Computer Graphics

8 - Lab - Lighting

Yoonsang Lee
Hanyang University

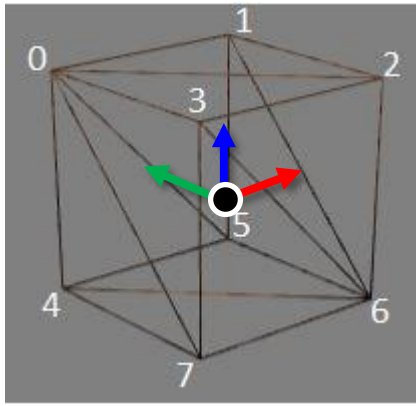
Spring 2025

Outline

- Setting Vertex Normal for Flat / Smooth Shading
- Render a Cube using Phong Illumination and Gouraud Shading
  - + Ambient component
  - + Diffuse component
  - + Specular component
- Render a Cube using Phong Illumination and Phong Shading
- Render a "Smooth" Cube using Phong Illumination and Gouraud / Phong Shading

Setting Vertex Normal for Flat / Smooth Shading

Example: a cube of length 2 again

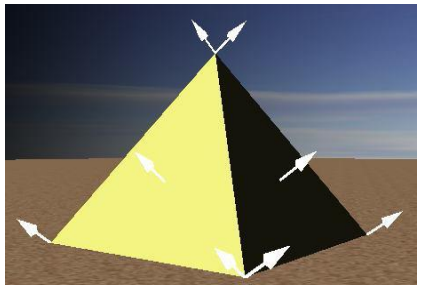


vertex index	position
0	(-1 , 1 , 1)
1	(1 , 1 , 1)
2	(1 , -1 , 1)
3	(-1 , -1 , 1)
4	(-1 , 1 , -1)
5	(1 , 1 , -1)
6	(1 , -1 , -1)
7	(-1 , -1 , -1)

Flat Shading in OpenGL

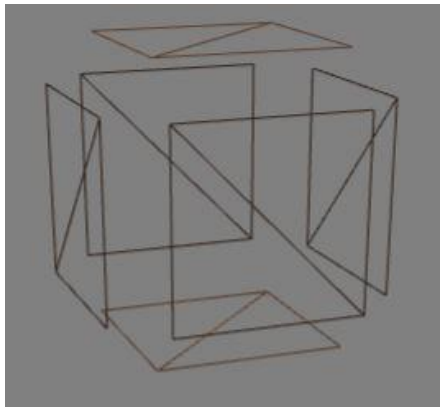
- Polygon shading method is determined by the vertex normal vectors you specify.

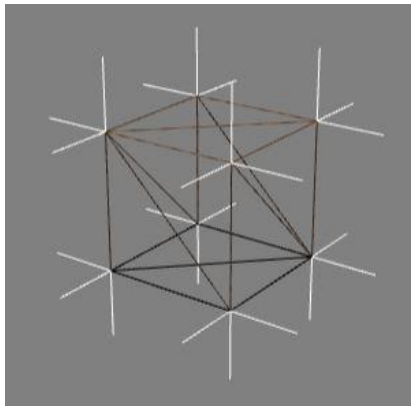
- Flat shading: Set each vertex normal to the face normal the vertex belongs to.

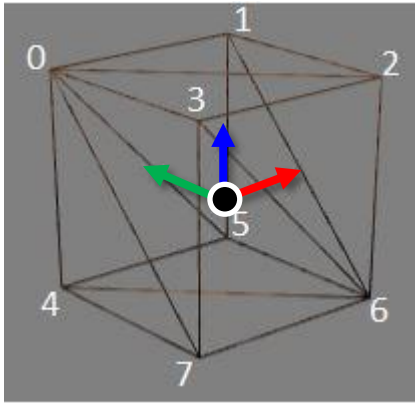


The normal at a vertex is the same as the face normal. Therefore, ***each vertex has as many normals as the number of faces it belongs****.*

