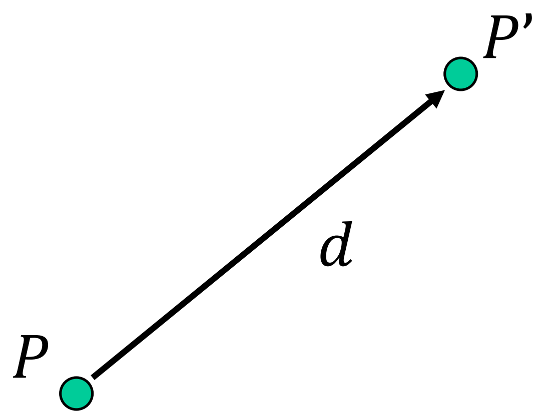
Transformations in OpenGL

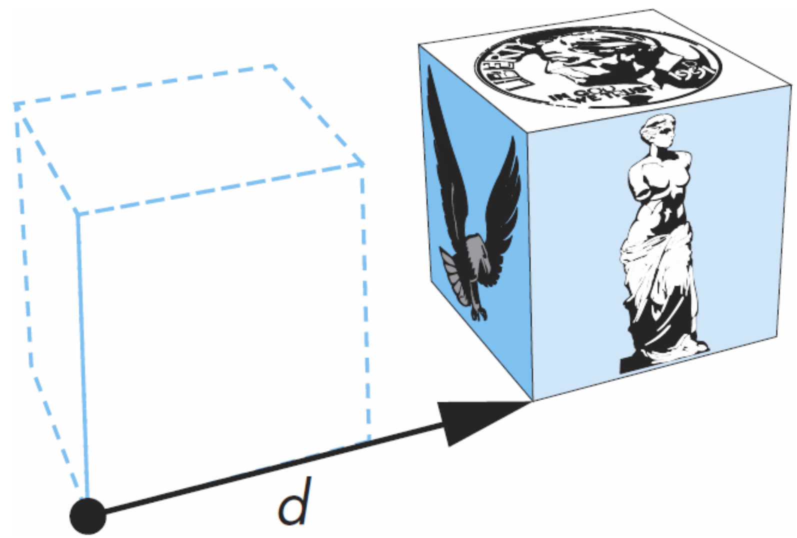
Computer Graphics
Instructor: Sungkil Lee

Standard (Affine) Transformations

Translation

- **Move (translate, displace) a point to a new location**





- **Displacement determined by a vector** d
  - 3 degrees of freedom
  - $P' = P + d$

Translation Matrix

- **We can also express translation using a 4×4 matrix T in homogeneous coordinates $p' = \mathrm{T}p$ where**

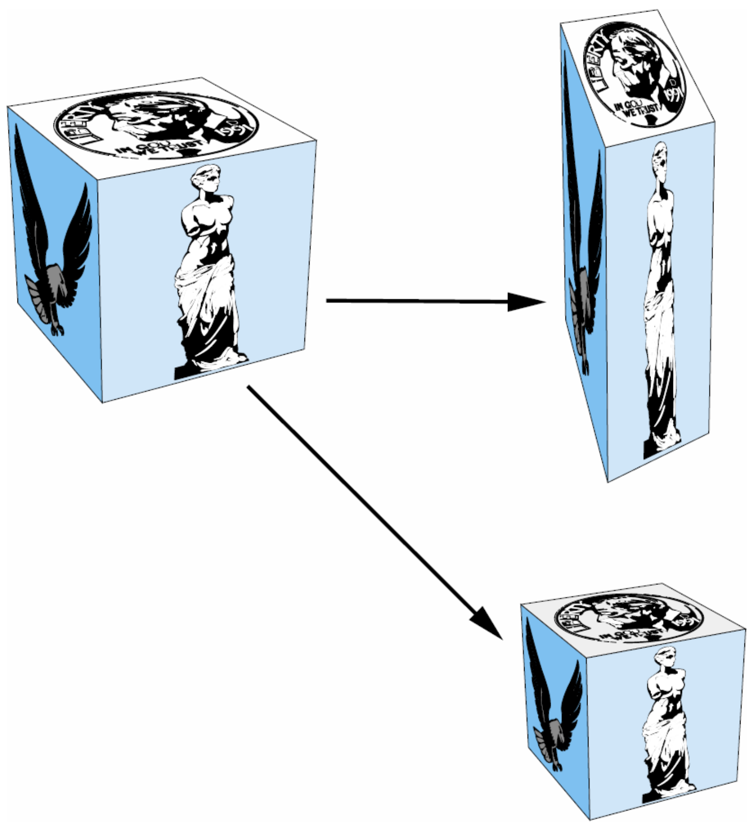
$$\boldsymbol{T} = \boldsymbol{T}(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **This form is better for implementation because**
  - all affine transformations can be expressed in this way (4×4 matrix)
  - and multiple transformations can be concatenated together by multiplying them together.

