Implementation: Transformation

Introduction to Computer Graphics

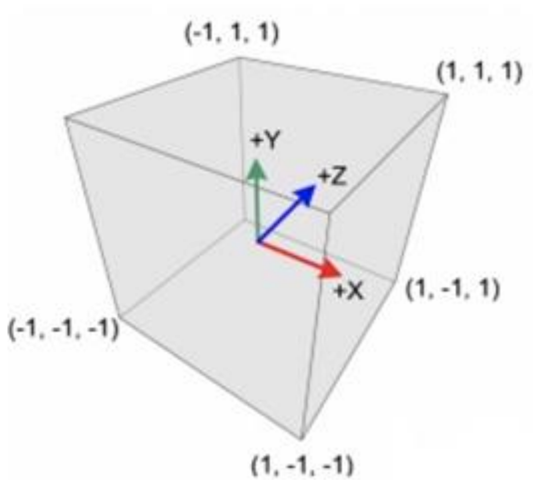
Yu-Ting Wu

Program Overview

Goals

- Learn how to build the transformation matrices
- Learn how to concatenate the transformation
- Learn how to transform a vertex from object space to screen space

Recap.



Object Space
(Local Space)

↓

World Space

↓

Camera Space
(View, Eye Space)

↓

Clip Space

Normalized Device Coordinate (NDC)

↓

Screen Space

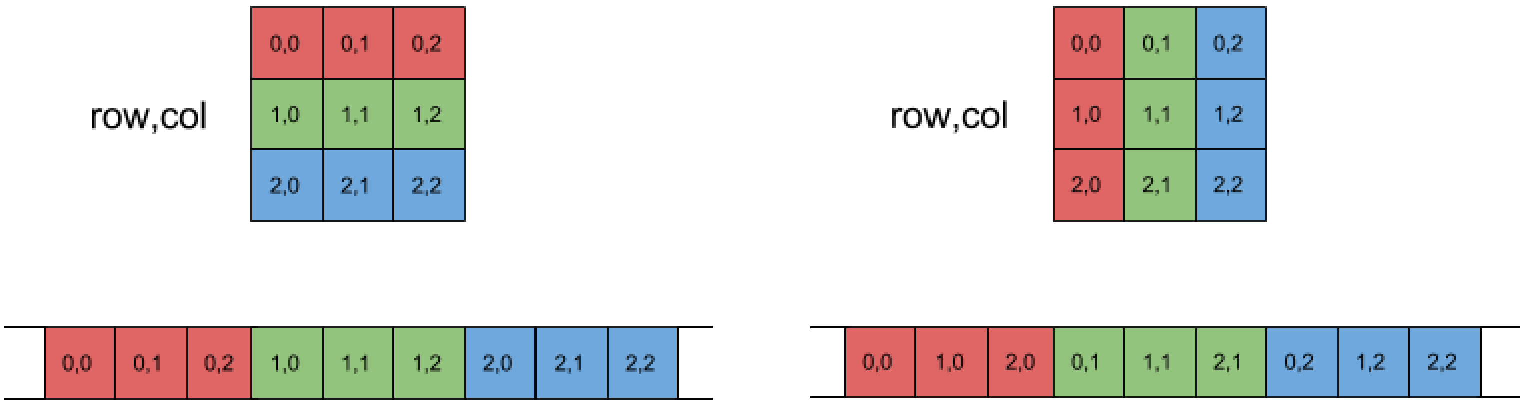
GLM Matrix

- 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



row-major column-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

```
GLM's Translation:
     1       0       0       0
     0       1       0       0
     0       0       1       0
   0.1     0.2     0.3       1
```

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;
}
```

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, 0.3f));
```

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
  const float angle, rotate amount in **radian**
  const glm::vec3& axis) rotate axis

  returned scaling matrix

```
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));
```

