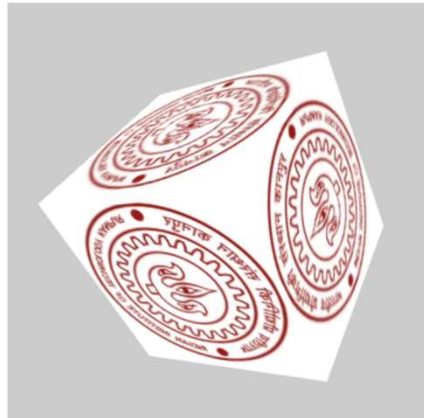


Introduction to Computer Graphics (CS360A)

Instructor: Soumya Dutta

Department of Computer Science and Engineering

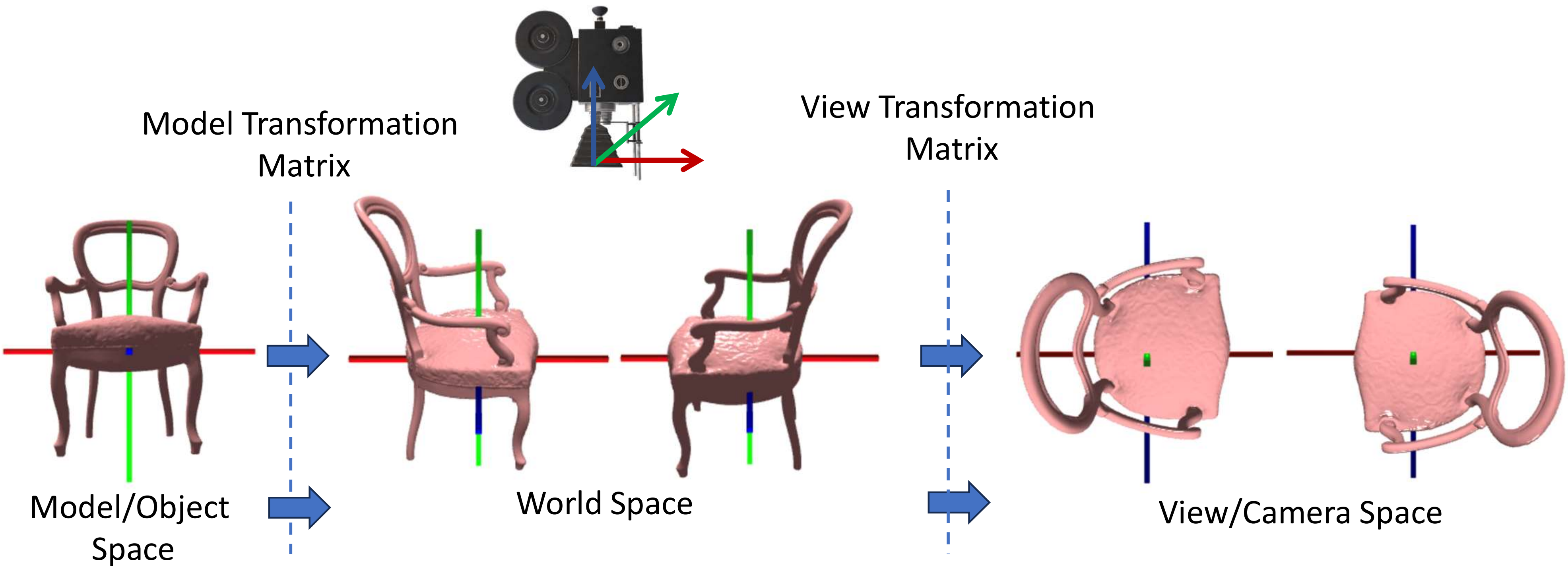
Indian Institute of Technology Kanpur (IITK)

