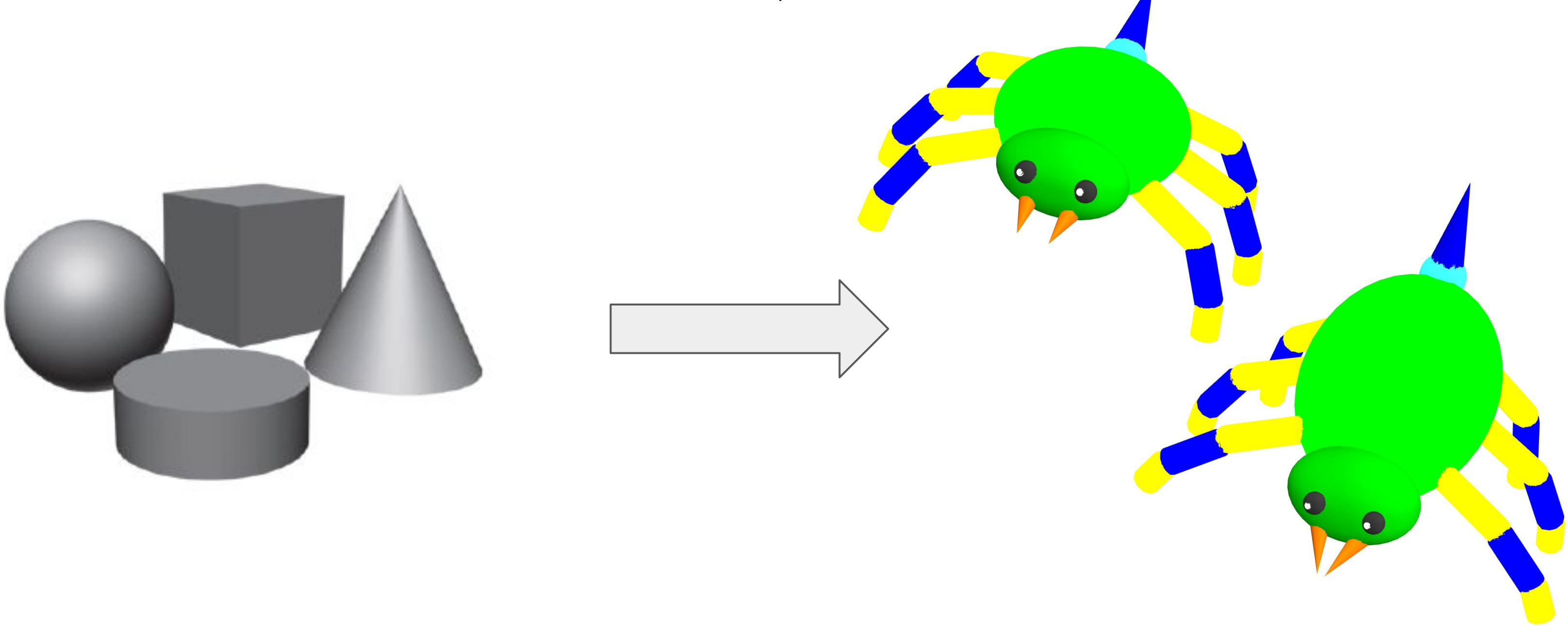
# PA2: 3D Modeling & Transformations

October 2, 2024

Daniel Scrivener, Zezhou Sun



# Assignment 2 is out!

- DUE: October 8 @ 11:59 PM
- Major goal: build a 3D model of a creature!
  - Set hierarchy of components (limbs, body segments)
  - Construct appropriate transformation matrix for each component
  - Set appropriate rotation behavior for components, create 5 poses
  - Finish UI so that individual components can be selected & rotated

# 3D Transformations

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

$$\text{translation:} \begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(from origin to *p*)

Euler rotations (roll, yaw, pitch)

$$R_x : \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta_x) & -sin(\theta_x) & 0 \\ 0 & sin(\theta_x) & cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y : \begin{bmatrix} cos(\theta_y) & 0 & sin(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta_y) & 0 & cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z : \begin{bmatrix} cos(\theta_z) & -sin(\theta_z) & 0 & 0 \\ sin(\theta_z) & cos(\theta_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Euler Rotations

- Must choose an ordering: ordering *does* matter!
  - (x, y, z), (z, y, x), (x, z, y)... etc.
  - Only commutative if you're performing rotations about the *same* axis

Example: what are these rotations? (angle? axis?)

