Transformation

Computer Graphics

Yu-Ting Wu

Outline

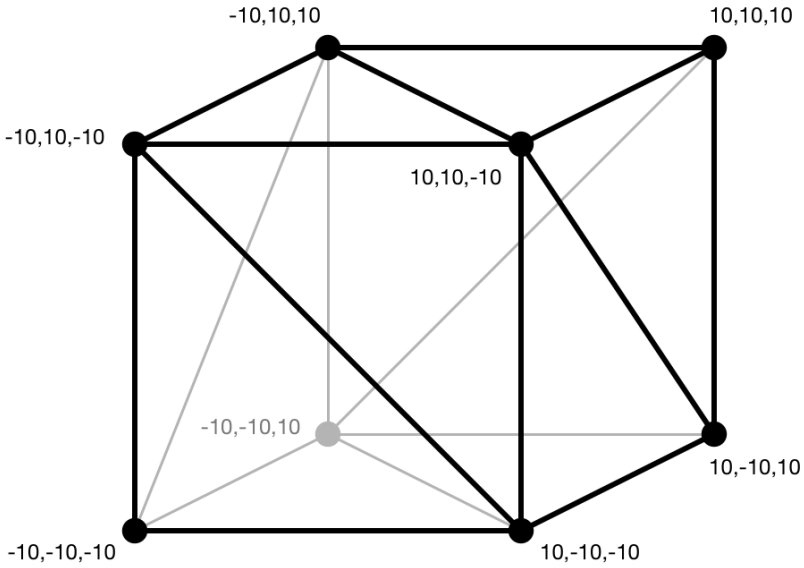
- Overview (world transformation)
- Transformation
- OpenGL implementation

Outline

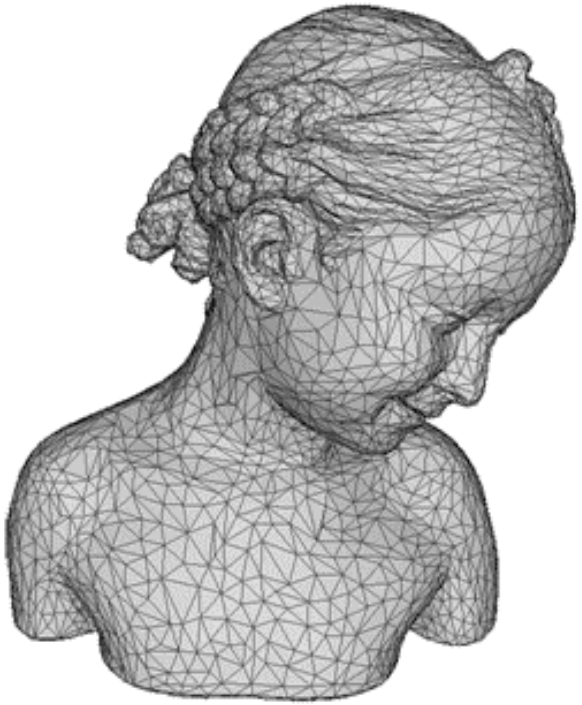
- **Overview (world transformation)**
- Transformation
- OpenGL implementation

Recap: Describing Geometry

- Geometry of an object is defined by specifying the coordinates of the **vertices** and **their adjacencies**



12 triangles



10K triangles

Describing Scenes

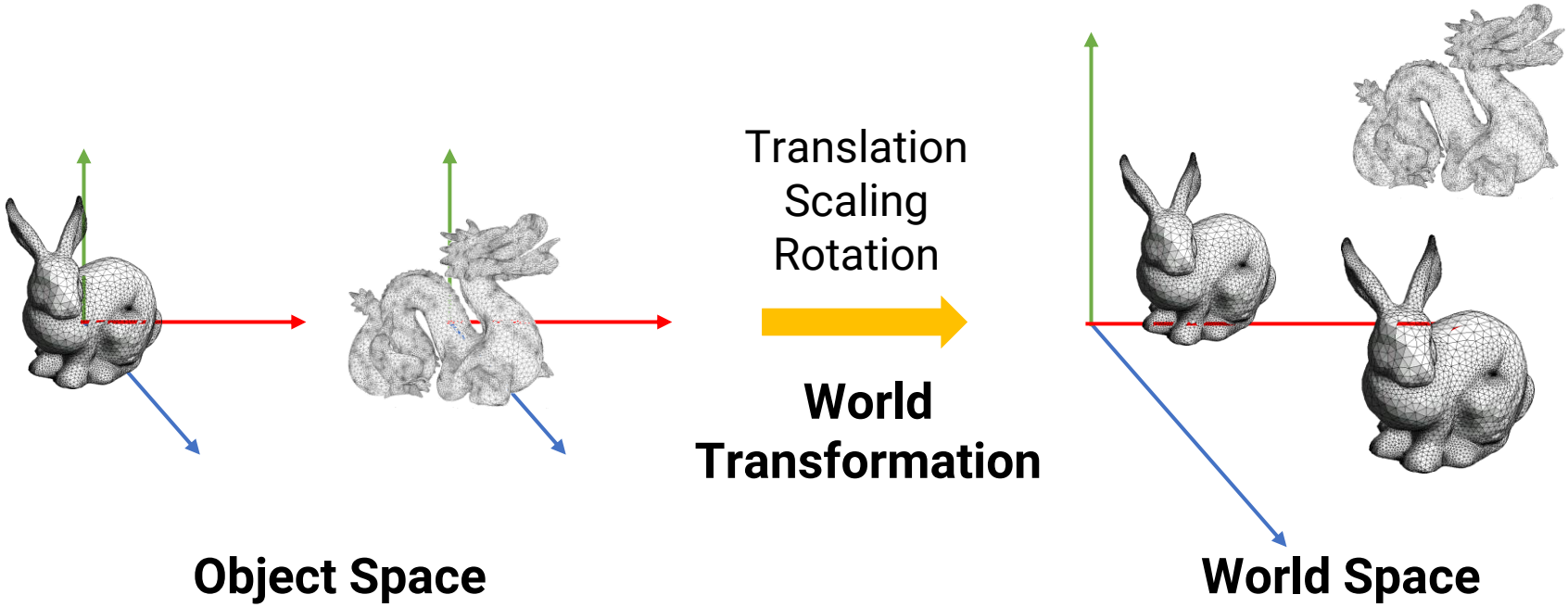
- A virtual scene usually consists of lots of objects

Object Space and World Space

- Objects are defined in **object space individually**

Object Space and World Space (cont.)

- Objects are defined in **object space individually**
- When building a scene, each object is transformed to a **global** and **unique** space called **world space**
- The transform is called **world transform**



World Space and World Coordinate (cont.)

