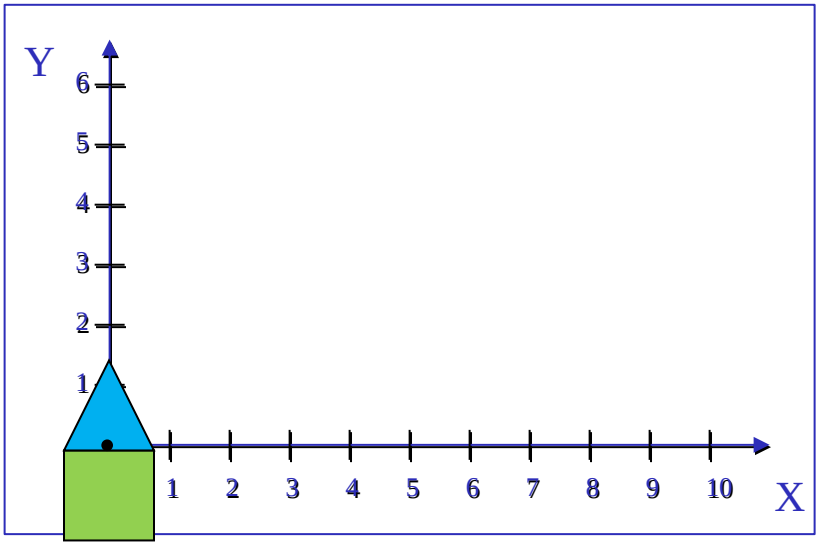
Translate() by
x=6, y=0
**then**
Rotate() by 45º




Translation → Rotation

# METHOD 1:
## 1a) Draw ‘house’ at origin

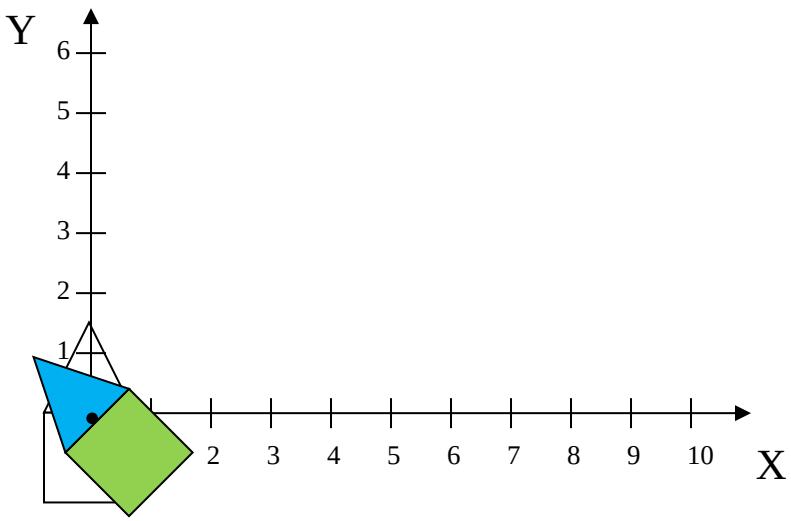
Rotate() by 45°
**then**
Translate() by
x=6, y=0




Rotation → Translation

# METHOD 1:
## 1b) Rotate all x,y coords,

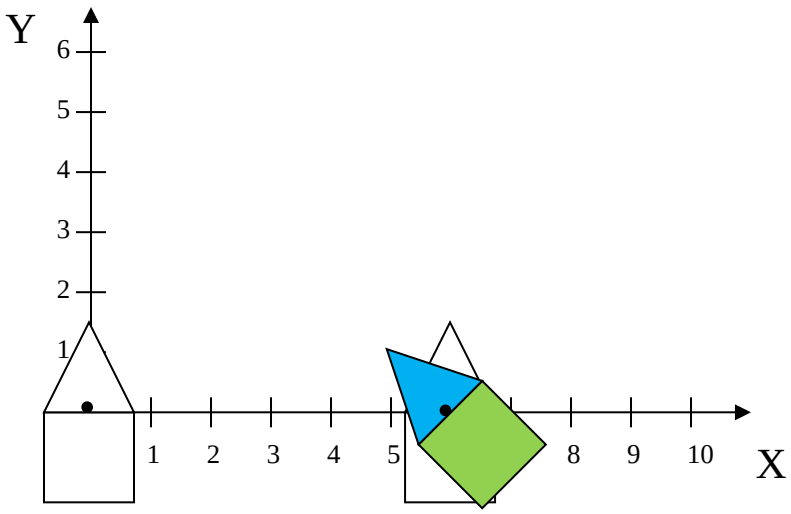
Rotate() by 45°
**then**
Translate() by
x=6, y=0




Rotation → Translation

# METHOD 1:
## 1c) Add 6 to all its x coords.

Rotate() by 45°
**then**
Translate()
by x=6, y=0

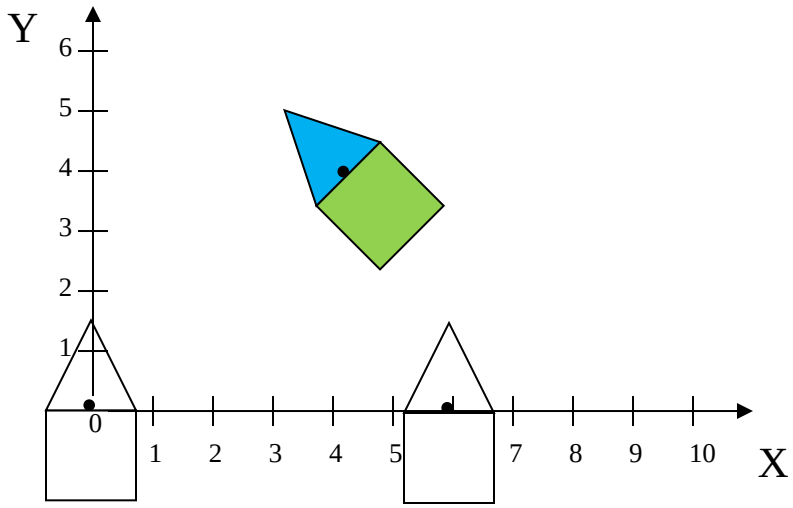



Rotation → Translation

# METHOD 1:

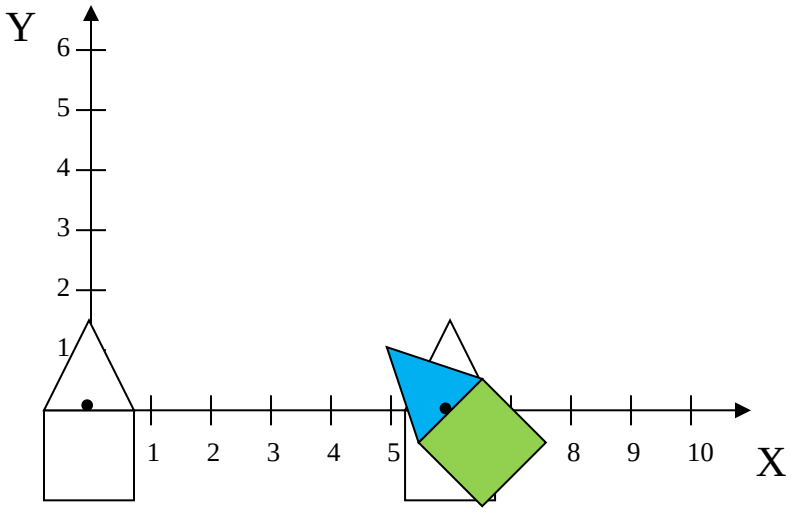
# Transformations are NOT Commutative!

Translate() by x=6, y=0 **then** Rotate() by 45º




Translation → Rotation

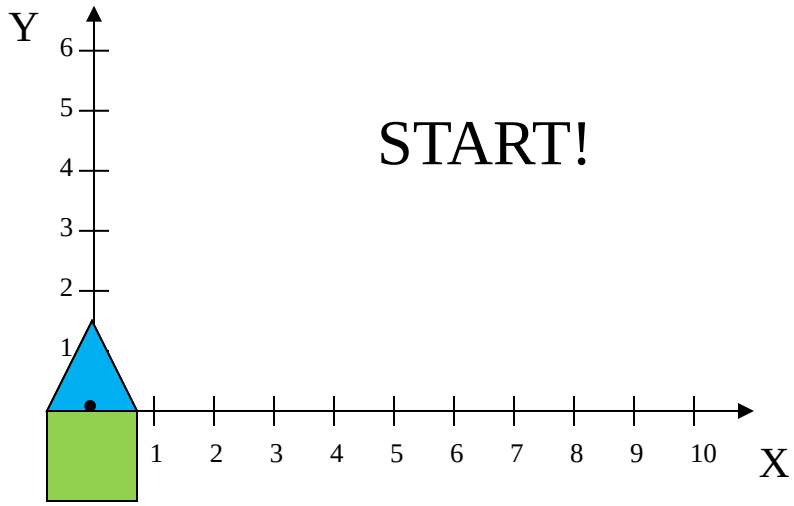
Rotate() by 45º **then** Translate() by x=6, y=0




Rotation → Translation

# METHOD 2:

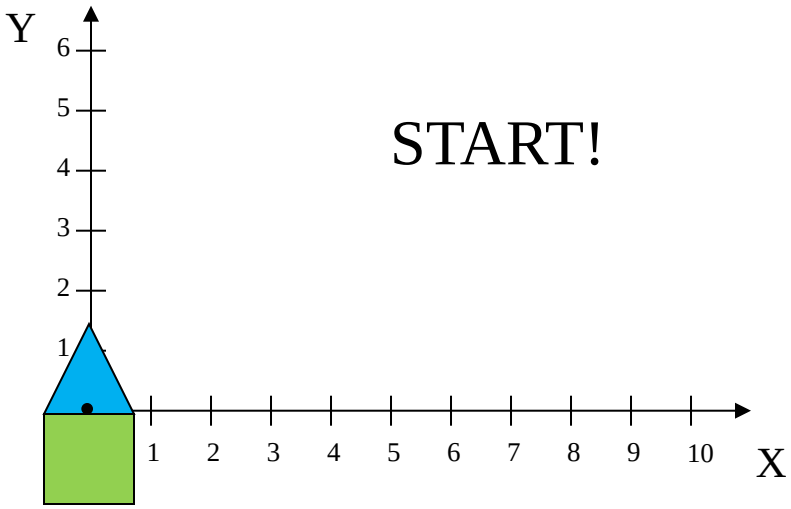
Translate()
by x=6, y=0
**then**
Rotate() by 45º




START!