Scaling

- **Expand or contract along each axis (fixed point of origin)**

$$x' = s_x x \qquad y' = s_y y \qquad z' = s_z z$$



Scaling Matrix

- **In homogeneous coordinates,**

$$\boldsymbol{p}' = \mathbf{S}\boldsymbol{p}$$

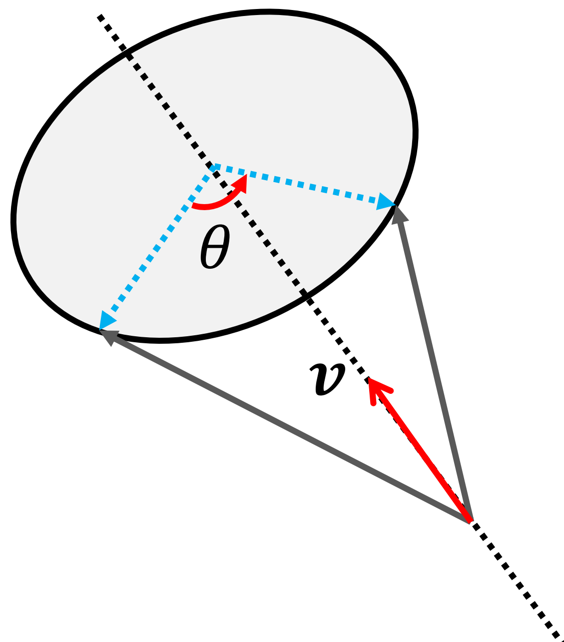
where

$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

- **Generally, rotation transformation can be described by the rotation angle θ and its revolution axis v.**

$$\boldsymbol{p}' = \boldsymbol{R}(\theta)\boldsymbol{p}$$



Rotation Matrix

- **Rotation matrix $R(\theta)$ revolving axis v is given as:**

$$\begin{bmatrix} v_x v_x(1-c)+c & v_x v_y(1-c)-v_z s & v_x v_z(1-c)+v_y s & 0 \\ v_x v_y(1-c)+v_z s & v_y v_y(1-c)+c & v_y v_z(1-c)-v_x s & 0 \\ v_x v_z(1-c)-v_y s & v_y v_z(1-c)+v_x s & v_z v_z(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where $\mathrm{c} = \cos\theta,\ s = \sin\theta$

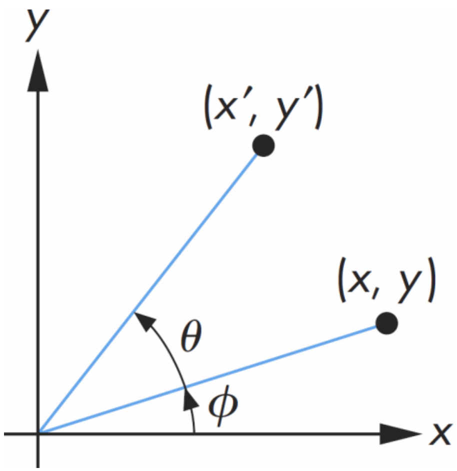
- This formulation is dervied using Quaternion (an extension of complex numbers with three imaginary numbers).
- Though, we do not prove this, because a rigorous proof for this goes far beyond the undergraduate level.

