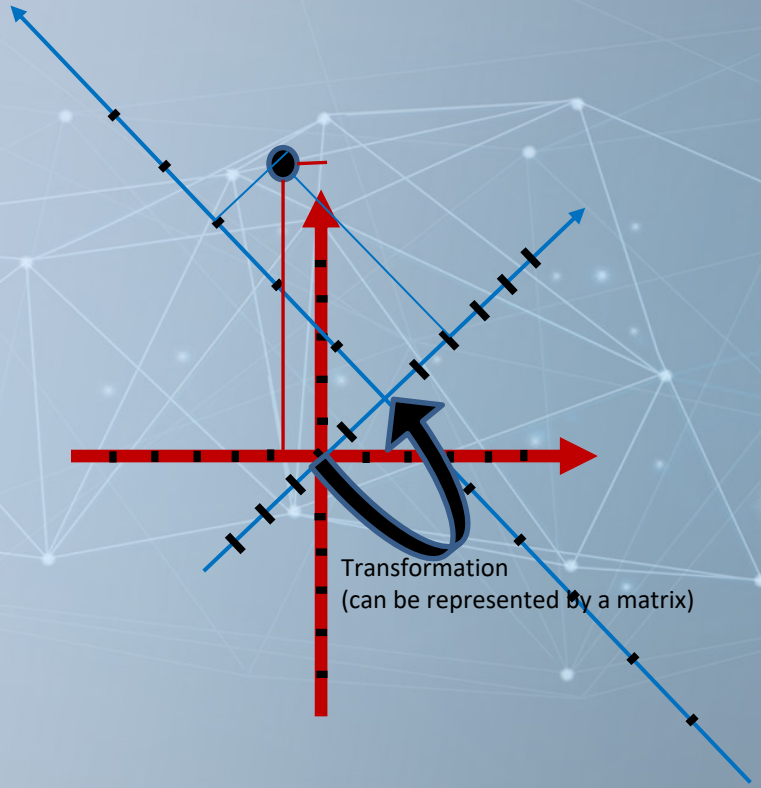
Transformation

CSU0021: Computer Graphics

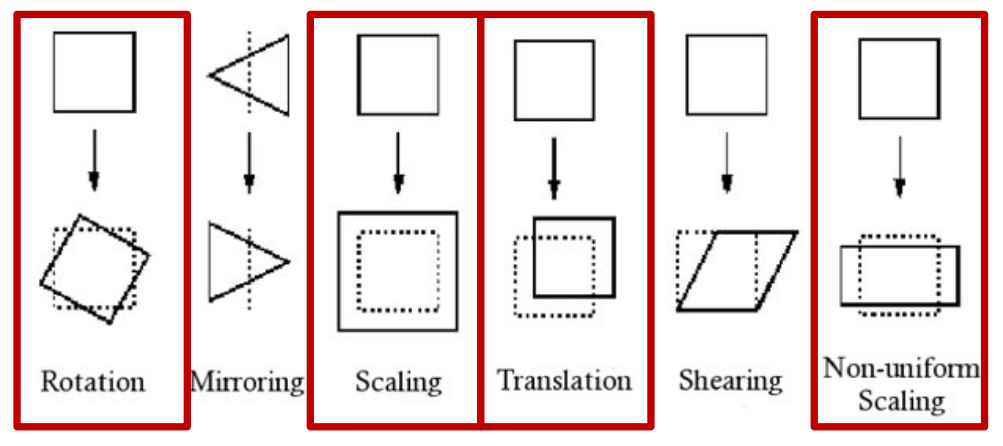
Transformation

- If
  - I know a point (x,y) according to the blue coordinate system
  - and I also know the transformation (relation) between red and blue coordinate systems
- what is the point coordinate according to the red coordinate system?



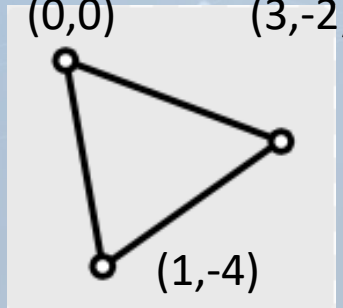
Transformation

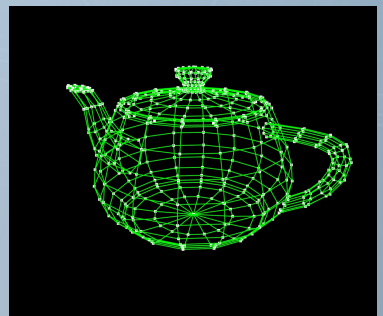
- What we talk here is “Affine” Transformation
  - A linear mapping to preserve points, straight lines, and planes
  - E.g. if lines are parallel, they are still parallel after affine transformation
- Types of affine transformation
  - Translation, rotation, scaling, mirroring, shearing, ……..
  - We only focus on translation, rotation and scale in this course



Why to Learn Transformation

- We usually predefine coordinates of vertices of an object in a file or code
  - Usually all vertices is defined according to a point within the object. E.g. (0,0) is defined at center of the teapot
- You may want to draw an object at different location, rotation angle and sizes
- If you are good at linear algebra, you know how to do it, but painful
- We need a more systematic way to do and understand it
  - Apply transformation to the original coordinates of all points the object
  - $v' = M(v)$
    - M: transformation operator
    - v: coordinate of a vertex of the object **before** transformation
    - v': coordinate of a vertex **after** transformation (where to draw it in global space)

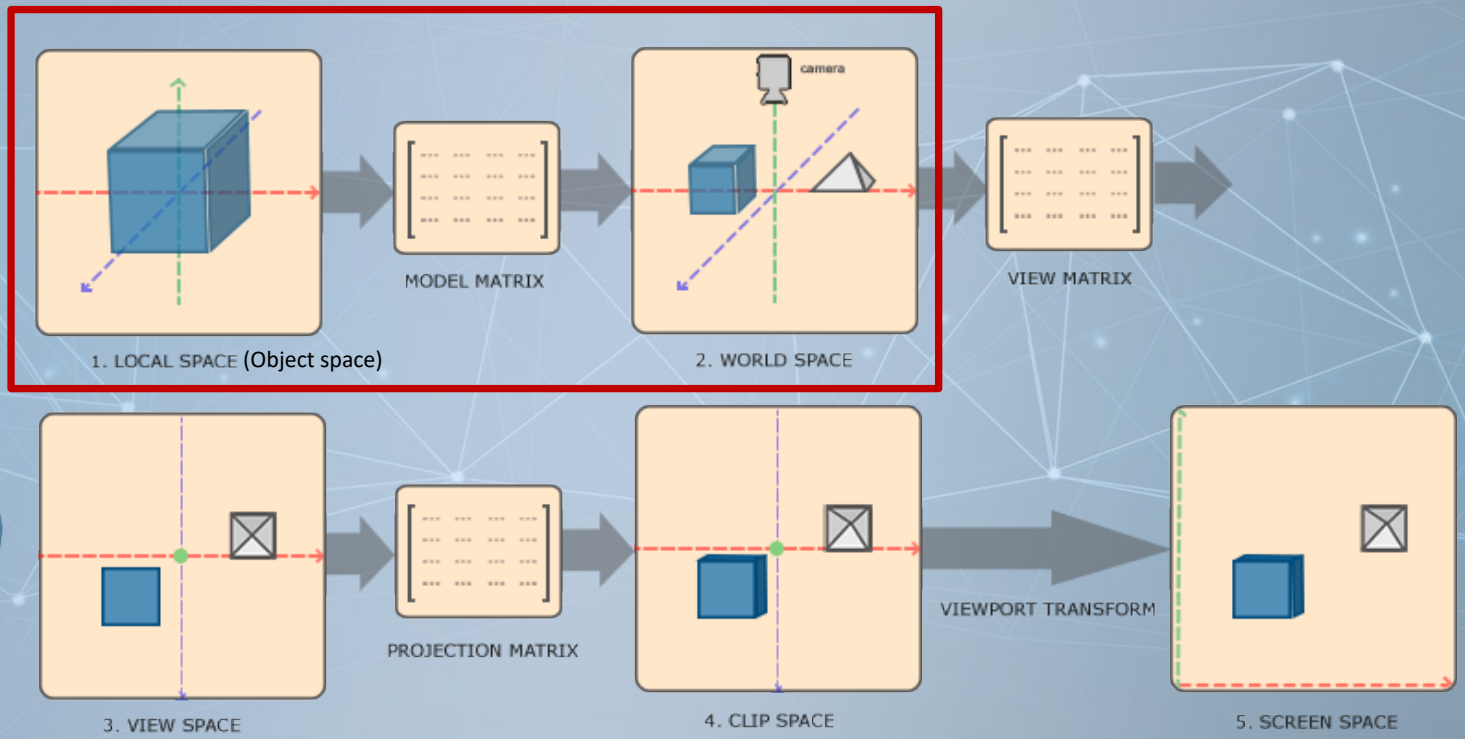




Coordinate System (Real 3D Pipeline Case)