What will the result look like if the order is…

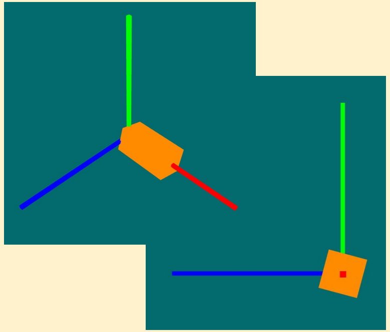
1, 2, 3 vs. 3, 2, 1?

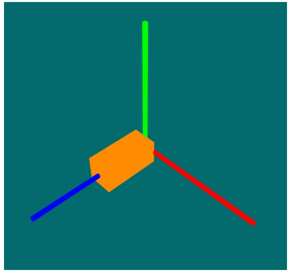
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{3}}{2} & -0.5 & 0 \\ 0 & 0.5 & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_x : \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{\sqrt{3}}{2} & -0.5 & 0 \\ 0 & 0.5 & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_y : \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad R_z : \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

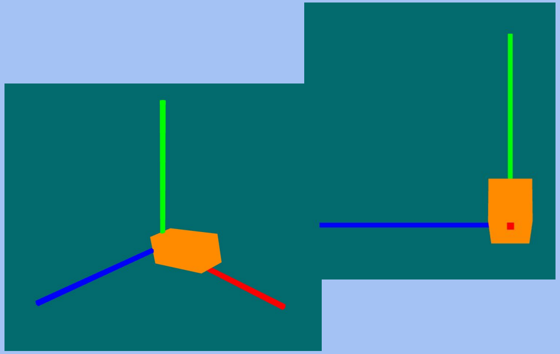
$$\theta = \frac{\pi}{6} \qquad \theta = \frac{\pi}{2} \qquad \theta = \frac{\pi}{4}$$

$$R_x R_y R_z : \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0.97 & 0.26 & 0 & 0 \\ -0.26 & 0.97 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

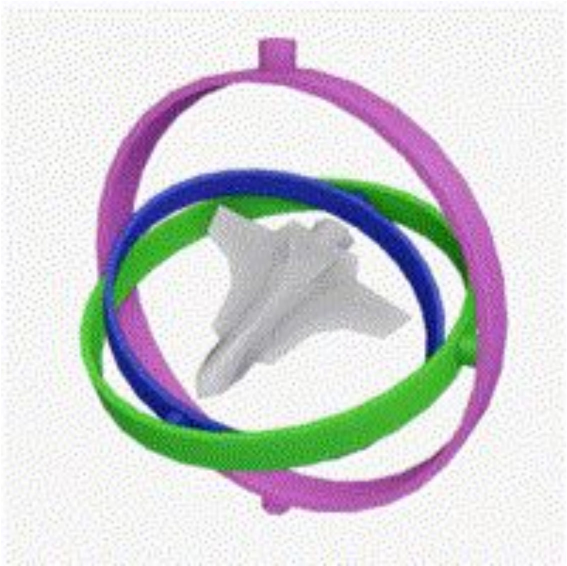




$$R_z R_y R_x : \begin{bmatrix} 0 & -0.26 & 0.97 & 0 \\ 0 & 0.97 & 0.26 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



# Gimbal Lock



- Occurs when two rotational axes “collapse” onto one another, effectively removing a rotational degree of freedom
    - Now, there are only two ways to rotate the airplane as opposed to one
- General case to avoid: rotating the “middle” axis by exactly 90 degrees
    - E.g. in $R_z R_x R_y$ order -> $\theta_x$ is the parameter to watch

# Quaternions

Allow us to rotate around any arbitrary axis rather than just the standard basis vectors in $\mathbb{R}^3$

$$q = (\cos\frac{\theta}{2}, u\sin\frac{\theta}{2})$$

$\theta$ is the angle of rotation

$u$ is the axis around which we'd like to rotate: must be a **unit** vector!