Normals of the Cube for Flat Shading

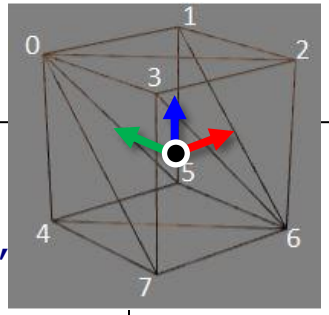






vertex index	position	**normal**
0	(-1 , 1 , 1)	**(0,0,1)**
2	(1 , -1 , 1)	**(0,0,1)**
1	(1 , 1 , 1)	**(0,0,1)**
0	(-1 , 1 , 1)	**(0,0,1)**
3	(-1 , -1 , 1)	**(0,0,1)**
2	(1 , -1 , 1)	**(0,0,1)**
4	(-1 , 1 , -1)	**(0,0,-1)**
5	(1 , 1 , -1)	**(0,0,-1)**
6	(1 , -1 , -1)	**(0,0,-1)**
4	(-1 , 1 , -1)	**(0,0,-1)**
6	(1 , -1 , -1)	**(0,0,-1)**
7	(-1 , -1 , -1)	**(0,0,-1)**
0	(-1 , 1 , 1)	**(0,1,0)**
1	(1 , 1 , 1)	**(0,1,0)**
5	(1 , 1 , -1)	**(0,1,0)**
0	(-1 , 1 , 1)	**(0,1,0)**
5	(1 , 1 , -1)	**(0,1,0)**
4	(-1 , 1 , -1)	**(0,1,0)**
3	(-1 , -1 , 1)	**(0,-1,0)**
6	(1 , -1 , -1)	**(0,-1,0)**
2	(1 , -1 , 1)	**(0,-1,0)**
3	(-1 , -1 , 1)	**(0,-1,0)**
7	(-1 , -1 , -1)	**(0,-1,0)**
6	(1 , -1 , -1)	**(0,-1,0)**
1	(1 , 1 , 1)	**(1,0,0)**
2	(1 , -1 , 1)	**(1,0,0)**
6	(1 , -1 , -1)	**(1,0,0)**
1	(1 , 1 , 1)	**(1,0,0)**
6	(1 , -1 , -1)	**(1,0,0)**
5	(1 , 1 , -1)	**(1,0,0)**
0	(-1 , 1 , 1)	**(-1,0,0)**
7	(-1 , -1 , -1)	**(-1,0,0)**
3	(-1 , -1 , 1)	**(-1,0,0)**
0	(-1 , 1 , 1)	**(-1,0,0)**
4	(-1 , 1 , -1)	**(-1,0,0)**
7	(-1 , -1 , -1)	**(-1,0,0)**

Vertex Data




```
def prepare_vao_cube():
    # 36 vertices for 12 triangles
    vertices = glm.array(glm.float32,
        # position      normal
        -1 ,  1 ,  1 ,  0, 0, 1, # v0
         1 , -1 ,  1 ,  0, 0, 1, # v2
         1 ,  1 ,  1 ,  0, 0, 1, # v1

        -1 ,  1 ,  1 ,  0, 0, 1, # v0
        -1 , -1 ,  1 ,  0, 0, 1, # v3
         1 , -1 ,  1 ,  0, 0, 1, # v2

        -1 ,  1 , -1 ,  0, 0,-1, # v4
         1 ,  1 , -1 ,  0, 0,-1, # v5
         1 , -1 , -1 ,  0, 0,-1, # v6

        -1 ,  1 , -1 ,  0, 0,-1, # v4
         1 , -1 , -1 ,  0, 0,-1, # v6
        -1 , -1 , -1 ,  0, 0,-1, # v7

        -1 ,  1 ,  1 ,  0, 1, 0, # v0
         1 ,  1 ,  1 ,  0, 1, 0, # v1
         1 ,  1 , -1 ,  0, 1, 0, # v5

        -1 ,  1 ,  1 ,  0, 1, 0, # v0
         1 ,  1 , -1 ,  0, 1, 0, # v5
        -1 ,  1 , -1 ,  0, 1, 0, # v4
        -1 , -1 ,  1 ,  0,-1, 0, # v3
         1 , -1 , -1 ,  0,-1, 0, # v6
         1 , -1 ,  1 ,  0,-1, 0, # v2

        -1 , -1 ,  1 ,  0,-1, 0, # v3
        -1 , -1 , -1 ,  0,-1, 0, # v7
         1 , -1 , -1 ,  0,-1, 0, # v6

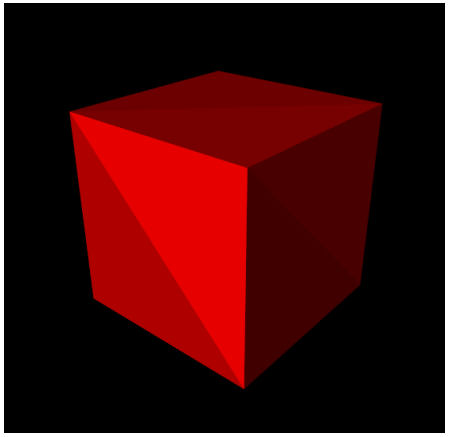
         1 ,  1 ,  1 ,  1, 0, 0, # v1
         1 , -1 ,  1 ,  1, 0, 0, # v2
         1 , -1 , -1 ,  1, 0, 0, # v6

         1 ,  1 ,  1 ,  1, 0, 0, # v1
         1 , -1 , -1 ,  1, 0, 0, # v6
         1 ,  1 , -1 ,  1, 0, 0, # v5

        -1 ,  1 ,  1 , -1, 0, 0, # v0
        -1 , -1 , -1 , -1, 0, 0, # v7
        -1 , -1 ,  1 , -1, 0, 0, # v3

        -1 ,  1 ,  1 , -1, 0, 0, # v0
        -1 ,  1 , -1 , -1, 0, 0, # v4
        -1 , -1 , -1 , -1, 0, 0, # v7
    )
    ...
```