Translation → Rotation

Rotate() by 45º
**then**
Translate() by
x=6, y=0

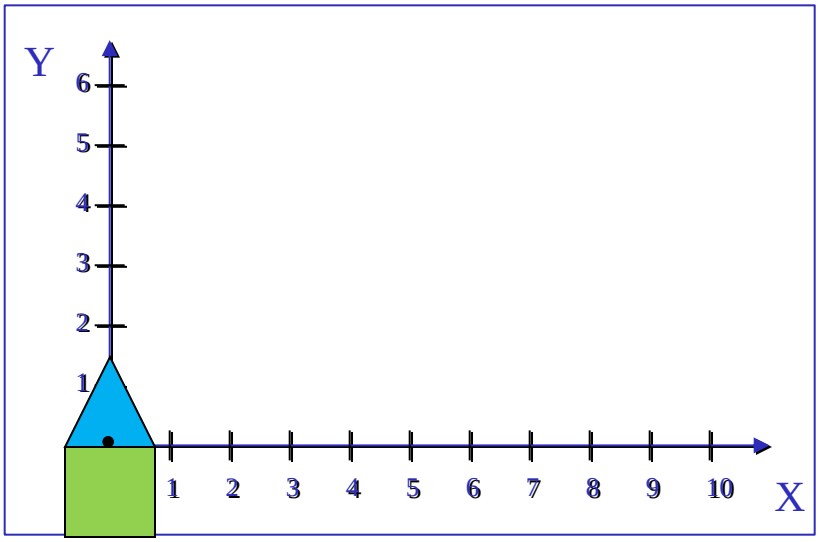



START!

Rotation → Translation

# METHOD 2:
# 1a) Copy: new coord system

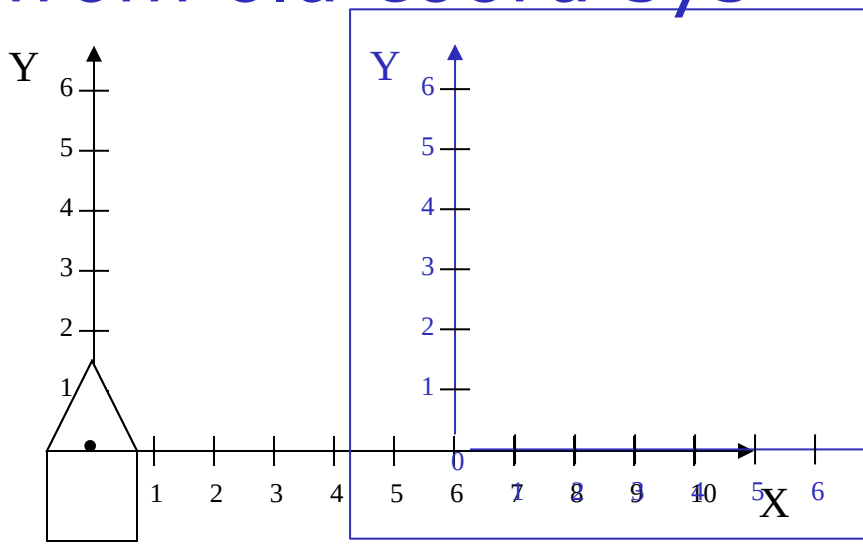
Translate()
by x=6, y=0
**then**
Rotate() by 45°




Translation → Rotation

# METHOD 2:
# 2a) Transform new coord sys as measured from old coord sys

**Translate()**
by x=6, y=0
**then**
**Rotate()** by 45°

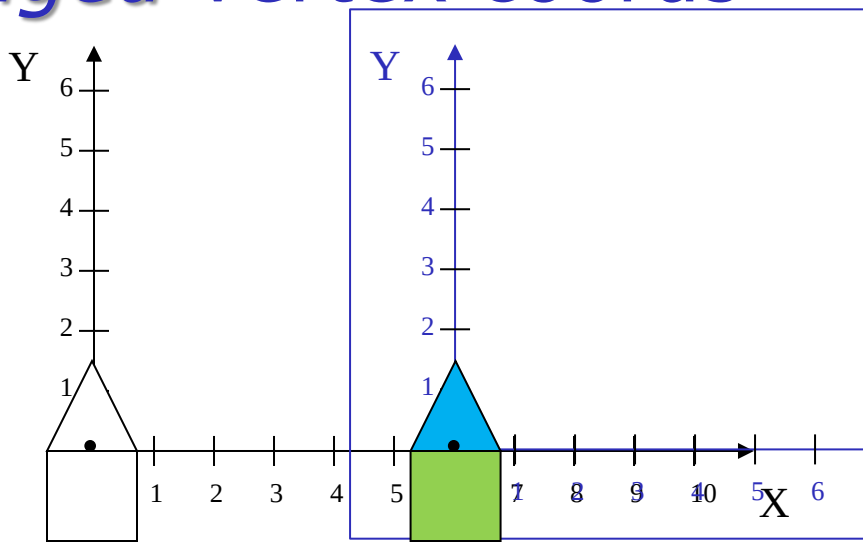



Translation → Rotation

# METHOD 2:

## 3a) Draw:in new coord system with *unchanged* vertex coords

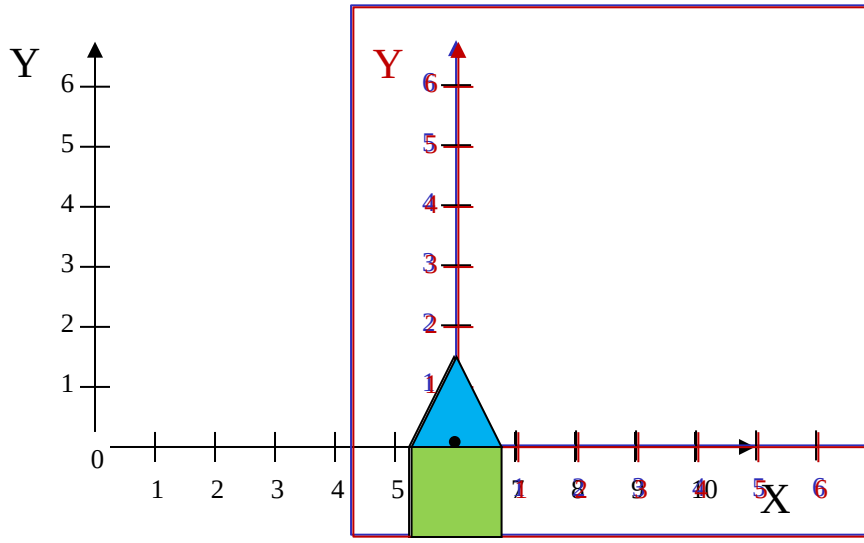
**Translate()**
by x=6, y=0
**then**
**Rotate()** by 45°