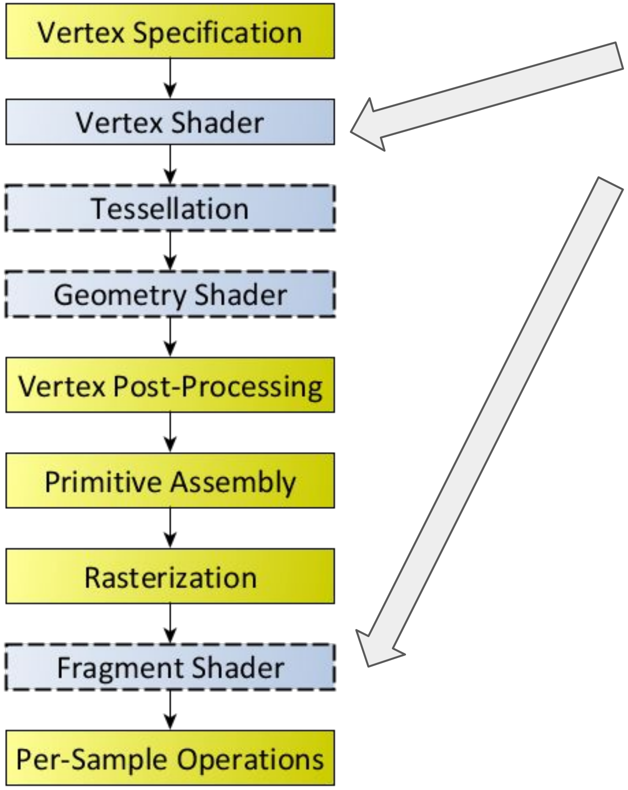
Quaternions encode linear transformations and can be represented in matrix form

This article provides a nice explanation of how imaginary numbers encode 2D rotation: now extend the case to 3D rotation with quaternions

# A Look at PA2: Code Infrastructure

- Primitives stored as meshes (.dae) and used to construct buffers that OpenGL can process
- VBO: vertex buffer object. Contains a list of vertices as well as their attributes (color, normal, uv-mapping)
- EBO: element buffer object. Contains a list of references to vertices (which are stored in a VBO)
- For each geometric primitive:
    - Send buffer data
    - Call glDrawElements(GL_TRIANGLES)
    - Post-process vertices and fragments (screen pixels) using a *shader*, defined in GLProgram.py

# Shader




A (generally short program) that processes primitive elements of a render: either *vertices* or *fragments*

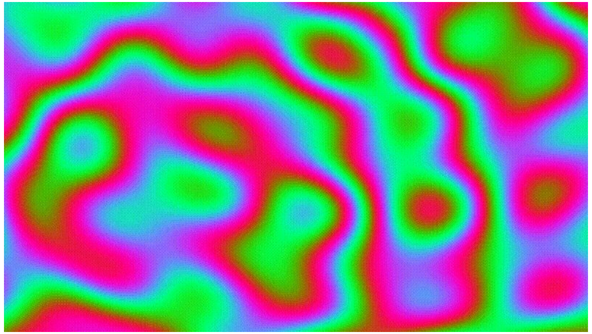
- Fragment: the data associated with a single screen pixel (position, color, z-depth, etc.)

OpenGL shaders are written in GLSL, a C-like language

The shader in PA2 does the following:

- (Vertex shader) Applies the complete set of transformations to each vertex
- (Fragment shader) Applies the appropriate (flat) color to each pixel

See [shadertoy.com](shadertoy.com) for cool examples!



*Converted old plasma* by jolle

# A Look at PA2: Code Infrastructure

- Shape classes defined in Shape.py: Sphere, Cone, Cube, Cylinder
  - Initialize with arguments: position, shaderProg, scale, color
  - position: Point
  - shaderProg: reference to a compiled shader program
  - size: tuple or list of 3 elements
  - color: ColorType
  - These functions return a Shape, which inherits from Component
- Can also initialize Components independently using the Component() constructor — useful for joints with no visible geometry
- See ModelLinkage.py for an example of how to combine shapes together
  - self.componentList: Python list of components
  - self.componentDict: Python dictionary of components (makes accessing individual components from Sketch.py easier)