Coordinates of Vertices of an object

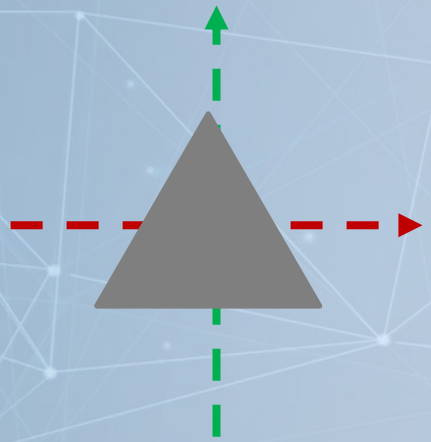
Coordinates of vertices in the world space after transformation



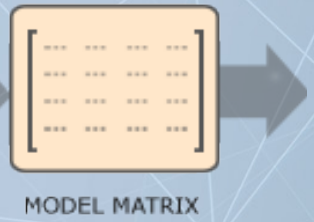
https://learnopengl.com/Getting-started/Coordinate-Systems

2D Example Today

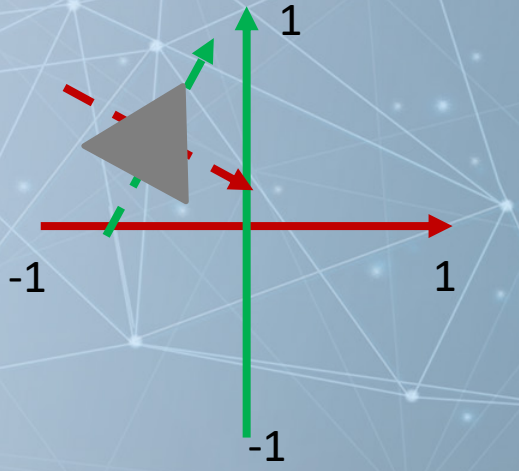
Coordinates of Vertices of an object



Object Space



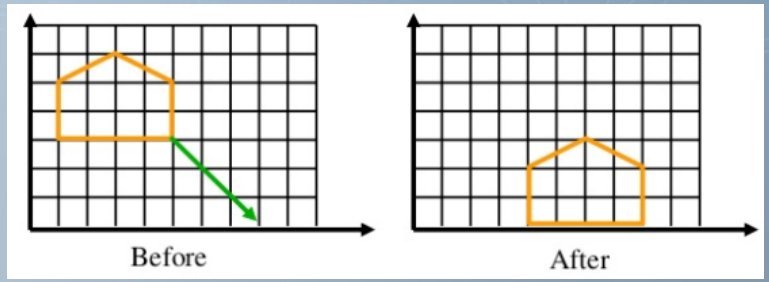
We want to know…
Coordinates in world space
to draw this object after transformation



World space

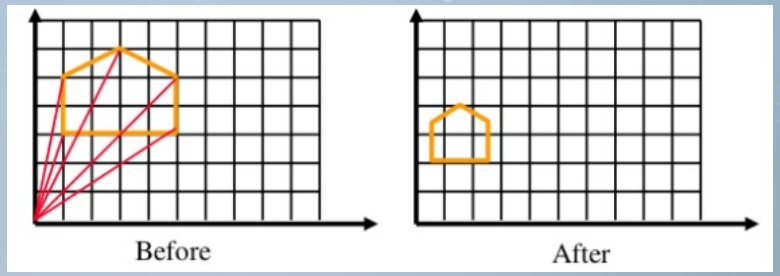
Translation

- Translate an object which consists of multiple vertices
  - Use 2D as an example
- Apply the following equations to all vertices of the object
  - $x' = x + t_x$
  - $y' = y + t_y$
- Or, let $P' = \begin{bmatrix} x' \\ y' \end{bmatrix}, P = \begin{bmatrix} x \\ y \end{bmatrix}, \mathrm{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$
  - $P' = P + T$



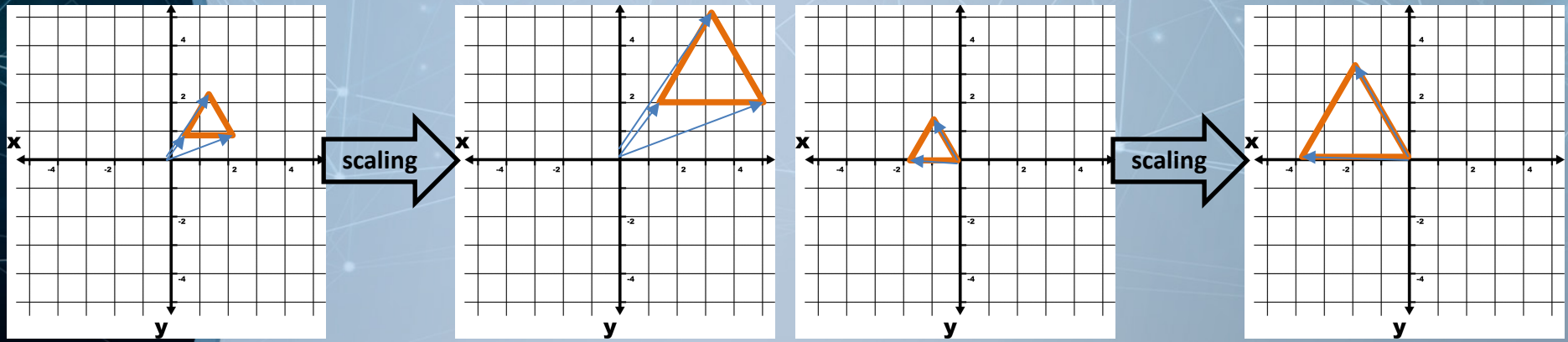
Scaling

- Apply the following equations to all vertices of the object
  - $x' = s_x x$
  - $y' = s_y y$
- Or, let $P' = \begin{bmatrix} x' \\ y' \end{bmatrix}, P = \begin{bmatrix} x \\ y \end{bmatrix}, S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$
  - $P' = S * P$
- Note: scaling operation scales up/down vertices according to the "origin"



Center for Scaling

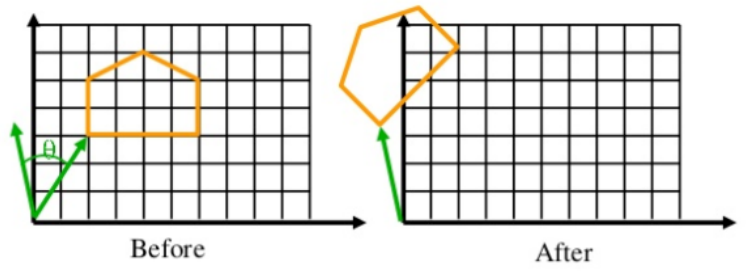
- Origin (0,0) in the local space is the center for scaling
  - The point does not move after scaling
    - Why? Check the equation in last slides again