- Advantages of using “transformation”
  - **Reuse model:** design a model and use it in several scenes
  - **Memory saving:** store a 4x4 matrix instead of duplicating the entire models

Outline

- Overview (world transformation)
- **Transformation**
- OpenGL implementation

2D Transformations

- 2D transformation of a point can be represented by the multiplication of a **column vector (point)** and a **transformation matrix**

transform matrix

$$p' = Mp$$

new point　　original point

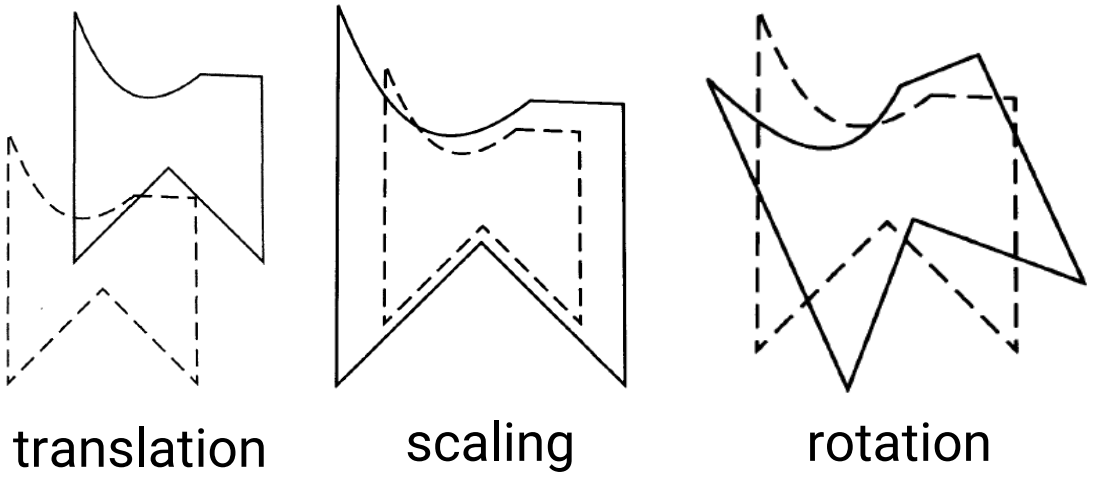
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}_{p'} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}_{p}$$

$$x' = ax + by + c$$
$$y' = dx + ey + f$$

- Common transformations include **translation**, **scaling**, and **rotation**

Common Transformations

- Translation
- Scaling
- Rotation

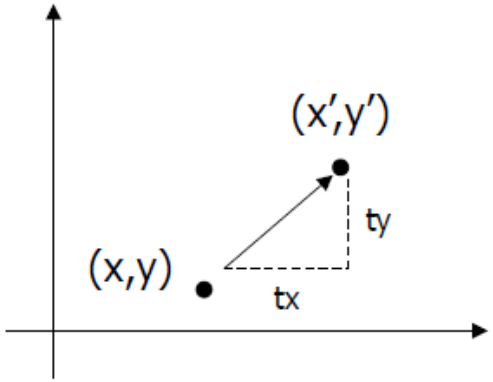


- We will start by introducing 2D transformations and then extend to the 3D cases

2D Translation

- Given a point **p(x, y)** and a **translation offset T(t_x, t_y)**, the new point **p'(x', y')** after translation is **p' = p + T**

$$x' = x + t_x$$
$$y' = y + t_y$$



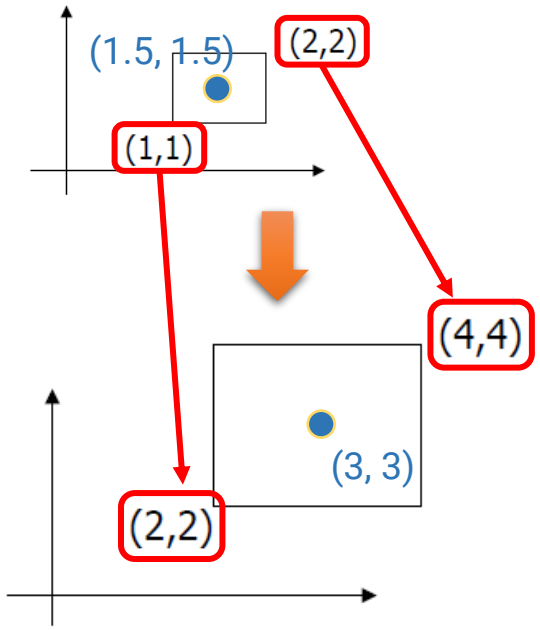
- **Can be represented as** Matrix-vector multiplication

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2D Scaling

- Given a point **p(x, y)** and a scaling factor **S(s_x, s_y)**, the new point **p'(x', y')** after scaling is **p' = S p**

$$x' = x * s_x$$
$$y' = y * s_y$$

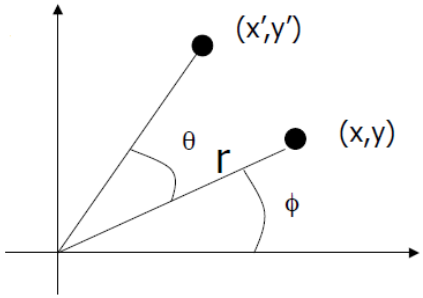


- Matrix-vector multiplication

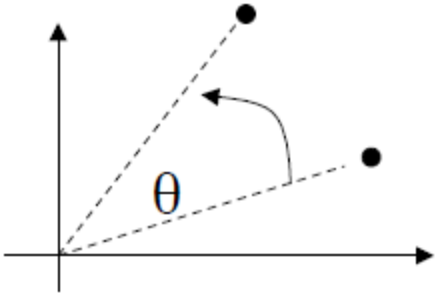
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2D Rotation

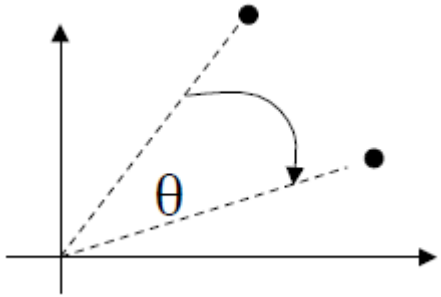
- Given a point **p(x, y)**, rotate it with respect to the **origin** by **Θ** and get the new point **p'(x', y')** after rotation



- Firstly we define



Θ > 0: rotate counterclockwise

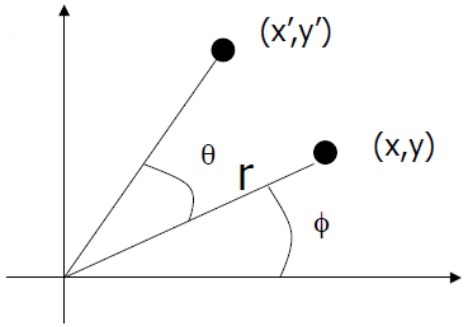


Θ < 0: rotate clockwise

2D Rotation (cont.)

- Given a point **p(x, y)**, rotate it with respect to the **origin** by **θ** and get the new point **p'(x', y')** after rotation

$$x = r\cos(\phi) \qquad y = r\sin(\phi)$$
$$x' = r\cos(\phi + \theta) \qquad y' = r\sin(\phi + \theta)$$



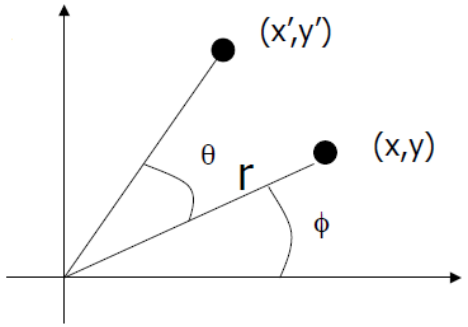
$$\begin{aligned} x' &= r\cos(\phi + \theta) \\ &= r\cos(\phi)\cos(\theta) - r\sin(\phi)\sin(\theta) \\ &= x\cos(\theta) - y\sin(\theta) \end{aligned}$$

$$\begin{aligned} y' &= r\sin(\phi + \theta) \\ &= r\sin(\phi)\cos(\theta) + r\cos(\phi)\sin(\theta) \\ &= y\cos(\theta) + x\sin(\theta) \end{aligned}$$

2D Rotation (cont.)