Translation → Rotation

# METHOD 2:
# 4a) Copy: new coord system

**Translate()**
by x=6, y=0
**then**
**Rotate()** by 45°

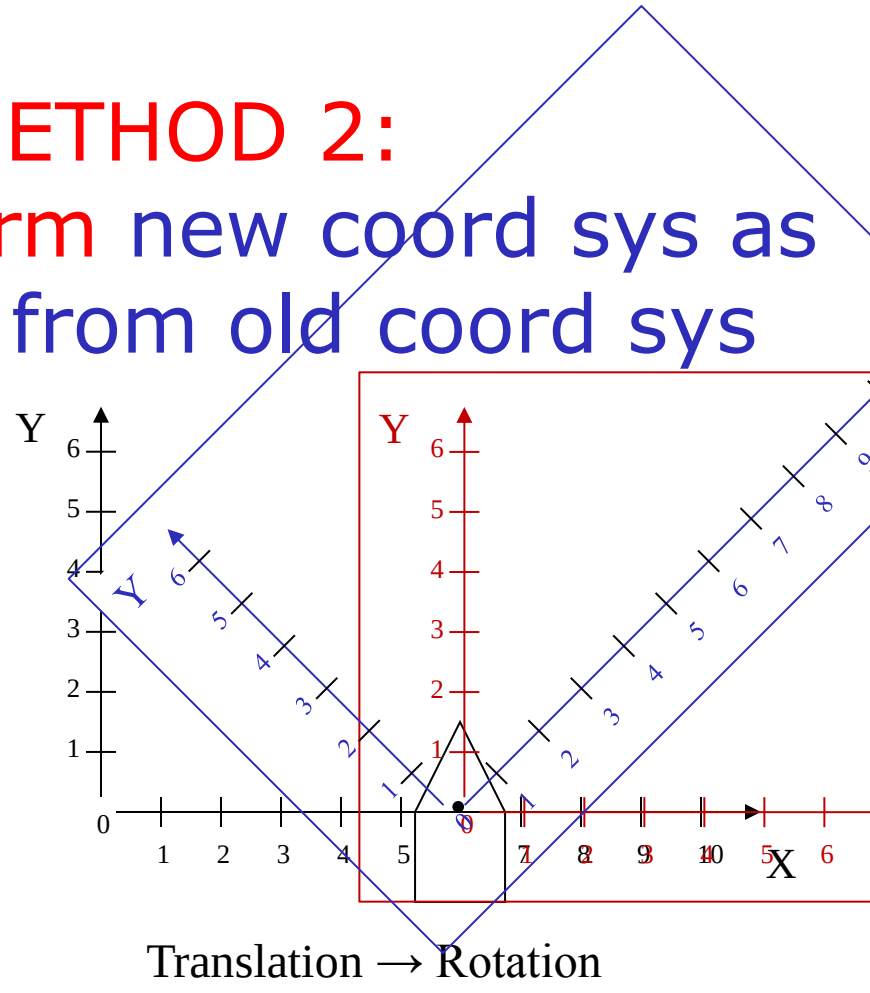



Translation → Rotation

# METHOD 2:

## 5a) Transform new coord sys as measured from old coord sys

**Translate()**
by x=6, y=0
**then**
Rotate() by 45°

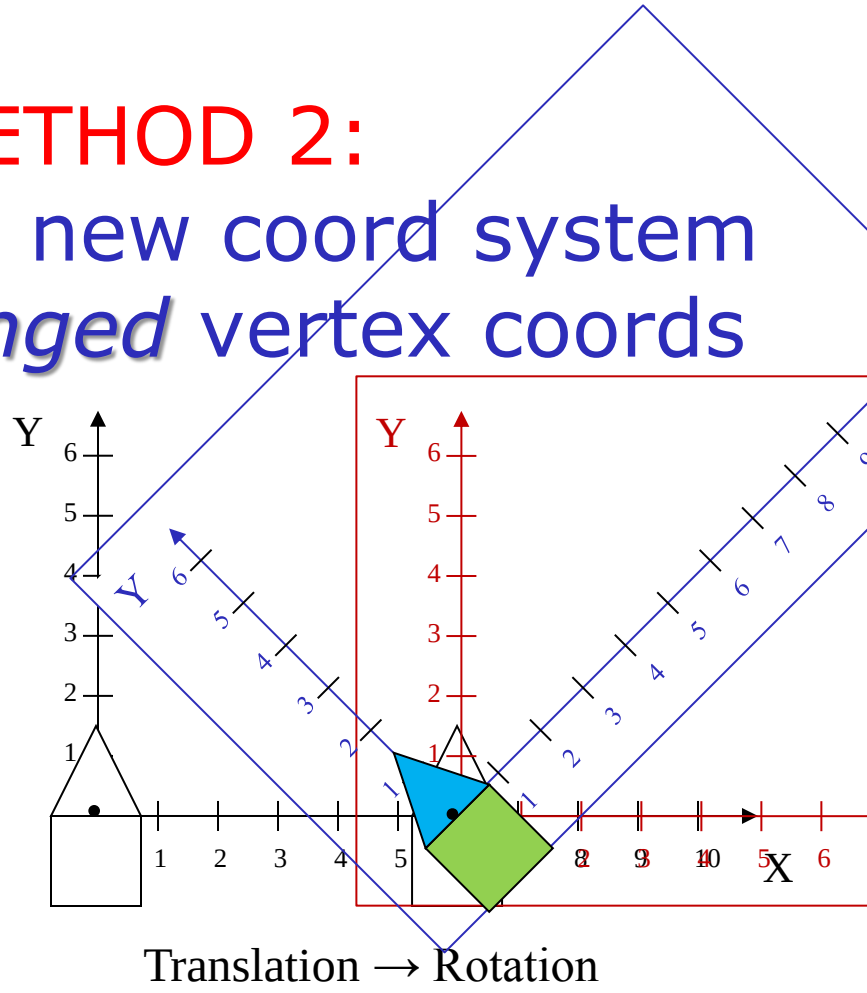



Translation → Rotation

# METHOD 2:

## 6a) Draw:in new coord system with *unchanged* vertex coords

Translate()
by x=6, y=0
**then**
Rotate() by 45°



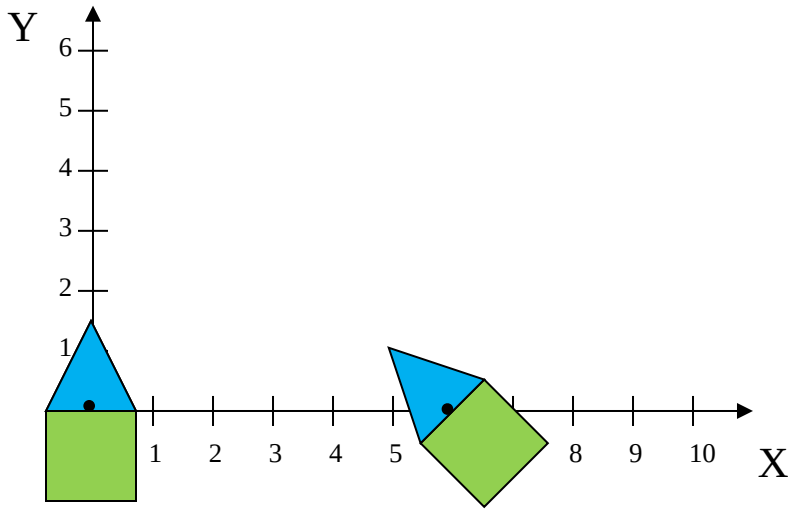

Translation → Rotation

# METHOD 2: Transformations are NOT Commutative!

Translate() by x=6, y=0

**then**

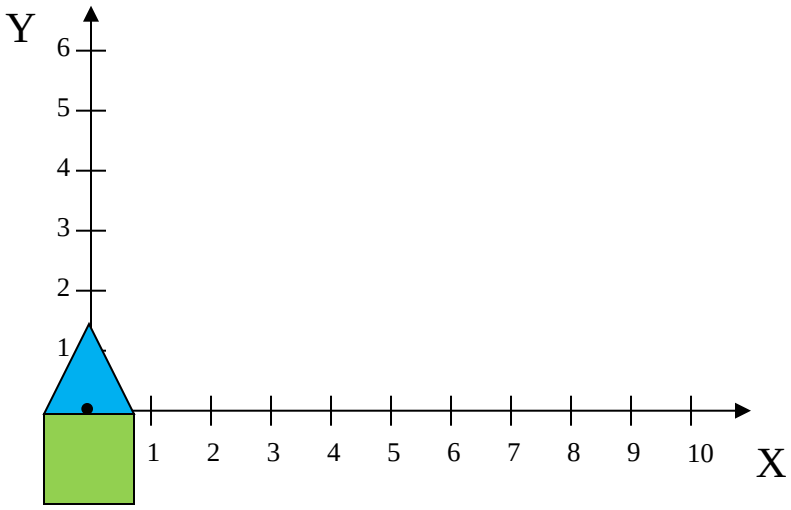