The object does not cover (0,0), all points of the object moves

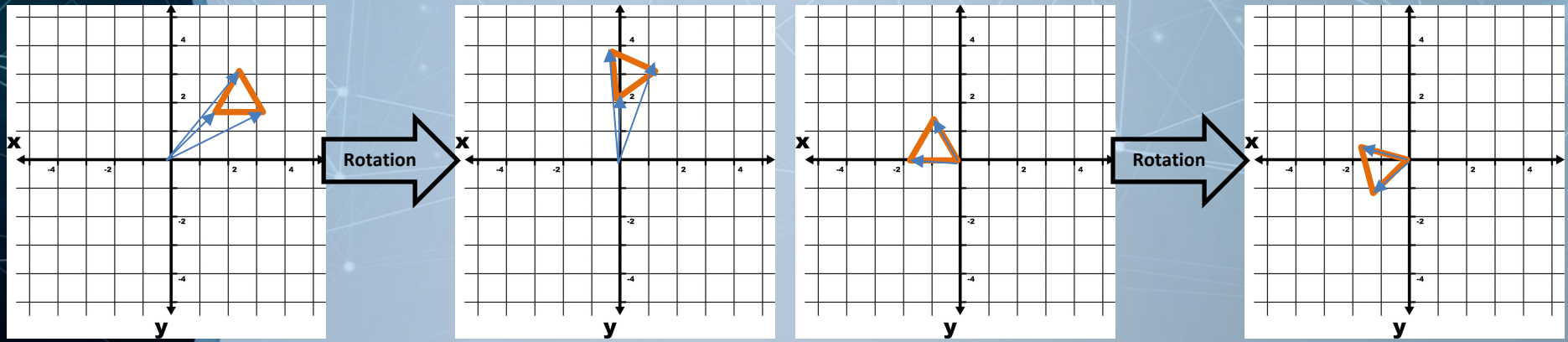
Rotation

- Apply the following equations to all vertices of the object
  - $x' = cos\theta x - sin\theta y$
  - $y' = sin\theta x + cos\theta y$
- Or, let $P' = \begin{bmatrix} x' \\ y' \end{bmatrix}, P = \begin{bmatrix} x \\ y \end{bmatrix}$, R $= \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$
  - $P' = R * P$
- Note: when rotating an object, there is a rotation axis which in the "origin"



Center for Rotation

- Origin (0,0) in the local space is the center (rotation axis) for rotation
  - The point does not move after rotation
    - Why? Check the equation in last slides again



The object does not cover (0,0), all points of the object moves

Summary of Translation, Rotation and Scaling (2D)

- $P' = \begin{bmatrix} x' \\ y' \end{bmatrix}$
- $P = \begin{bmatrix} x \\ y \end{bmatrix}$
- $\mathrm{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$
- $\mathrm{S} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$
- $\mathrm{R} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$

- Translation

  $P' = P + T$

- Scaling

  $P' = S * P$

- Rotation

  $P' = R * P$

Summary of Translation, Rotation and Scaling (3D)

- $P' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$
- $P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$
- $\mathrm{T} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$
- $\mathrm{S} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$
- $\mathrm{R} = ?$

- Translation

  $P' = P + T$

- Scaling

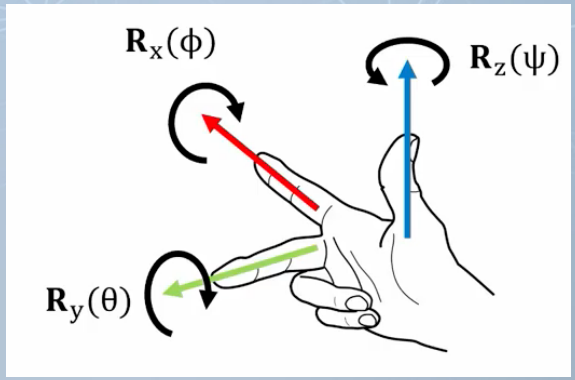
  $P' = S * P$

- Rotation

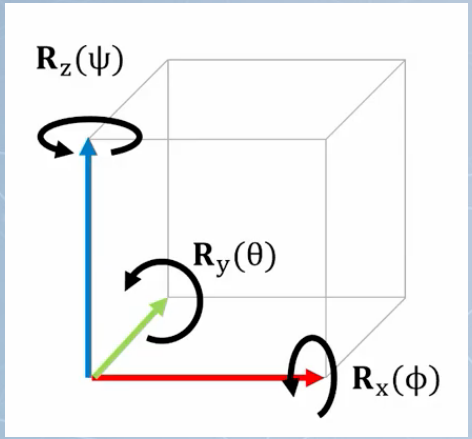
  ?

3D Rotation

- In WebGL, we use “Right hand rule”
  - If 2 axes are fixed, where is the positive direction of the last axis
  - For example, if x-axis and y-axis are fixed
    - You can use right hand index finger pointing to x-axis and middle finger to pointing y-axis. Z-axis positive direction should be the thumb finger direction.



3D Rotation



Rotation direction?
Same, use your right hand

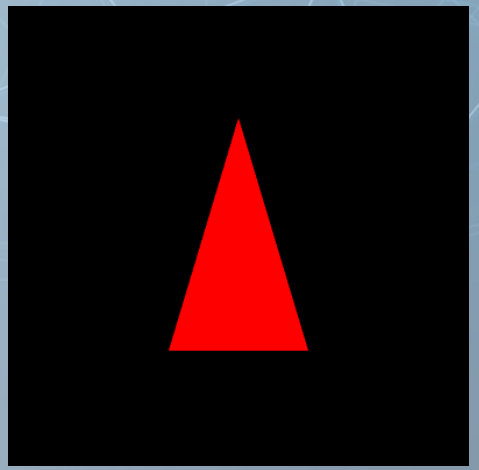
- Rotation matrix R by angle θ and rotation axis is $u = (u_x, u_y, u_z)$ and $u_x^2 + u_y^2 + u_z^2 = 1$ (u is an unit vector)

$$R = \begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}$$

For more details, check https://en.wikipedia.org/wiki/Rotation_matrix

Example (Ex03-1)

- Download Ex03-1 from Moodle
- This example can draw a triangle
  - Let try to apply translate, rotation, scale on it, and see what happen
  - We will ignore some detail in the code here



- Files
  - index.html
  - WebGL.js
  - cuon-matrix.js

Example (Ex03-1)

- index.html

For matrix calculation, but we do not introduce it here.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>WebPage Title </title>
    </head>

    <body onload="main()">
        <canvas id="webgl" width = "400" height = "400">
            Please use a browser that support "canvas"
        </canvas>
        <script src="cuon-matrix.js"></script>
        <script src="WebGL.js"></script>
    </body>
</html>
```

Example (Ex03-1)

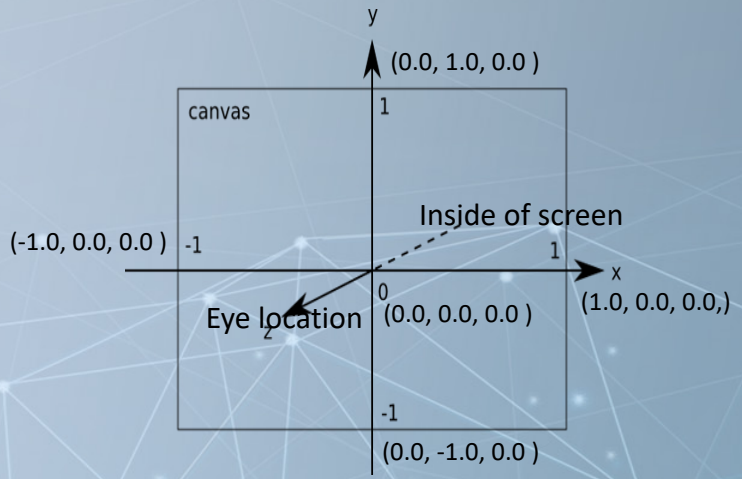
- WebGL.js

Where to setup the translation, scaling or rotation

Pass transformation information to shader and draw the triangle