email: soumyad@cse.iitk.ac.in

Transformations for Shading

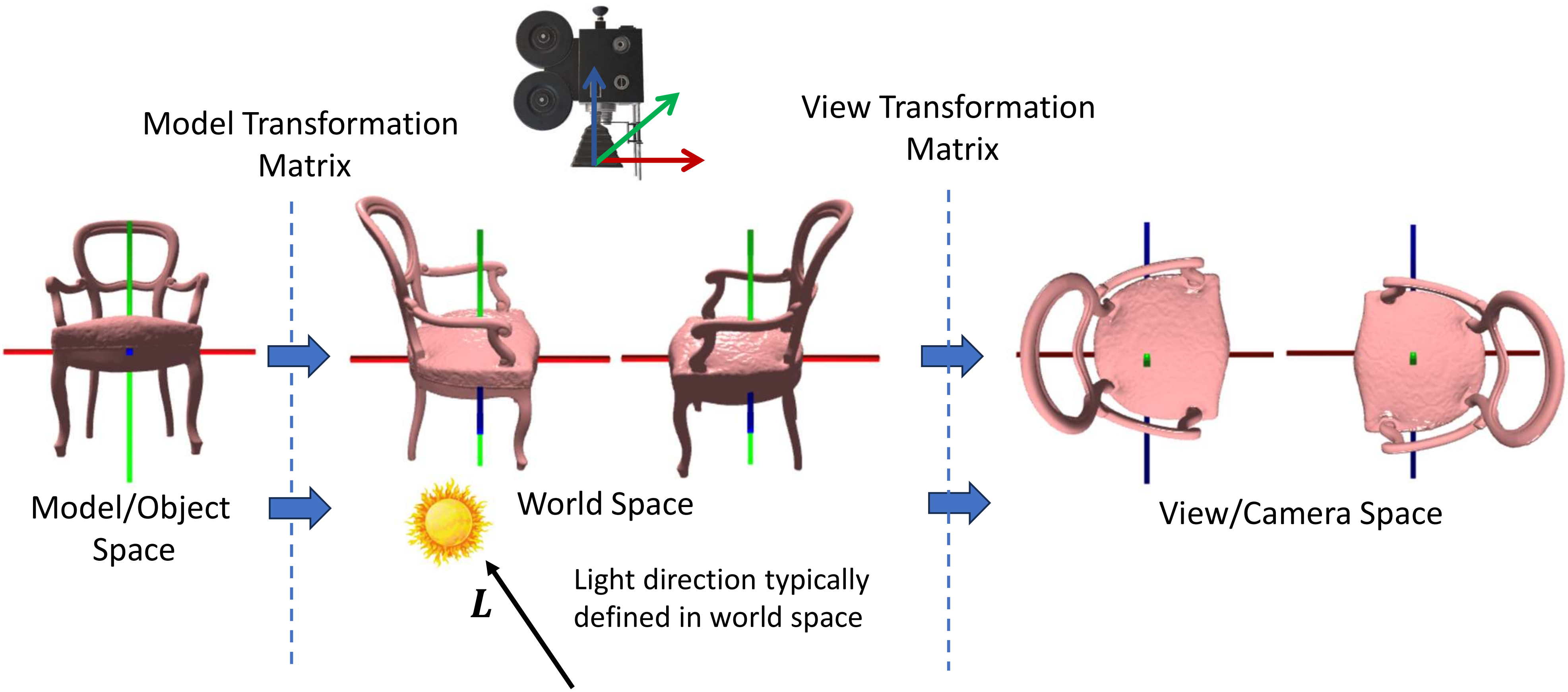
Shading Transformations

- While computing shading, all vectors and positions should be in the same coordinate frame so that operations on them make sense