- Given a point **p(x, y)**, rotate it with respect to the **origin** by **Θ** and get the new point **p'(x', y')** after rotation

$$
\begin{aligned}
x' &= r\cos(\phi+\theta) \\
&= x\cos(\theta) - y\sin(\theta) \\
y' &= r\sin(\phi+\theta) \\
&= y\cos(\theta) + x\sin(\theta)
\end{aligned}
$$



- Matrix-vector multiplication

$$
\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}
$$

2D Translation, Scaling, and Rotation

- Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Scaling

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Using a 3x3 matrix allows us to perform all transformations using matrix/vector multiplications
  - We can also **pre-multiply (concatenate)** all the matrices

Homogeneous Coordinate

- We call the (x, y, 1) representation the **homogeneous coordinate** for (x, y)

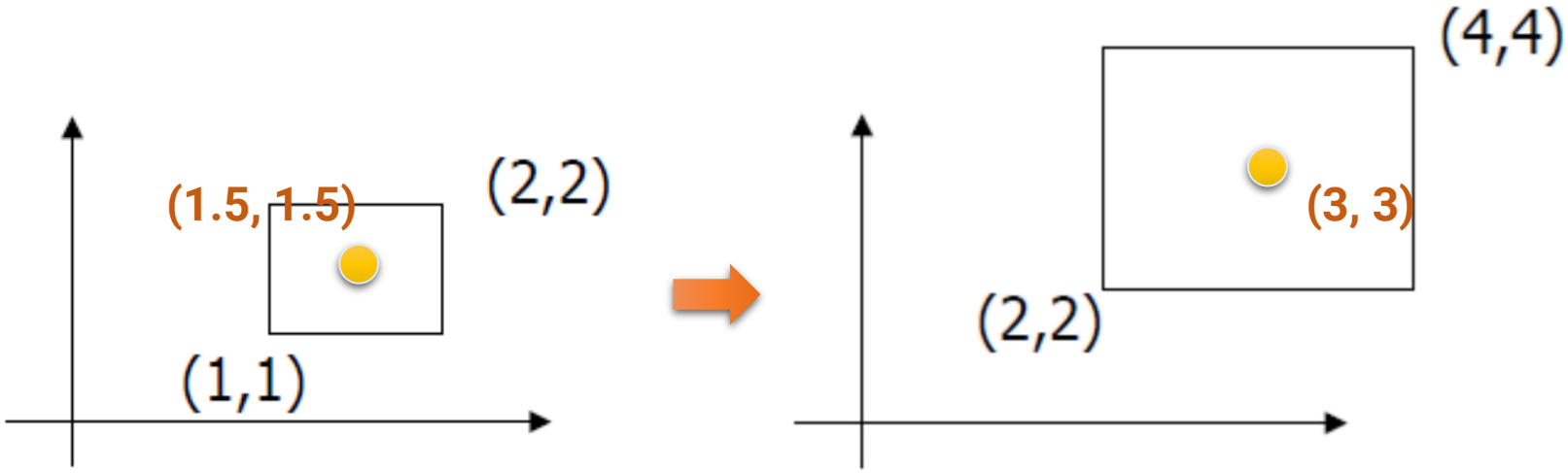
$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- If w is not equal to 1, to make the transformed coordinate also homogeneous, we need to divide the x and y components by w

$$x' = x'/w \qquad y' = y'/w \qquad w = 1$$

Revisit 2D Scaling

- The standard scaling matrix will only anchor at (0, 0)
  - Otherwise, the object center got shifted

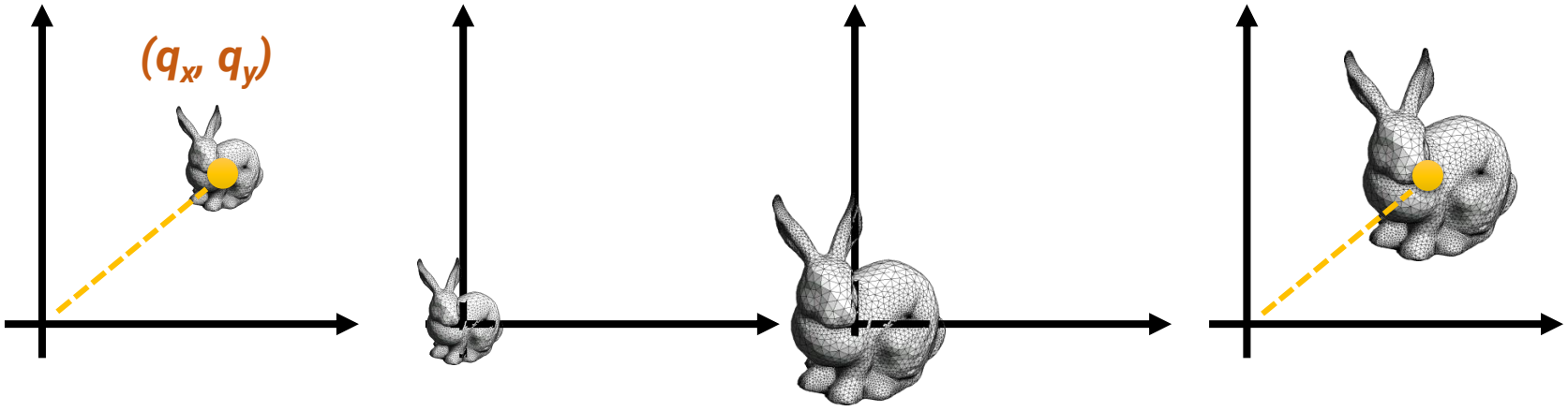


- What if we want the object to be scaled w.r.t its center?

Revisit 2D Scaling (cont.)

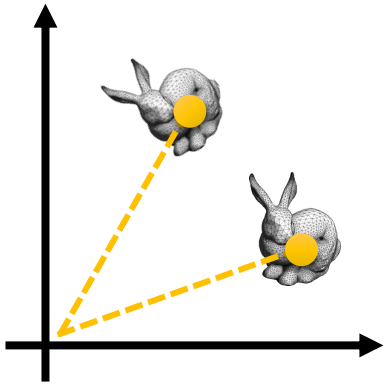
- Scaling about an arbitrary pivot point **$Q(q_x, q_y)$**
  - Translate the objects so that Q will coincide with the origin: **$T(-q_x, -q_y)$**
  - Scale the object: **$S(s_x, s_y)$**
  - Translate the object back: **$T(q_x, q_y)$**
- The final scaling matrix can be written as $T(q)S(s)T(-q)$