```
function main(){
    //////Get the canvas context
    var canvas = document.getElementById('webgl');
    var gl = canvas.getContext('webgl2');
    if(!gl){
        console.log('Failed to get the rendering context for WebGL');
        return ;
    }

    program = compileShader(gl, VSHADER_SOURCE, FSHADER_SOURCE);

    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);

    gl.useProgram(program);
    u_modelMatrix = gl.getUniformLocation(gl.getParameter(gl.CURRENT_PROGRAM), 'u_modelMatrix');

    transformMat.setIdentity();

    //transformMat.translate(0.0, 0.5, 0.0); //translate(tx, ty, tz) ..in 2D tz is useless, do not change it
    //transformMat.scale(0.75, 0.25, 1.0); //scale(sx, sy, sz) ...same, do not change sz
    //transformMat.rotate(45, 0, 0, 1); //rotate(angle in degree, rx, ry, rz)

    gl.uniformMatrix4fv(u_modelMatrix, false, transformMat.elements); //pass the transformation matrix to shader
    /////Draw a triangle/////
    triangleVertices = [0.0, 0.5, -0.3, -0.5, 0.3, -0.5]; //define the triangle in object space
    var triangleColor = [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0 ]; //red triangle
    buffer0 = initArrayBuffer(gl, new Float32Array(triangleVertices), 2, gl.FLOAT, 'a_Position');
    buffer1 = initArrayBuffer(gl, new Float32Array(triangleColor), 3, gl.FLOAT, 'a_Color');
    gl.drawArrays(gl.TRIANGLES, 0, triangleVertices.length/2);
}
```

Example (Ex03-1)



- Let's try
  - Only uncomment 'translate(…)'
    - Try to modify tx and tx
  - Only uncomment 'scale(…)'
    - Try to modify sx and sy
  - Only uncomment 'rotate(…)'
    - Try to modify the 'angle in degree'
  - Uncomment translate() and rotate() at the same time (still comment scale())

```
transformMat.translate(0.0, 0.5, 0.0);
transformMat.rotate(45, 0, 0, 1);
```

    - swap the order of translate and rotate, what is the difference?