# Component.py Hierarchical Transformations

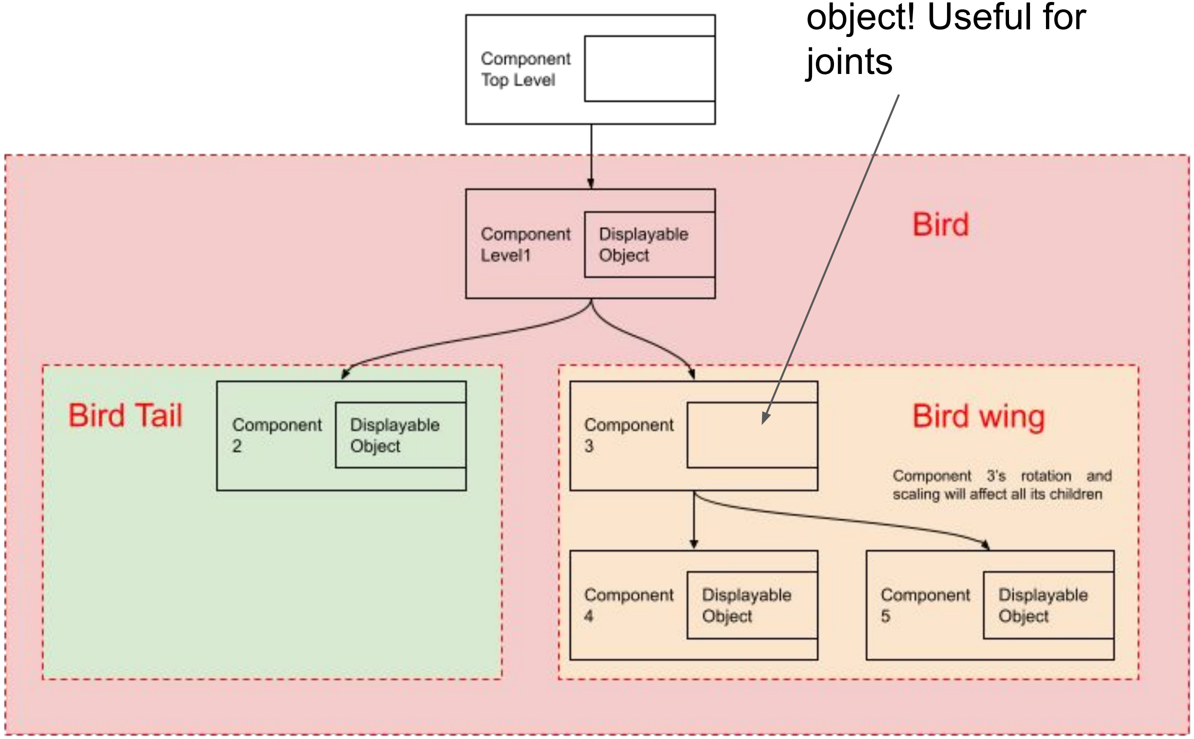
- Objects of the Component class have built-in methods for setting transformation parameters
  - setCurrentColor, setCurrentPosition, setCurrentAngle
  - Can also set defaults: setDefaultColor, setDefaultAngle, etc.
  - Access the object’s local coordinate system with object.uAxis, object.vAxis, object.wAxis
  - setQuaternion can be used to set a Quaternion for rotation (see Quaternion class). overrides any Euler angles that are set

```python
class ModelAxes(Component):
    """
    Define our linkage model
    """

    components = None
    contextParent = None

    def __init__(self, parent, position, shaderProg, display_obj=None):
        super().__init__(position, display_obj)
        self.components = []
        self.contextParent = parent

        xAxis = Cube(Point((0,0,0)), shaderProg, [0.05, 0.05, 2], Ct.RED)
        xAxis.setDefaultAngle(90, xAxis.vAxis)
        yAxis = Cube(Point((0,0,0)), shaderProg, [0.05, 0.05, 2], Ct.GREEN)
        yAxis.setDefaultAngle(-90, yAxis.uAxis)
        zAxis = Cube(Point((0,0,0)), shaderProg, [0.05, 0.05, 2], Ct.BLUE)
        self.addChild(xAxis)
        self.addChild(yAxis)
        self.addChild(zAxis)

        self.components = [xAxis, yAxis, zAxis]
```