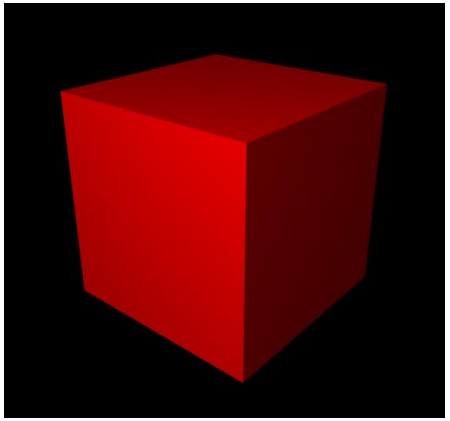
Flat Shading in OpenGL



- However, in modern OpenGL, "true" flat shading in the strict sense ("calculating color once per polygon") often produces visually odd results.
  - You can do this by using `flat` qualifier.
  - ```
    flat in vec4 vout_color;
    // in fragment shader
    ```

- This is because a polygon always is a triangle in modern OpenGL.
    ```

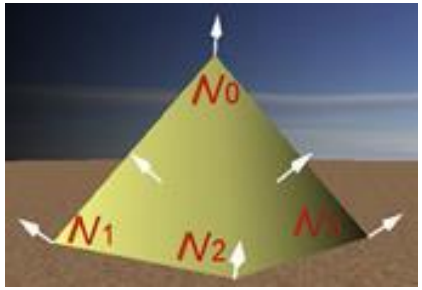
Flat Shading in OpenGL



- For the purpose of flat shading, Gouraud or Phong shading with "single normal per polygon" is generally used instead, by
  - Setting each vertex normal to the face normal the vertex belongs to.
  - Calculating **color per vertex or fragment**.
    - Even though the normal vector is the same, the light direction vector at each vertex or fragment is slightly different, so the calculated color is also slightly different.

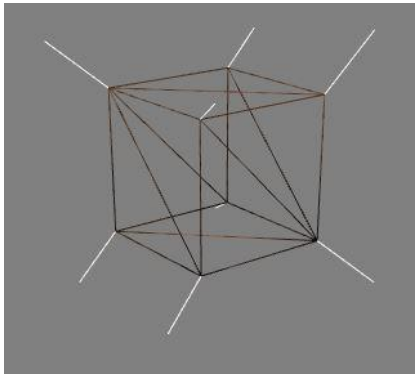
Smooth Shading in OpenGL

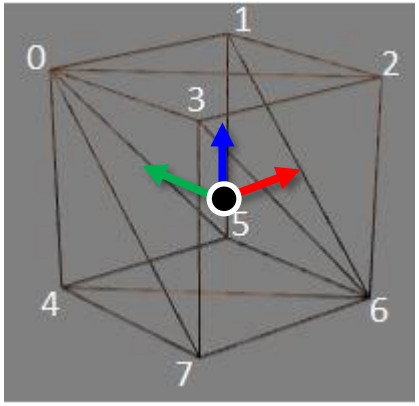
- Smooth shading: Set each vertex normal to the average of normals of all faces sharing the vertex.



***Only one vertex normal per vertex**; average of face normals of the faces the vertex is part of*

Normals of the Cube for Smooth Shading

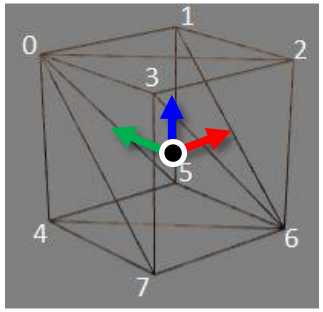




vertex index	position	normal
0	(-1 , 1 , 1)	(-0.5773502691896258 , 0.5773502691896258 , 0.5773502691896258)
1	(1 , 1 , 1)	(0.8164965809277261 , 0.4082482904638631 , 0.4082482904638631)
2	(1 , -1 , 1)	(0.4082482904638631 , -0.4082482904638631 , 0.8164965809277261)
3	(-1 , -1 , 1)	(-0.4082482904638631 , -0.8164965809277261 , 0.4082482904638631)
4	(-1 , 1 , -1)	(-0.4082482904638631 , 0.4082482904638631 , -0.8164965809277261)
5	(1 , 1 , -1)	(0.4082482904638631 , 0.8164965809277261 , -0.4082482904638631)
6	(1 , -1 , -1)	(0.5773502691896258 , -0.5773502691896258 , -0.5773502691896258)
7	(-1 , -1 , -1)	(-0.8164965809277261 , -0.4082482904638631 , -0.4082482904638631)

Vertex and Index Data

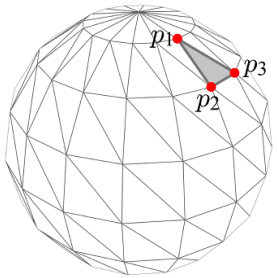
```
# 8 vertices
vertices = glm.array(glm.float32,
    # position      normal
    -1 ,  1 ,  1 , -0.577,  0.577,  0.577, # v0