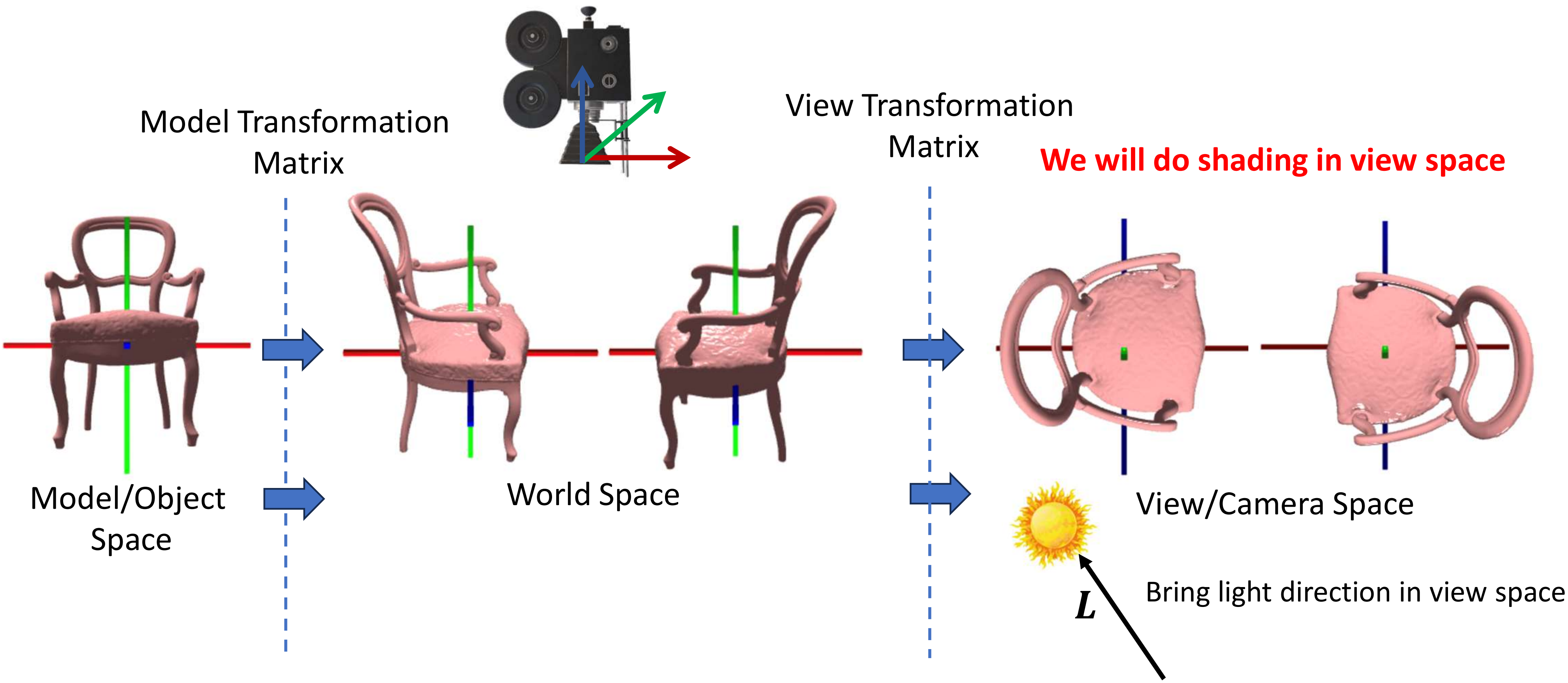
Shading Transformations

- In which space should we perform shading computation?



Shading Transformations

- In which space should we perform shading computation?



