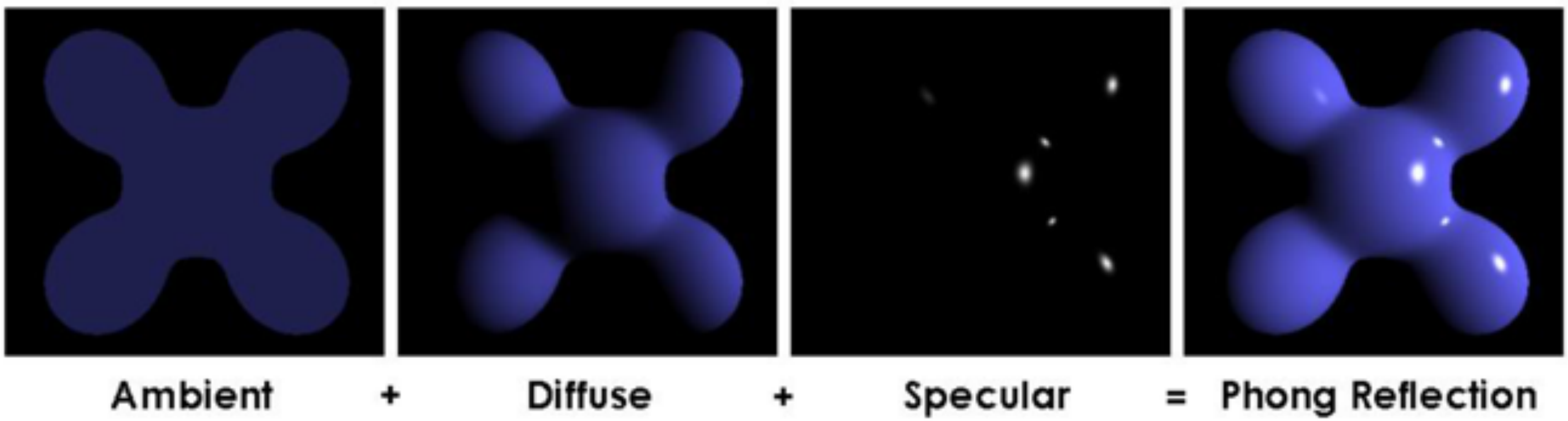
ICG 2016 Fall Homework 1 Guide

劉志豪

Requirements

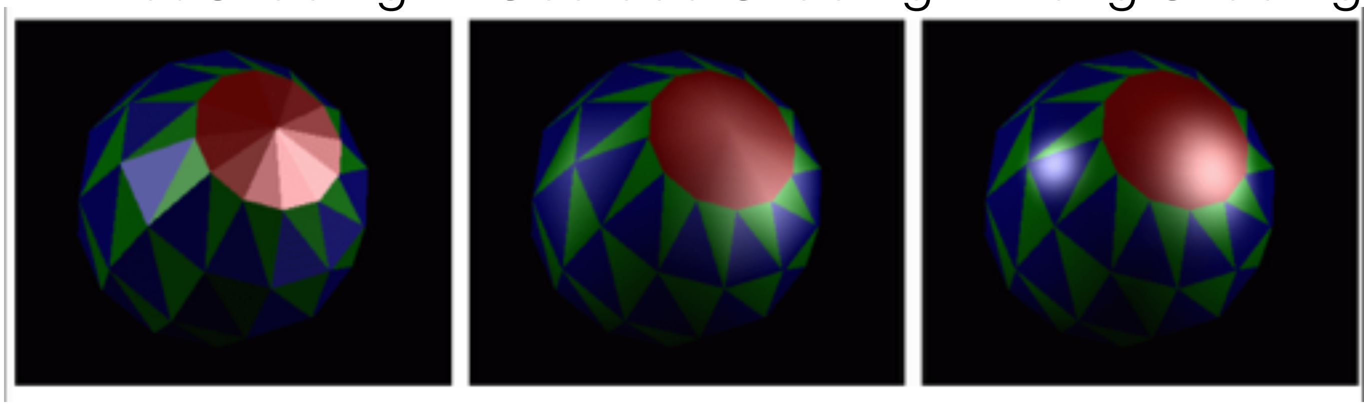
- Flat, Gouraud, and Phong shading with Phong illumination model in shaders. You can demonstrate the three shading computation in a single object. (3pts)
- Enable multiple shaders and transformation on multiple objects in a scene. You are free to use those provided model files and arrange them to form the scene on your own style. You **must show the three shading simultaneously** on different objects in your scene. (3pts)
- Bonus: Special effects on animation

Phong Illumination Model



Shading

Flat Shading Gouraud Shading Phong Shading



Do not confuse phong shading with phong illumination(refection) model
(Flat shading + phong illumination is OK.)

