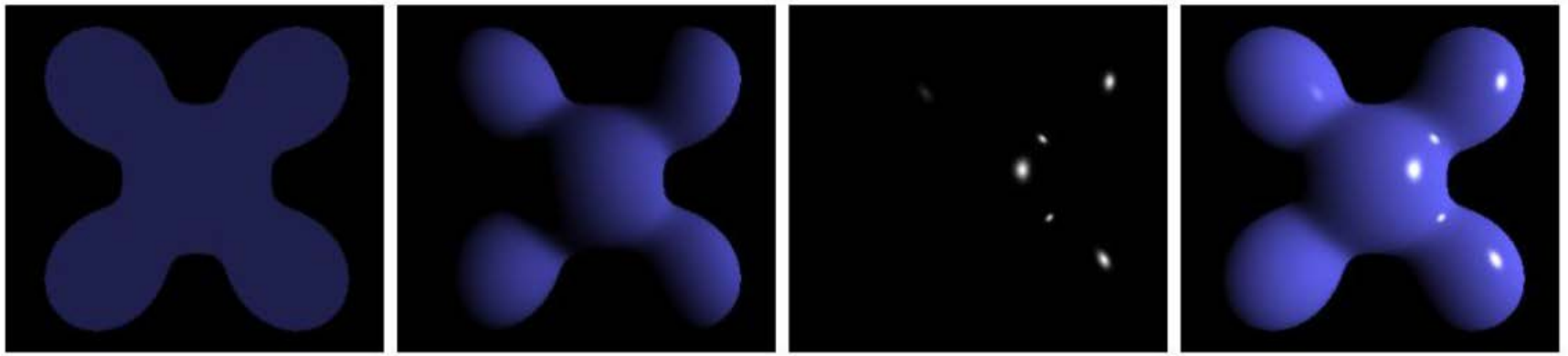
ICG 2018 Fall Homework1 Guidance 20181011

網媒所碩一　周家宇

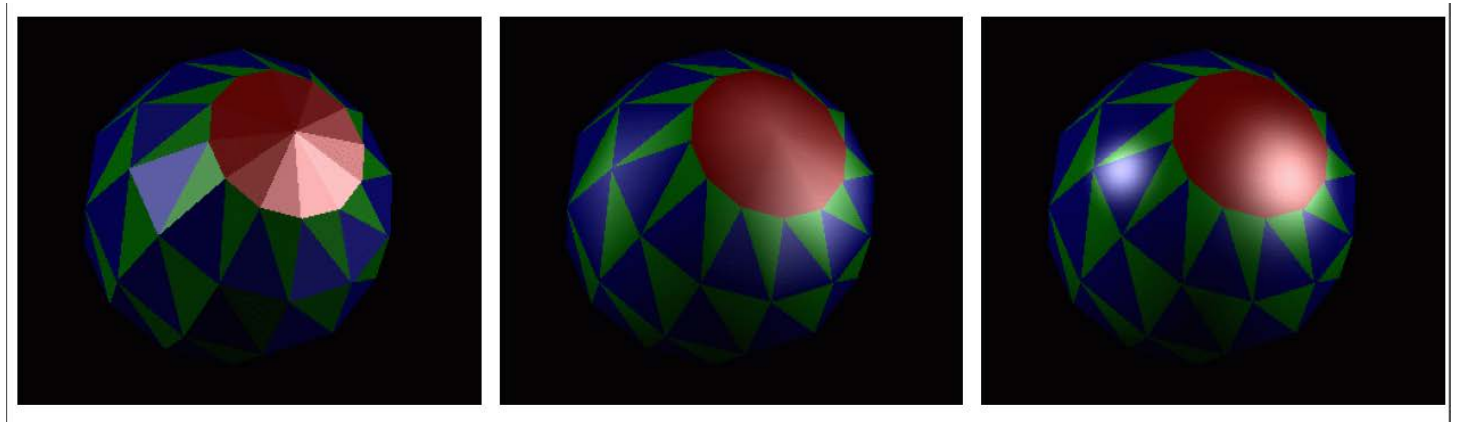
Requirements (Due to 20181108)

- **Flat, Gouraud, and Phong shading with Phong reflection model** in shaders. You can demonstrate the three shading computation in a single object.
- **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.
- At least **3 objects** & at least **3 light sources**
- Bonus: Special effects on shading / lighting / animation, …