Shading in View Space

View Space ⟵ Model Space

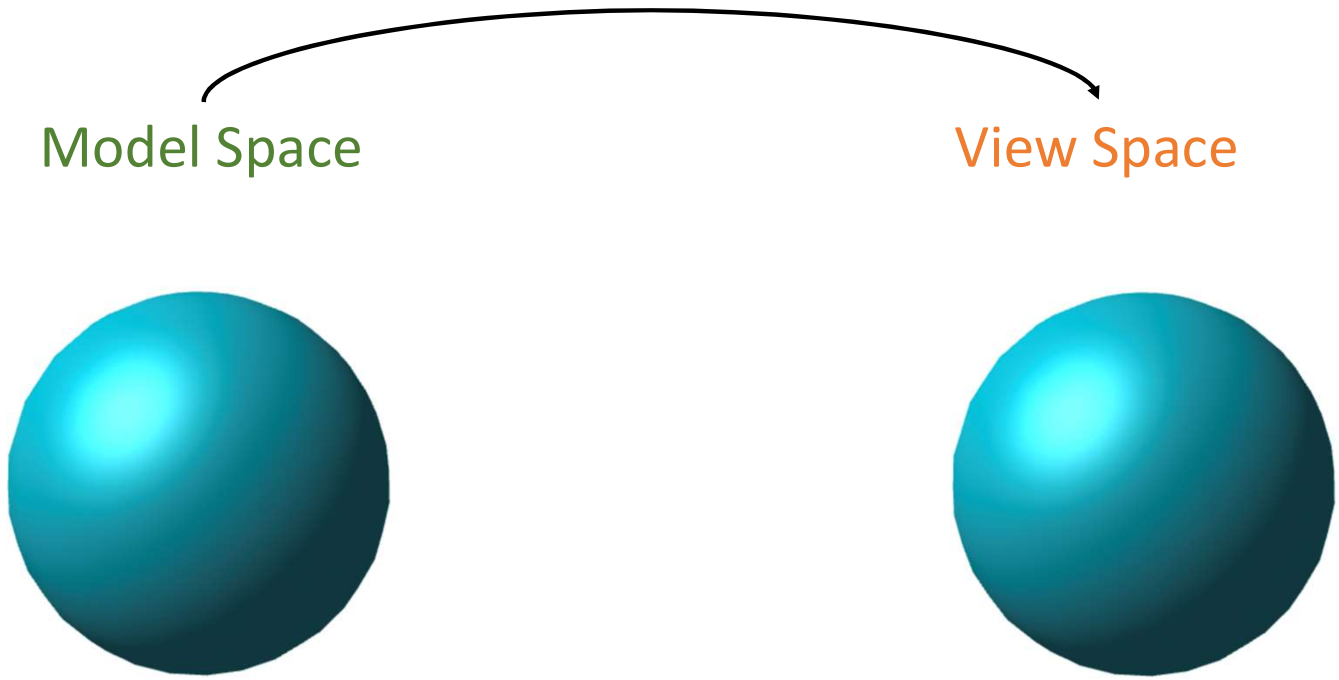
$p' = Mp$ ⟵ Positions

$n' = ?n$ ⟵ Normals

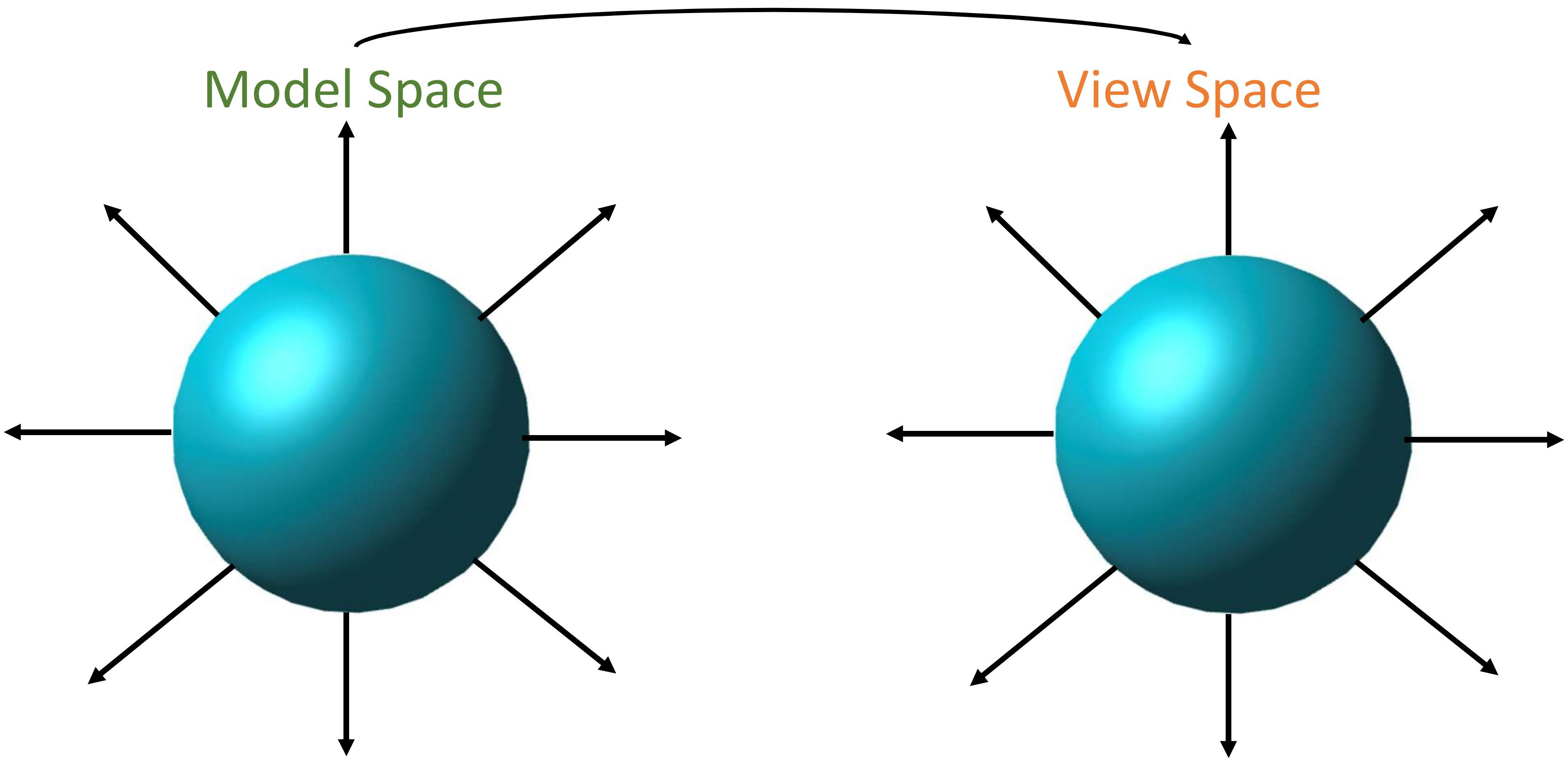