Rotate() by 45º




Translation → Rotation

# METHOD 2:
# Transformations are NOT Commutative!

Rotate() by 45°
**then**
Translate() by
x=6, y=0

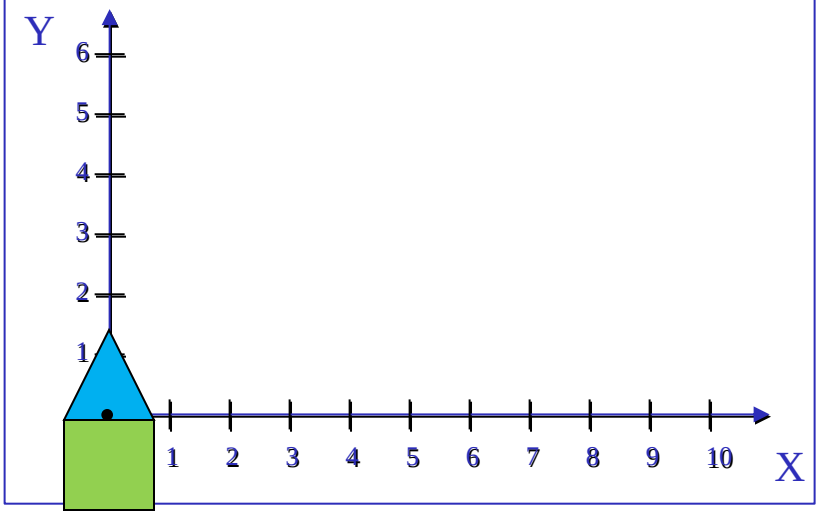



Rotation → Translation

# METHOD 2:
# 1b) Copy: new coord system

Translation → Rotation

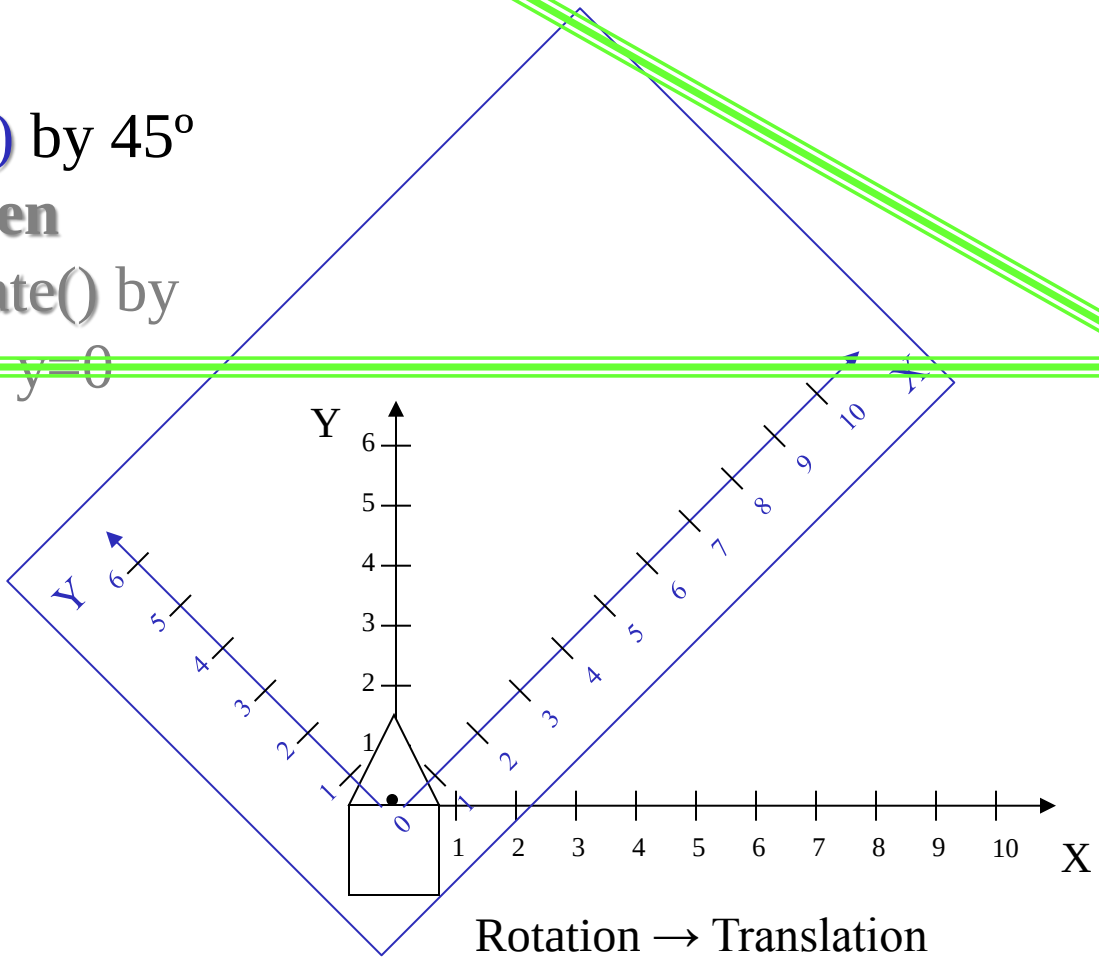
Rotate() by 45°
**then**
Translate() by
x=6, y=0




Rotation → Translation

# METHOD 2:
## 2b) Transform new coord sys as measured from old coord sys

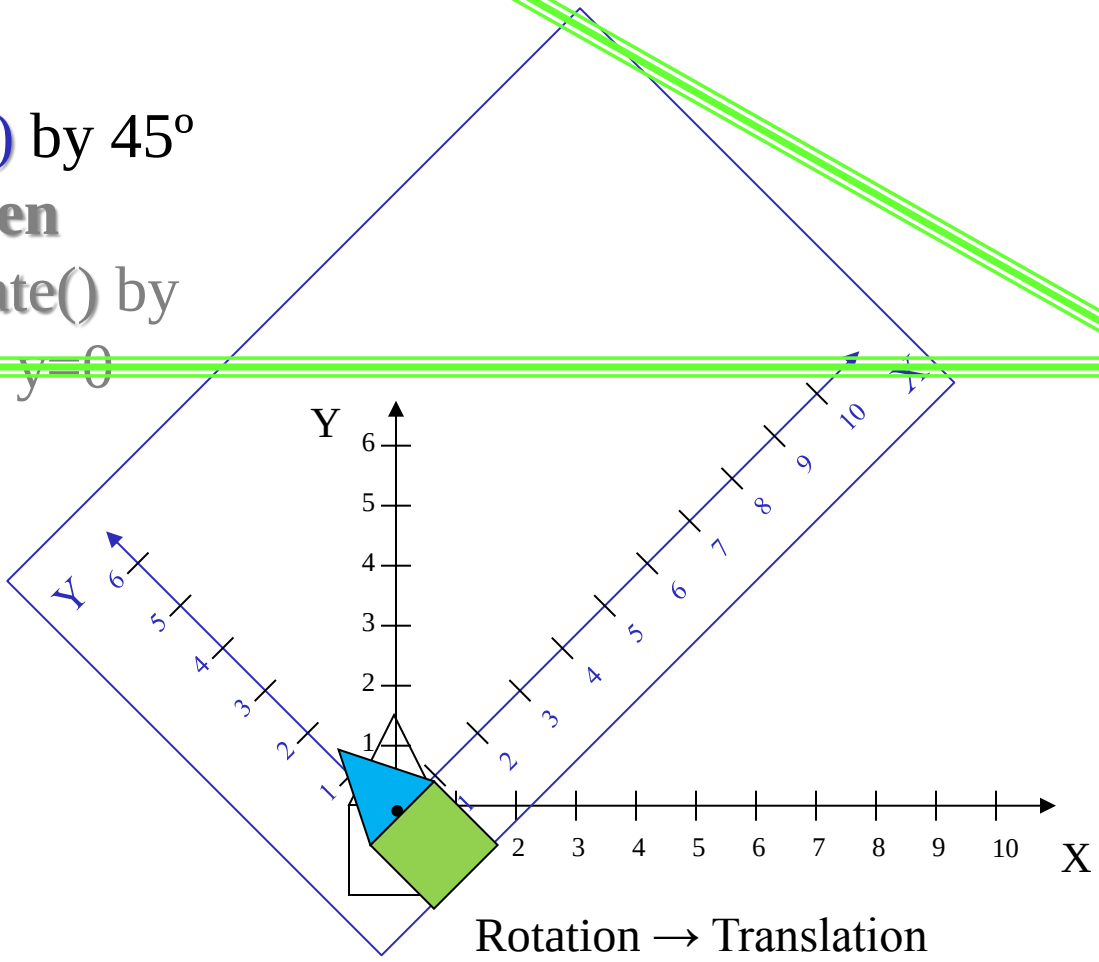
Rotate() by 45°
**then**
Translate() by
x=6, y=0



# METHOD 2:

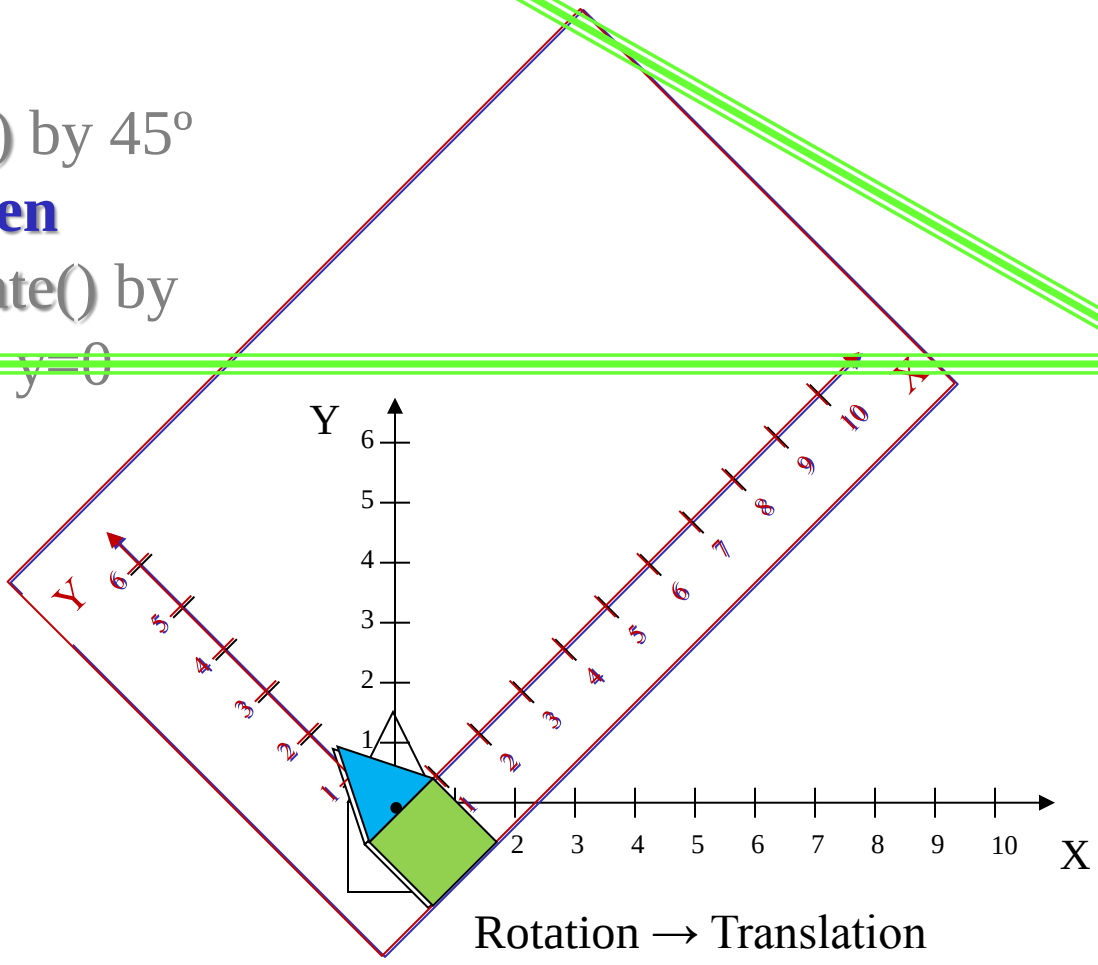
## 3b) Draw:in new coord system with *unchanged* vertex coords