Camera Matrix

(P_x, P_y, P_z) is the camera's position

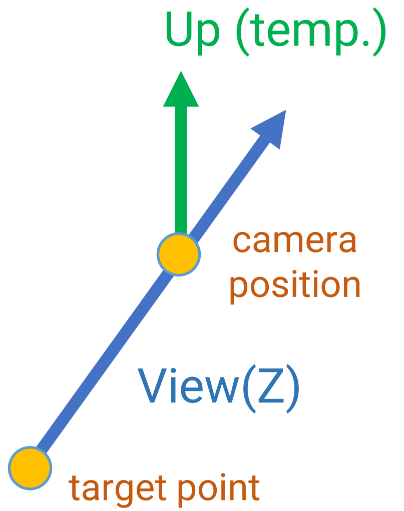
$$\begin{array}{l} \text{right vector} \\ \text{up vector} \\ \text{viewing vector} \\ \end{array} \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- glm::mat4x4 lookAt(const glm::vec3& eye,
  const glm::vec3& target,
  const glm::vec3& up)

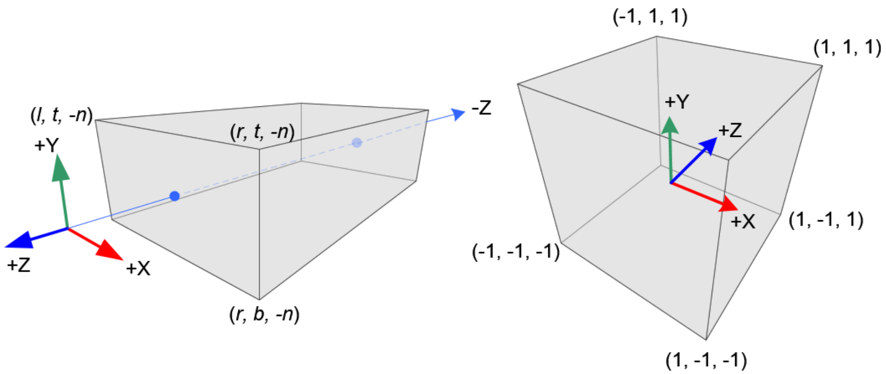
returned camera matrix

temporal up vector

```
glm::vec3 camPos = glm::vec3(3, 5, 10);
glm::vec3 target = glm::vec3(0, 1, 0);
glm::vec3 up = glm::vec3(0, 1, 0);
glm::mat4x4 gV = glm::lookAt(camPos, target, up);
```



Ortho Projection Matrix



$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- glm::mat4x4 ortho(const float left, const float right,
  const float bottom, const float bottom,
  const float near, const float far)

```
glm::mat4x4 goP = glm::ortho(-5.0f, 5.0f, -5.0f, 5.0f, 0.01f, 100.0f);
```

Perspective Projection Matrix

$$\begin{bmatrix} \frac{1}{ar\cdot\tan(\frac{\alpha}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\alpha}{2})} & 0 & 0 \\ 0 & 0 & \frac{-nearZ-farZ}{nearZ-farZ} & \frac{2\cdot farZ\cdot nearZ}{nearZ-farZ} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- glm::mat4x4 perspective(const float fovy ,
  const float aspectRatio ,
  const float near ,
  const float far)

use radian, not degree

```
float fovy = glm::radians(30.0f);
float aspectRatio = 640.0f / 360.0f;
float nearZ = 0.1f;
float farZ = 100.0f;
glm::mat4x4 gP = glm::perspective(fovy, aspectRatio, nearZ, farZ);
```

width / height

Apply the Transformation on CPU

- To transform a vertex from object space to clip space, we multiply its position with the **model-view-projection (MVP)** matrix
- We can pre-multiply part of the matrix if some of them are fixed
    - For example, we can pre-multiply the camera (view) and the projection matrix to form a VP matrix, and change the model matrix to perform object animation
- Remember to do the **perspective division**

Apply the Transformation on CPU (cont.)

