University of Minnesota
Driven to Discover®

Homogeneous Coordinates

CSCI 4611: Programming Interactive Computer Graphics and Games

Evan Suma Rosenberg | CSCI 4611 | Fall 2022

Review: 2D Translation

Translation is the component-wise addition of vectors

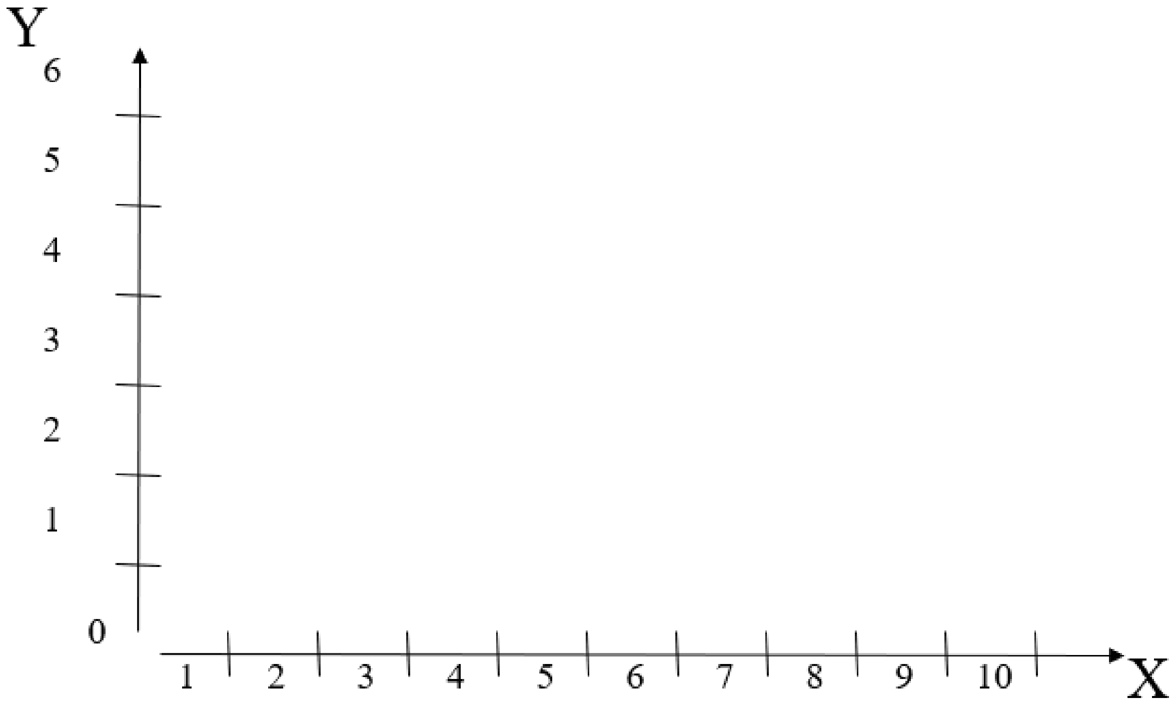
v' = v + t where

$$v = \begin{bmatrix} x \\ y \end{bmatrix}, \quad v' = \begin{bmatrix} x' \\ y' \end{bmatrix}, \quad t = \begin{bmatrix} dx \\ dy \end{bmatrix}$$

and x' = x + dx

y' = y + dy

Operation is isometric (preserves lengths)



Review: 2D Scaling

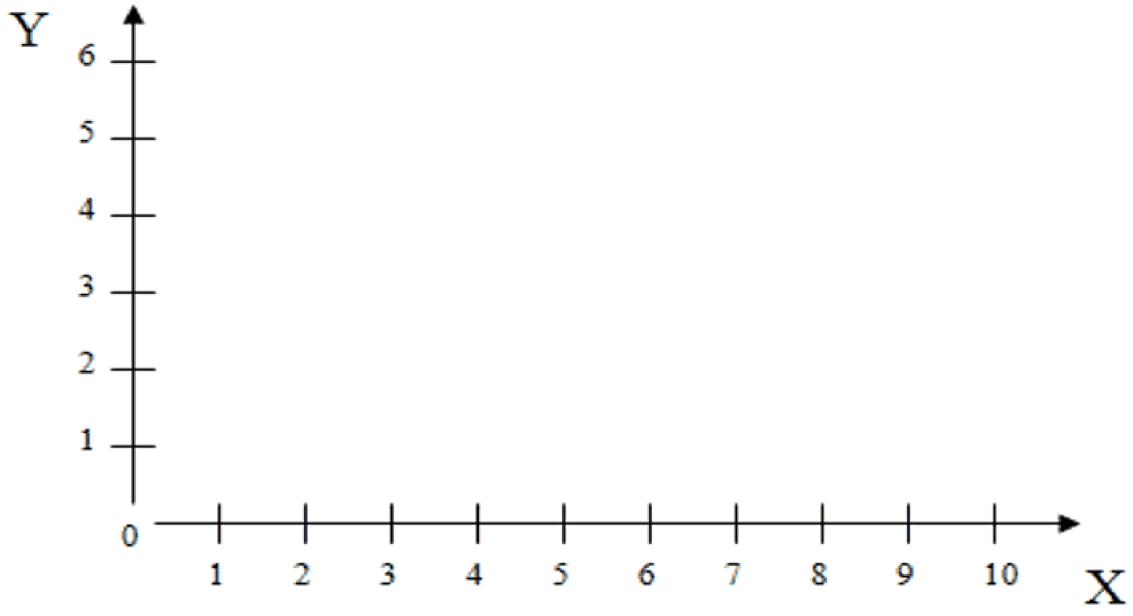
Scaling is the component-wise multiplication of vectors

v' = Sv where

$$v = \begin{bmatrix} x \\ y \end{bmatrix}, \quad v' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

and

$$S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \qquad \begin{aligned} x' &= s_x x \\ y' &= s_y y \end{aligned}$$