Rotation Matrix

- **Rotation about z axis in 3D**
  - With z-axis ($v_x = 0, v_y = 0, v_z = 1$), the formulation is reduced to the well-known form.
  - equivalent to 2-D rotation in planes of constant z, like slicing 3D into multiple plane slices at height z and rotating in each such plane.

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

$$\boxed{\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned}}$$



Rotation Matrix

- **Similarly, rotation matrix along x- and y-axes are:**

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

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

Standard 2D Transformation Matrices

2D Transformation in 4x4 Matrix

- **Use 4x4 matrices instead of 2x2 or 3x3 matrices**
  - Graphics pipeline is optimized for 4x4 matrix, and thus, it is better to use 4x4 matrices even for 2D transformation.
  - It is consistent, when mixing with 3D transformations.

- **It is trivial to derive 2D transformations from 3D transformations.**
  - 2D translation: $\boldsymbol{T} = \boldsymbol{T}(d_x, d_y, 0)$
  - 2D Scaling: $\mathbf{S} = \mathbf{S}(s_x, s_y, 1)$
  - 2D rotation (with z-axis): $\mathbf{R} = \mathbf{R}_z(\theta)$

Implementations

Matrix Library for transformation

- **Methods to apply 3D transformation**
  - Static methods immediately return the corresponding matrix.
  - Instance methods set its internals to the corresponding matrix and return itself.

Matrix Library for transformation

- **Add these methods to mat4 class in your "cgmath.h."**
  - Implement member functions as taught in the theory lecture.

```
struct mat4
{
    ...
    static mat4 translate( const vec3& v ){ return mat4().setTranslate(v); }
    static mat4 translate( float x, float y, float z ){ return mat4().setTranslate(x,y,z); }
    static mat4 scale( const vec3& v ){ return mat4().setScale(v); }
    static mat4 scale( float x, float y, float z ){ return mat4().setScale(x,y,z); }
    static mat4 rotateX( float theta ){ return mat4().setRotateX(theta); }
    static mat4 rotateY( float theta ){ return mat4().setRotateY(theta); }
    static mat4 rotateZ( float theta ){ return mat4().setRotateZ(theta); }
    static mat4 rotate( const vec3& axis, float angle ){ return mat4().setRotate(axis,angle); }

    inline mat4& set_translate( const vec3& v ){ ... }
    inline mat4& set_translate( float x, float y, float z ){ ... }
    inline mat4& set_scale( const vec3& v ){ ... }
    inline mat4& set_scale( float x, float y, float z ){ ... }
    inline mat4& set_rotateX( float theta ){ ... }
    inline mat4& set_rotateY( float theta ){ ... }
    inline mat4& set_rotateZ( float theta ){ ... }
    inline mat4& set_rotate( const vec3& axis, float angle ){ ... }
    …
};
```

Examples

- **set_translate()**

```
inline mat4& set_translate( const vec3& v)
{
   set_identity();          // reset to identity matrix
   _14=v.x; _24=v.y; _34=v.z;
   return *this;            // return itself.
}

inline mat4& set_translate( float x, float y, float z )
{
   set_identity();
   _14=x; _24=y; _34=z;
   return *this;
}
```

Examples

- **translate()**

```
static mat4 translate( const vec3& v )
{
    return mat4().set_translate(v);
}

static mat4 translate( float x, float y, float z )
{
    return mat4().set_translate(x,y,z);
}
```

Setting a Transformation

- **Calculate the desired transformation.**

```
void render()
{
    ...

    // configure transformation parameters
    float t = float(glfwGetTime());
    float theta= t*((k%2)-0.5f)*float(k+1)*0.5f;
    float move= ((k%2)-0.5f)*200.0f*float((k+1)/2);

    // build the model matrix
    mat4 model_matrix =
        mat4::translate( move, 0.0f, -abs(move) ) *
        mat4::translate( cam.at ) *
        mat4::rotate( vec3(0,1,0), theta ) *
        mat4::translate( -cam.at );
    ...
}
```

Updating Uniform Variables

- **Connecting the matrices to the uniform variables**
  - Provide GL_TRUE for the third parameter to glUniformMatrix4fv().
  - This will be explained in the last pages.

```
void render( )
{
   ...

    // update the uniform model matrix and render
    GLint uloc = glGetUniformLocation( program, "model_matrix" );
    if(uloc>-1) glUniformMatrix4fv( uloc, 1, GL_TRUE, model_matrix );
}
```