Shading

- Flat Shading: Constant normal on the whole surface
- Gouraud Shading: Different vertex normal, interpolated vertex color on a fragment
- Phong Shading: Different vertex normal, interpolated vertex normal on a fragment

Vertex Shader

```
//specify GLSL version
#version 330 core
// Get vertex data from the VBO according to the vertex attribute id. The vertex data will stored in declared variable "vertex_position"
layout(location = 0) in vec3 vertex_position;

void main(){
// gl_Position is a built-in variable in GLSL, which is an output variable of the vertex shader
gl_Position = vec4(vertex_position, 1.0);

}
```

note: vertex shader "must" output vertex position (in clipping coordinate space) to let OpenGL system perform scan conversion

Fragment Shader

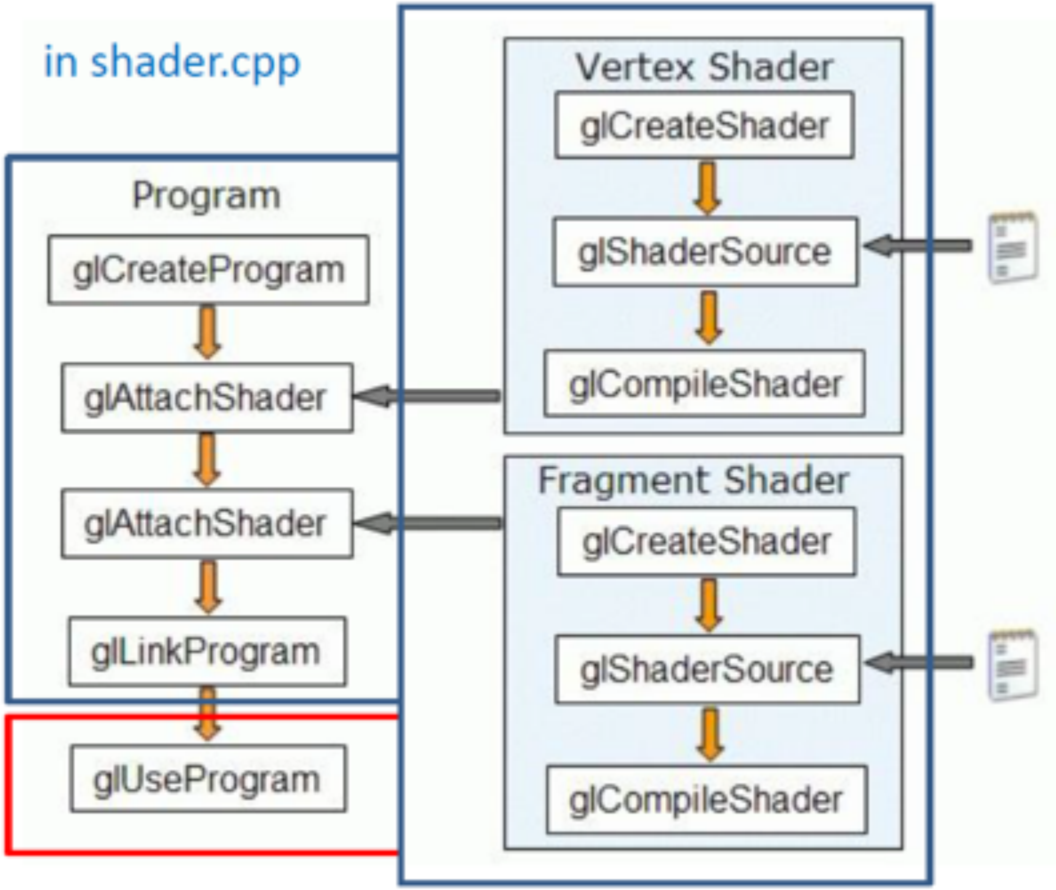
```
#version 330
//declare an output variable “color” to the image
out vec3 color;
void main()
{
// output red color for each segment
color = vec3(1,0,0);
}
```

Note:

In fragment shader, it receives a fragment in a “triangle” when vertex shader finish processing three vertices (remember GL_TRIANGLE in client code?)