```
transformMat.rotate(45, 0, 0, 1);
transformMat.translate(0.0, 0.5, 0.0);
```

Homogeneous Coordinate System

- What is this?
  - Add one more component (w) to describe a point
- 2D
  - (x, y, w) is a point defined in homogeneous system.
  - $(\frac{x}{w}, \frac{y}{w})$ is the same point defined in 2D cartesian coordinate system
  - Ex: $\begin{bmatrix}3\\5\end{bmatrix}_{car} = \begin{bmatrix}3\\5\\1\end{bmatrix}_{hom} = \begin{bmatrix}1.5\\2.5\\0.5\end{bmatrix}_{hom} = \begin{bmatrix}9\\15\\3\end{bmatrix}_{hom}$
- 3D
  - (x, y, z, w) is a point defined in homogeneous system.
  - $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})$ is the same point defined in 3D cartesian coordinate system

Why

- Why to use this?
  - Easy for affine transformation calculation
- Affine transformation (translation, rotation, scaling) can be represented as a matrix if we use homogeneous coordinate system
  - Easier to apply multiple transformations on an object (you may wan to translate a triangle first, them rotate it)
  - $P' = R*T*P$

Translation in Homogeneous Coordinate System (2D)

$$\begin{bmatrix} \frac{x'}{w'} \\ \frac{y'}{w'} \end{bmatrix}_{car} = \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}_{hom} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}_{hom}$$

Example: $\boldsymbol{x = 3, y = 6, w = 1, t_x = 4, t_y = 2}$

$$\begin{bmatrix} \frac{7}{1} \\ \frac{8}{1} \end{bmatrix}_{car} = \begin{bmatrix} 7 \\ 8 \\ 1 \end{bmatrix}_{hom} = \begin{bmatrix} 1 & 0 & 4 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \\ 1 \end{bmatrix}_{hom}$$

Translation in Homogeneous Coordinate System (2D)

$$\begin{bmatrix} \frac{x'}{w'} \\ \frac{y'}{w'} \end{bmatrix}_{car} = \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}_{hom} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}_{hom}$$

Example: $\boldsymbol{x = 3, y = 6, w = 1, t_x = 4, t_y = 2}$

$$\begin{bmatrix} 7 \\ 8 \end{bmatrix}_{car} = \begin{bmatrix} \frac{7}{1} \\ \frac{8}{1} \end{bmatrix}_{car} = \begin{bmatrix} 7 \\ 8 \\ 1 \end{bmatrix}_{hom} = \begin{bmatrix} 1 & 0 & 4 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \\ 1 \end{bmatrix}_{hom}$$

Translation in Homogeneous Coordinate System (2D)

$$\begin{bmatrix} \frac{x'}{w'} \\ \frac{y'}{w'} \end{bmatrix}_{car} = \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}_{hom} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}_{hom}$$

Example: $\boldsymbol{x = 3, y = 6, w = 1}$, $\boldsymbol{t_x = 4, t_y = 2}$

$$\begin{bmatrix} 7 \\ 8 \end{bmatrix}_{car} = \begin{bmatrix} \frac{7}{1} \\ \frac{8}{1} \end{bmatrix}_{car} = \begin{bmatrix} 7 \\ 8 \\ 1 \end{bmatrix}_{hom} = \begin{bmatrix} 1 & 0 & 4 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \\ 1 \end{bmatrix}_{hom}$$

Example: $\boldsymbol{x = 1.5, y = 3, w = 0.5}$, $\boldsymbol{t_x = 4, t_y = 2}$

$$\begin{bmatrix} 7 \\ 8 \end{bmatrix}_{car} = \begin{bmatrix} \frac{3.5}{0.5} \\ \frac{4}{0.5} \end{bmatrix}_{car} = \begin{bmatrix} 3.5 \\ 4 \\ 0.5 \end{bmatrix}_{hom} = \begin{bmatrix} 1 & 0 & 4 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1.5 \\ 3 \\ 0.5 \end{bmatrix}_{hom}$$

Transformation in Homogeneous Coordinate System (2D)

Translation:

$$\begin{bmatrix} \frac{x'}{w'} \\ \frac{y'}{w'} \end{bmatrix}_{car} = \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}_{hom} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}_{hom}$$

Scaling:

$$\begin{bmatrix} \frac{x'}{w'} \\ \frac{y'}{w'} \end{bmatrix}_{car} = \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}_{hom} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}_{hom}$$

Rotation:

$$\begin{bmatrix} \frac{x'}{w'} \\ \frac{y'}{w'} \end{bmatrix}_{car} = \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}_{hom} = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}_{hom}$$

Transformation in Homogeneous Coordinate System (3D)

Translation:

$$\begin{bmatrix} \frac{x'}{w'} \\ \frac{y'}{w'} \\ \frac{z'}{w'} \end{bmatrix}_{car} = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}_{hom} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}_{hom}$$

Scaling:

$$\begin{bmatrix} \frac{x'}{w'} \\ \frac{y'}{w'} \\ \frac{z'}{w'} \end{bmatrix}_{car} = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}_{hom} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}_{hom}$$

Rotation:

$$\begin{bmatrix} \frac{x'}{w'} \\ \frac{y'}{w'} \\ \frac{z'}{w'} \end{bmatrix}_{car} = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}_{hom} = \begin{bmatrix} & R & & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}_{hom}$$

$$R = \begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}$$

If you want to apply “multiple transformation” (translations, rotations and scalings) to an object, what is the order to apply these transformations?

There is no gold rule.

It depends on what you want to do….

(The following slides many help you answering this question)

Multiple Transformations

translate(2,1) -> rotate(45degrees)->scale(1, 2)

$$\begin{bmatrix}1 & 0 & 2\\0 & 1 & 1\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos(\frac{PI}{4}) & -\sin(\frac{PI}{4}) & 0\\\sin(\frac{PI}{4}) & \cos(\frac{PI}{4}) & 0\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\0 & 2 & 0\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}2\\3\\1\end{bmatrix}=\begin{bmatrix}-0.828\\6.656\\1\end{bmatrix}$$

Order here is defined by the order of matrix multiplication

Coordinate in object space

Where to draw in world space

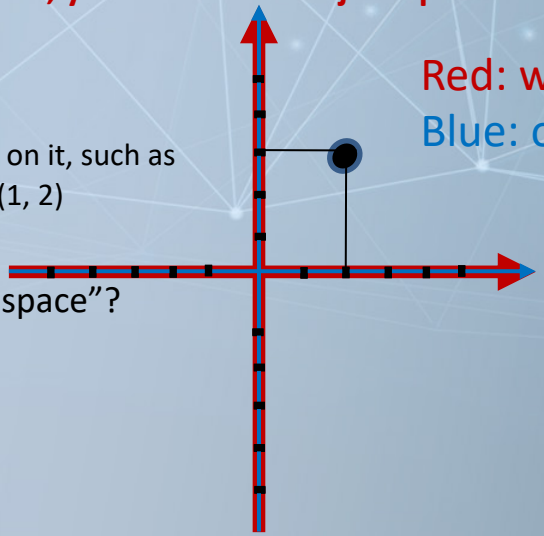
One Way to Interpret Transformation

- Imagine that there are two coordinate system
  - World space (never change)
  - Object space (change by transformation)
    - When you draw any object, you draw the object according to the drawing coordinate system
- These two systems overlap perfectly if you do not apply any transformation
- **When you apply transformation, you move the ”object space”**

(x=2.0, y=3.0) in “object space”

If we want to apply some transformations on it, such as translate(2,1) -> rotate(45degrees)->scale(1, 2)

Where should we draw it in “world space”?



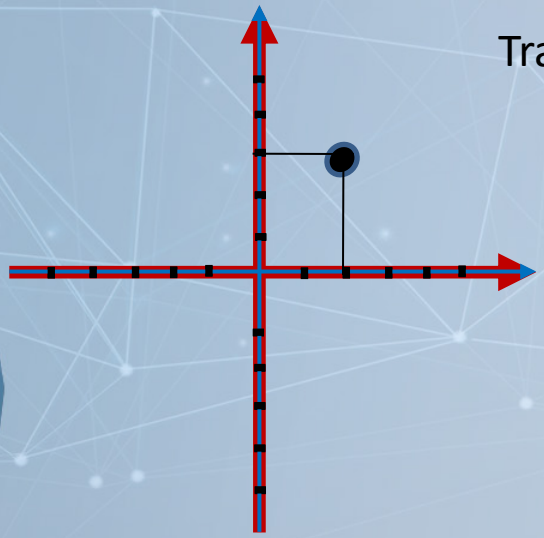
Red: world space

Blue: object space

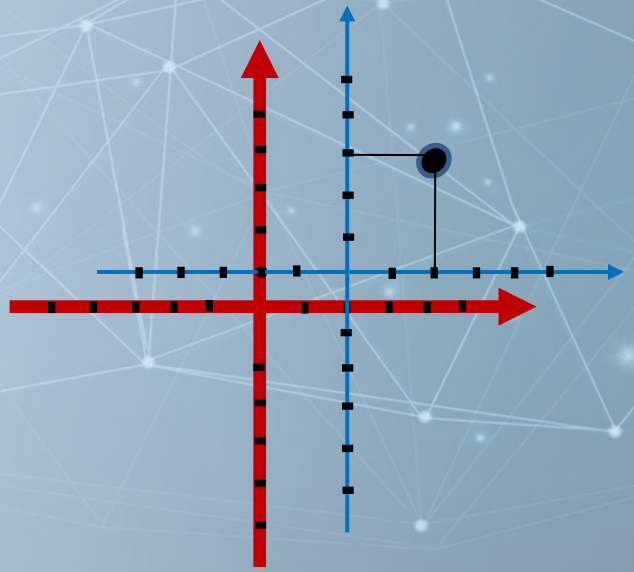
One Way to Interpret Transformation

Red: world space

Blue: object space



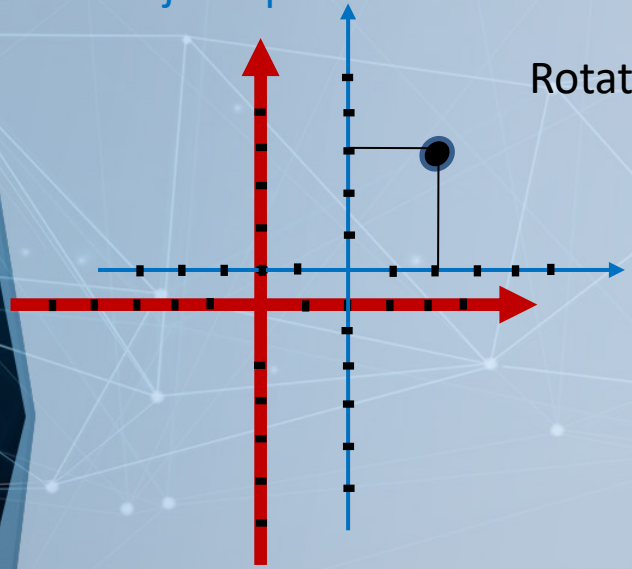
Translate(tx=2, ty=1)



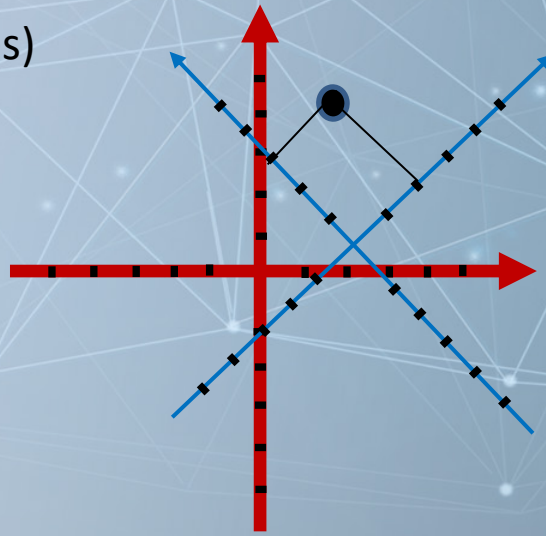
One Way to Interpret Transformation

Red: world space

Blue: object space



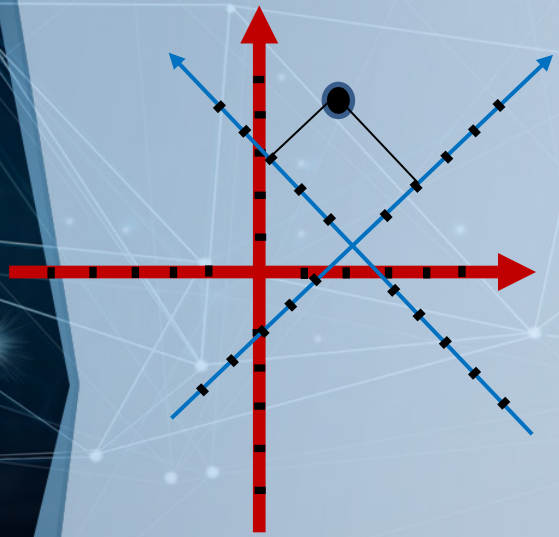
Rotate(45degrees)



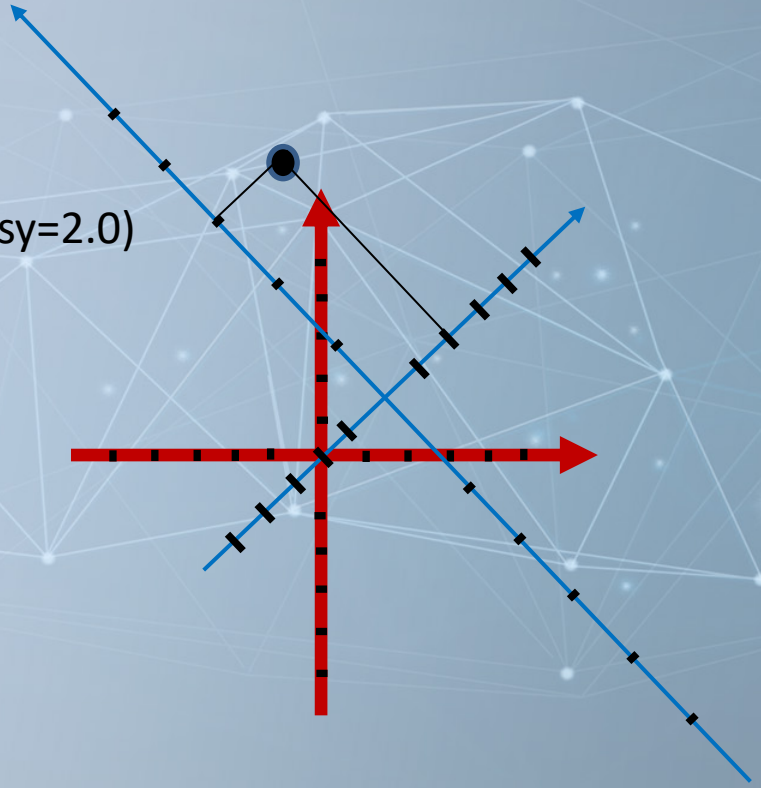
One Way to Interpret Transformation

Red: world space

Blue: object space



Scale(sx=1.0, sy=2.0)



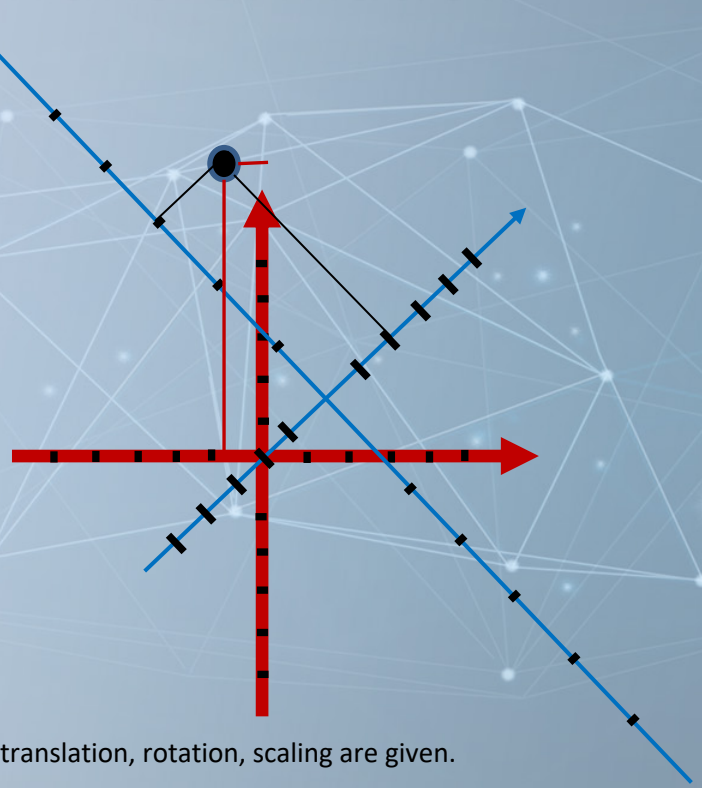
One Way to Interpret Transformation

Red: world space

Blue: object space

Draw a point at
(x=2.0, y=3.0)

$$\begin{bmatrix}1 & 0 & 2\\ 0 & 1 & 1\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos(\frac{PI}{4}) & -\sin(\frac{PI}{4}) & 0\\ \sin(\frac{PI}{4}) & \cos(\frac{PI}{4}) & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}2\\ 3\\ 1\end{bmatrix} = \begin{bmatrix}-0.828\\ 6.656\\ 1\end{bmatrix}$$



Not only for you to understand what happens when a sequence of translation, rotation, scaling are given.

But also help to come up with a sequences of translations, rotations, scalings for a complex transformation

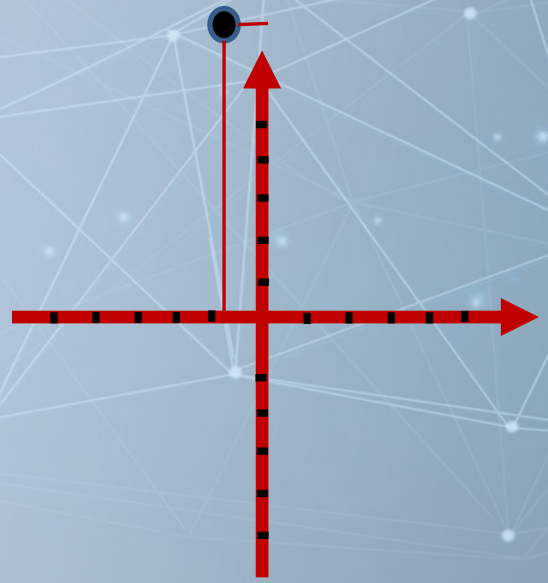
One Way to Interpret Transformation

Red: world space

Blue: object space

Draw a point at
(x=2.0, y=3.0)

$$\begin{bmatrix}1 & 0 & 2\\ 0 & 1 & 1\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos(\frac{PI}{4}) & -\sin(\frac{PI}{4}) & 0\\ \sin(\frac{PI}{4}) & \cos(\frac{PI}{4}) & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}2\\ 3\\ 1\end{bmatrix} = \begin{bmatrix}-0.828\\ 6.656\\ 1\end{bmatrix}$$



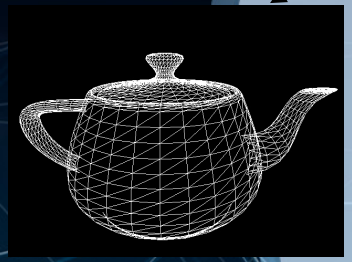
Not only for you to understand what happens when a sequence of translation, rotation, scaling are given.

But also help to come up with a sequences of translations, rotations, scalings for a complex transformation

True Power of Homogenous Coordinate and Matrix Calculation

- Homogenous coordinate system enables the matrix calculation for transformation
- Matrix calculation significantly **reduce the computation** for transformation
- For example, a teapot model consists of a lot of vertices. If you want to move, scale or rotate the teapot, you will apply the same calculation on all vertices.

If this teapot consists of **1000 vertices**



True Power of Homogenous Coordinate and Matrix Calculation

You have to repeat this for 1000 times

$$\begin{bmatrix}1 & 0 & 2\\ 0 & 1 & 1\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos(\frac{PI}{4}) & -\sin(\frac{PI}{4}) & 0\\ \sin(\frac{PI}{4}) & \cos(\frac{PI}{4}) & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & 2 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}v_0\\ v_1\\ v_2\end{bmatrix}$$

Pre-multiply matrices

$$\begin{bmatrix}m_{00} & m_{01} & m_{02}\\ m_{10} & m_{11} & m_{12}\\ m_{20} & m_{21} & m_{22}\end{bmatrix}\begin{bmatrix}v_0\\ v_1\\ v_2\end{bmatrix}$$

Only repeat this for 1000 times if you pre-multiply the transformation matrices

Matrix Calculation in WebGL

- WebGL shader supports matrix type variable (e.g. Mat4)
- However, WebGL API does NOT support transformation matrix operation
  - E.g. create a matrix which represent a transformation, tx=2, ty=1
  - E.g. Multiply two matrices

- We have to implement the transformation matrix operation by ourself or use the library(.js) already implemented by others
  - We already use a matrix operation library (cuon-matrix.js) in Ex3-1 (but we do not introduce it in detail)
    - Remember, we just simply call “transformMat.translate(0.0, 0.5, 0.0);” to have a translation matrix
  - Note: “cuon-matrix.js” just one of the transformation matrix implementation. You can find others or implement yours.

cuon-matrix.js

Category	Method	Description
Creation	Matrix4() ex: var **m**=new Matrix4();	Create a Mat4 object (4x4 matrix)
Set Matrix	m.setIdentity()	Set Matrix4(m) to identity matrix
	m.setTranslate(tx,ty,tz)	set a translation on Matrix4(m)
	m.setRotate(angle, rx, ry, rz)	set a rotation on Matrix4(m). rx, ry, rz define the rotation axis. Angle is in degree.
	m.setScale(sx, sy, sz)	set a scaling on Matrix4(m)
	m.set(k)	Set Matrix4(m) to k, k is also an instance of Matrix4
Apply	m.translate(tx, ty, tz)	apply a translation on Matrix4(m), m = m * (translate matrix)
	m.rotate(angle, rx, ry, rz)	Apply a rotation on Matrix4(m). rx, ry, rz define the rotation axis. Angle is in degree. m = m * (rotation matrix)
	m.scale(sx, sy, sz)	Apply a scaling on Matrix4(m). m = m * (scaling matrix)
Other	m.elements	Return a Float32Array (column-major) which is the 16 elements of Matrix4 (m)

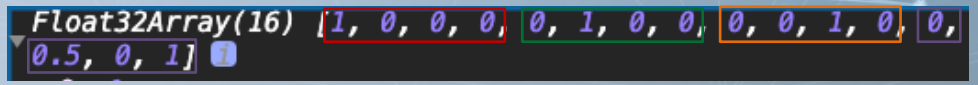
*Check “cuon-matrix.js” to get more information or add more functions to it

cuon-matrix.js

- WebGL Matrix (Mat4 in shader) is ”column major”
- If you use console.log() and Mat4.elements to print the content of Mat4
- Example
  - Matrix4.translate(0.0, 0.5, 0.0) and use console.log() to print the matrix

–

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Use cuon-matrix.js for 2D Transformation

- cuon-matrix.js is designed for 3D transformation
  - The matrix it creates is 4x4 matrix for 3D (homogenous coordinate system)

- If you want to use it for 2D (x-y plane)
  - setTranslate(), translate(): set tz to 0
  - setScale(), scale(): set sz to 1
  - setRotate(), rotate(): always set rotation axis to (rx=0, ry=0, rz=1)

How to Make This Matrix using cuon-matrix.js

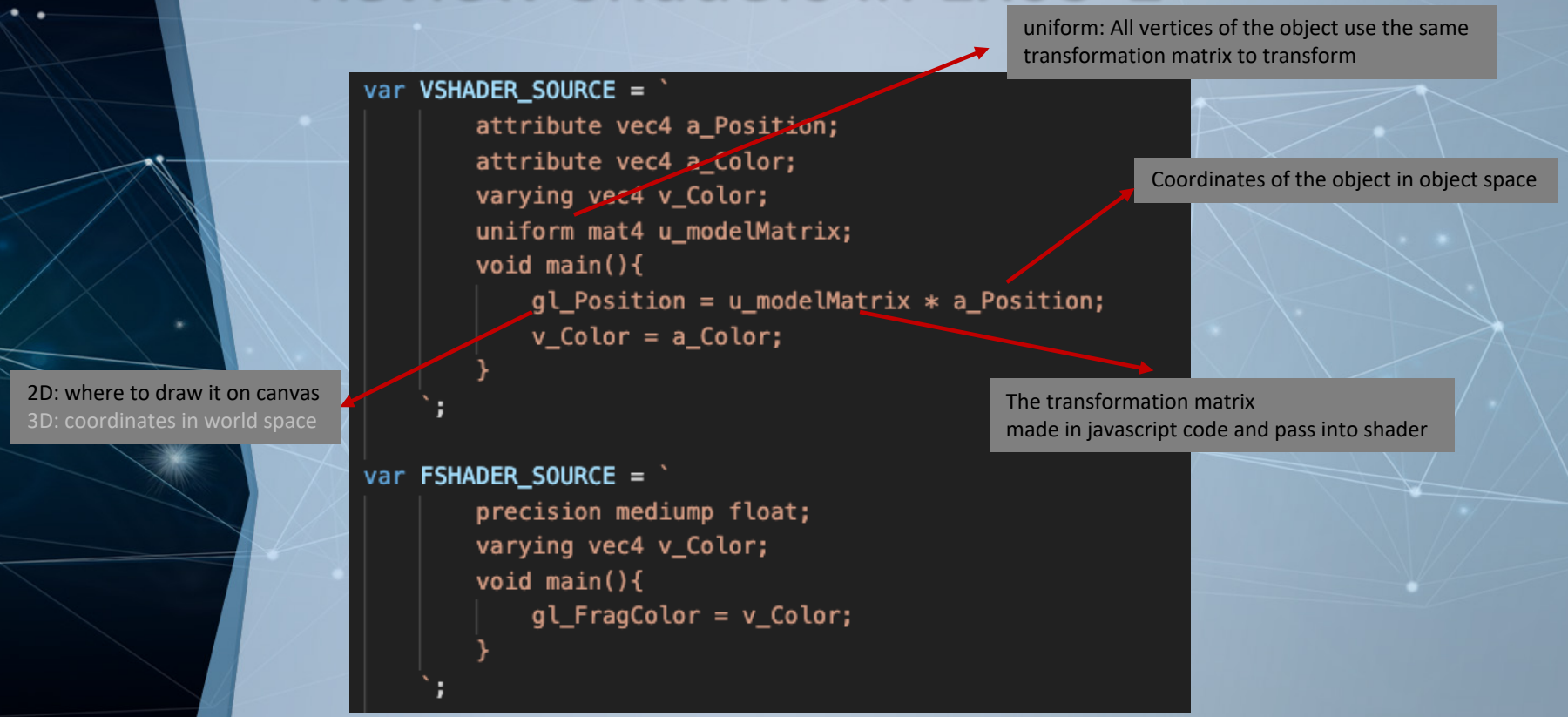
translate(2, 1, 0) -> rotate(45degrees, axis=(0,0,1))->scale(1, 2, 1)

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\left(\frac{PI}{4}\right) & -\sin\left(\frac{PI}{4}\right) & 0 & 0 \\ \sin\left(\frac{PI}{4}\right) & \cos\left(\frac{PI}{4}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -0.828 \\ 6.656 \\ 0 \\ 1 \end{bmatrix}$$