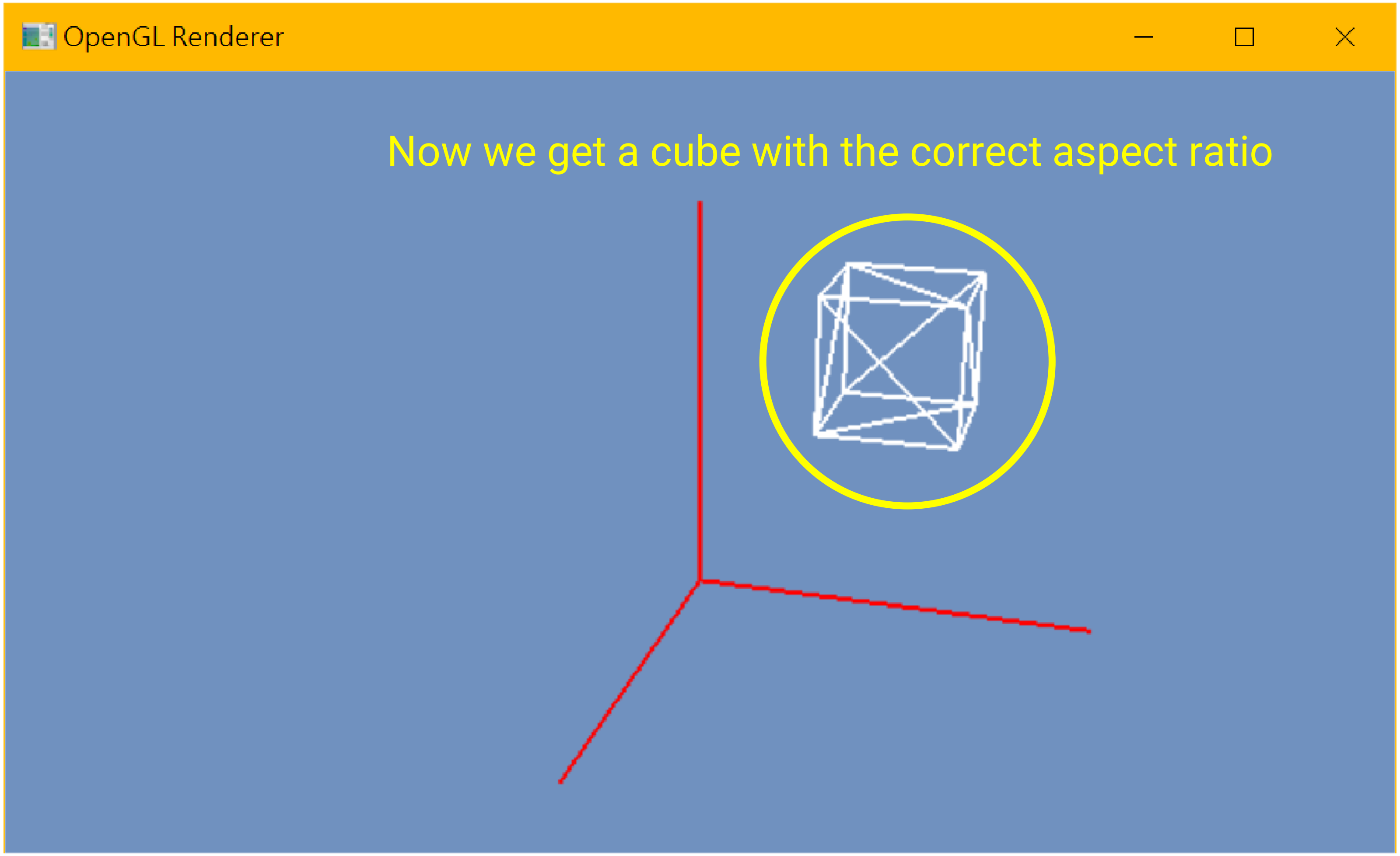
```
void ApplyTransformCPU(std::vector<glm::vec3>& vertexPositions, const glm::mat4x4& mvpMatrix)
{
    for (unsigned int i = 0 ; i < vertexPositions.size(); ++i) {
        glm::vec4 p = mvpMatrix * glm::vec4(vertexPositions[i], 1.0f);
        if (p.w != 0.0f) {
            float inv = 1.0f / p.w;
            vertexPositions[i].x = p.x * inv;
            vertexPositions[i].y = p.y * inv;
            vertexPositions[i].z = p.z * inv;
        }
    }
}
```

perspective division

- A useful coding technique available in shader programming
- It combines a 3d vector and a 1d scalar to form a 4d vector
- You can also write