# Component.py Hierarchical Transformations

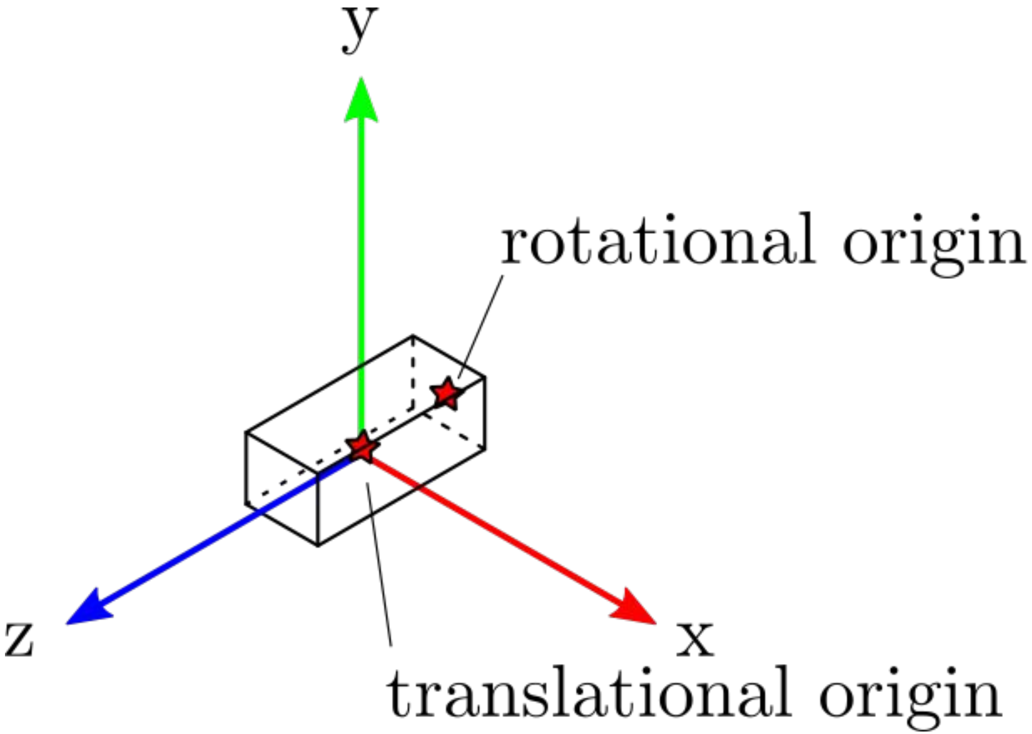
- Components inherit transformations from their parents: see Component.update()

Not every component needs a Displayable object! Useful for joints



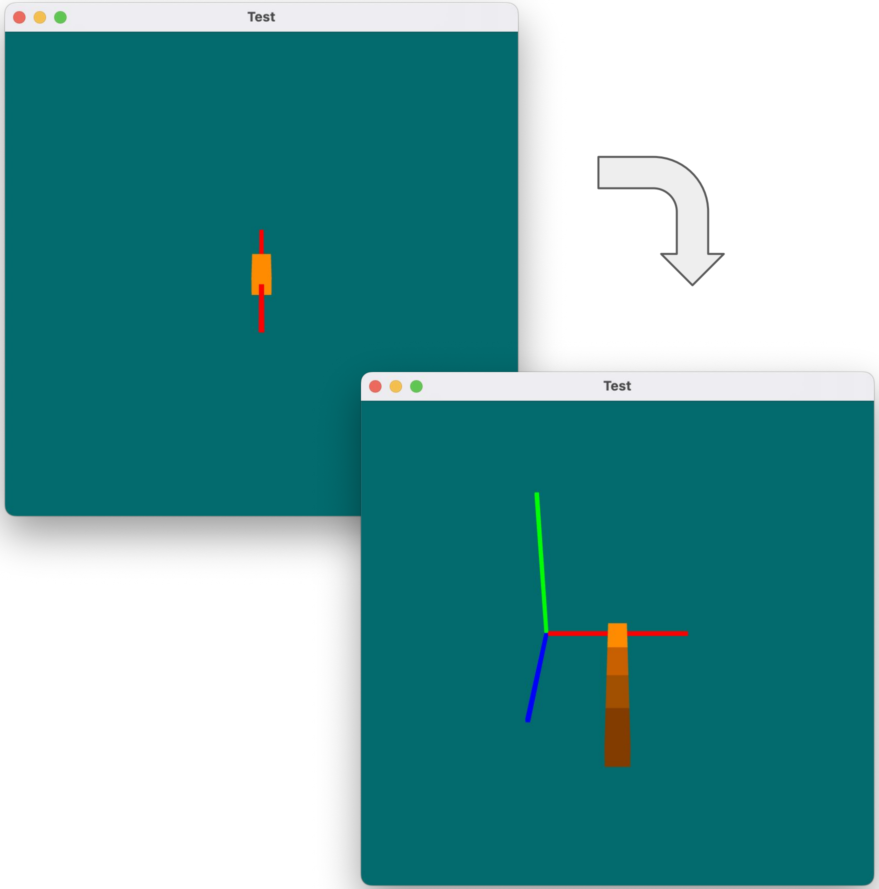
# Default Rotational Origin

- By default: all shapes rotate about a specific endpoint (when drawn from origin along local **z-axis**)
    - Pass limb=False to constructor to have shapes rotate about their centers (useful for ball joints, eyes)



# Component.py Hierarchical Transformations

- TODO 1: Finish the update() function in Component.py!
    - This function is responsible for generating the transformation matrix that will be applied to each component
    - You must choose the order in which transformations are applied!
    - Make sure your composition includes:
        - Scaling (scalingMat)
        - Rotations (rotationMatU, rotationMatV, rotationMatW)
        - Translation (translationMat)



# Code Demo