Phong Reflection Model

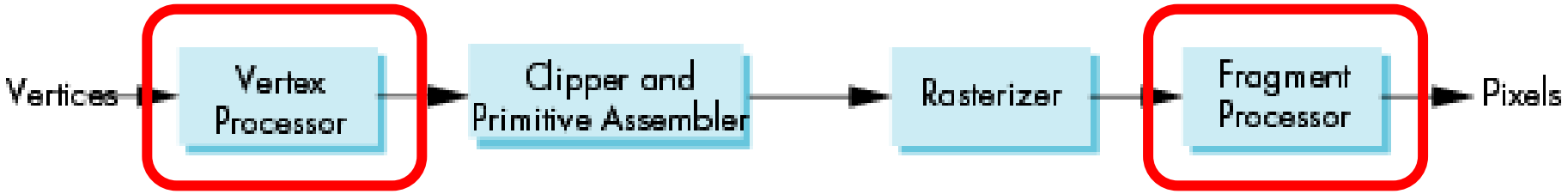


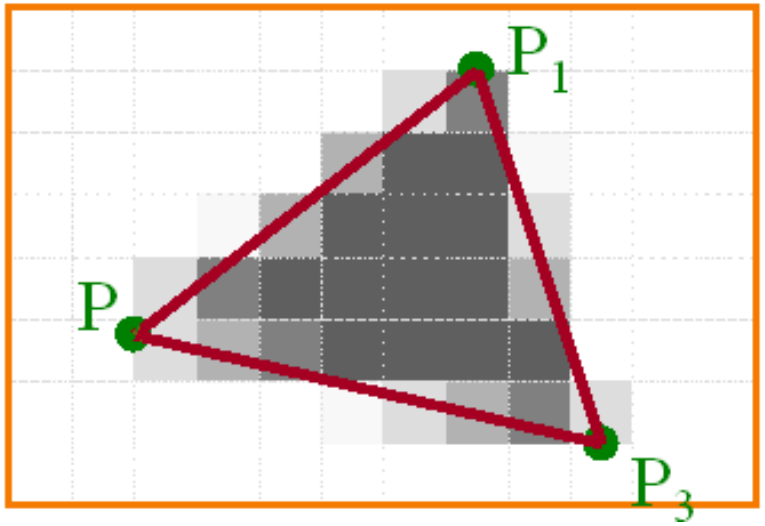
Shading



- 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

Rendering Pipeline





Shaders

```
<script id="fragmentShader" type="fragment">
    precision mediump float;
    varying vec4 fragcolor;
    void main(void) {
        gl_FragColor = fragcolor;
    }
</script>
```

// gl_FragColor is a built-in variable, it is the output of the fragment shader

```
<script id="vertexShader" type="vertex">
    attribute vec3 aVertexPosition;
    attribute vec2 aTextureCoord;

    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;

    varying vec4 fragcolor;
    uniform sampler2D uSampler;
```

// gl_Position is a built-in variable, it is the output of the vertex shader

```
    void main(void) {
        gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
        vec4 fragmentColor;
        fragmentColor = texture2D(uSampler, vec2(aTextureCoord.s, aTextureCoord.t
            ));
        fragcolor = vec4(fragmentColor.rgb, fragmentColor.a);;
    }
</script>
```

// another output pass to fragment shader

Shader data

```
<script id="fragmentShader" type="fragment">
    precision mediump float;
    varying vec4 fragcolor;
    void main(void) {
        gl_FragColor = fragcolor;
    }
</script>

<script id="vertexShader" type="vertex">
    attribute vec3 aVertexPosition;
    attribute vec2 aTextureCoord;

    uniform mat4 uMVMatrix;
    uniform mat4 uPMatrix;

    varying vec4 fragcolor;
    uniform sampler2D uSampler;

    void main(void) {
        gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
        vec4 fragmentColor;
        fragmentColor = texture2D(uSampler, vec2(aTextureCoord.s, aTextureCoord.t
            ));
        fragcolor = vec4(fragmentColor.rgb, fragmentColor.a);;
    }
</script>
```

Shader data

SHADER DATA

"Per-object constant"

Uniform

= Shared Constant

Vertex Data

= ANYTHING YOU WANT!

Example?

Positions…

Normals…

Colors…

Texture Coordinates…

Built-in Variables

Shader programs communicate with the graphics pipeline using pre-defined input and output variables.

A *Vertex Shader*'s Outputs

A *vertex shader* has two outputs:

- `gl_Position` : a `vec4` position of a vertex in "clip coordinates". Clip coordinates were described in detail in s
  - The values for *x* and *y* are in the range [-1.0,+1.0] and represent the location of a vertex in the viewing
  - The *z* value is in the range [-1.0,+1.0] and represents the distance of the vertex from the camera.
  - The *w* value is 1.0 for orthogonal projections, or the *w* value is the perspective divide value for perspec
  - Any vertex outside the clipping cube is clipped to the cube's boundaries.
- `gl_PointSize` : the number of pixels to use to render a point. It is a float value that can have a fractional part lines and triangles. If no value is specified, its default value it 1.0.