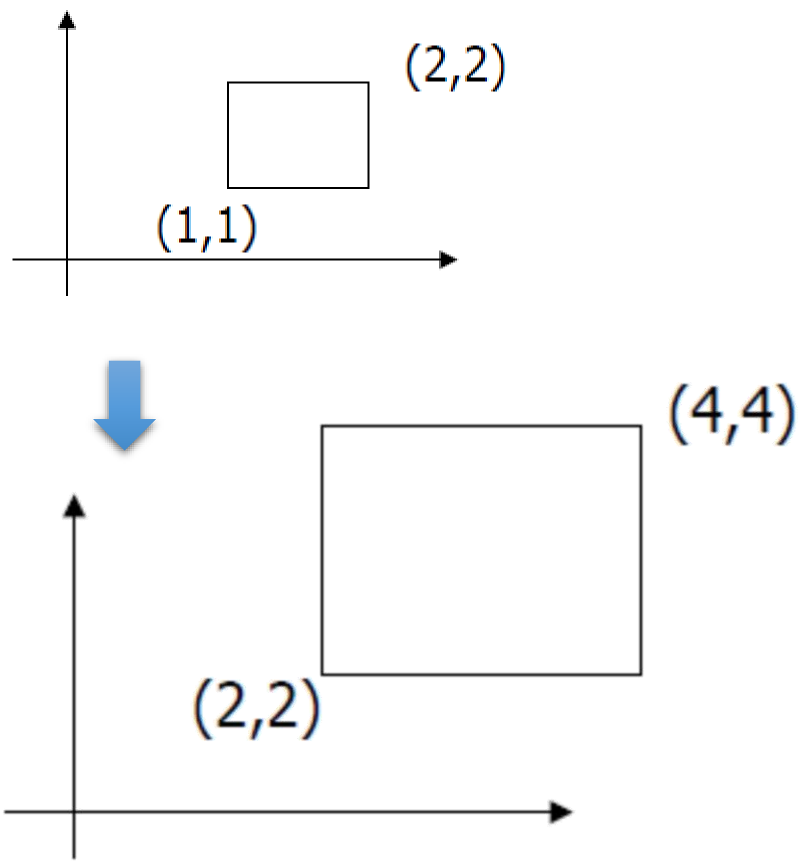
glm::vec4(vertexPositions[i].x,
vertexPositions[i].y,
vertexPositions[i].z)

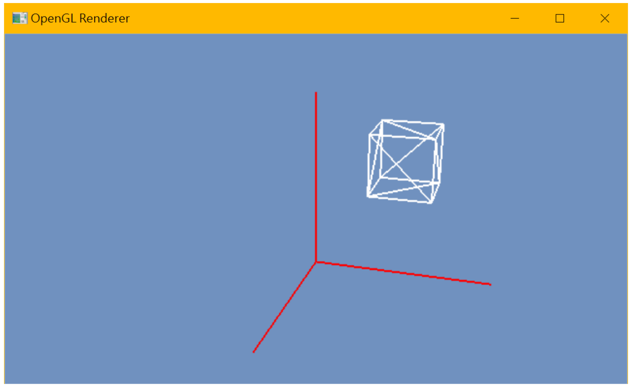
Apply the Transformation on CPU (cont.)