Rotate() by 45°
**then**
Translate() by
x=6, y=0



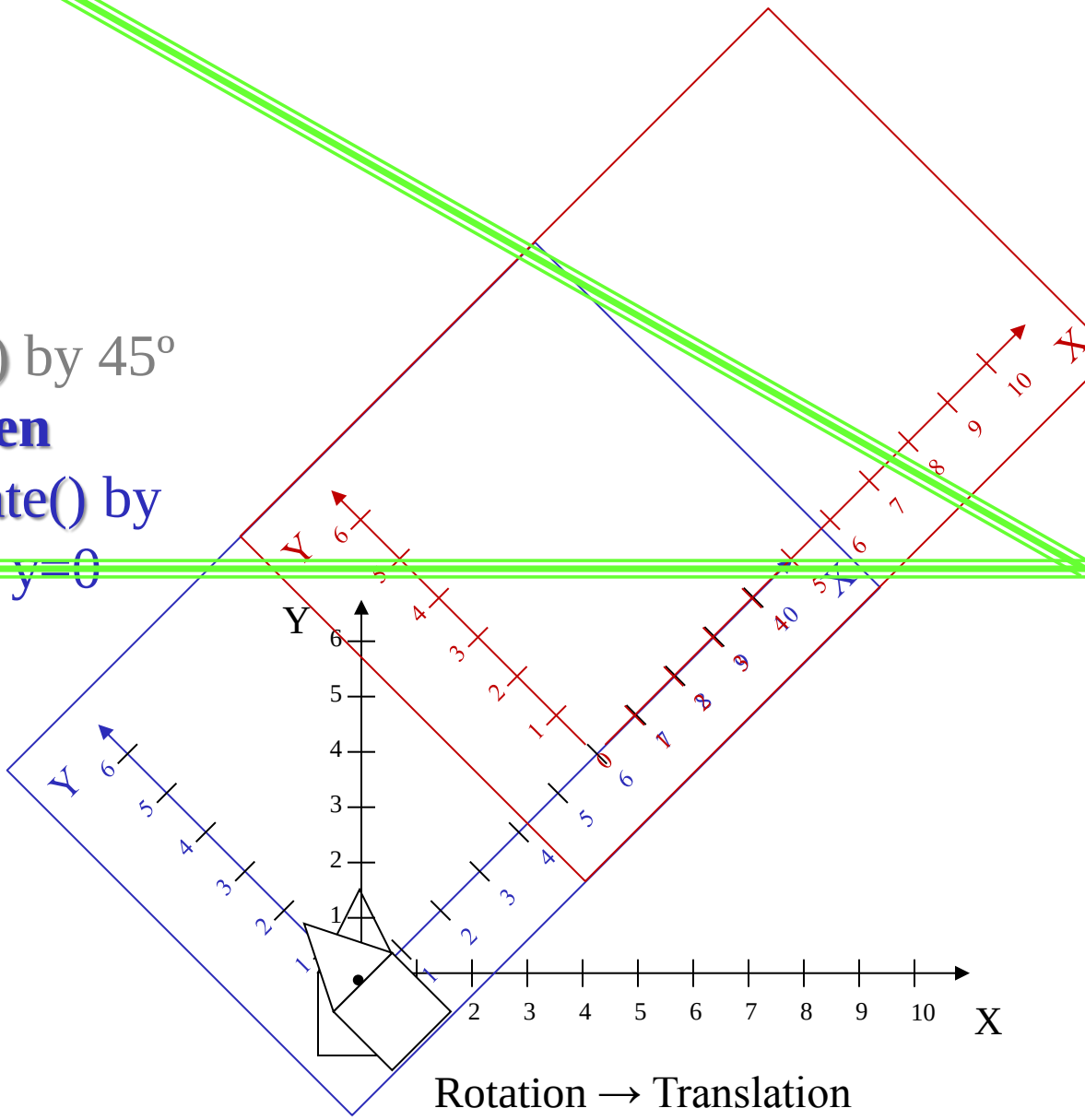
# METHOD 2:
# 4b) Copy: new coord system

Rotate() by 45°
**then**
Translate() by
x=6, y=0



# METHOD 2:
# 5b) Transform new coord sys as measured from old coord sys

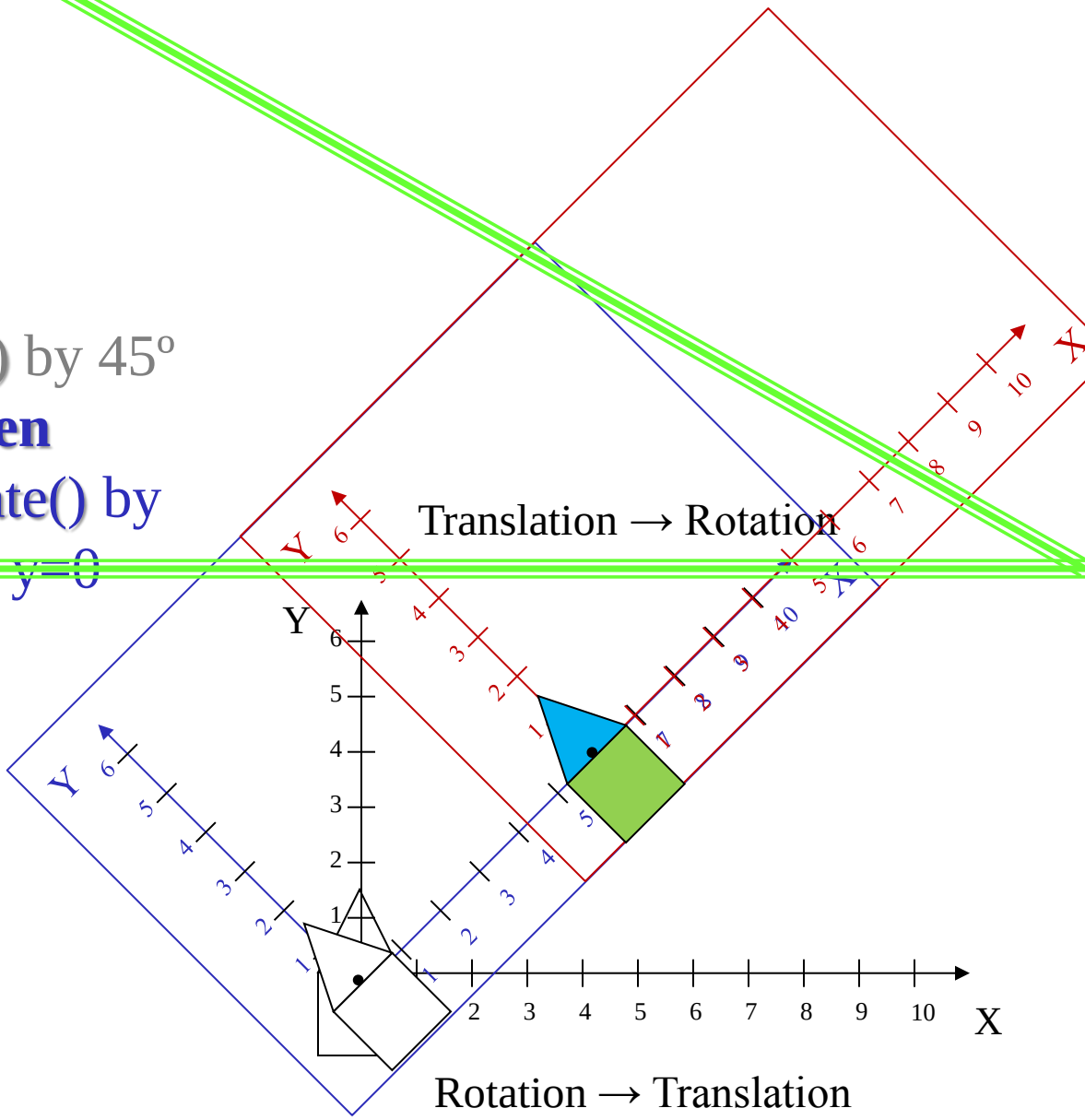
Rotate() by 45°
**then**
Translate() by
x=6, y=0