     1 ,  1 ,  1 ,  0.816,  0.408,  0.408, # v1
     1 , -1 ,  1 ,  0.408, -0.408,  0.816, # v2
    -1 , -1 ,  1 , -0.408, -0.816,  0.408, # v3
    -1 ,  1 , -1 , -0.408,  0.408, -0.816, # v4
     1 ,  1 , -1 ,  0.408,  0.816, -0.408, # v5
     1 , -1 , -1 ,  0.577, -0.577, -0.577, # v6
    -1 , -1 , -1 , -0.816, -0.408, -0.408, # v7
)
# 12 triangles
indices = glm.array(glm.uint32,
    0,2,1,
    0,3,2,
    4,5,6,
    4,6,7,
    0,1,5,
    0,5,4,
    3,6,2,
    3,7,6,
    1,2,6,
    1,6,5,
    0,7,3,
    0,4,7,
)
```

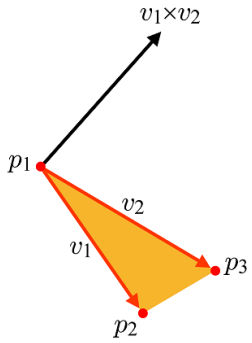


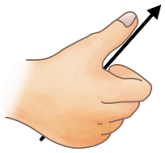
How to Get Vertex Normals

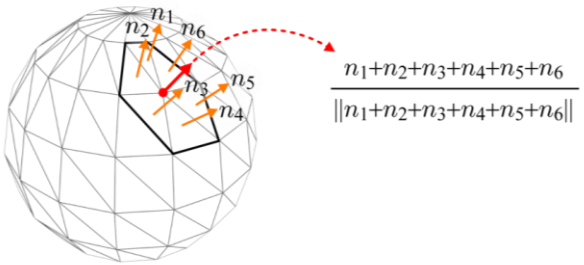
- Hard-code vertex normals in vertex data array
  - The previous code samples. Not commonly used.

- Compute normals from vertex positions









- Read normals from a model file such as .obj
  - Most commonly used.

Render a Cube using Phong Illumination and Gouraud Shading - Adding components one by one

Light & Material Phong Illumination Components

- Recall that the visible color of objects is computed by **element-wise multiplication** of the light and material RGB color values.

- Similarly, each Phong illumination component (ambient, diffuse, specular) is computed by **element-wise multiplication** of the light and material component color values.
  - e.g., diffuse_color = light_diffuse_color * material_diffuse_color

Good Settings for Light & Material Phong Illumination Components

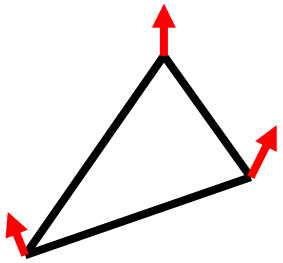
- Light
  - diffuse, specular: color of the light source
  - ambient: the same color, but at much reduced intensity (about 10%)

- Material
  - diffuse, ambient: color of the object
  - specular:
    - white (non-metal)
    - color of the object (metal)

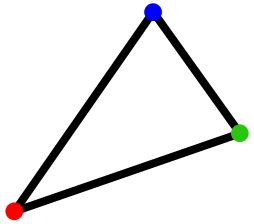
Recall: Gouraud Shading

Henri Gouraud
(1944~)

- Use a single vertex normal for each vertex



- Calculate color (by illumination) **at each vertex**
  → Lighting computation is done in **vertex shader**



- Interpolate vertex colors across polygon
  - Barycentric interpolation

[Code] 1-ambient-only-gouraud-facenorm

- Vertex shader