The outputs of a *vertex shader* are used by the graphics pipeline to determine the pixels (i.e., the fragments) in the the graphics pipeline creates a fragment for each pixel, calculates the interpolated values for any `varying` variable

Inputs to a *Fragment Shader*

- `gl_FragCoord` : a `vec4` value that holds the (x,y,z,w) value of the fragment. This is the value of `gl_Position` perspective divide has been performed. Therefore, the (x,y) values are the location of the fragment in the ima Note that these are floating point values and that the (x,y) values are the center of a pixel. For example, the b
- `gl_FrontFacing` : a Boolean value that is true if this fragment is part of a front-facing primitive. This only appl pointing toward the camera. The normal vector is calculated from the triangle's vertices, and a counter-clockv vector.
- `gl_PointCoord` : is a `vec2` value where its (x,y) values are in the range [0.0,1.0]. The values indicate the rela point's total size, in pixels, comes from the `gl_PointSize` output variable. The origin of the coordinate syster covers the point. (This only applies to rendered points, gl.POINTS. It is undefined when rendering gl.LINES a

Outputs from a *Fragment Shader*

A WebGL *fragment shader* has one output – a color value for its fragment.

- `gl_FragColor` : a RGBA (`vec4`) value that is placed into the *color buffer* for the fragment it is processing.

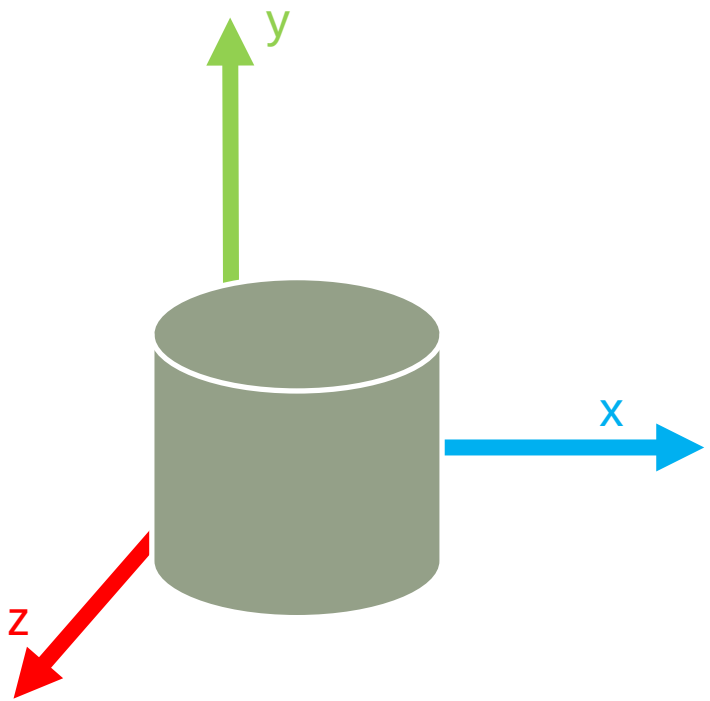
Load models

- 已經將大部分課程網的 tri 模型轉成 json 檔

Example　Teapot.json

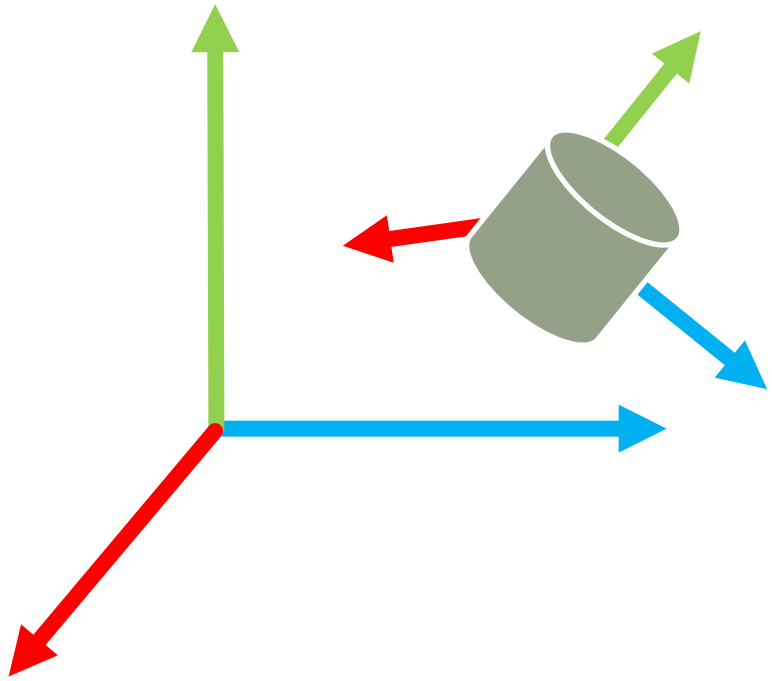
```
{
    "vertexPositions" : [5.929688,4.125,0,5.387188,4.125,2.7475,5.
    "vertexNormals" : [-0.966742,-0.255752,0,-0.893014,-0.256345,-
    "vertexTextureCoords" : [2,2,1.75,2,1.75,1.975,2,1.975,1.75,1.
    "indices" : [0,1,2,2,3,0,3,2,4,4,5,3,5,4,6,6,7,5,7,6,8,8,9,7,1
}
```