Does not preserve lengths

Does not preserve angles

(unless scaling is uniform)

Review: 2D Rotation

Rotation of Θ about the origin

$v' = R_{\Theta}$ where

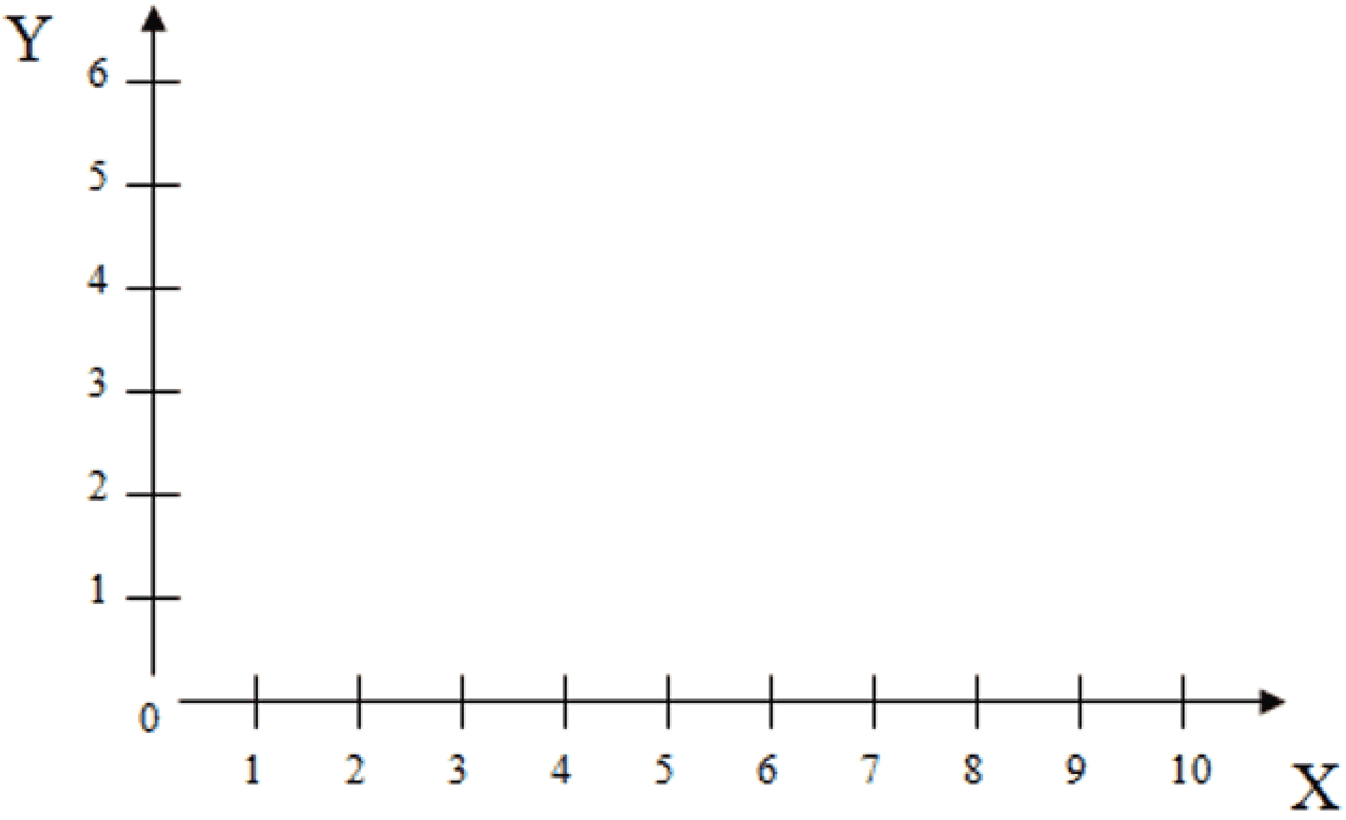
$$v = \begin{bmatrix} x \\ y \end{bmatrix}, \quad v' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

$$R_{\theta} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

and $x' = x\cos\Theta - y\sin\Theta$

$y' = x\sin\Theta - y\cos\Theta$

A rotation of zero (no rotation) is results in the identity matrix



2D Rotation and Scale are Relative to the Origin

Suppose we want to scale and rotate an object that is **not centered at origin**.

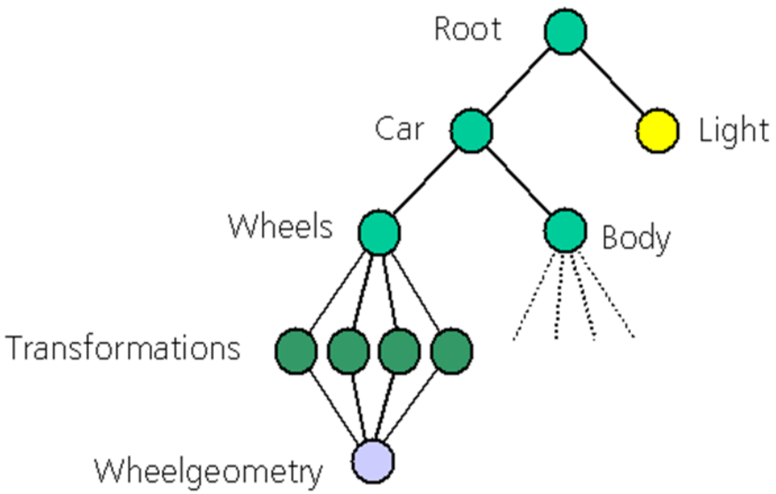
Solution: move to the origin, scale and/or rotate in its local coordinate system, then move it back.