```
#version 330 core
layout (location = 0) in vec3 vin_pos;
layout (location = 1) in vec3 vin_normal;
out vec4 vout_color;
uniform mat4 MVP;

void main()
{
    vec4 p3D_in_hcoord = vec4(vin_pos.xyz, 1.0);
    gl_Position = MVP * p3D_in_hcoord;

    // light and material properties
    vec3 light_color = vec3(1,1,1);
    vec3 material_color = vec3(1,0,0);

    // light components
    vec3 light_ambient = 0.1*light_color;

    // material components
    vec3 material_ambient = material_color;

    // ambient
    vec3 ambient = light_ambient * material_ambient;

    vec3 color = ambient;
    vout_color = vec4(color, 1.);
}
```

[Code] 1-ambient-only-gouraud-facenorm

- Fragment shader

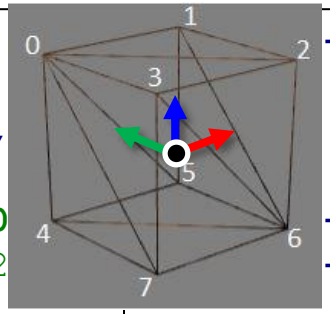
```
#version 330 core

in vec4 vout_color; // interpolated color

out vec4 FragColor;

void main()
{
    FragColor = vout_color;
}
```

[Code] 1-ambient-only-gouraud-facenorm




```
def prepare_vao_cube():
    # 36 vertices for 12 triangles
    vertices = glm.array(glm.float32,
        # position      normal
        -1 ,  1 ,  1 ,  0, 0, 1, # v0
         1 , -1 ,  1 ,  0, 0, 1, # v2
         1 ,  1 ,  1 ,  0, 0, 1, # v1

        -1 ,  1 ,  1 ,  0, 0, 1, # v0
        -1 , -1 ,  1 ,  0, 0, 1, # v3
         1 , -1 ,  1 ,  0, 0, 1, # v2

        -1 ,  1 , -1 ,  0, 0,-1, # v4
         1 ,  1 , -1 ,  0, 0,-1, # v5
         1 , -1 , -1 ,  0, 0,-1, # v6

        -1 ,  1 , -1 ,  0, 0,-1, # v4
         1 , -1 , -1 ,  0, 0,-1, # v6
        -1 , -1 , -1 ,  0, 0,-1, # v7

        -1 ,  1 ,  1 ,  0, 1, 0, # v0
         1 ,  1 ,  1 ,  0, 1, 0, # v1
         1 ,  1 , -1 ,  0, 1, 0, # v5

        -1 ,  1 ,  1 ,  0, 1, 0, # v0
         1 ,  1 , -1 ,  0, 1, 0, # v5
        -1 ,  1 , -1 ,  0, 1, 0, # v4
        -1 , -1 ,  1 ,  0,-1, 0, # v3
         1 , -1 , -1 ,  0,-1, 0, # v6
         1 , -1 ,  1 ,  0,-1, 0, # v2

        -1 , -1 ,  1 ,  0,-1, 0, # v3
        -1 , -1 , -1 ,  0,-1, 0, # v7
         1 , -1 , -1 ,  0,-1, 0, # v6

         1 ,  1 ,  1 ,  1, 0, 0, # v1
         1 , -1 ,  1 ,  1, 0, 0, # v2
         1 , -1 , -1 ,  1, 0, 0, # v6

         1 ,  1 ,  1 ,  1, 0, 0, # v1
         1 , -1 , -1 ,  1, 0, 0, # v6
         1 ,  1 , -1 ,  1, 0, 0, # v5

        -1 ,  1 ,  1 , -1, 0, 0, # v0
        -1 , -1 , -1 , -1, 0, 0, # v7
        -1 , -1 ,  1 , -1, 0, 0, # v3

        -1 ,  1 ,  1 , -1, 0, 0, # v0
        -1 ,  1 , -1 , -1, 0, 0, # v4
        -1 , -1 , -1 , -1, 0, 0, # v7
    )
    ...
```

(Same as page 5)

[Code] 1-ambient-only-gouraud-facenorm