Vertex Shader Example

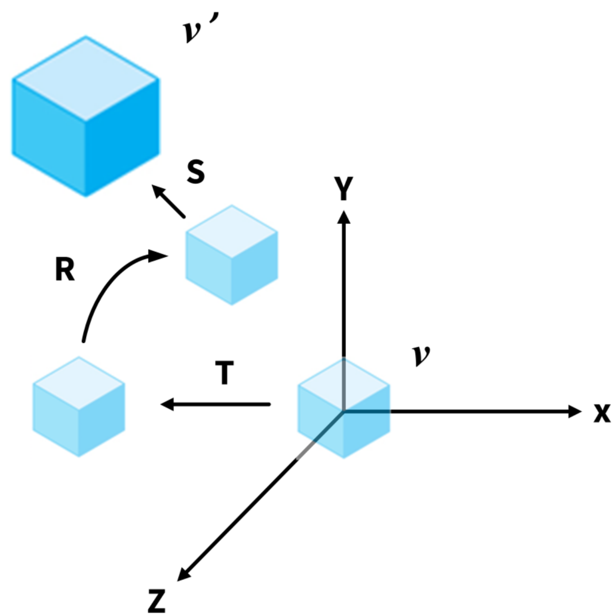
- **Multiply with position vector**

```
// vertex attributes
layout(location=0) in vec3 position;
layout(location=1) in vec3 normal;
layout(location=2) in vec2 texcoord;

// vertex shader output
out vec3 norm;

// matrices
uniform mat4 model_matrix, view_matrix, projection_matrix;

void main()
{
    // transform the vertex position by model matrix.
    vec4 wpos = model_matrix * vec4(position, 1.0);
    // transform the position to the eye-space position (taught later)
    vec4 epos = view_matrix * wpos;
    // project the eye-space position to the canonical view volume (taught later)
    gl_Position = projection_matrix * epos;
    // pass normal vector to fragment shader
    norm = normal;
}
```

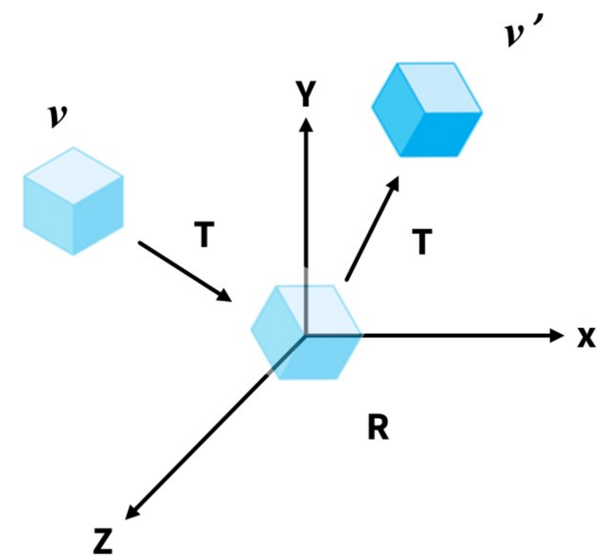
More Examples on Transformation

- **Although you build various combinations of matrices, you can use the same vertex shader.**