This sequence suggest the need to compose successive transformations....

Hierarchical Transformations

The scene graph contains a hierarchical representation of the spatial relationship between objects.

This also requires us to be able to compose multiple transformations!



Composing Transformations

Translation, scaling and rotation are expressed as:

$$\text{translation:} \quad v' = v + t$$

$$\text{scale:} \quad v' = Sv$$

$$\text{rotation:} \quad v' = Rv$$

Composition is difficult to express because does not use matrix multiplication!

Homogeneous Coordinates

translation: $v' = v + t$

scale: $v' = Sv$

rotation: $v' = Rv$

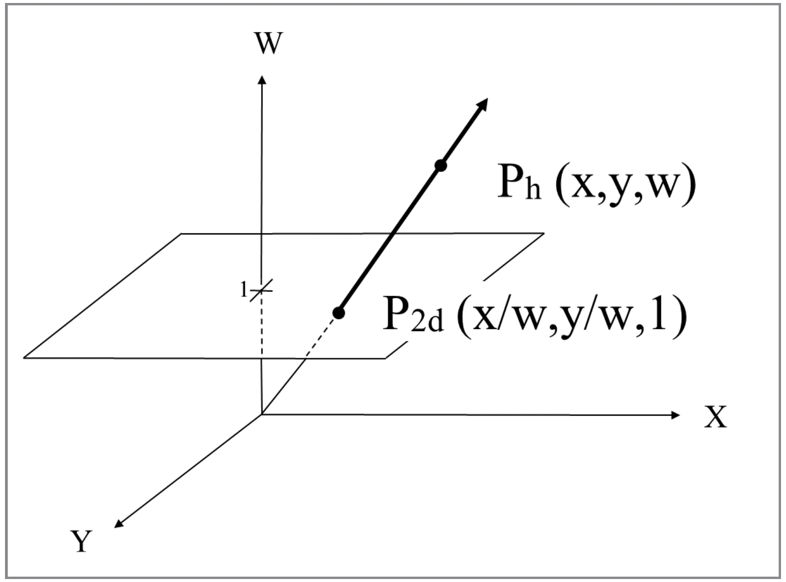
Homogeneous coordinates allow expression of all three transformations as 3x3 matrices for easy composition.

$$\mathrm{p} = (\mathrm{x}, \mathrm{y}) \quad \text{becomes} \quad \mathrm{p} = (\mathrm{x}, \mathrm{y}, 1)$$

This conversion does not transform *p*. It only changes notation to show that it can be viewed as a point on $w = 1$ hyperplane.

What is $\begin{bmatrix} x \\ y \\ w \end{bmatrix}$?

P_{2d} is intersection of line determined by P_h with the $w = 1$ plane.



Two sets of coordinates that are proportional denote the same point of projective space:
For any non-zero scalar c, (cx, cy, ..., cw) denotes the same point.

Homogeneous Transformations in 2D

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

For points written in homogeneous coordinates, translation, scaling and rotation relative to the origin are expressed homogeneously as:

$$T_{(d_x,d_y)} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \qquad v' = T_{(d_x,d_y)}v$$

$$S_{(s_x,s_y)} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad v' = S_{(s_x,s_y)}v$$

$$R_{(\phi)} = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad v' = R_{(\phi)}v$$

Examples

Translate [1,3] by [7,9]

$$\begin{bmatrix} 1 & 0 & 7 \\ 0 & 1 & 9 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 8 \\ 12 \\ 1 \end{bmatrix}$$

Scale [2,3] by 5 in the X direction and 10 in the Y direction

$$\begin{bmatrix} 5 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 10 \\ 30 \\ 1 \end{bmatrix}$$

Rotate [2,2] by 90° (π/2)

$$\begin{bmatrix} \cos(\pi/2) & -\sin(\pi/2) & 0 \\ \sin(\pi/2) & \cos(\pi/2) & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ 2 \\ 1 \end{bmatrix}$$

Points and Vectors in Homogeneous Coordinates

- **For points add a 1.**

  e.g. point (2,3) becomes (2,3,1)

- **For vectors add a 0.**

  e.g. vector <1,4> becomes <1,4,0>

- **This is pretty slick…**

  Try the examples on the last slide, but with vectors rather than points.

  Change the w coordinate to zero so [2 3 1] becomes [2 3 0] and [1 3 1] becomes [1 3 0].