```
def main():
    ...
    vao_cube = prepare_vao_cube()
    while not glfwWindowShouldClose(window):
        ...
        glUseProgram(shader_program)
        P = glm.perspective(45, 1, 1, 10) # projection matrix

        view_pos = glm.vec3(5*np.sin(g_cam_ang),g_cam_height,5*np.cos(g_cam_ang))
        V = glm.lookAt(view_pos, glm.vec3(0,0,0), glm.vec3(0,1,0)) # view matrix

        M = glm.mat4()
        ...
        # # try applying rotation
        # M = R

        # update uniforms
        MVP = P*V*M
        glUniformMatrix4fv(loc_MVP, 1, GL_FALSE, glm.value_ptr(MVP))

        # draw cube w.r.t. the current frame MVP
        glBindVertexArray(vao_cube)
        glDrawArrays(GL_TRIANGLES, 0, 36)
        ...
```

[Code] 2-ambient-diffuse-gouraud-shading

- Vertex shader

```
#version 330 core

layout (location = 0) in vec3 vin_pos;
layout (location = 1) in vec3 vin_normal;

out vec4 vout_color;

uniform mat4 MVP;
uniform mat4 M;

void main()
{
    vec4 p3D_in_hcoord = vec4(vin_pos.xyz, 1.0);
    gl_Position = MVP * p3D_in_hcoord;

    // light and material properties
    vec3 light_pos = vec3(3,2,4);
    vec3 light_color = vec3(1,1,1);
    vec3 material_color = vec3(1,0,0);
```

[Code] 2-ambient-diffuse-gouraud-shading

```
    // light components
    vec3 light_ambient = 0.1*light_color;
    vec3 light_diffuse = light_color;

    // material components
    vec3 material_ambient = material_color;
    vec3 material_diffuse = material_color;

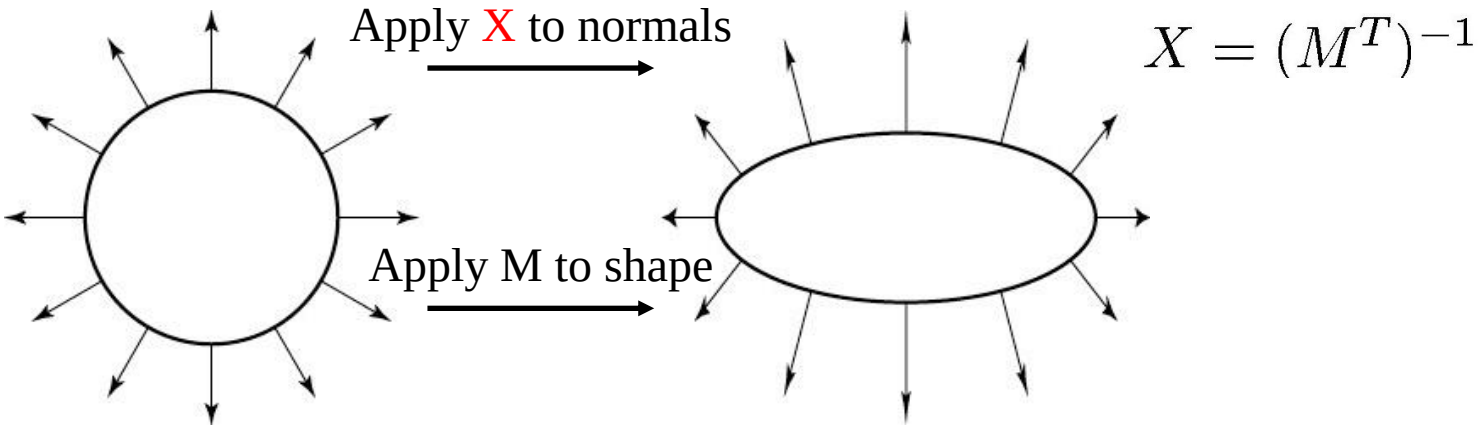
    // ambient
    vec3 ambient = light_ambient * material_ambient;

    // diffuse
    vec3 normal = normalize( mat3(inverse(transpose(M))) * vin_normal);
    vec3 surface_pos = vec3(M * vec4(vin_pos, 1)); // in world space
    vec3 light_dir = normalize(light_pos - surface_pos);
    float diff = max(dot(normal, light_dir), 0);
    vec3 diffuse = diff * light_diffuse * material_diffuse;

    vec3 color = ambient + diffuse;
    vout_color = vec4(color, 1.);
}
```

[Code] 2-ambient-diffuse-gouraud-shading