World transform



Model coordinates

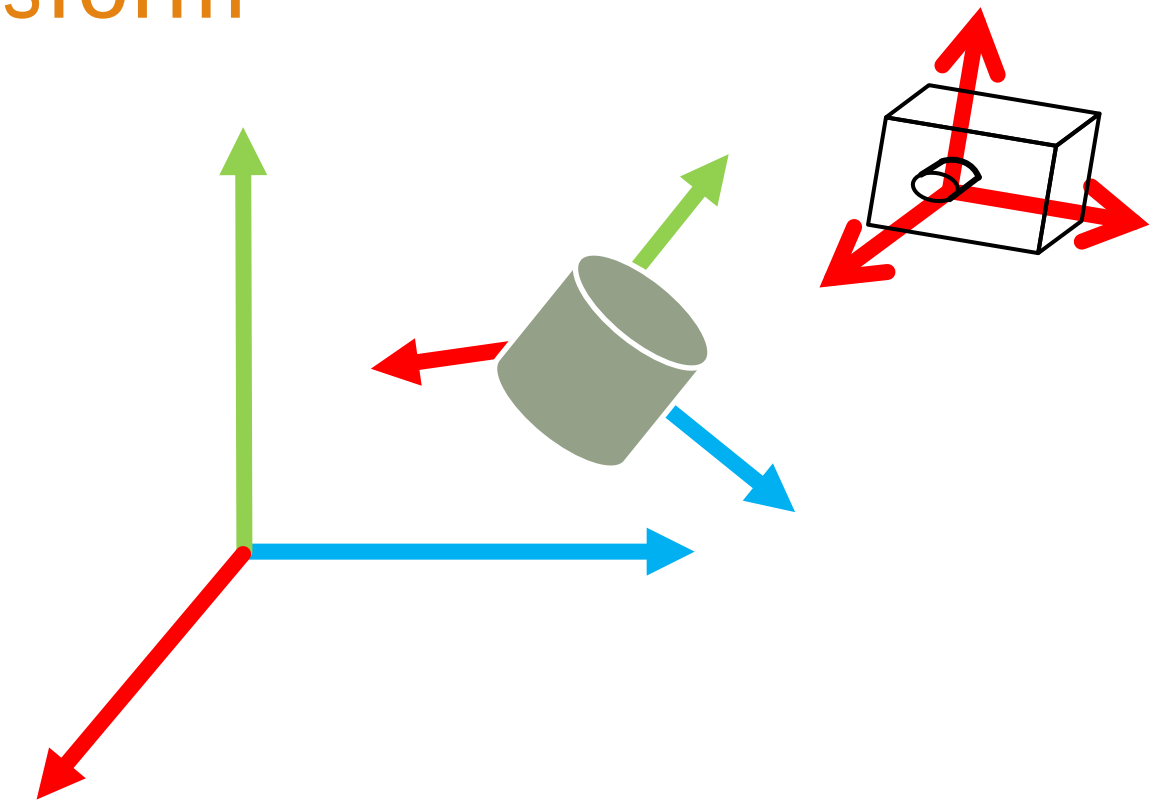
World transform



World coordinates

```
gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
```

World transform



Camera coordinates

```
gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
```

Transformations

- Homogeneous Coordinates

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

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

Translation

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 :

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

Scale

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

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

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

Rotate

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

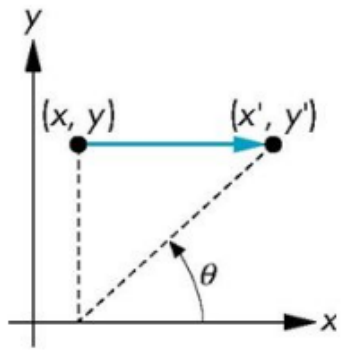
$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

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

Shear

Consider simple shear along x axis

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



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

Requirements Again (Due to 20181108)

- **Flat, Gouraud, and Phong shading with Phong reflection 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)
- At least **3 objects** & at least **3 light sources**
- Bonus: Special effects on shading / lighting / animation, …

Requirements Again

Reference

- https://webglfundamentals.org/
- http://learningwebgl.com/blog/?page_id=1217
- https://learnopengl.com/

TA Hours

- CSIE R506
- 周家宇
  - r07944038@csie.ntu.edu.tw
  - 星期五 10:00 ~ 12:00
- 楊子由
  - mukyu99@cmlab.csie.ntu.edu.tw
  - 星期二 10:00 ~ 12:00

Q & A