```
var transformMat = new Matrix4();
transformMat.setTranslate(2,1,0);
transformMat.rotate(45, 0, 0, 1);
transformMat.scale(1, 2, 1);
console.log(transformMat.elements);
```

I extend the old 2D example into 3D space, but all z coordinates of the object is still 0

Review Shaders in Ex03-1



Pass Matrix into Shader (GPU)

- cuon-matrix.js just help you do matrix calculation in Javascript
- **After calculation you still have to pass the matrix in javascript variable to shader variable**

- You needs three steps
  1. Of course, you have to declare “matrix variable” in shader to store the matrix
  2. In Javascript, you have to get the reference of the “matrix variable” in shader
  3. Pass the matrix content from javascript to shader

Pass Matrix into Shader (GPU)

1. **Of course, you have to declare "matrix variable" in shader to store the matrix**
2. In Javascript, you have to get the reference of the "matrix variable" in shader
3. Pass the matrix content from javascript to shader

- Matrix type in shader
  - mat2 : 2x2 matrix
  - mat3 : 3x3 matrix
  - mat4 : 4x4 matrix

- Example of declaring 4x4 matrix variable in shader
  - uniform mat4 u_modelMatrix;

- More information of shader matrix type: https://www.khronos.org/opengl/wiki/Data_Type_(GLSL)#Matrix_constructors

Pass Matrix into Shader (GPU)

1. **Of course, you have to declare "matrix variable" in shader to store the matrix**
2. In Javascript, you have to get the reference of the "matrix variable" in shader
3. Pass the matrix content from javascript to shader

- If you declare the matrix as an 'uniform' type (ex: uniform mat4 u_modelMatrix;), we should use 'gl.getUniformLocation' to get the reference of 'u_modelMatrix'