- **You will see that we can scale and rotate vectors, but we can't translate them!**

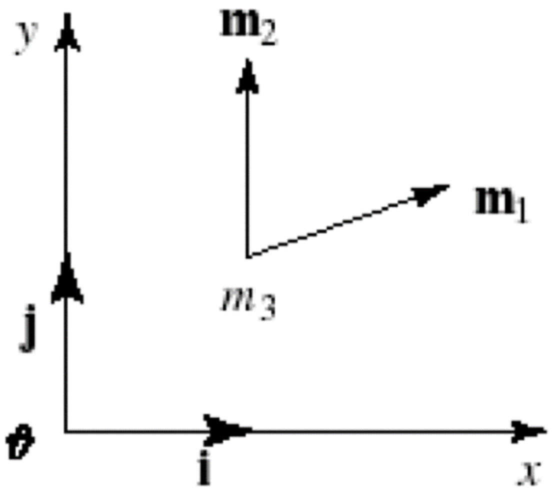
  This makes sense given the definitions of points and vectors that we talked about last class. It doesn't make sense to "translate" a vector because it doesn't have a location! So, this is perfect. Linear algebra rocks!

What do the Columns Tell Us?

- The first two columns are:
  - vectors (3rd component is 0)
  - the X-axis and Y-axis of the coordinate frame specified by the transformation
  - if a rotation matrix, these columns will show the vectors into which the X and Y-axes rotate.

- The third column is:
  - a point (3rd component is 1)
  - the origin of the coordinate frame

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix} = \left(m_1 \mid m_2 \mid m_3 \right)$$



Homogeneous Coordinates in Practice

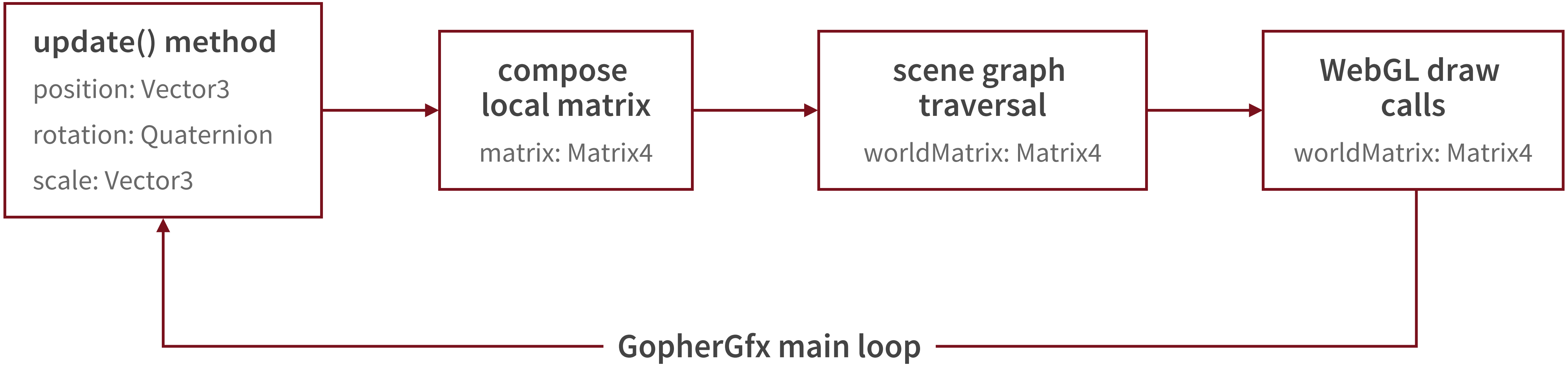
Everyone uses homogeneous coordinates and matrices.

Inside the low-level software and hardware, the *w* is always there... it needs to be in order to do projection and transformations, etc.

In practice, the w component (1=point, 0=vector) is often hidden inside an API.

GopherGfx uses the Vector3 class for both points and vectors, but then it needs to include two separate matrix methods that implicitly use the 1 or 0.

What happens under the hood?



Composing Transformations

Apply a sequence of transformations:

$$\mathbf{v}' = \mathbf{M}_4 \left(\mathbf{M}_3 \left(\mathbf{M}_2 \left(\mathbf{M}_1 \ \mathbf{v}\right)\right)\right)$$

Because matrix algebra obeys the associative law, we can regroup this as:

$$\mathbf{v}' = \left(\mathbf{M}_4 \ \mathbf{M}_3 \ \mathbf{M}_2 \ \mathbf{M}_1\right) \mathbf{v}$$

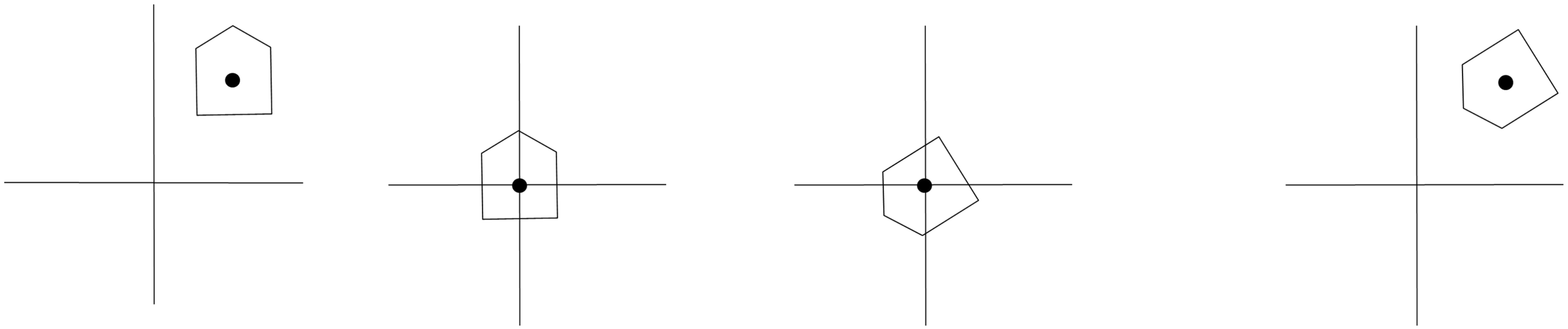
This allows us to compose the transformations into a single matrix:

$$\mathbf{M}_{total} = \mathbf{M}_4 \ \mathbf{M}_3 \ \mathbf{M}_2 \ \mathbf{M}_1$$

$$\mathbf{v}' = \mathbf{M}_{total} \ \mathbf{v}$$

Matrix Multiplication is NOT Commutative

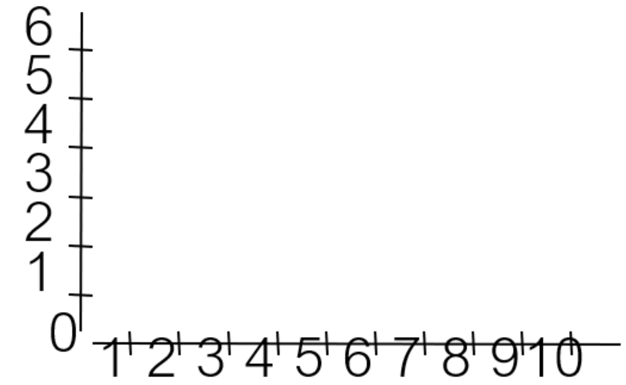
$House\ (H)$ $\quad T(dx, dy)H \quad R(\theta)T(dx, dy)H \quad T(-dx, -dy)R(\theta)T(dx, dy)H$



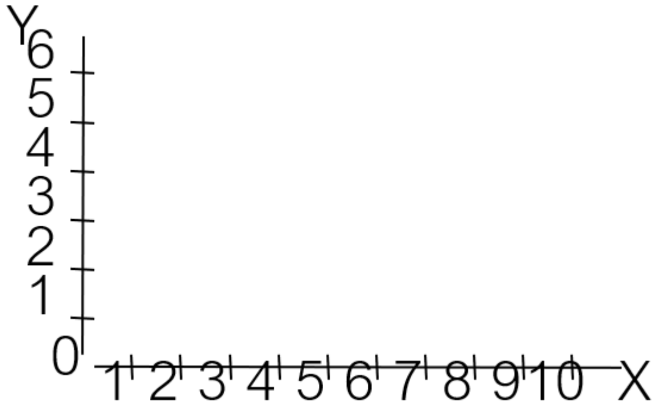
Matrix multiplication is not **commutative**. The order of transformations matters!

Matrix Multiplication is NOT Commutative

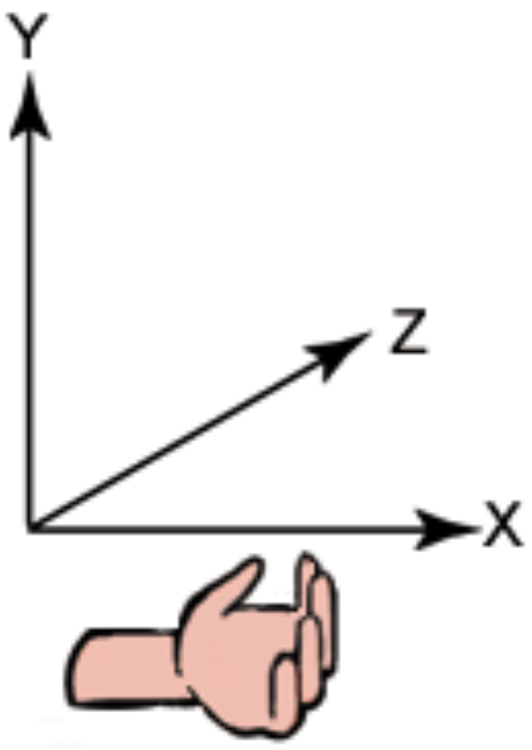
Translate by
x=6, y=0 then
rotate by 45º



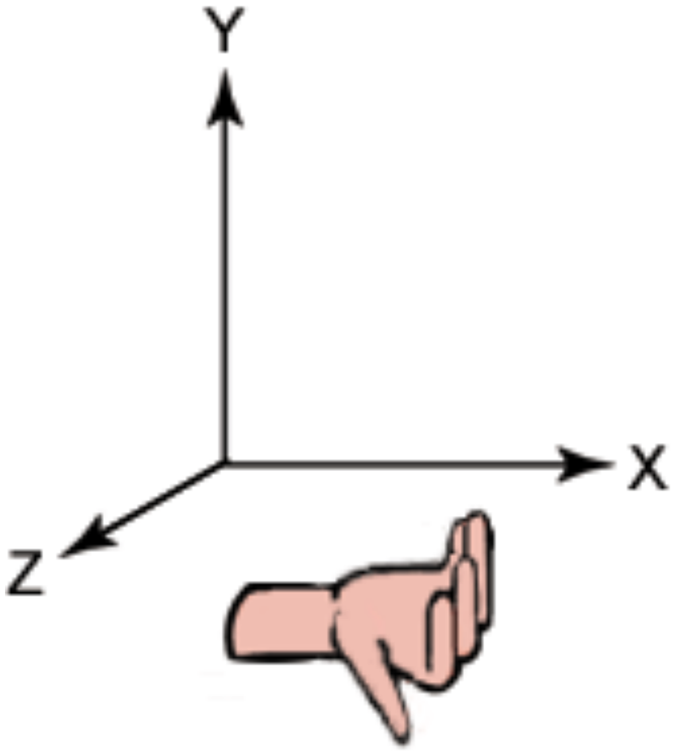
Rotate by 45º
then translate
by x=6, y=0