Concatenation of matrices



Revisit 2D Rotation

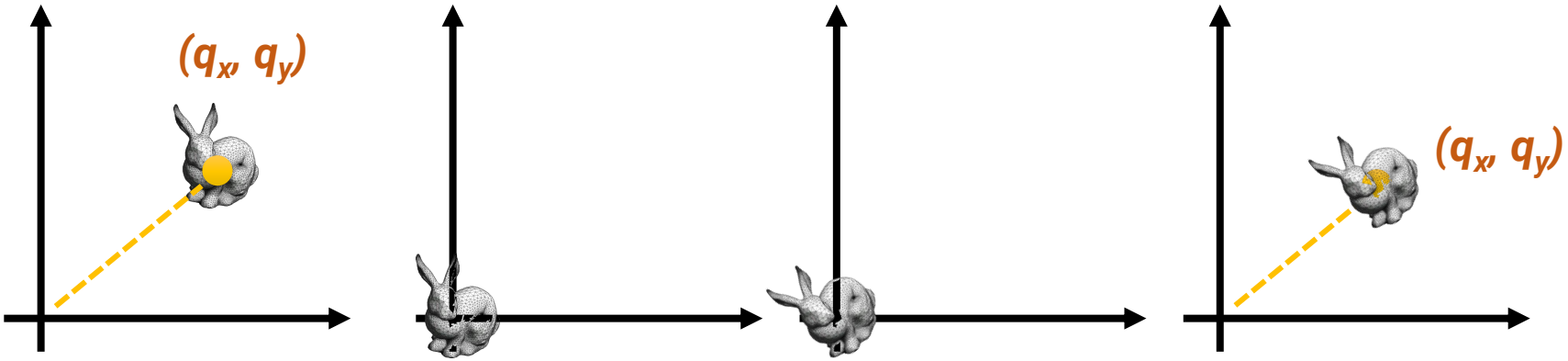
- The standard rotation matrix is used to rotate about the origin (0, 0)



- What if we want the object to be rotated w.r.t a specific pivot?

Revisit 2D Rotation (cont.)

- Rotate about an arbitrary pivot point **Q(q_x, q_y)** by **Θ**
  - Translate the objects so that Q will coincide with the origin: **T(-q_x, -q_y)**
  - Rotate the object: **R(Θ)**
  - Translate the object back: **T(q_x, q_y)**
- The final rotation matrix can be written as $T(q)R(\theta)T(-q)$



Translation (3D) and Scaling (3D)

- A 3D transformation is represented as a **4x4 matrix**, with **homogeneous coordinate**

translation

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

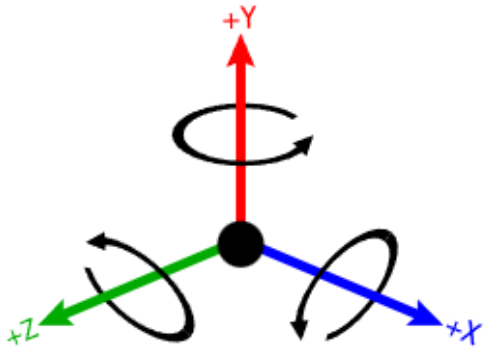
scaling

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

2D 3D

Rotation (3D)

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



2D

rotation w.r.t 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 w.r.t 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 w.r.t z-axis

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

3D

Outline

- Overview (world transformation)
- Transformation
- **OpenGL implementation**

Goals

- Learn how to build the transformation matrices
- Learn how to code with GLM matrices
- Learn how to concatenate the transformation

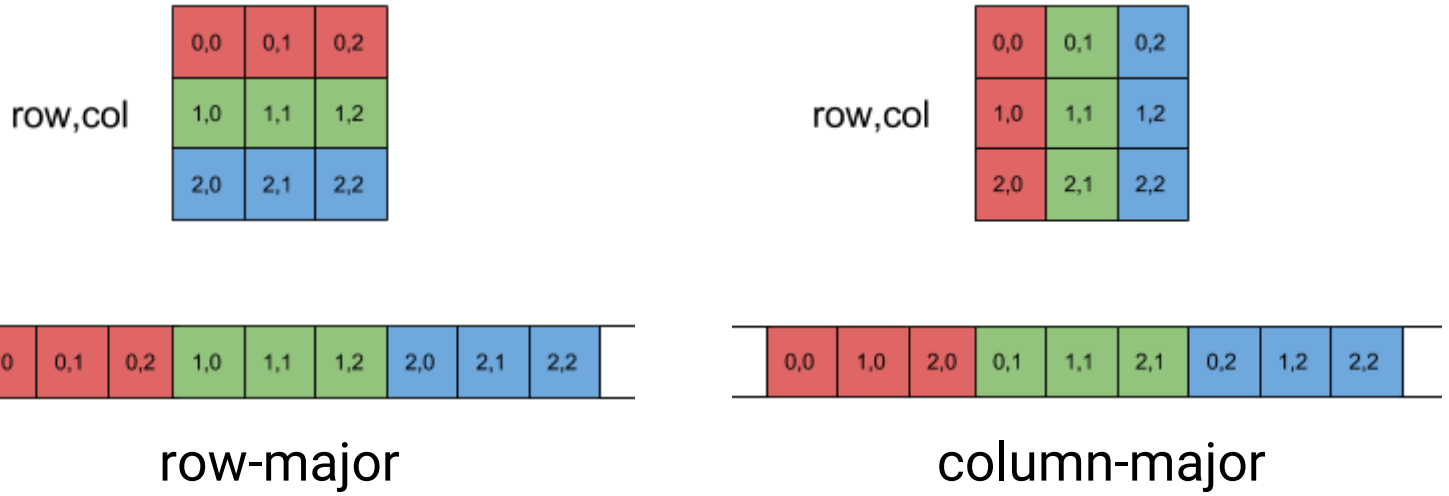
GLM Matrix

- GLM provides several classes to support matrices with different rows and columns
  - Square matrix
    - glm::mat2 (equals to glm::mat2x2)
    - glm::mat3 (equals to glm::mat3x3)
    - glm::mat4 (equals to glm::mat4x4)
  - Non-square matrix
    - glm::mat***m***x***n*** (***m*** and ***n*** are in the range from 2 to 4)

- Declare a **zero** 4x4 matrix: glm::mat4x4(**0.0f**);
- Declare an **identity** 4x4 matrix: glm::mat4x4(**1.0f**);

Matrix Representation: Column/Row Major

- A 2-dimensional matrix can be accessed by either column-major or row-major



- By default, OpenGL (and thus GLM) supplies matrix data in **column-major**

Translation Matrix

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

base matrix

- glm::mat4x4 translate(const glm::mat4x4& m ,
  const glm::vec3& v)

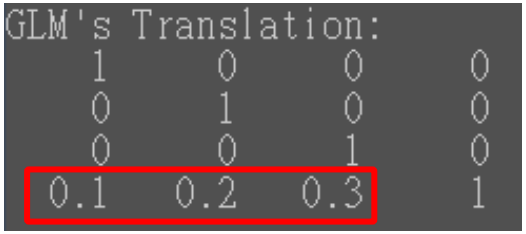
returned translation matrix

translation vector

```
glm::mat4x4 gT = glm::translate(glm::mat4x4(1.0f),
                                glm::vec3(0.1f, 0.2f, 0.3f));
```

Translation Matrix (cont.)