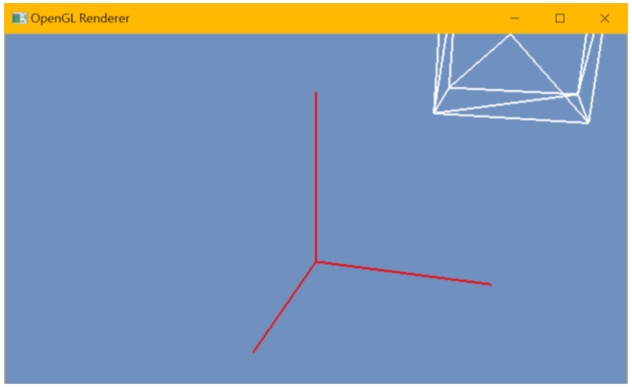
Example: 3D Scaling in Place

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





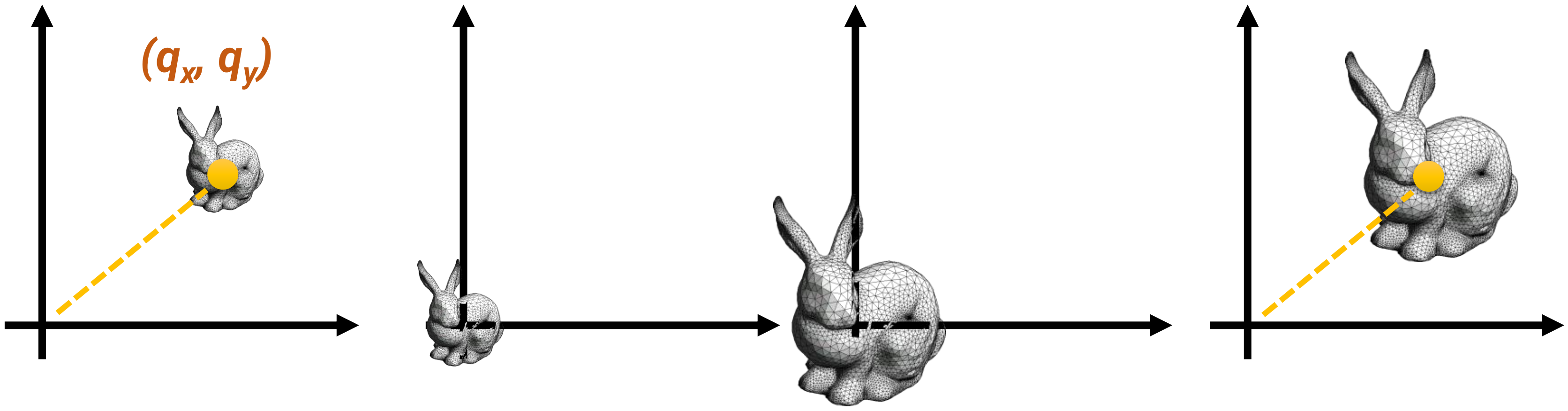
cube center at (1.5, 2.0, 0.0)



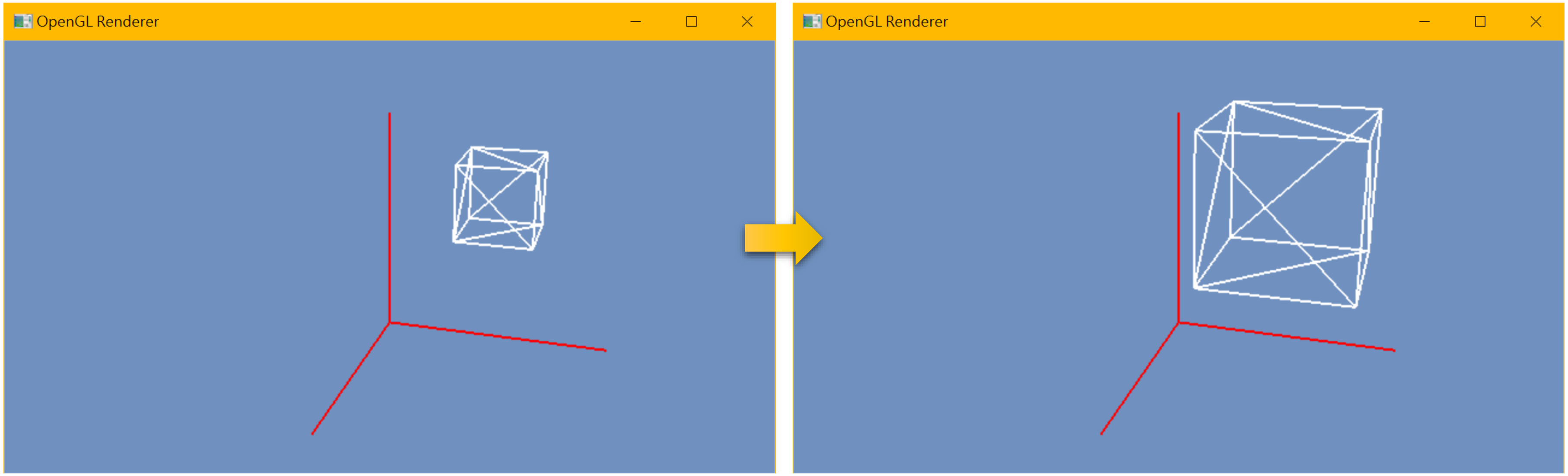
cube center at (3.0, 4.0, 0.0)