# METHOD 2:

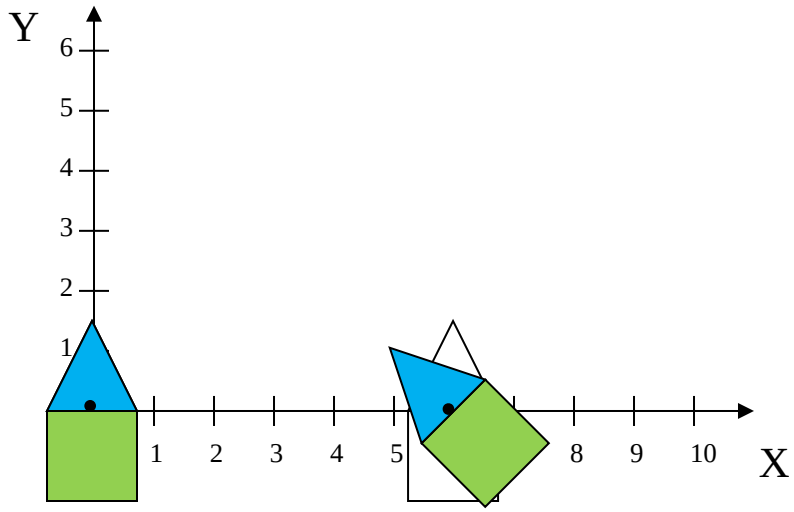
## 6b) Draw:in new coord system with *unchanged* vertex coords

Rotate() by 45°
**then**
Translate() by
x=6, y=0



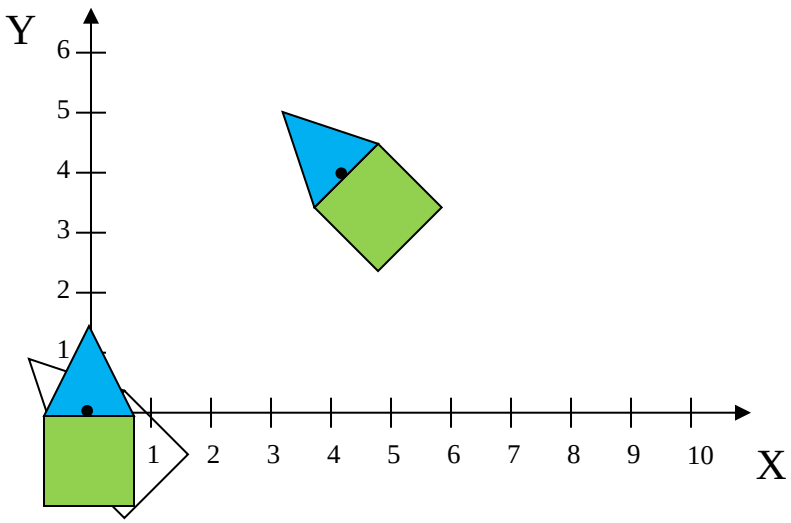
# METHOD 2:
# Transformations are NOT Commutative!

Translate by x=6, y=0
**then**
Rotate by 45º




Translation → Rotation

Rotate by 45º
**then**
Translate by x=6, y=0

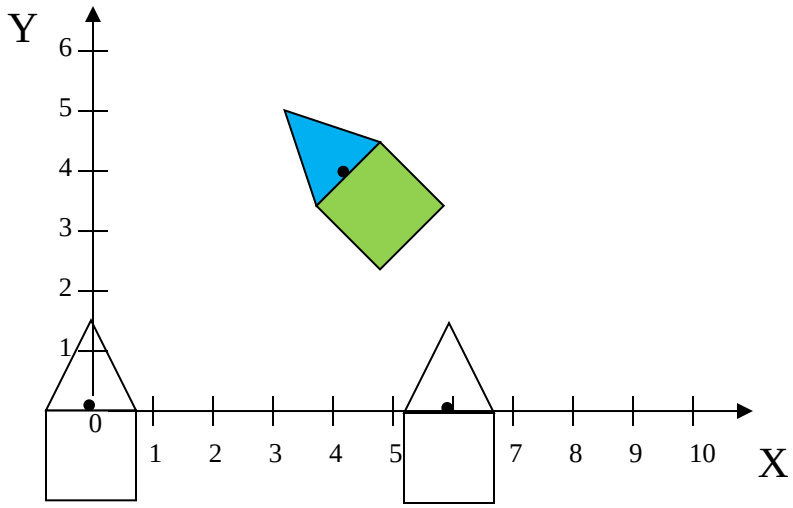



Rotation → Translation

# METHOD 1:

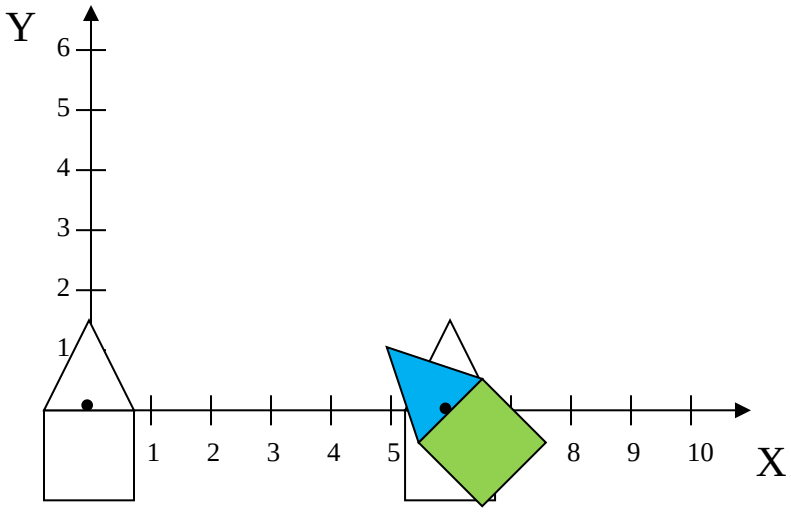
# Transformations are NOT Commutative!

Translate() by x=6, y=0 **then** Rotate() by 45º




Translation → Rotation

Rotate() by 45º **then** Translate() by x=6, y=0




Rotation → Translation

# METHOD 1, 2 Inverse?

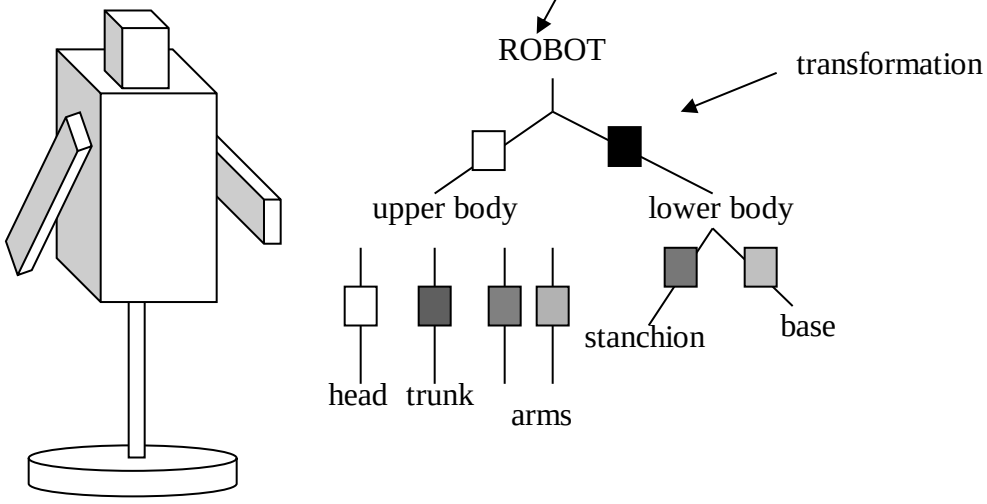
- Yep, that's right….
- Method 1:

$$[M] = [M_{new}][M_{old}]$$

- Method 2:

$$[M] = [M_{old}][M_{new}]$$

# How Can We Organize MANY Geometric Transformations (T,R,S) To Build Jointed Objects?




- **Answer:**

  A TREE of TRANSFORMATIONS, SHAPES, & ATTRIBUTES

- An "is-composed-of" hierarchy, arranged as a DAG (Directed Acyclic Graph)
- that performs a 'coarse-to-fine' decomposition of a jointed object made of individual parts, such as this robot…

- The completed tree we will devise is called a

**"SceneGraph"**

# Scene Graphs Assembly (1/4)

- A *scene graph* is a directed acyclic graph (DAG) that describes a complete 3D scene + cameras
- Examples:
  - Open Scene Graph (used in the Cave)
  - Sun's Java3D™
  - X3D ™ (VRML ™ was a precursor to X3D)

**Typical scene graph node types:**

- **Object Nodes** (cubes, sphere, cone, triangle etc.) describe re-usable shapes as fixed sets of connected vertices (default: unit size, centered at origin)