- If you print the matrix produced by glm::translate, you will get the following result

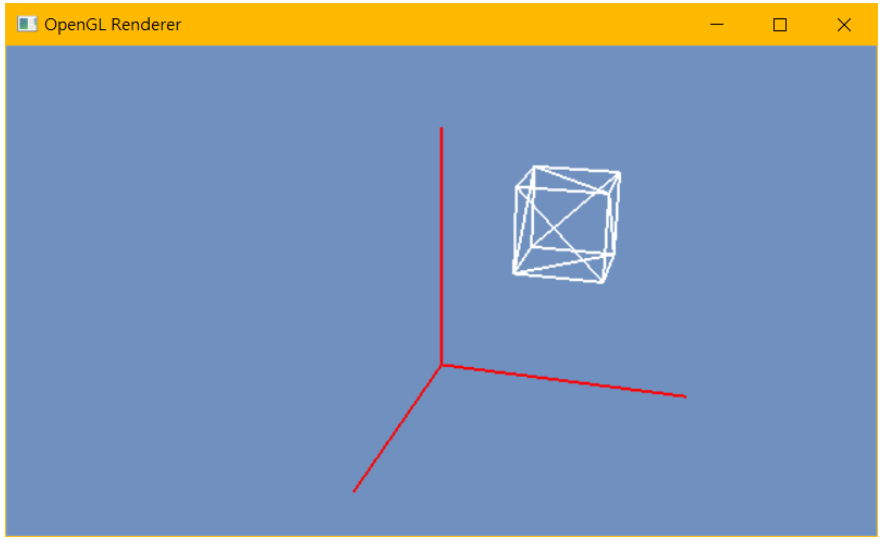


Why? OpenGL and GLM use column-major representation!

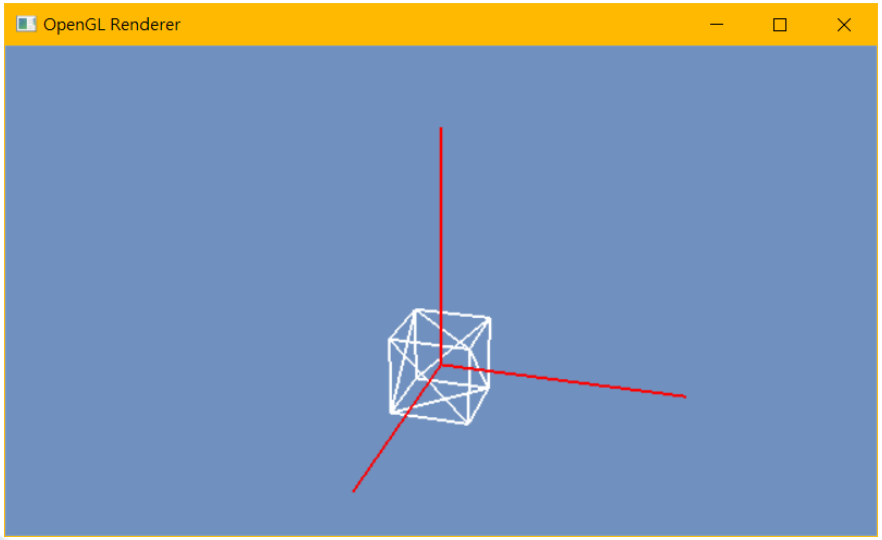
- If you want to build the matrix on your own, remember to transpose the matrix

```
void BuildTranslationMatrix(glm::mat4x4& T, const glm::vec3& tr)
{
    T[0][0] = 1.0f;  T[0][1] = 0.0f;  T[0][2] = 0.0f;  T[0][3] = 0.0f;
    T[1][0] = 0.0f;  T[1][1] = 1.0f;  T[1][2] = 0.0f;  T[1][3] = 0.0f;
    T[2][0] = 0.0f;  T[2][1] = 0.0f;  T[2][2] = 1.0f;  T[2][3] = 0.0f;
    T[3][0] = tr.x;  T[3][1] = tr.y;  T[3][2] = tr.z;  T[3][3] = 1.0f;
}
```

Translation Matrix (cont.)



cube center (1.5, 2.0, 0.0)



cube center (0.0, 0.0, 0.0)

apply a translation of (-1.5, -2.0, 0.0)

Scaling Matrix

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

base matrix

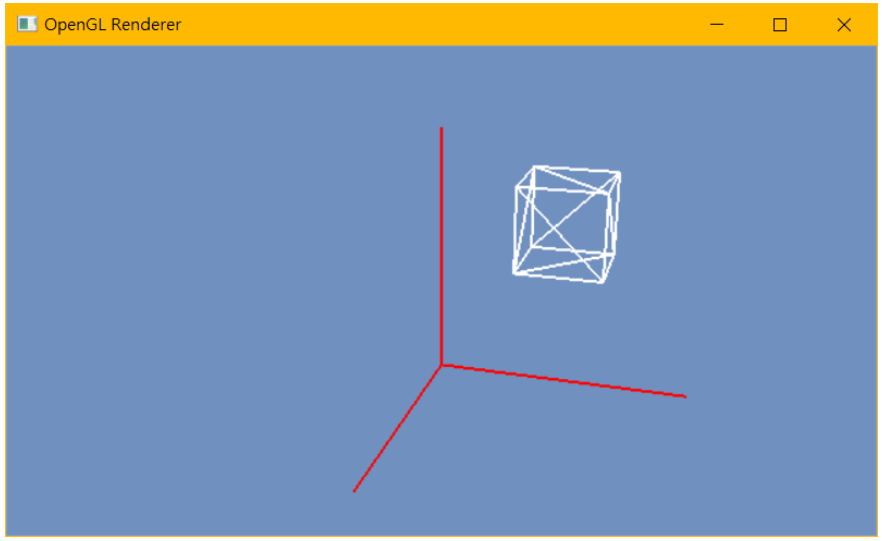
- glm::mat4x4 scale(const glm::mat4x4& m ,
  const glm::vec3& v)

returned scaling matrix

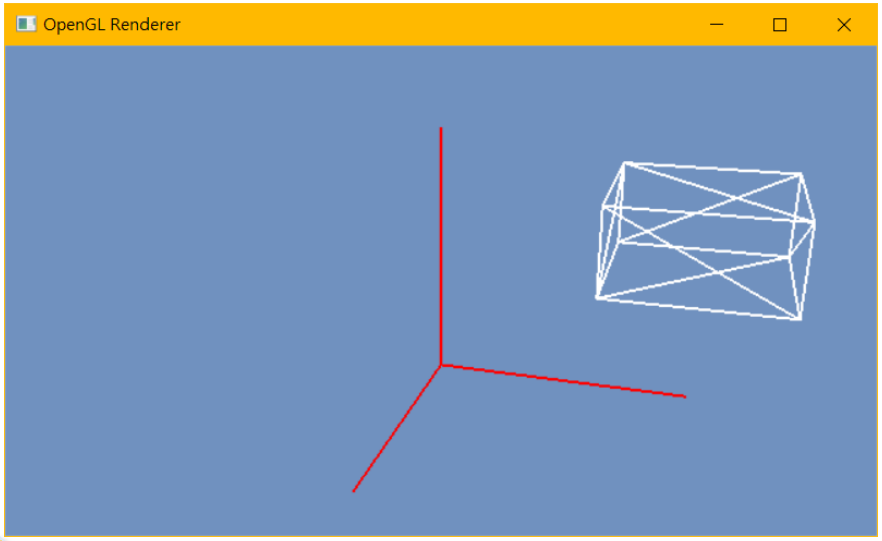
scaling vector

```
glm::mat4x4 gS = glm::scale(glm::mat4x4(1.0f),
                            glm::vec3(0.5f, 0.4f, 0.2f));
```

Translation Matrix (cont.)



cube center (1.5, 2.0, 0.0)



cube center (3.0, 2.0, 0.0)

apply a scaling of (2.0, 1.0, 2.0)