The fragment is already in window coordinate here. We can derive the coordinate from the built-in variable for other application

Bind shader program to OpenGL



(You must call glUseProgram each time before you define the vertex attribute array)

RENDER LOOP(CLIENT)

```
while (running):
        initialize window
        load shader program
        clear frame buffer
        Update transformation
        Update Objects
        Draw Object
        SwapBuffers
```

a linear array on GPU memory

SHADER DATA

“Per-object constant”

<u>Uniform</u>

= Shared Constant

<u>Vertex Data</u>

= ANYTHING YOU WANT!

<u>Example</u>?

Positions…

Normals…

Colors…

Texture Coordinates…

(from "progressive openGL" slides, 2012)

SHADER DATA AND GLSL

OpenGL 4.2 Reference card: **Blue means deprecated**

www.khronos.org/files/**opengl**42-quick-**reference**-card.pdf

Qualifiers

Storage Qualifiers [4.3]

Declarations may have one storage qualifier.

none	(default) local read/write memory, or input parameter
const	global compile-time constant, or read-only function parameter, or read-only local variable
in	linkage into shader from previous stage
out	linkage out of a shader to next stage
attribute	same as **in** for vertex shader
uniform	linkage between a shader, OpenGL, and the application
varying	same as **in** for vertex shader, same as **out** for fragment shader

"Per-object constant"

http://en.wikibooks.org/wiki/GLSL_Programming/Rasterization

(from "progressive openGL" slides, 2012)