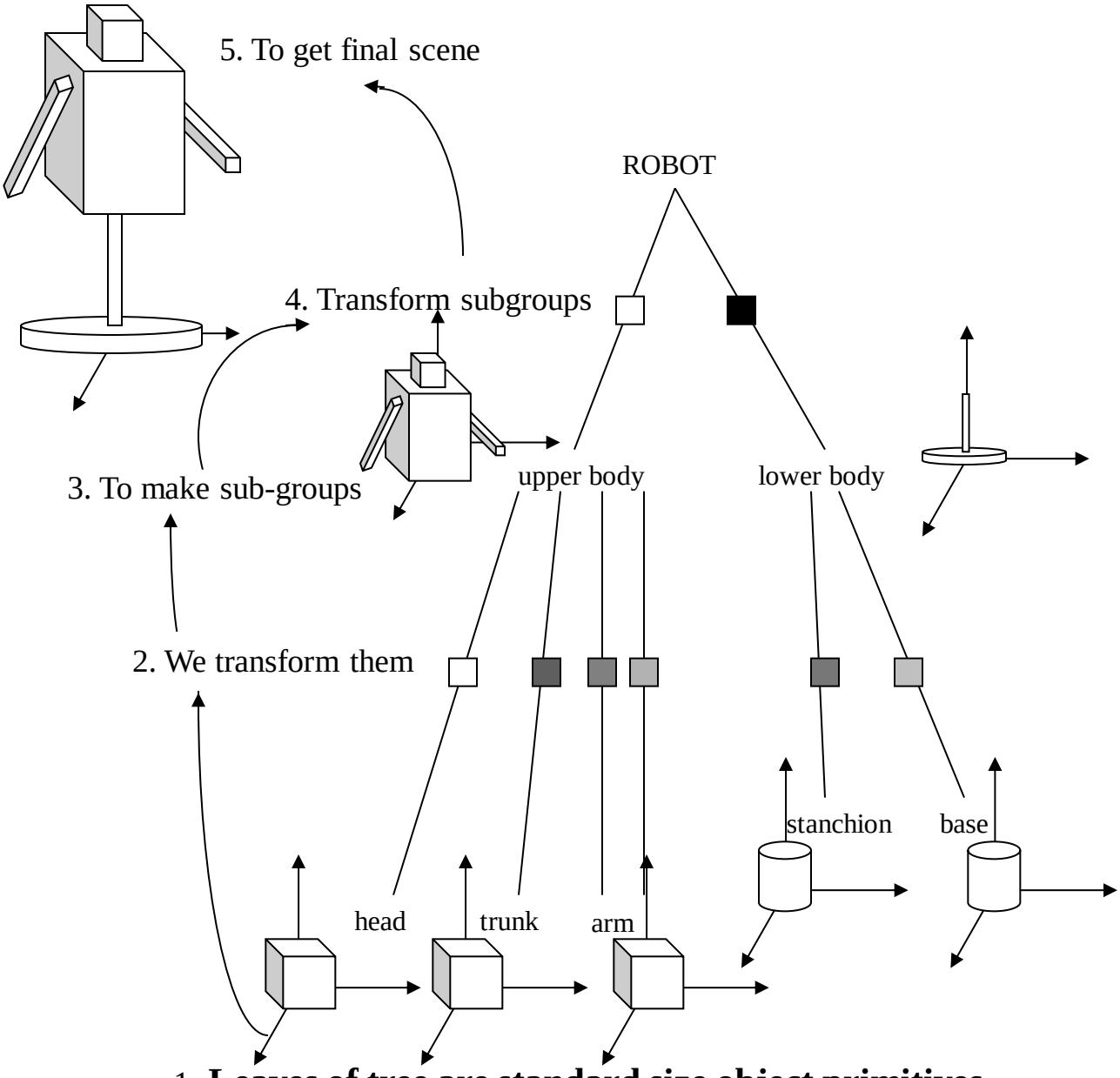
- **Drawing Attribute Nodes** (Apply a new color, line width, shading type, texture map, etc.)
  Describe how to render the nodes below us…

- **Transformation Nodes** describe parameterized T,R,S matrices used to 'pose' the nodes below it

- **Group Nodes** describe collections that share all the attributes & transforms above it in the scene graph
  Helps with 'instancing': modified copies of objects

# Scene Graph Assembly (2/4)

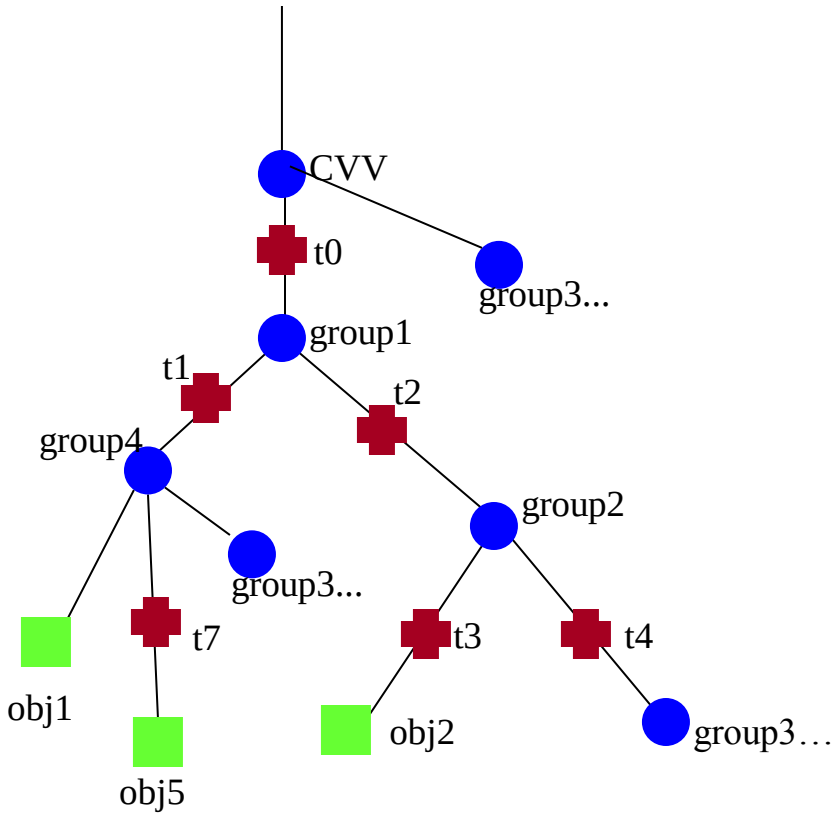
How to Draw the Robot:




1. **Leaves of tree are standard size object primitives**

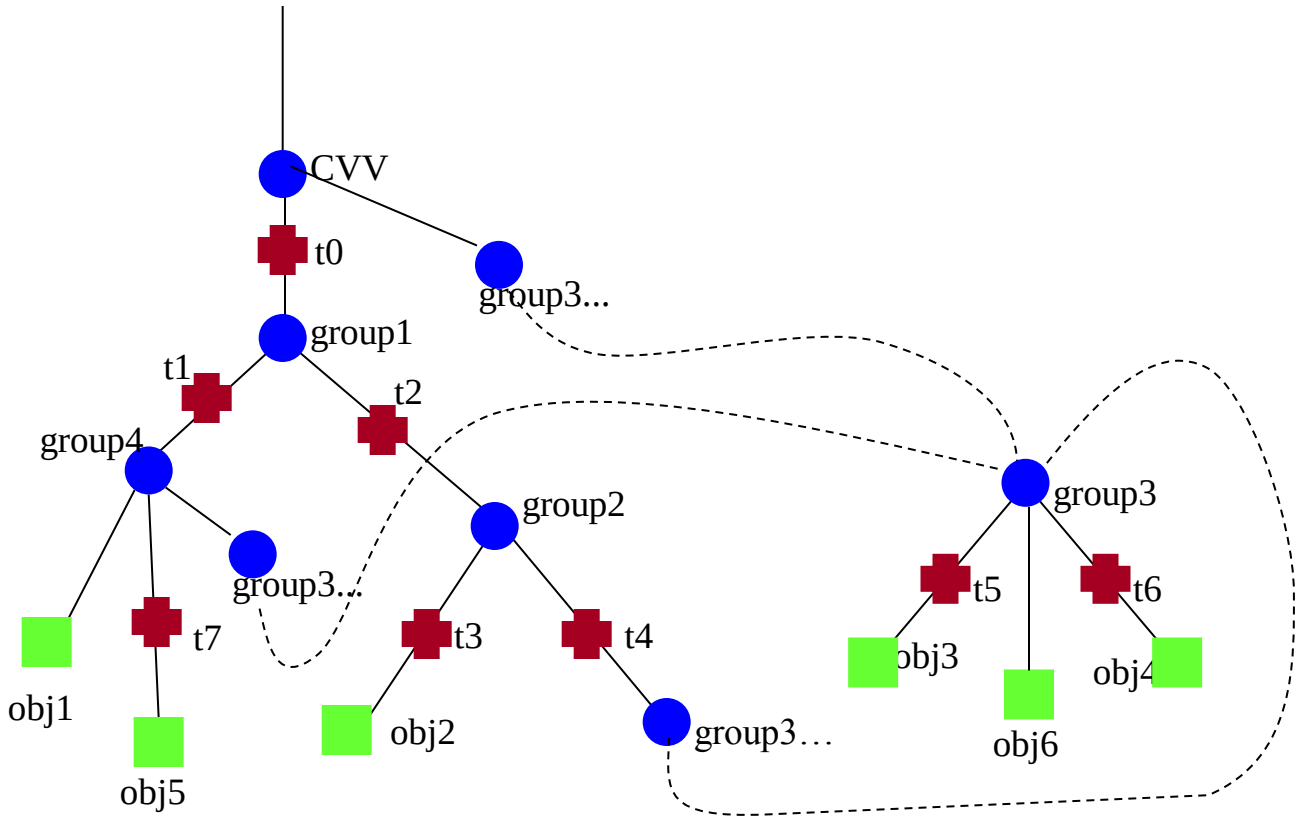
# Scene Graph Assembly (3/4)

- **Group Nodes:**
  - Defines one shared coordinate system used by multiple child nodes below
  - enables 'instancing' – easy object re-use
  - Depicts complex graphs in smaller pieces