Rotation Matrix

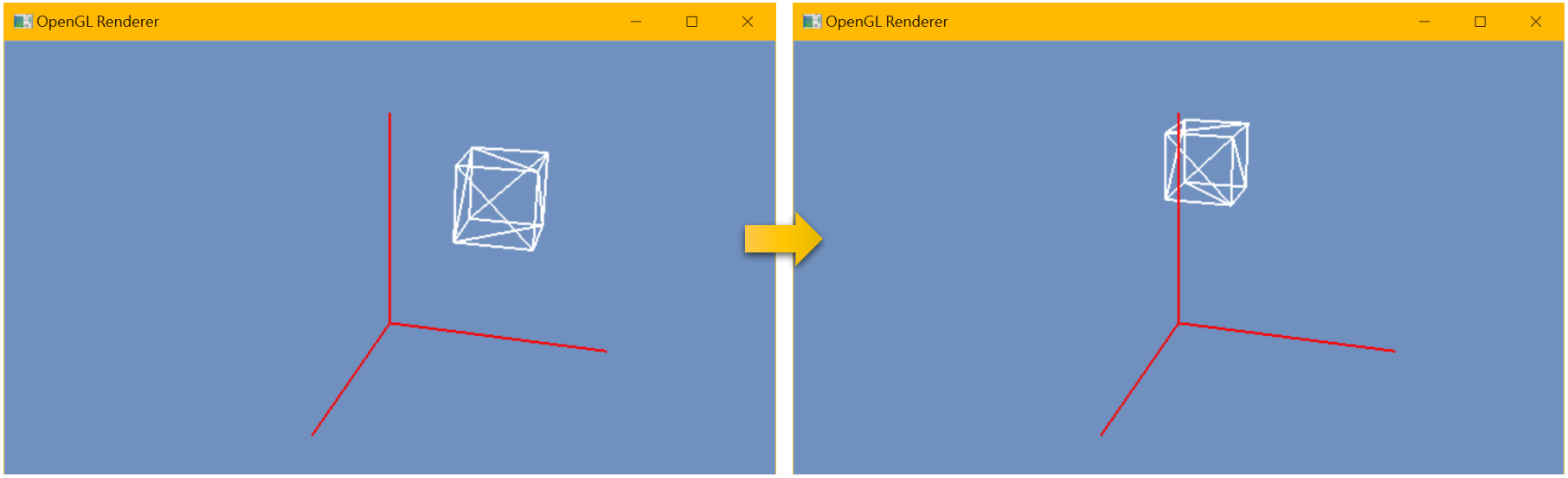
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos\theta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

rotation w.r.t x-axis | rotation w.r.t y-axis | rotation w.r.t z-axis

- glm::mat4x4 rotate(const glm::mat4x4& m , base matrix
  - returned scaling matrix
  - const float angle , rotate amount in **radian**
  - const glm::vec3& axis) rotate axis

```
glm::mat4x4 gRx = glm::rotate(glm::mat4x4(1.0f), glm::radians(30.0f), glm::vec3(1, 0, 0));
glm::mat4x4 gRy = glm::rotate(glm::mat4x4(1.0f), glm::radians(45.0f), glm::vec3(0, 1, 0));
glm::mat4x4 gRz = glm::rotate(glm::mat4x4(1.0f), glm::radians(60.0f), glm::vec3(0, 0, 1));
```

3D Rotating in Place (cont.)

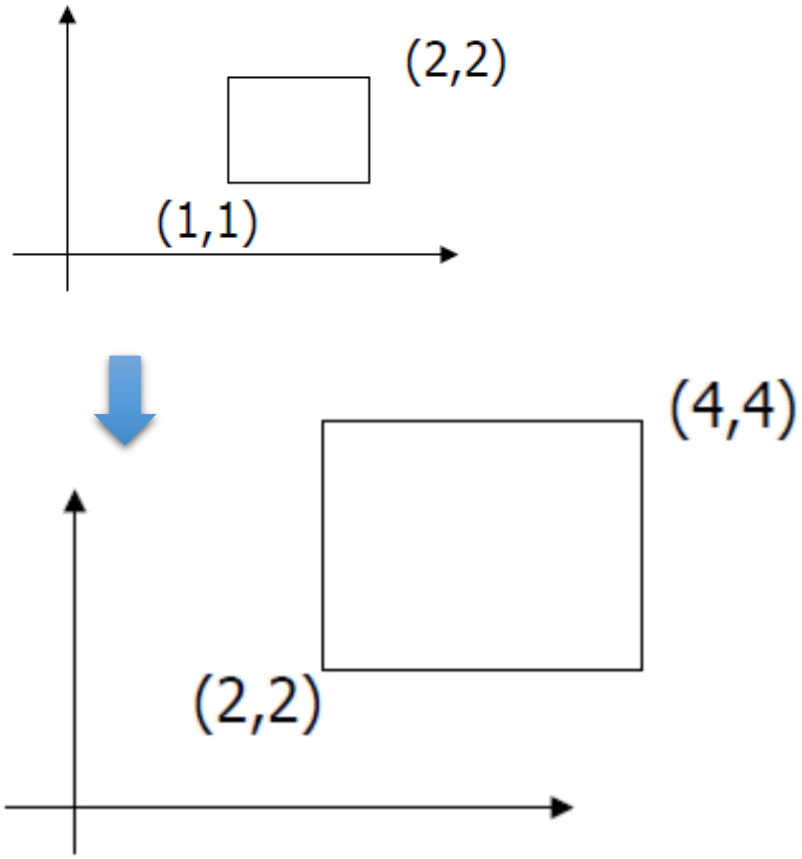


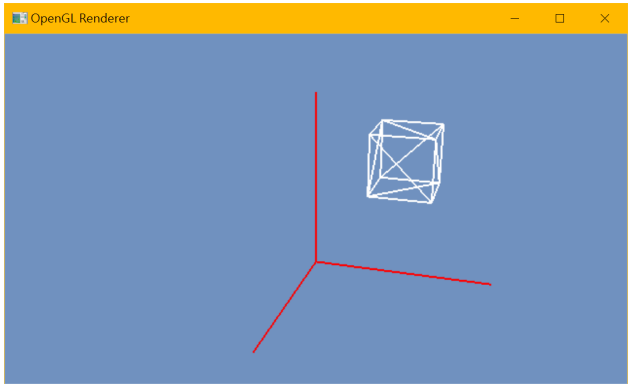

```
glm::mat4x4 RY = glm::rotate(glm::mat4x4(1.0f), glm::radians(90.f), glm::vec3(0, 1, 0));
worldMatrix = RY;
```

rotate w.r.t the global Y axis

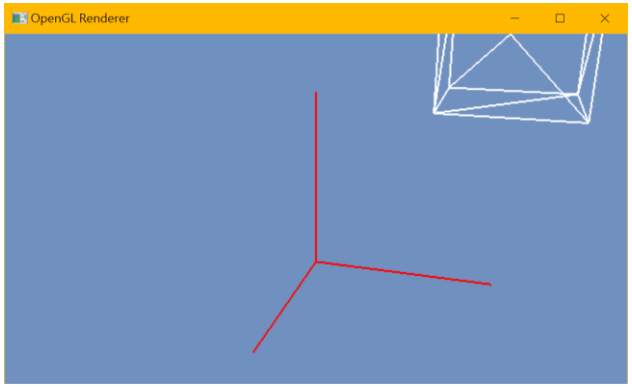
3D Scaling in Place

- The standard scaling matrix will only anchor at (0, 0)