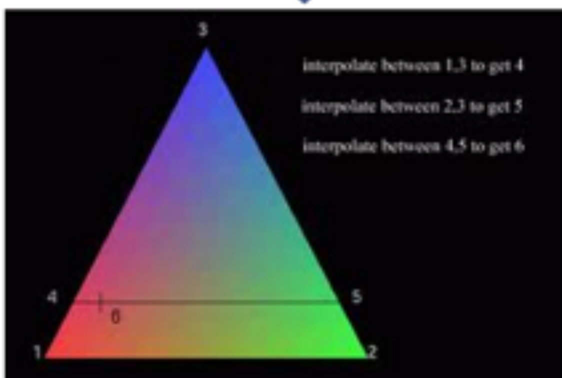
IN VS. OUT

GPU Memory

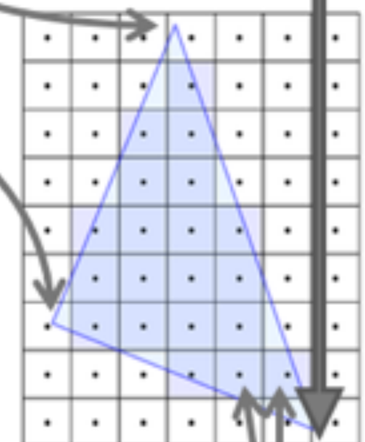
`in` = Data from **Vertex Buffer**

`out` = Rasterizer `in`

Vertex Shader



Rasterizer



Fragment Shader

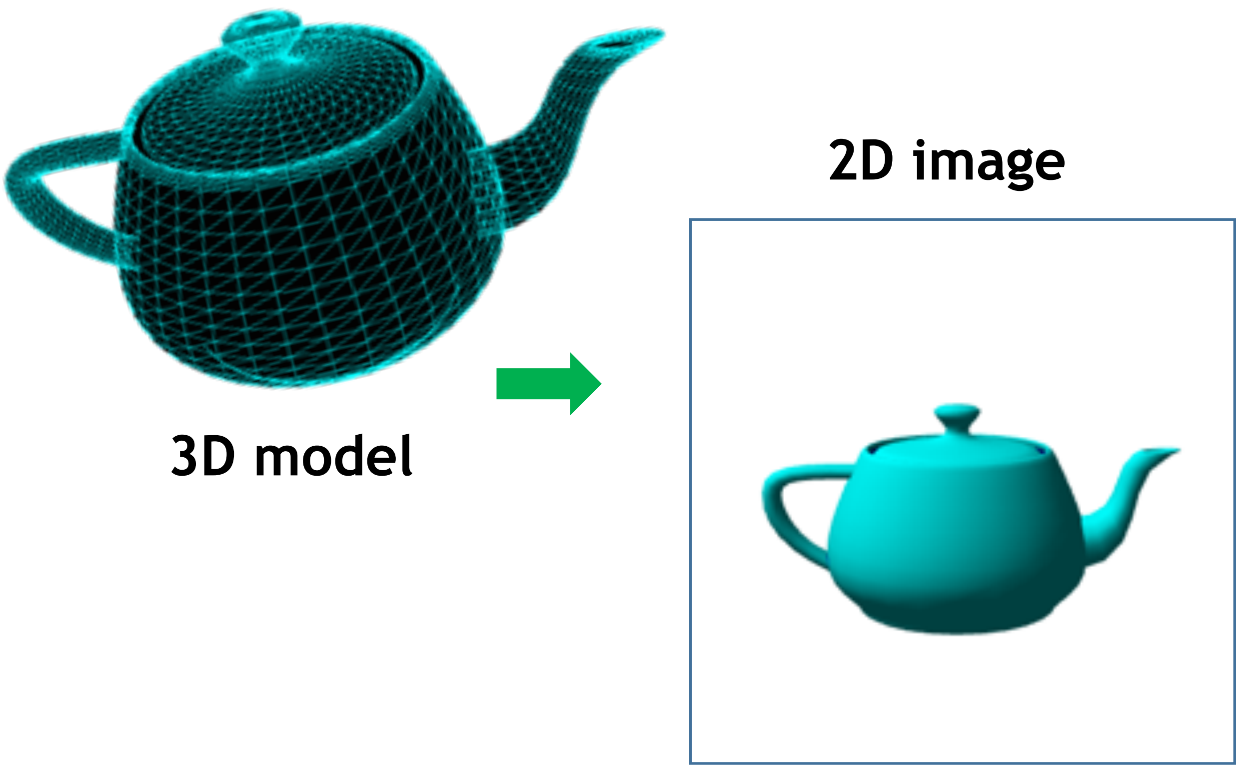
`in` = Rasterizer `out`

`out` = **ANYTHING YOU WANT!**
… but usually pixel **color** and **depth**

Fragment ≈ Pixel

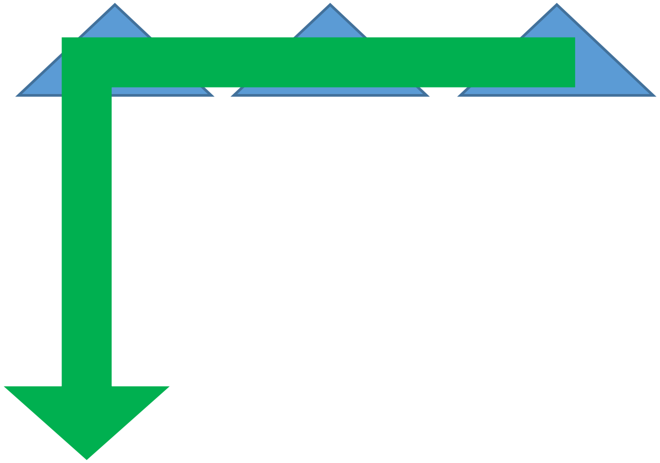
http://en.wikibooks.org/wiki/GLSL_Programming/Rasterization