```
gl.useProgram(program);
u_modelMatrix = gl.getUniformLocation(gl.getParameter(gl.CURRENT_PROGRAM), 'u_modelMatrix');
```

Just a javascript variable

Pass Matrix into Shader (GPU)

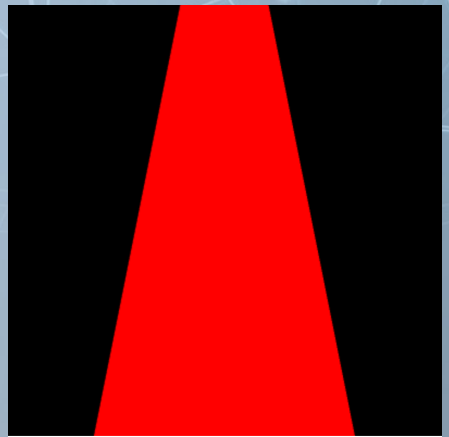
1. **Of course, you have to declare "matrix variable" in shader to store the matrix**
2. In Javascript, you have to get the reference of the "matrix variable" in shader
3. Pass the matrix content from javascript to shader

- gl.uniformMatrix2fv(location, transpose, array) – for 2x2 matrix
- gl.uniformMatrix3fv(location, transpose, array) – for 3x3 matrix
- gl.uniformMatrix4fv(location, transpose, array) – for 4x4 matrix
  - location: the uniform matrix variable
  - transpose: do you want to transpose the matrix (usually false)
  - array: the matrix array

- Example
  - gl.uniformMatrix4fv(u_modelMatrix, false, transformMat.elements)

Example (Ex03-2)

- Download Ex03-2 from Moodle
- This example can draw a triangle
  - Draw a big triangle (but, what it draws is not important)
  - We use console.log() to print out the content of matrix after calling cuon-matrix.js
    - This example let you observe how the matrix change after calling the functions
- Files
  - index.html
  - WebGL.js
  - cuon-matrix.js



Example (Ex03-2)

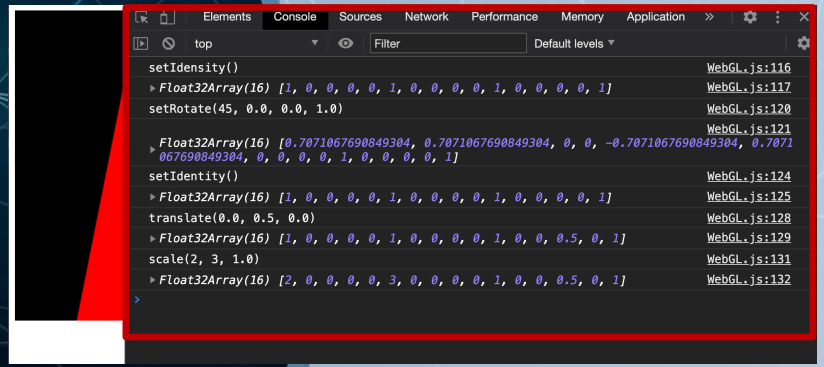
- WebGL.js

Create the matrix and store in a javascript variable