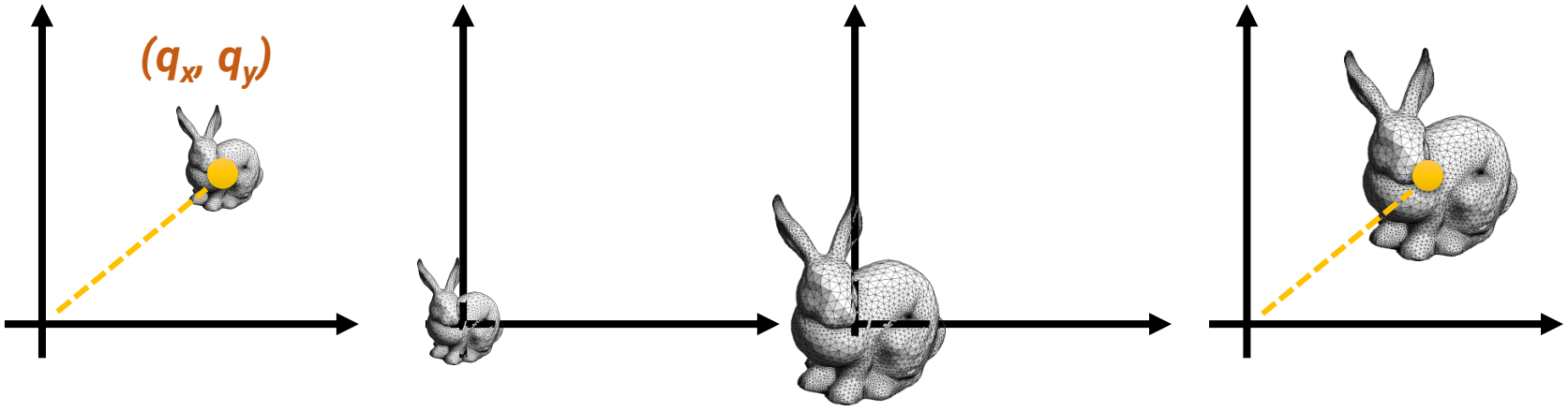
cube center at (1.5, 2.0, 0.0)



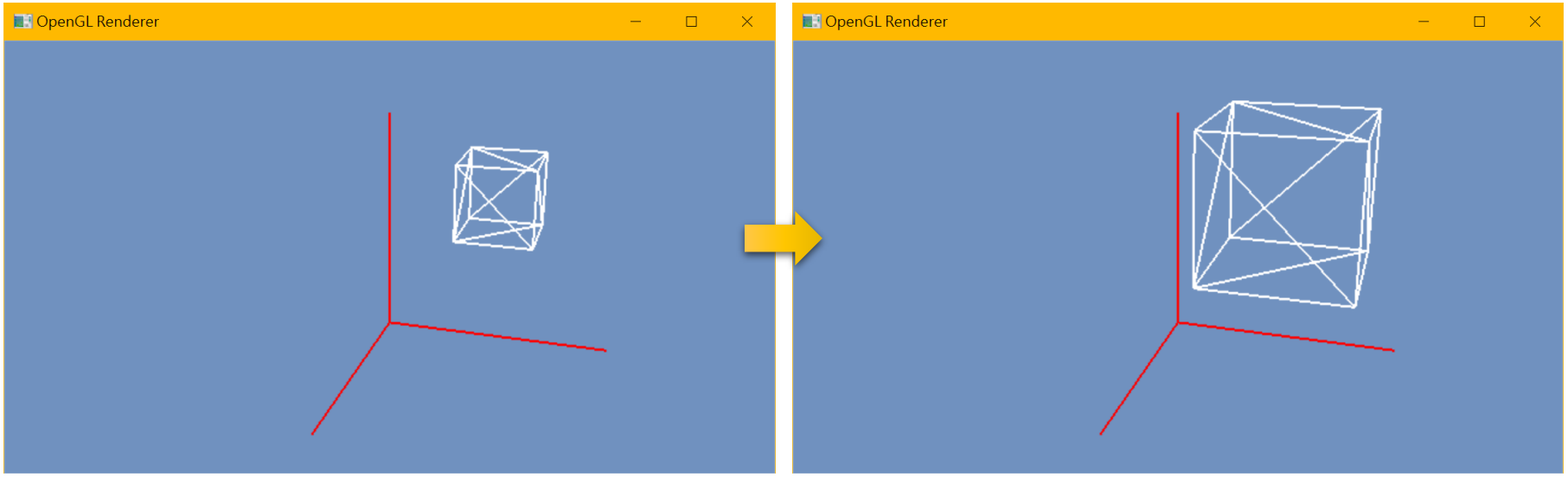
cube center at (3.0, 4.0, 0.0)

3D Scaling in Place (cont.)

- Scaling about an arbitrary pivot point $Q(q_x, q_y)$
  - Translate the objects so that Q will coincide with the origin: $T(-q_x, -q_y)$
  - Scale the object: $S(s_x, s_y)$
  - Translate the object back: $T(q_x, q_y)$
- The final scaling matrix can be written as $T(q)S(s)T(-q)$



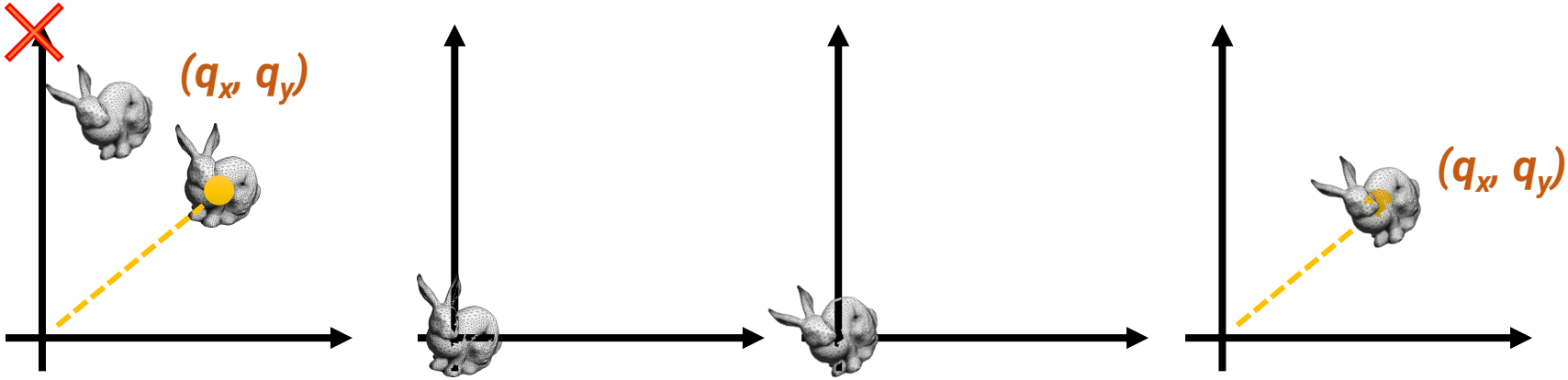
3D Scaling in Place (cont.)