```
vec3 normal = normalize( mat3(inverse(transpose(M))) * vin_normal);
```



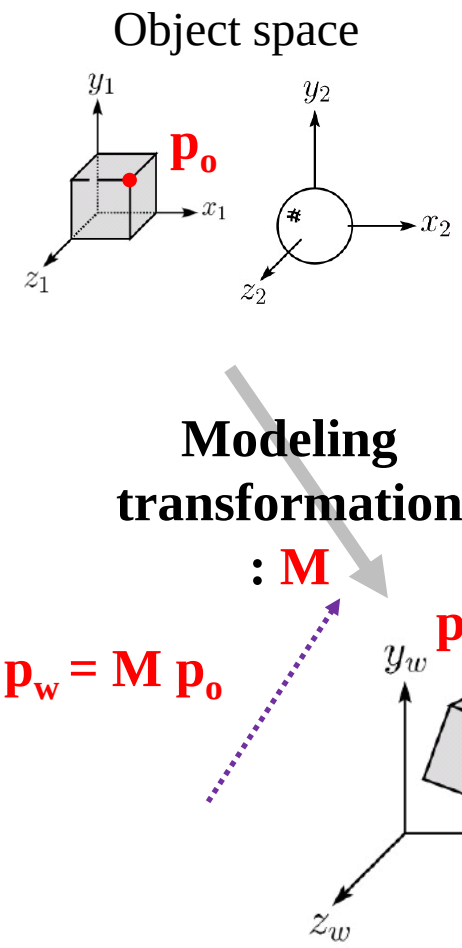

```
float diff = max(dot(normal, light_dir), 0);
```

- $\mathbf{I}_d = \mathbf{l}_d * \mathbf{m}_d \cos(\theta) = \mathbf{l}_d * \mathbf{m}_d (\mathbf{L} \cdot \mathbf{N})$
- `max()` is used because we don't want negative color values.

[Code] 2-ambient-diffuse-gouraud-shading

```
vec3 surface_pos = vec3(M * vec4(vin_pos, 1)); // in world space
```

- The position (in **world space**) of the surface point to which this lighting calculation is applied.
- We're doing all lighting calculation in **world space**.
  - This means that all positions and vectors used for lighting calculations are expressed w.r.t world frame.
- However, doing lighting in **view space** is preferred.
  - Advantage: view position is always (0,0,0) in view space.
  - But today's examples use world space because it is more intuitive for the purpose of learning.
  - You can try lighting in view space yourself later.



[Code] 3-all-components-gouraud-facenorm

- Vertex shader

```
...
uniform mat4 MVP;
uniform mat4 M;
uniform vec3 view_pos;

void main()
{
    vec4 p3D_in_hcoord = vec4(vin_pos.xyz, 1.0);
    gl_Position = MVP * p3D_in_hcoord;

    // light and material properties
    vec3 light_pos = vec3(3,2,4);
    vec3 light_color = vec3(1,1,1);
    vec3 material_color = vec3(1,0,0);
    float material_shininess = 32.0;

    // light components
    vec3 light_ambient = 0.1*light_color;
    vec3 light_diffuse = light_color;
    vec3 light_specular = light_color;
```

```
    // material components
    vec3 material_ambient = material_color;
    vec3 material_diffuse = material_color;
    vec3 material_specular = vec3(1,1,1);  // for non-metal material

    // ambient
    vec3 ambient = light_ambient * material_ambient;

    // for diffiuse and specular
    vec3 normal = normalize(mat3(inverse(transpose(M))) * vin_normal);
    vec3 surface_pos = vec3(M * vec4(vin_pos, 1));
    vec3 light_dir = normalize(light_pos - surface_pos);

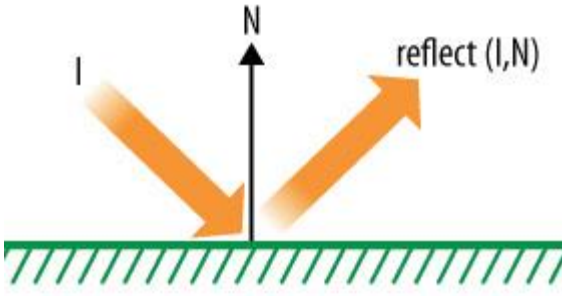
    // diffuse
    float diff = max(dot(normal, light_dir), 0);
    vec3 diffuse = diff * light_diffuse * material_diffuse;

    // specular
    vec3 view_dir = normalize(view_pos - surface_pos);
    vec3 reflect_dir = reflect(-light_dir, normal);
    float spec = pow( max(dot(view_dir, reflect_dir), 0.0),
material_shininess);
    vec3 specular = spec * light_specular * material_specular;

    vec3 color = ambient + diffuse + specular;
    vout_color = vec4(color, 1.);
}
```

[Code] 3-all-components-gouraud-facenorm