Rasterization



Rasterization

- transformation
- clipping
- scan conversion

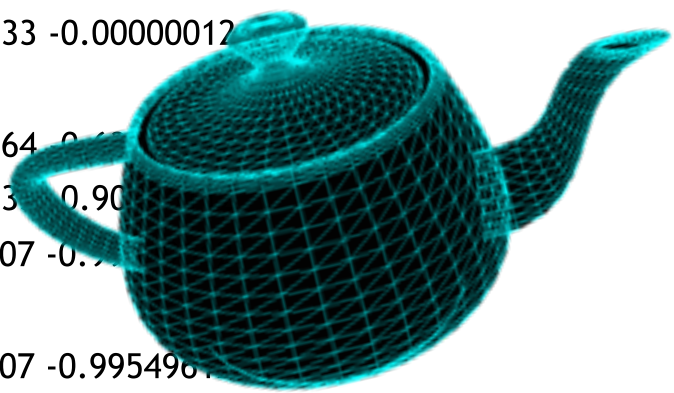


The pipeline is suited for hardware acceleration

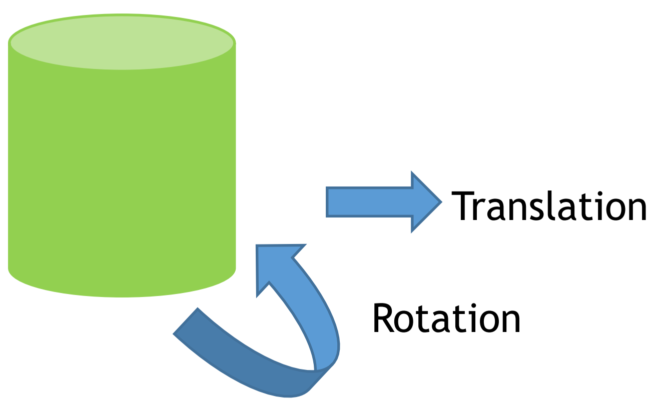
loading model file

- Triangle
- 126.000000 202.500000 0.000000 -0.902861 -0.429933 -0.00000012
- 89.459999 202.500000 89.459999 -0.637936 -0.431364 -0.63793612
- 88.211967 209.144516 88.211967 -0.703896 0.095197 -0.70389512
- Triangle
- 88.211967 209.144516 88.211967 -0.703896 0.095197 -0.70389512
- 124.242210 209.144516 0.000000 -0.995496 0.094807 -0.00000012
- 126.000000 202.500000 0.000000 -0.902861 -0.429933 -0.00000012
- Triangle
- 89.459999 202.500000 89.459999 -0.637936 -0.431364
- 0.000000 202.500000 126.000000 -0.000000 -0.42993
- 0.000000 209.144516 124.242210 -0.000000 0.094807
- Triangle
- 0.000000 209.144516 124.242210 -0.000000 0.094807 -0.99549
- 88.211967 209.144516 88.211967 -0.703895 0.095197 -0.70389612
- 89.459999 202.500000 89.459999 -0.637936 -0.431364 -0.63793612