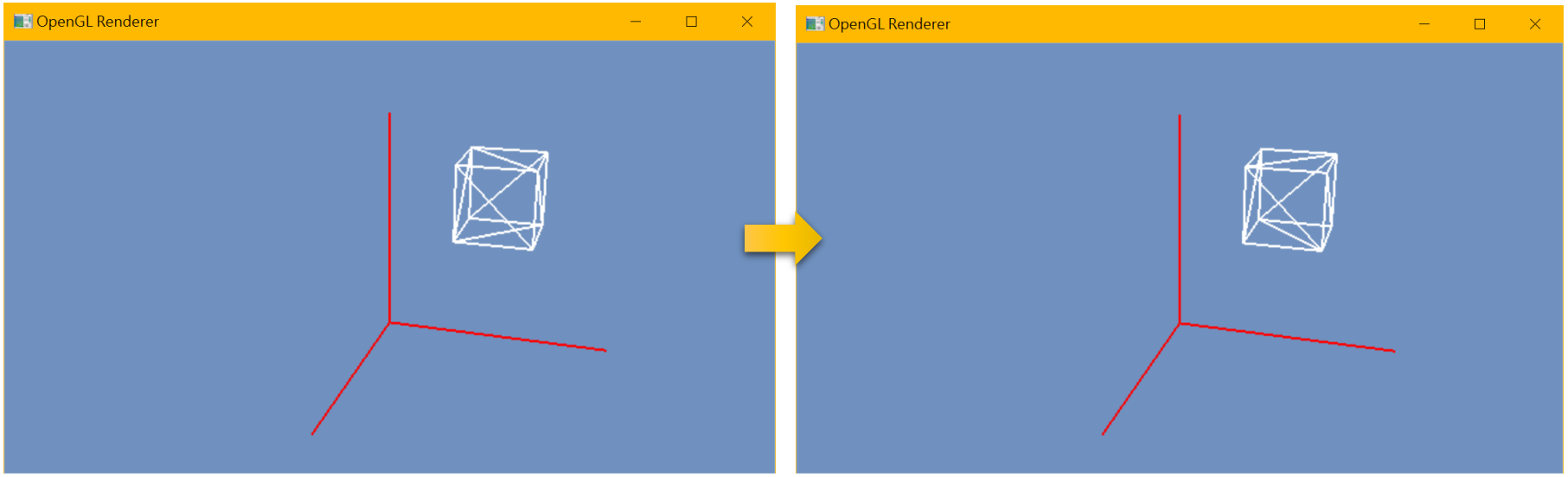

```
glm::mat4x4 T1 = glm::translate(glm::mat4x4(1.0f), glm::vec3(-1.5f, -2.0f, 0.0f));
glm::mat4x4 S = glm::scale(glm::mat4x4(1.0f), glm::vec3(2.0f, 2.0f, 2.0f));
glm::mat4x4 T2 = glm::translate(glm::mat4x4(1.0f), glm::vec3( 1.5f,  2.0f, 0.0f));
worldMatrix = T2 * S * T1;
```

3D Rotating in Place (cont.)

- The standard rotation matrix rotates about an **axis**
- Rotate about an arbitrary pivot point $Q(q_x, q_y)$ by **Θ**
  - Translate the objects so that Q will coincide with the origin: **$T(-q_x, -q_y)$**
  - Rotate the object: **$R(\Theta)$**
  - Translate the object back: **$T(q_x, q_y)$**
- The final rotation matrix can be written as $T(q)R(\theta)T(-q)$



3D Rotating in Place (cont.)

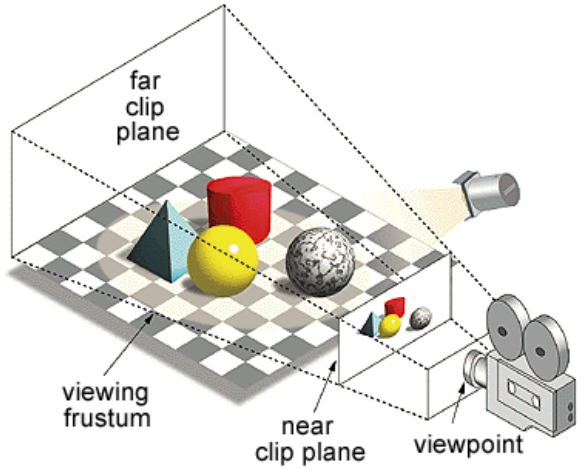



```
glm::mat4x4 T1 = glm::translate(glm::mat4x4(1.0f), glm::vec3(-1.5f, -2.0f, 0.0f));
glm::mat4x4 RY = glm::rotate(glm::mat4x4(1.0f), glm::radians(90.f), glm::vec3(0, 1, 0));
glm::mat4x4 T2 = glm::translate(glm::mat4x4(1.0f), glm::vec3( 1.5f,  2.0f, 0.0f));
worldMatrix = T2 * RY * T1;
```

rotate in place!

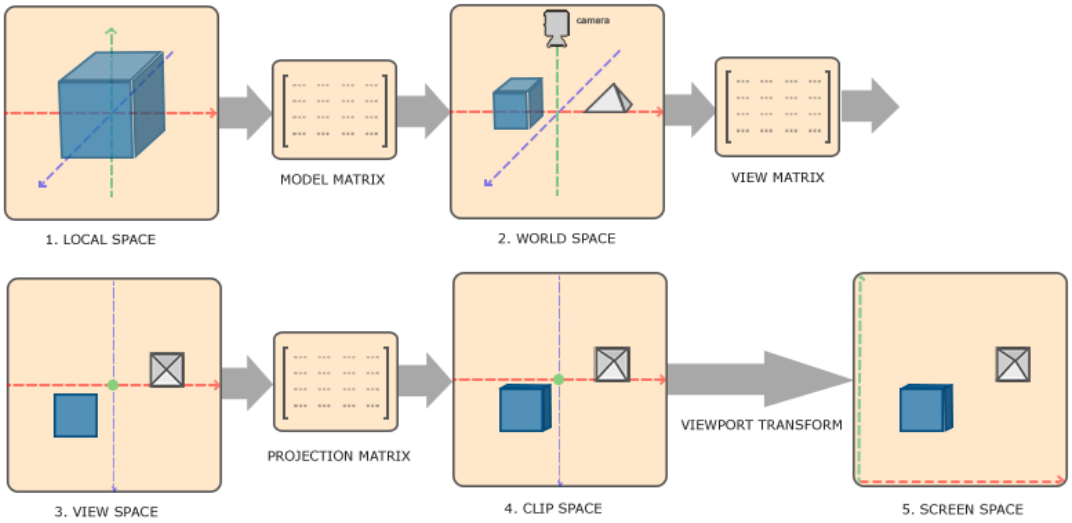
Where is the Camera and Projection?

- The typical flow of bringing a 3D point to the 2D screen involves the **camera projection**
- For now, we specify neither the camera nor the projection, so you can consider that we set the **“projected”** positions of the vertices directly
- In the next topic (camera), we will go through the full transformation



Spoiler

- There are other spaces
- We will introduce **camera space**, **clip space**, and **NDC** in the next slides



for assisting rendering

for building scene

Object Space
(Local Space)

World Space

Camera Space
(View, Eye Space)

Clip Space

Normalized Device Coordinate (NDC)

Screen Space

for displaying


END OF PRESENTATION
ANY QUESTIONS?