Review: Left Hand vs. Right Hand Coordinate Systems



Left-Handed Coordinate System



Right-Handed Coordinate System

Image Credit: https://learn.microsoft.com/en-us/previous-versions/windows/desktop/bb324490(v=vs.85)

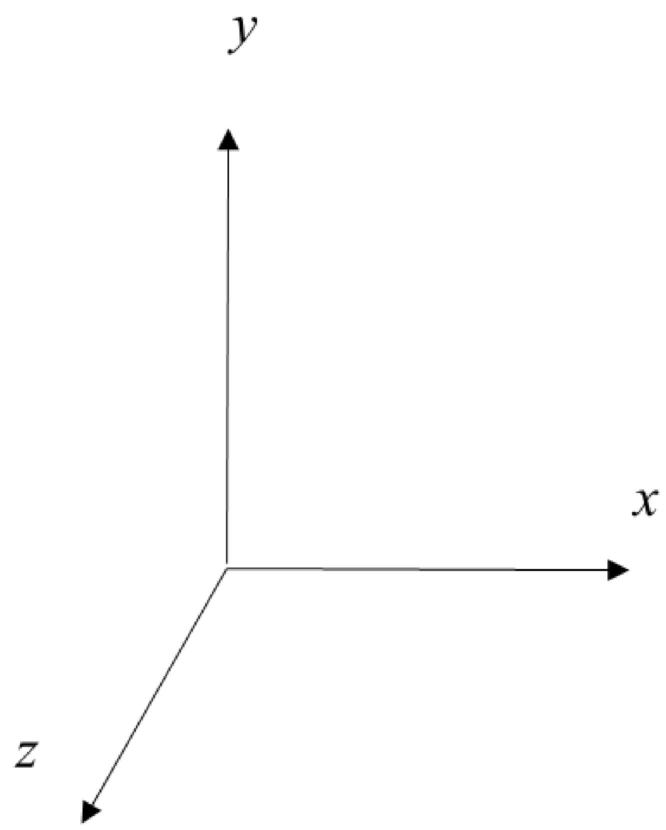
Review: 3D Transformations (Right Handed)

Translation

$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Review: 3D Transformations (Right Handed)

Rotation about X-axis

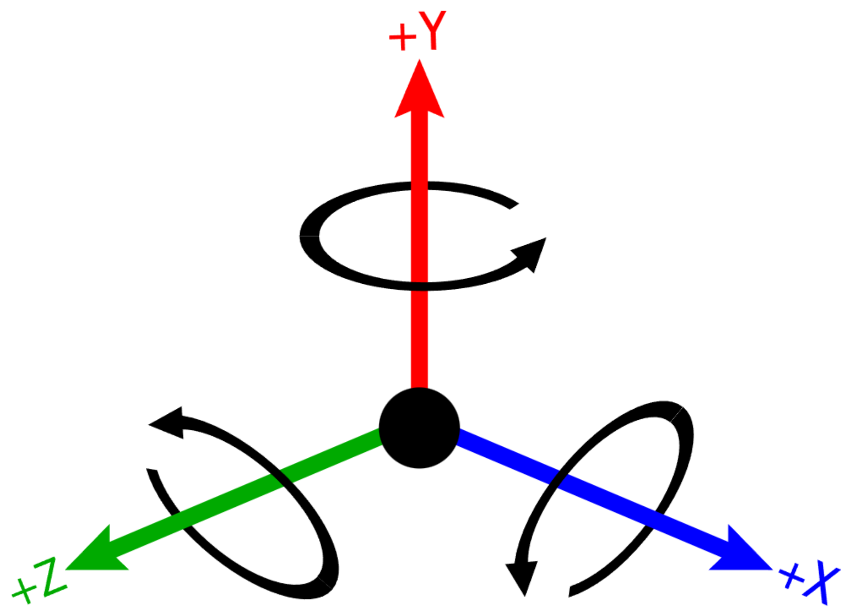
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about Y-axis

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about Z-axis

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Programming with Transforms

- **GopherGfx has a Matrix4 class.**

  It stores 4x4 transformation matrices.

- **Why Matrix4 when we are in 3D?**

  Homogenous coordinates!

- **You can often set up objects in a scene graph without manipulating matrices directly.**

  Under the hood, transformations are ultimately represented as matrices.

  The object's world matrix is computed each frame by composing hierarchical transformations using matrix multiplication.

Programming with Transforms

Matrix4 has special utility routines to help us create 4x4 matrices for the basic transformations.

```
matrix.makeTranslation(v: Vector3);
matrix.makeScale(v: Vector3);
matrix.makeRotationX(angle: number);
matrix.makeRotationY(angle: number);
matrix.makeRotationZ(angle: number);
```

What about rotation about an arbitrary axis?