$$v' = \mathrm{SRT}\, v \qquad\qquad v' = \mathrm{TRT}\, v$$



+



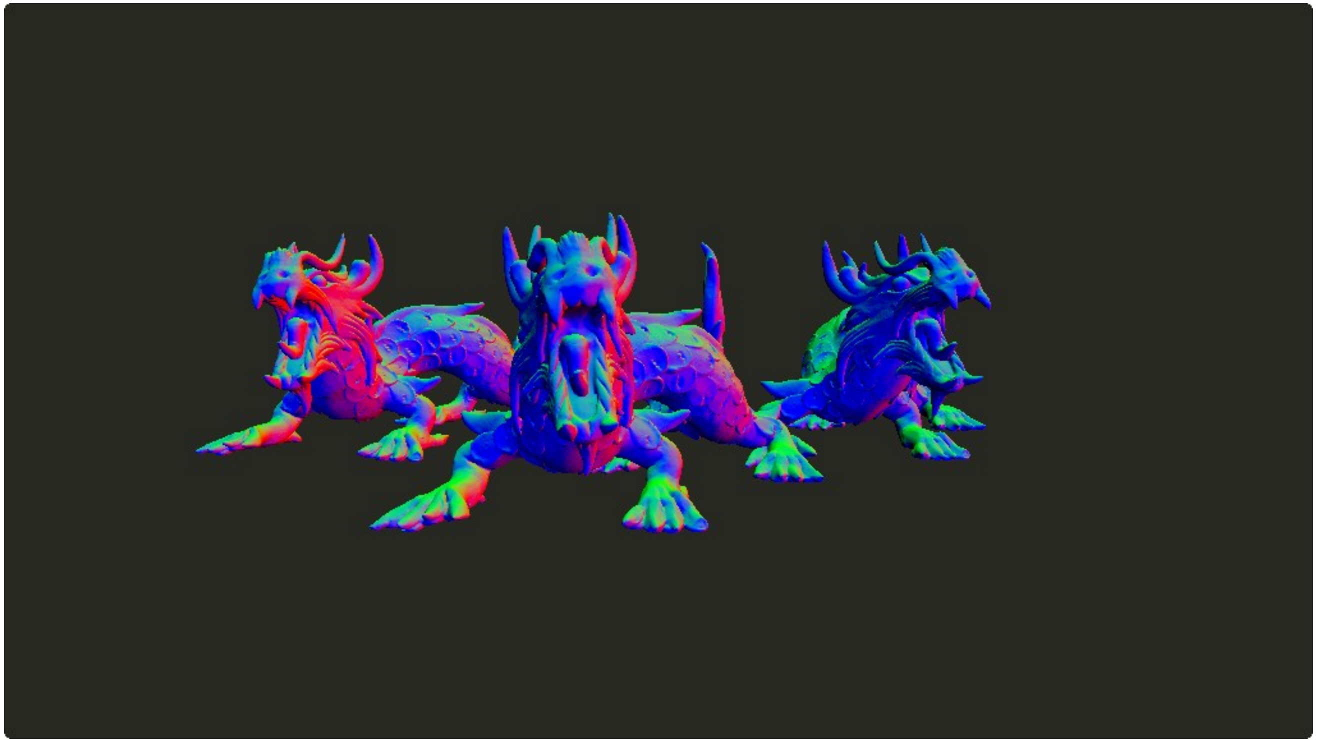
Instancing

- **Instance transformation**
  - We can render many (identical) objects using one geometry definition.
  - Call draw function multiple times.

```
void render( )
{
    ...

    // render the same object for n-times.
    for( int k=0; k < NUM_INSTANCE; k++)
    {
        // configure transformation parameters for object k ...
        // build the model matrix for object k ...

        // update the uniform model matrix and render
        GLint uloc = glGetUniformLocation( program, "model_matrix" );
        glUniformMatrix4fv( uloc, 1, GL_TRUE, model_matrix );
        glDrawElements( GL_TRIANGLES, pMesh->indexList.size(), GL_UNSIGNED_INT, nullptr );
    }
}
```

Instancing Example

- **Three dragons**
  - Call draw function three times using the same dragon model but with different transformations.