```
var transformMat = new Matrix4();
var matStack = [];
var u_modelMatrix;
```

```
gl.useProgram(program);
u_modelMatrix = gl.getUniformLocation(gl.getParameter(gl.CURRENT_PROGRAM), 'u_modelMatrix');

transformMat.setIdentity();
console.log("setIdensity()");
console.log(transformMat.elements);

transformMat.setRotate(45, 0.0, 0.0, 1.0);
console.log("setRotate(45, 0.0, 0.0, 1.0)");
console.log(transformMat.elements);

transformMat.setIdentity();
console.log("setIdentity()");
console.log(transformMat.elements);

transformMat.translate(0.0, 0.5, 0.0);
console.log("translate(0.0, 0.5, 0.0)");
console.log(transformMat.elements);
transformMat.scale(2, 3, 1.0);
console.log("scale(2, 3, 1.0)");
console.log(transformMat.elements);

gl.uniformMatrix4fv(u_modelMatrix, false, transformMat.elements); //pass the transformation matr
/////Draw a triangle/////
triangleVertices = [0.0, 0.5, -0.3, -0.5, 0.3, -0.5]; //define the triangle in object space
var triangleColor = [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0 ]; //red triangle
buffer0 = initArrayBuffer(gl, new Float32Array(triangleVertices), 2, gl.FLOAT, 'a_Position');
buffer1 = initArrayBuffer(gl, new Float32Array(triangleColor), 3, gl.FLOAT, 'a_Color');
gl.drawArrays(gl.TRIANGLES, 0, triangleVertices.length/2);

}
```

Call functions to do matrix calculation and print out the matrix



Enlarge the console window and refresh the page again to see the matrix values directly

Example (Ex03-2)

- WebGL.js

A Javascript variable to store the var reference in shader

```
var transformMat = new Matrix4();
var matStack = [];
var u_modelMatrix;
```

```
gl.useProgram(program);
u_modelMatrix = gl.getUniformLocation(gl.getParameter(gl.CURRENT_PROGRAM), 'u_modelMatrix');
```

The shader variable reference

```
transformMat.setIdentity();
console.log("setIdensity()");
console.log(transformMat.elements);

transformMat.setRotate(45, 0.0, 0.0, 1.0);
console.log("setRotate(45, 0.0, 0.0, 1.0)");
console.log(transformMat.elements);

transformMat.setIdentity();
console.log("setIdentity()");
console.log(transformMat.elements);

transformMat.translate(0.0, 0.5, 0.0);
console.log("translate(0.0, 0.5, 0.0)");
console.log(transformMat.elements);
transformMat.scale(2, 3, 1.0);
console.log("scale(2, 3, 1.0)");
console.log(transformMat.elements);
```

Pass matrix into shader

```
gl.uniformMatrix4fv(u_modelMatrix, false, transformMat.elements); //pass the transformation mat
/////Draw a triangle/////
triangleVertices = [0.0, 0.5, -0.3, -0.5, 0.3, -0.5]; //define the triangle in object space
var triangleColor = [1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0 ]; //red triangle
buffer0 = initArrayBuffer(gl, new Float32Array(triangleVertices), 2, gl.FLOAT, 'a_Position');
buffer1 = initArrayBuffer(gl, new Float32Array(triangleColor), 3, gl.FLOAT, 'a_Color');
gl.drawArrays(gl.TRIANGLES, 0, triangleVertices.length/2);

}
```

Shader code

```
var VSHADER_SOURCE = `
    attribute vec4 a_Position;
    attribute vec4 a_Color;
    varying vec4 v_Color;
    uniform mat4 u_modelMatrix;
    void main(){
        gl_Position = u_modelMatrix * a_Position;
        v_Color = a_Color;
    }
`;

var FSHADER_SOURCE = `
    precision mediump float;
    varying vec4 v_Color;
    void main(){
        gl_FragColor = v_Color;
    }
`;
```

Transform vertices of the triangle

Let’s try (5mins)

- Download Ex03-2 from Moodle
- Try to understand what is going on in shader codes