Example: 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 $T(q)S(s)T(-q)$



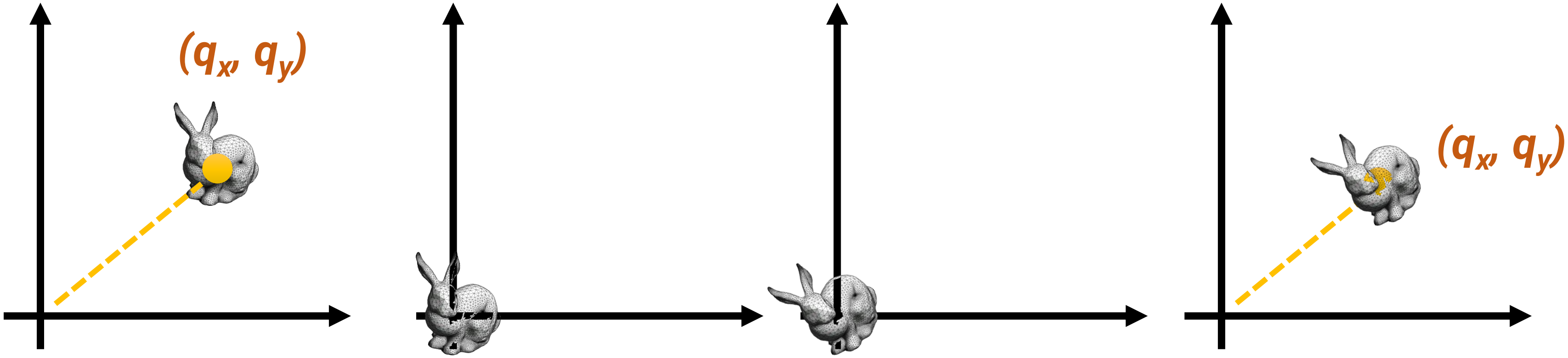
Example: 3D Scaling in Place (cont.)



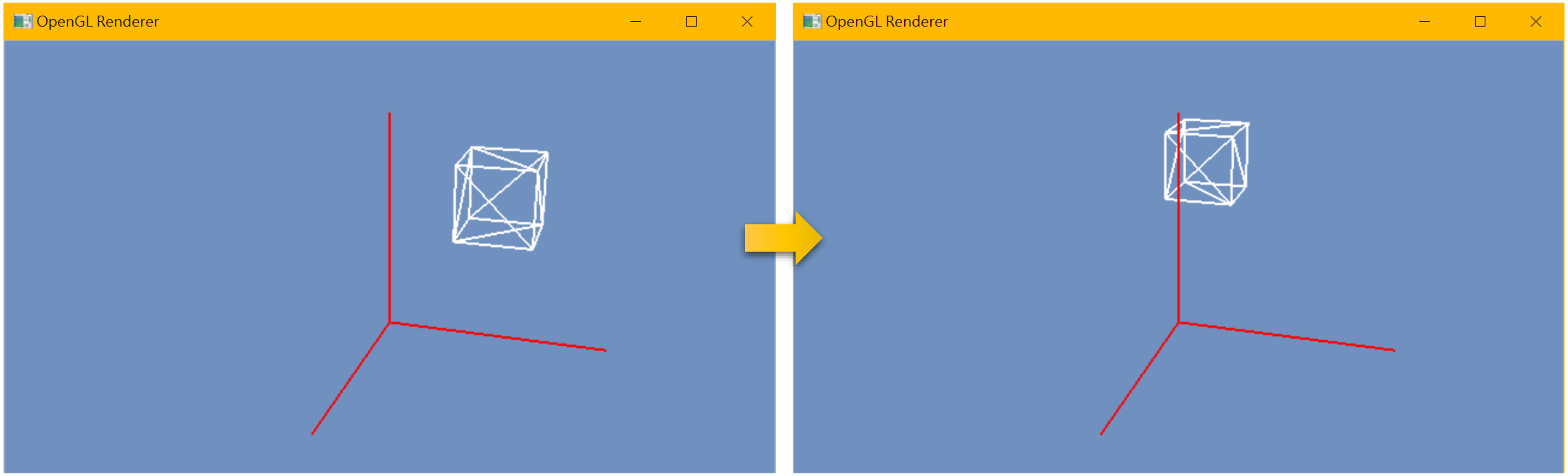

```
glm::mat4x4 T1 = glm::translate(glm::mat4x4(1.0f), glm::vec3(-1.5f, -2.0f, 0.0f));
glm::mat4x4 S = glm::scale(glm::mat4x4(1.0f), glm::vec3(2.0f, 2.0f, 2.0f));
glm::mat4x4 T2 = glm::translate(glm::mat4x4(1.0f), glm::vec3( 1.5f,  2.0f, 0.0f));
worldMatrix = T2 * S * T1;
```