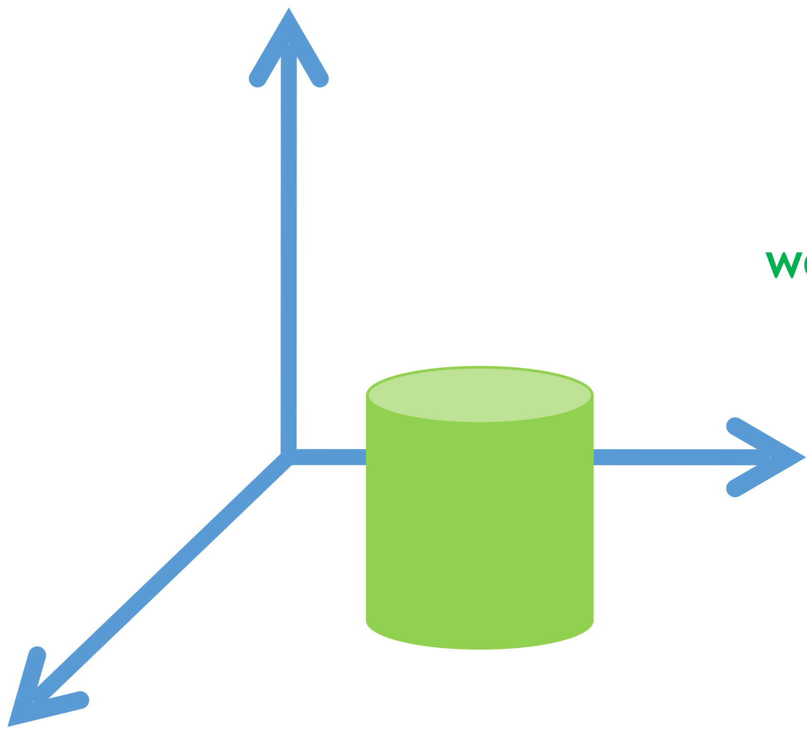
Coordinate
Normal orientation



Move and rotate object



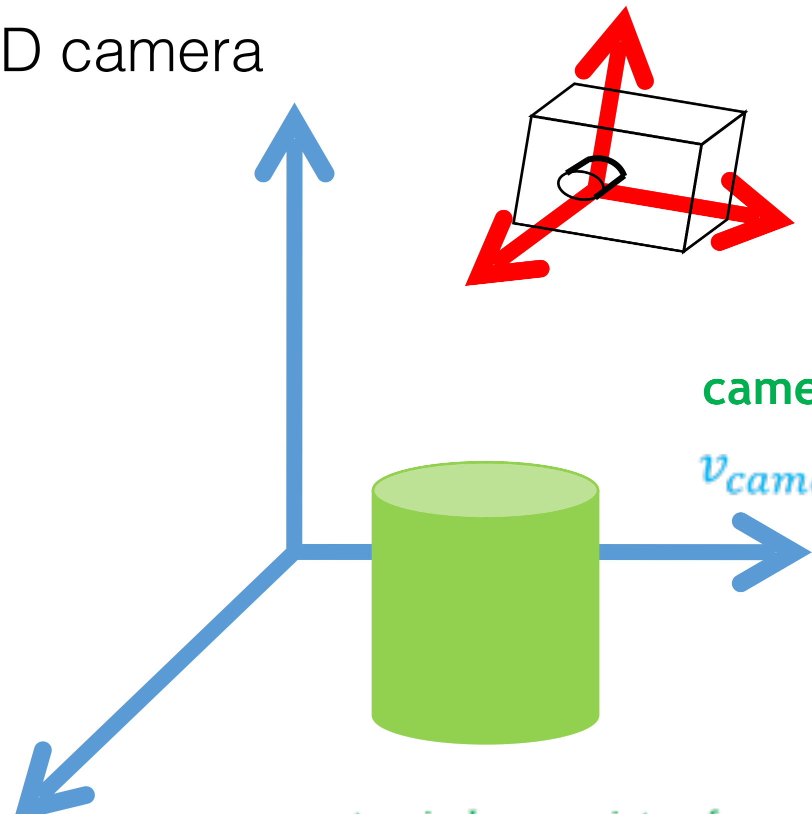
$$R \cdot T \cdot v_{object}$$



world coordinate

$$v_{world} = R \cdot T \cdot v_{object}$$

ADD camera



camera coordinate

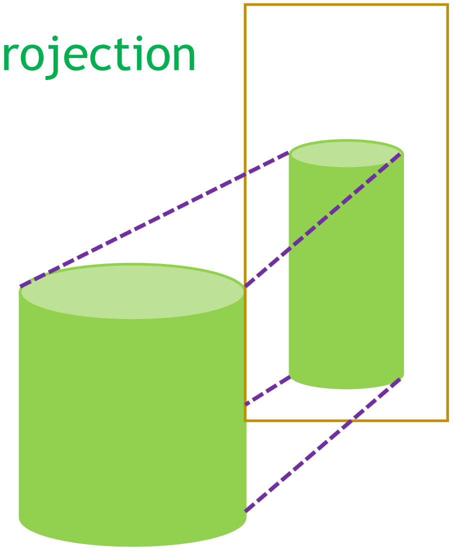
$$v_{camera} = L \cdot v_{world}$$

camera parameters in L: eye point, reference point, up vector

Now we enter Clipping space

Clipping space denotes a viewing volume in 3D space.

The viewing volume is related to projection model.



Homogeneous coordinates

Until then, we only considered 3D vertices as a (x,y,z) triplet. Let's introduce w. We will now have (x,y,z,w) vectors.

This will be more clear soon, but for now, just remember this :

- If w == 1, then the vector (x,y,z,1) is a position in space.
- If w == 0, then the vector (x,y,z,0) is a direction.

(In fact, remember this forever.)

What difference does this make ? Well, for a rotation, it doesn't change anything. When you rotate a point or a direction, you get the same result. However, for a translation (when you move the point in a certain direction), things are different. What could mean "translate a direction" ? Not much.

Homogeneous coordinates allow us to use a single mathematical formula to deal with these two cases.