```
vec3 reflect_dir = reflect(-light_dir, normal);
```




```
float spec = pow(max(dot(view_dir,reflect_dir),0.0),material_shininess);
```

- $\mathbf{I}_s = \mathbf{l}_s * \mathbf{m}_s \cos^n(\alpha) = \mathbf{l}_s * \mathbf{m}_s (\mathbf{V} \cdot \mathbf{R})^n$
- `max()` is used because we don't want negative color values.

Quiz 3

- Go to https://www.slido.com/
- Join #**cg-ys**
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2021123456: 4.0**

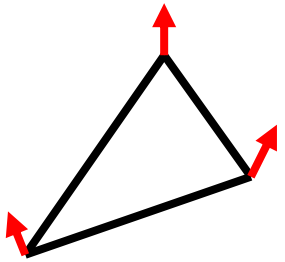
- Note that your quiz answer must be submitted **in the above format** to be counted for attendance.

Render a Cube using Phong Illumination and Phong Shading

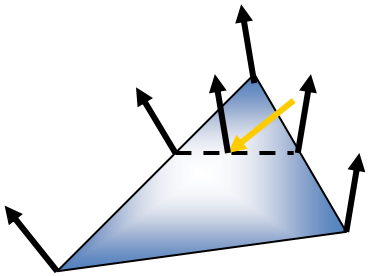
Recall: Phong Shading

Bùi Tường Phong
(1942 – 1975)

- Use a single vertex normal for each vertex



- Interpolate vertex normals across polygon



- Calculate color (by illumination) **at each pixel** in polygon using the interpolated normal
  → Lighting computation is done in **fragment shader**

[Code] 4-all-components-phong-facenorm

- Vertex shader

```
#version 330 core

layout (location = 0) in vec3 vin_pos;
layout (location = 1) in vec3 vin_normal;

out vec3 vout_surface_pos;
out vec3 vout_normal;

uniform mat4 MVP;
uniform mat4 M;

void main()
{
    vec4 p3D_in_hcoord = vec4(vin_pos.xyz, 1.0);
    gl_Position = MVP * p3D_in_hcoord;

    vout_surface_pos = vec3(M * vec4(vin_pos, 1));
    vout_normal = normalize( mat3(inverse(transpose(M))) * vin_normal);
}
```

[Code] 4-all-components-phong-facenorm

- Fragment shader

```
#version 330 core

in vec3 vout_surface_pos; // interpolated surface position
in vec3 vout_normal; // interpolated normal

out vec4 FragColor;

uniform vec3 view_pos;

void main()
{
    // light and material properties
    vec3 light_pos = vec3(3,2,4);
    vec3 light_color = vec3(1,1,1);
    vec3 material_color = vec3(1,0,0);
    float material_shininess = 32.0;

    // light components
    vec3 light_ambient = 0.1*light_color;
    vec3 light_diffuse = light_color;
    vec3 light_specular = light_color;
```

```
    // material components
    vec3 material_ambient = material_color;
    vec3 material_diffuse = material_color;
    vec3 material_specular = vec3(1,1,1);  // or can be material_color

    // ambient
    vec3 ambient = light_ambient * material_ambient;

    // for diffiuse and specular
    vec3 normal = normalize(vout_normal);
    vec3 surface_pos = vout_surface_pos;
    vec3 light_dir = normalize(light_pos - surface_pos);

    // diffuse
    float diff = max(dot(normal, light_dir), 0);
    vec3 diffuse = diff * light_diffuse * material_diffuse;

    // specular
    vec3 view_dir = normalize(view_pos - surface_pos);
    vec3 reflect_dir = reflect(-light_dir, normal);
    float spec = pow( max(dot(view_dir, reflect_dir), 0.0),
material_shininess);
    vec3 specular = spec * light_specular * material_specular;

    vec3 color = ambient + diffuse + specular;
    FragColor = vec4(color, 1.);
}
```

Render a "Smooth" Cube using Phong Illumination and Gouraud / Phong Shading

[Code]

- '5-all-components-gouraud-avgnorm'
  :Same as '3-all-components-gouraud-facenorm'
  except prepare_vao_cube() and glDrawElements()
  call

- '6-all-components-phong-avgnorm'
  : :Same as '4-all-components-phong-facenorm'
  except prepare_vao_cube() and glDrawElements()
  call

Time for Assignment

- Let's start today's assignment.
- TA will guide you.