Example: 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(θ)***
  - Translate the object back: ***T(q_x, q_y)***
- The final rotation matrix can be written as $\boxed{T(q)R(\theta)T(-q)}$



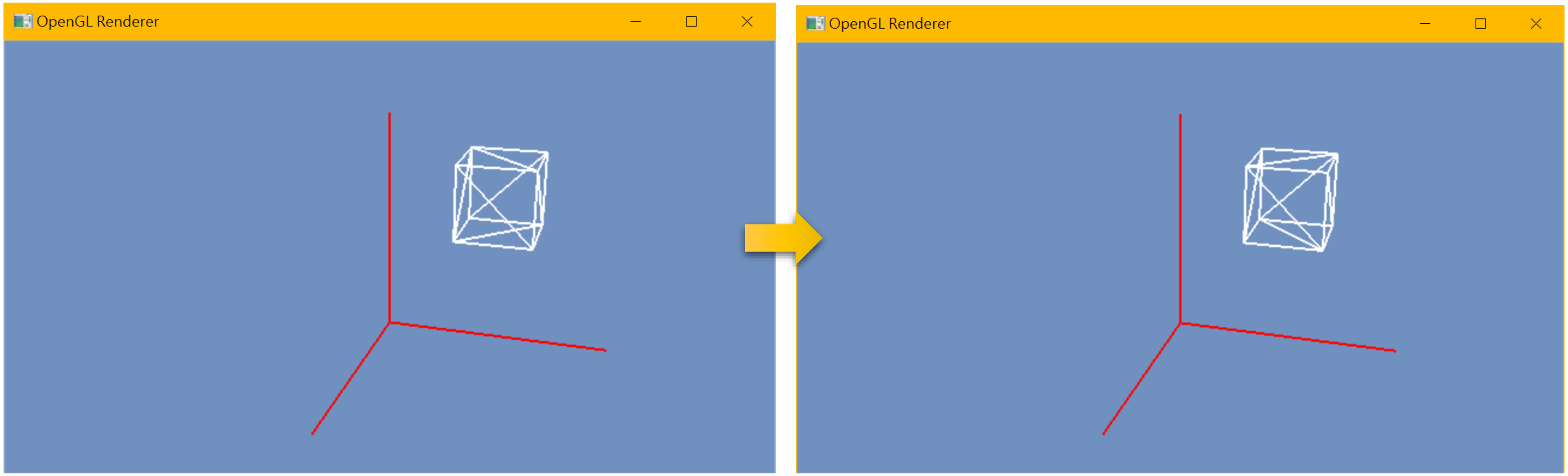
Example: 3D Rotating in Place (cont.)




```
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

Example: 3D Rotating in Place (cont.)

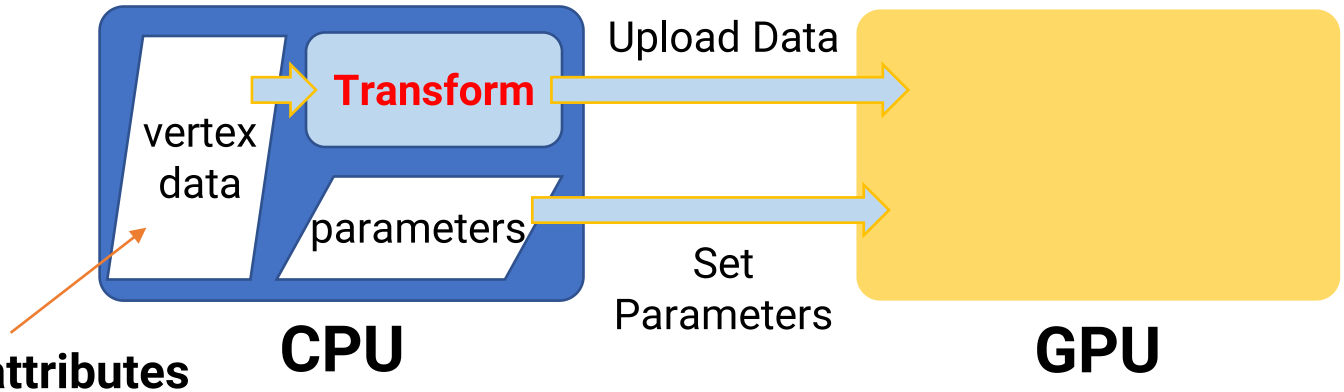