Review: Combining 3D Rotations

One way to build a rotation in 3D is by composing three elementary rotation transformations:

an x-rotation(**pitch**),

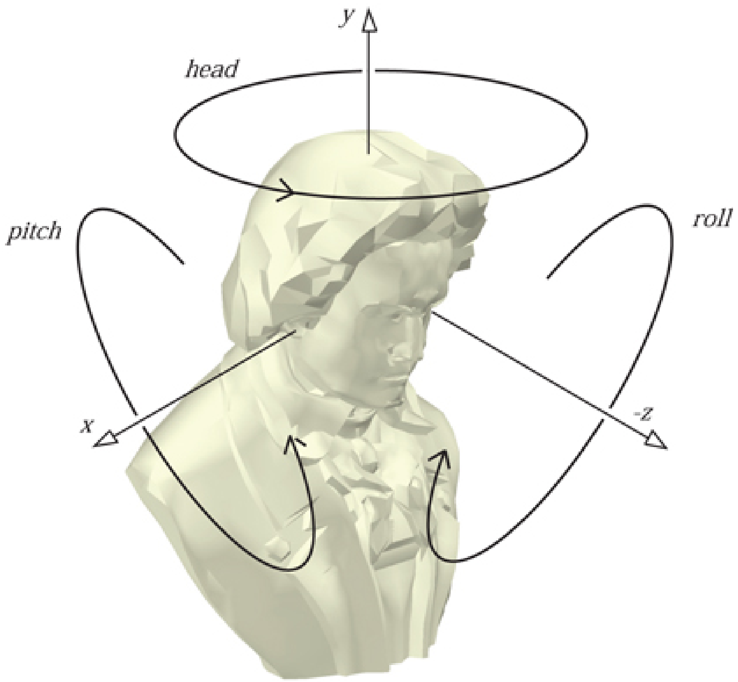
followed by a y-rotation (**yaw** or **head**),

and then a z-rotation (**roll**).

The overall rotation is given by:

$M = Rz(\beta 3)Ry(\beta 2)Rx(\beta 1)$

In this context the angles β1, β2, and β3 are often called **Euler angles.**



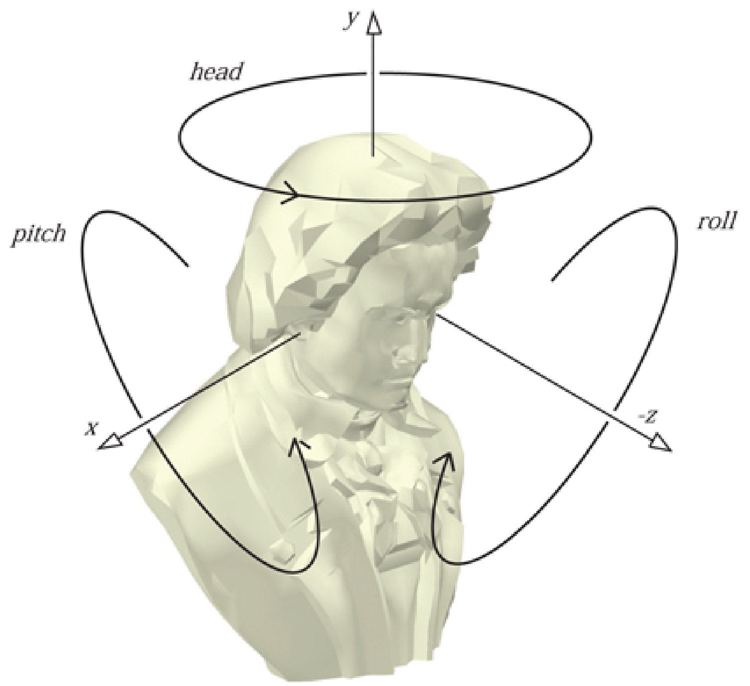
Review: Combining 3D Rotations

3D rotation matrices are not commutative!

M = Rz(β3)Ry(β2)Rx(β1)

If you want to describe this rotation to me...

- I need to know B1, B2, B3
- AND, I need to know the order of rotation



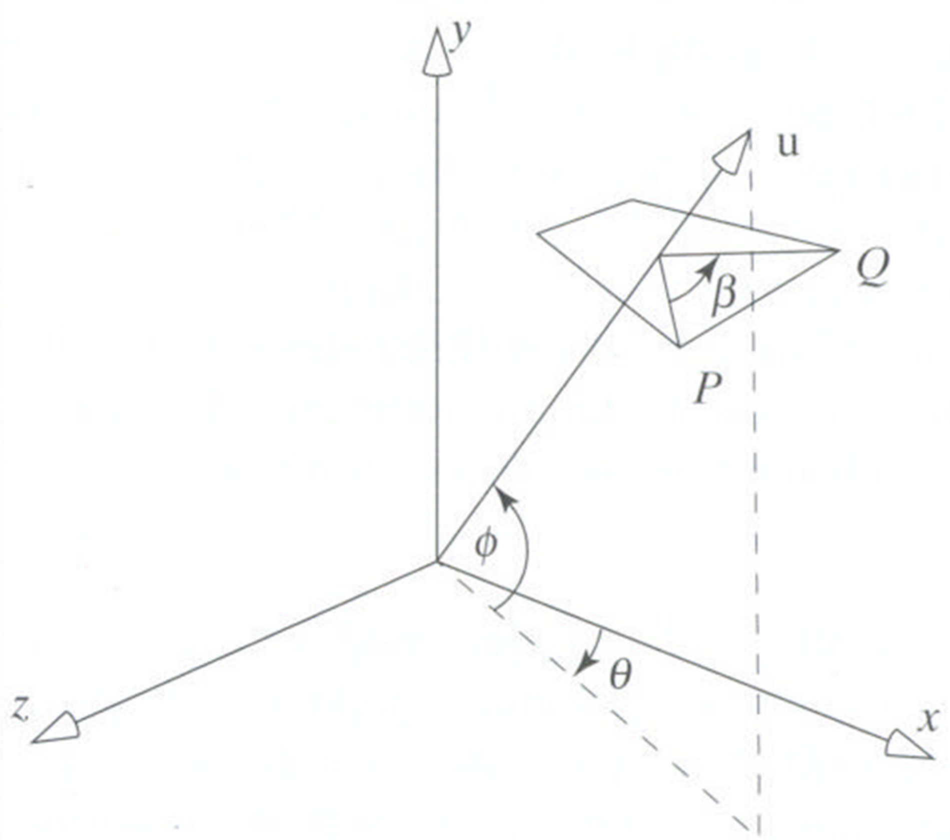
Rotation about an Arbitrary Axis

Euler's theorem states that any sequence of rotations can be represented as one rotation about some axis.