More on OpenGL Matrix

Modern-Style OpenGL Matrices

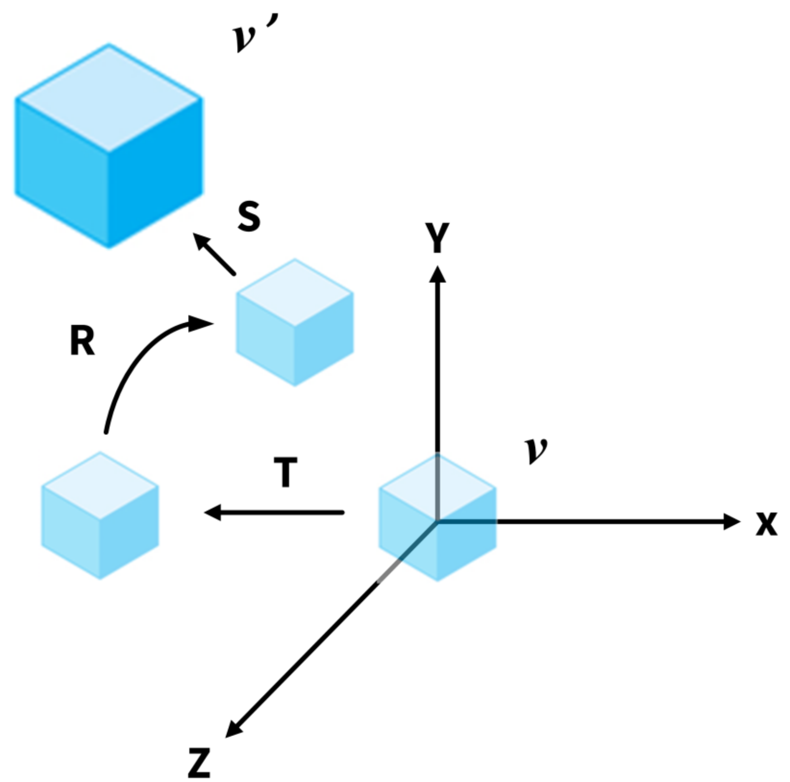
- **A user can specify any matrices, yet a user also need to form desired matrices by hand.**
  - Matrices are stored as one dimensional array of 16 elements which are the components of the 4 x 4 matrix.

- **Usually, we maintain transformation matrices in application program, and pass them to shader programs via uniform variables.**
  - Nothing special in forming transformation matrices.
  - Just use the matrix definitions covered before.
  - But, pay attention in uploading matrices to uniform variables.

Row-Major or Column-Major?

- **OpenGL matrix multiplication uses *row-major* order as usual.**
  - e.g., If we apply $\mathbf{T}, \mathbf{R}$, and $\boldsymbol{S}$ onto $\boldsymbol{v}$ sequentially, operations would be like:

$$\boldsymbol{v}' = \mathbf{SRT}\,\boldsymbol{v}$$



Row-Major or Column-Major?

- **However, the internal memory layout of OpenGL matrices is column-major.**
  - Memory layout: Given C array $a[0..15]$, OpenGL will store it as:

$$\begin{bmatrix} a[0] & a[1] & a[2] & a[3] \\ a[4] & a[5] & a[6] & a[7] \\ a[8] & a[9] & a[10] & a[11] \\ a[12] & a[13] & a[14] & a[15] \end{bmatrix}$$

We use row-major ordering in CPU.

$$\begin{bmatrix} a[0] & a[4] & a[8] & a[12] \\ a[1] & a[5] & a[9] & a[13] \\ a[2] & a[6] & a[10] & a[14] \\ a[3] & a[7] & a[11] & a[15] \end{bmatrix}$$

However, the internal memory of OpenGL is laid out as column-major.

  - If you want to pass a row-major matrix to GLSL, you need to transpose the matrix beforehand.

Row-Major or Column-Major?

- **Rule in this course:**

Apply only row-major multiplications, yet pass matrices to GLSL with transposition.

- **Transpose only when uploading uniform**
  - OpenGL functions that have matrices as parameters allow the application to send the matrix or its transpose.

```
void glUniformMatrix4fv( GLint location, GLsizei count,
     GLboolean transpose, const GLfloat* value );
```

  - Use `GL_TRUE` for ***`transpose`*** to send the transpose.

Comparison with DirectX

- **How to do in DirectX**
  - The matrix operator uses ***column-major*** operators, and the memory layout also uses ***column-major*** order as follows:

$$\begin{bmatrix} a[0] & a[1] & a[2] & a[3] \\ a[4] & a[5] & a[6] & a[7] \\ a[8] & a[9] & a[10] & a[11] \\ a[12] & a[13] & a[14] & a[15] \end{bmatrix}$$

  - DirectX is more consistent in comparison to OpenGL;
    you can also use row-major multiplication for row-major memory layout.