```
glm::mat4x4 T1 = glm::translate(glm::mat4x4(1.0f), glm::vec3(-1.5f, -2.0f, 0.0f));
glm::mat4x4 RY = glm::rotate(glm::mat4x4(1.0f), glm::radians(90.f), glm::vec3(0, 1, 0));
glm::mat4x4 T2 = glm::translate(glm::mat4x4(1.0f), glm::vec3( 1.5f,  2.0f, 0.0f));
worldMatrix = T2 * RY * T1;
```

rotate in place!

Apply the Transformation on CPU

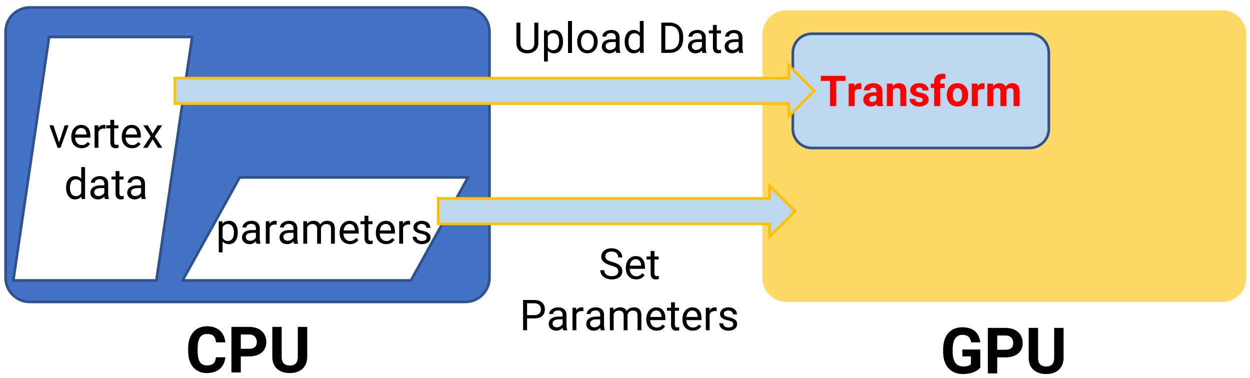
- So far we have performed the transformation of vertices on the CPU



Vertex attributes
Position
Normal
Texture coordinate

Apply the Transformation on CPU

- However, doing this job on CPU is not cost-effective
  - CPU is good at doing sequential, complex jobs
- In the next slides, we will introduce the **GPU graphics pipeline** and the **vertex shaders** for **parallel** processing



Any Questions?