Transformation matrices

Matrix x Vertex (in this order !!) = TransformedVertex

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix}$$

In C++, with GLM:

```
glm::mat4 myMatrix;
glm::vec4 myVector;
// fill myMatrix and myVector somehow
glm::vec4 transformedVector = myMatrix * myVector; // Again, in this order ! this is important.
```

In GLSL :

```
mat4 myMatrix;
vec4 myVector;
// fill myMatrix and myVector somehow
vec4 transformedVector = myMatrix * myVector; // Yeah, it's pretty much the same than GLM
```

Translation matrices

These are the most simple tranformation matrices to understand. A translation matrix look like this :

$$\begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where X,Y,Z are the values that you want to add to your position.

So if we want to translate the vector (10,10,10,1) of 10 units in the X direction, we get :

$$\begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 10 \\ 10 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*10+0*10+0*10+10*1 \\ 0*10+1*10+0*10+0*1 \\ 0*10+0*10+1*10+0*1 \\ 0*10+0*10+0*10+1*1 \end{bmatrix} = \begin{bmatrix} 10+0+0+10 \\ 0+10+0+0 \\ 0+0+10+0 \\ 0+0+0+1 \end{bmatrix} = \begin{bmatrix} 20 \\ 10 \\ 10 \\ 1 \end{bmatrix}$$

Scaling matrices

Scaling matrices are quite easy too :

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So if you want to scale a vector (position or direction, it doesn't matter) by 2.0 in all directions :

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} 2*x+0*y+0*z+0*w \\ 0*x+2*y+0*z+0*w \\ 0*x+0*y+2*z+0*w \\ 0*x+0*y+0*z+1*w \end{bmatrix} = \begin{bmatrix} 2*x+0+0+0 \\ 0+2*y+0+0 \\ 0+0+2*z+0 \\ 0+0+0+1*w \end{bmatrix} = \begin{bmatrix} 2*x \\ 2*y \\ 2*z \\ w \end{bmatrix}$$

and the w still didn't change. You may ask : what is the meaning of "scaling a direction" ? Well, often, not much, so you usually don't do such a thing, but in some (rare) cases it can be handy.

(notice that the identity matrix is only a special case of scaling matrices, with (X,Y,Z) = (1,1,1). It's also a special case of translation matrix with (X,Y,Z)=(0,0,0), by the way)

In C++ :

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>
glm::mat4 myScalingMatrix = glm::scale(2.0f, 2.0f ,2.0f);
```

Rotation matrices

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

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

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

In C++ :

```
// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>
glm::vec3 myRotationAxis( ??, ??, ??);
glm::rotate( angle_in_degrees, myRotationAxis );
```

Rotation matrices

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

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

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

In C++ :

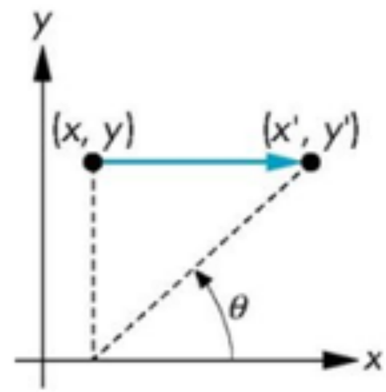
```
// Use #include <glm/gtc/matrix_transform.hpp> and #include <glm/gtx/transform.hpp>
glm::vec3 myRotationAxis( ??, ??, ??);
glm::rotate( angle_in_degrees, myRotationAxis );
```

Shear matrices

Consider simple shear along x axis

$$x' = x + y \cot\theta$$
$$y' = y$$
$$z' = z$$

$$\mathbf{H}(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Reference

- http://www.opengl-tutorial.org/
- http://learningwebgl.com/blog/?page_id=1217

Reference

- https://www.tutorialspoint.com/computer_graphics/3d_transformation.htm

P/S: This website is using pre-multiply. Transpose the matrix to get post-multiply.

TAs @ R506

- 劉志豪
  - R05922017@ntu.edu.tw
  - 0988950026
  - 星期一(Mon.) – 12:00~13:45
  - 星期三(Wed.) – 13:00~17:00
- 楊騏瑄
  - R05922101@ntu.edu.tw
  - 0978520930
  - 星期三(Wed.) – 10:30~12:00
  - 星期四(Thu.) – 10:30~12:00