To rotate around arbitrary axis *u* by angle B:

- Use 2 rotations to align u with the X-axis.
- Rotate around the X-axis (an X roll) by angle B.
- Undo the original 2 rotations.

$$R_u(\beta) = R_y(-\theta)R_z(\phi)R_x(\beta)R_z(-\phi)R_y(\theta)$$



Rotation about an Arbitrary Axis in GopherGfx

```
// Construct rotation matrix
const axis = gfx.Vector3.normalize(new gfx.Vector3(1, 1, 1));
const angle = 45 * Math.PI / 180;
const matrix = gfx.Matrix4.makeAxisAngle(axis, angle);

// You can transform a Vector3 using a Matrix4 (this uses homogeneous coordinates under the hood)
const point = new gfx.Vector3(100, 200, 300);
point.transform(matrix);
```

```
// Construct a rotation quaternion
const axis = new gfx.Vector3(1, 1, 1).normalize();
const angle = 45 * Math.PI / 180;
const quat = gfx.Quaternion.makeAxisAngle(axis, angle);

// You can rotate a Vector3 using a Quaternion
const point = new gfx.Vector3(100, 200, 300);
point.rotate(q);
```

Note: some of the function signatures in these slides may be slightly different in the Assignment 2 version due to a recent code refactor.

Other Useful Functions

The Matrix4.lookAt() and Quaternion.lookAt() functions construct a rotation looking from an eye point towards a target point, given a defined up vector.

```
const eye = new gfx.Vector3(0, 0, 0);
const target = new gfx.Vector3(0, 0, -1);
const up = new gfx.Vector3(0, 1, 0);

const matrix = new gfx.Matrix4();
matrix.lookAt(eye, target, up);

const quat = new gfx.Quaternion();
quat.lookAt(eye, target, up);
```

Composing Transformations

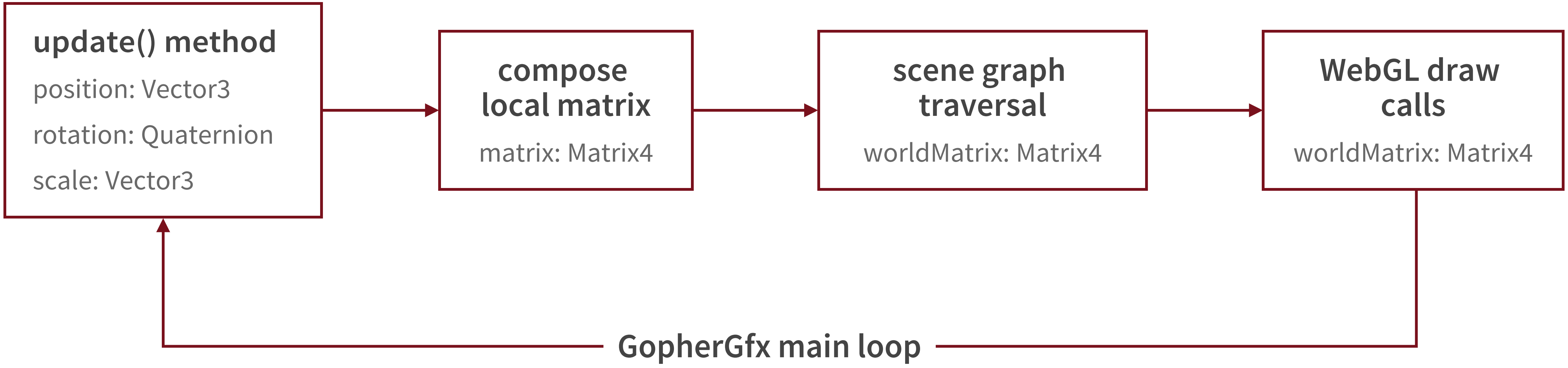
Remember, composing transformations is done mathematically via matrix multiplication.

```
const T = new gfx.Matrix4.makeTranslation(1, 0, 0);
const S = new gfx.Matrix4.makeScale(2, 2, 2);
const Rx = new gfx.Matrix4().makeRotationX(45 * Math.PI / 180);

// There is no * operator for objects
// This will give you a syntax error
const combo = T * S;

// Of course, there is a function to do this
const combo = new gfx.Matrix4();
combo.multiply(T, S);
combo.multiply(combo, Rx);
```

Composing and Decomposing a Matrix



Composing/Decomposing Transform3 Matrices

```
// This disables automatic matrix composition and lets you set it manually.
// Otherwise, the matrix will be overwritten every frame after update() completes.
transform.autoUpdateMatrix = false;
transform.matrix.compose(position, rotation, scale);
```

```
// If you have changed the object's position, rotation, or scale then
// you can force the world matrix to be recomputed and then decompose the matrix.
transform.updateWorldMatrix()
const [worldPosition, worldRotation, worldScale] = transform.worldMatrix.decompose();
```