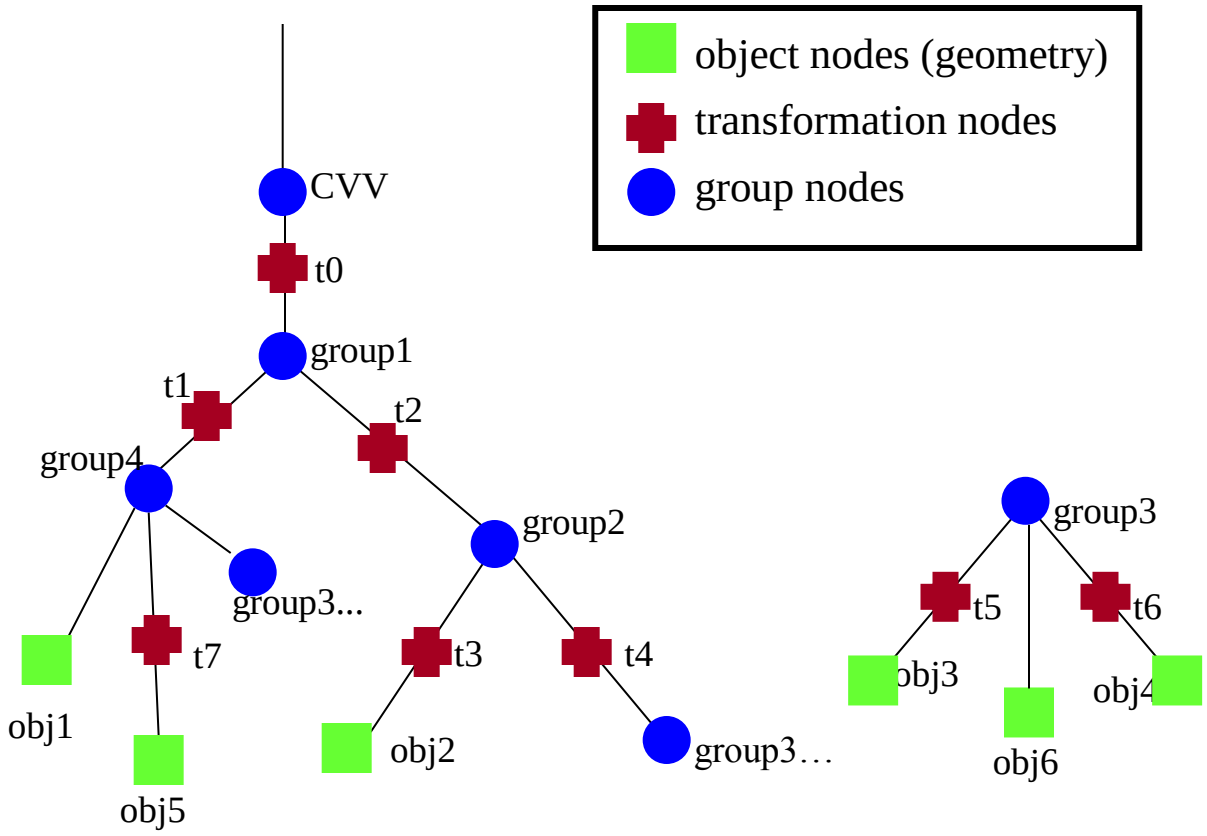
# Scene Graph Assembly (3/4)

- **Group Nodes:**
  - Defines one shared coordinate system used by multiple child nodes below
  - enables 'instancing' – easy object re-use
  - Depicts complex graphs in smaller pieces
  - EXAMPLE: **group3** used three times below: in the **world** group, in **group2** and in **group4**



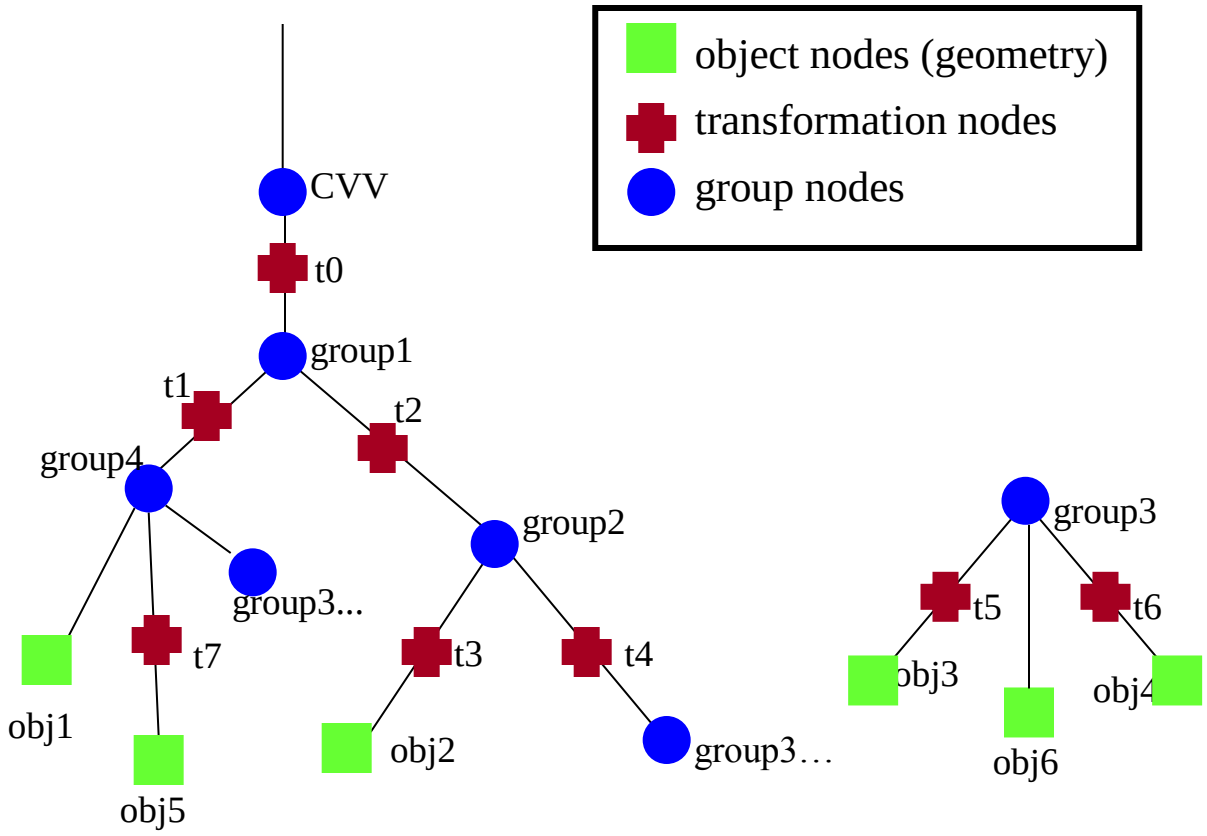
# Scene Graph Assembly (4/4)

- **Rendering with WebGL commands:**
  - Traverse graph from root (top)
    to leaf (bottom)
    in depth-first order
  - Drop into Group Node? Call glPushMatrix()
  - Rise above Group Node? Call glPopMatrix().
  - Arrive at Object Node? Call it's '`drawMe()` fcn.



# Scene Graph Duality:

- **WebGL commands** traverse **top→bottom**
  **Vertex coordinates** traverse **bottom→top**
- from memory to screen; through matrices (GL_MODELVIEW→GL_PROJECTION →CVV clip →viewport→ display window pixels.



# Scene Graph Duality:

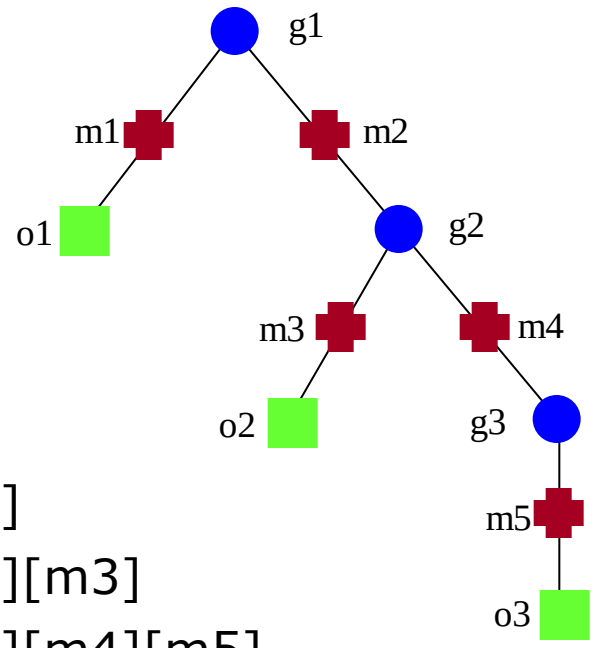
- Example:

**g**: group nodes

**m**: matrices of transform nodes

**o**: object nodes

**CTM**: composed transform. matrix




- for object o1, CTM = [m1]
- for object o2, CTM = [m2][m3]
- for object o3, CTM = [m2][m4][m5]

To convert vertex coordinate values **v** in **o3** to its 'world' or 'root' coordinates values **r:**

**r** = [m2][m4][m5]v

To build that matrix in WebGL:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gl<Do-the-M2-transform>();
gl<Do-the-M4-transform>();
gl<Do-the-M5-transform>();
```

**"Reversed!" (Duality)**