M = ModelView Matrix

$= ViewMat * ModelMat$

Shading Transformations: Positions



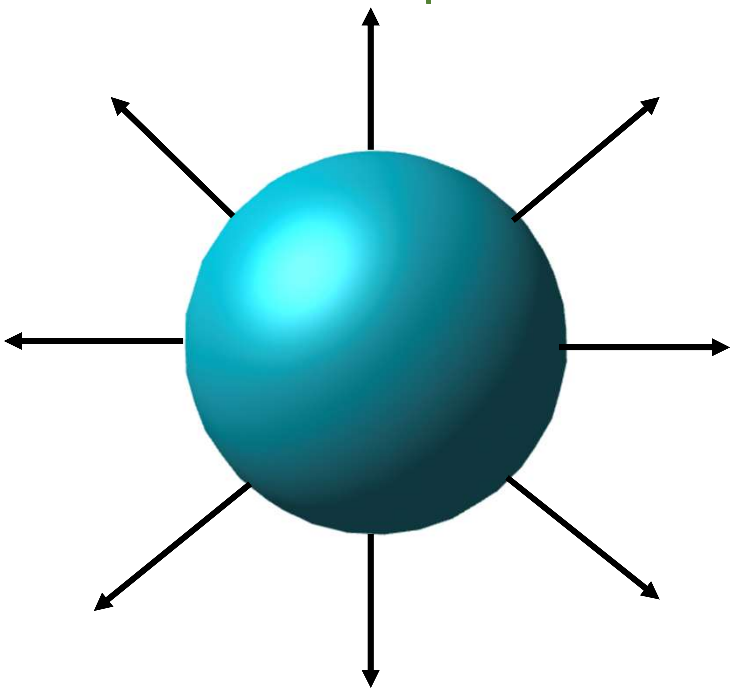
Shading Transformations: Normals



Shading Transformations: Normals

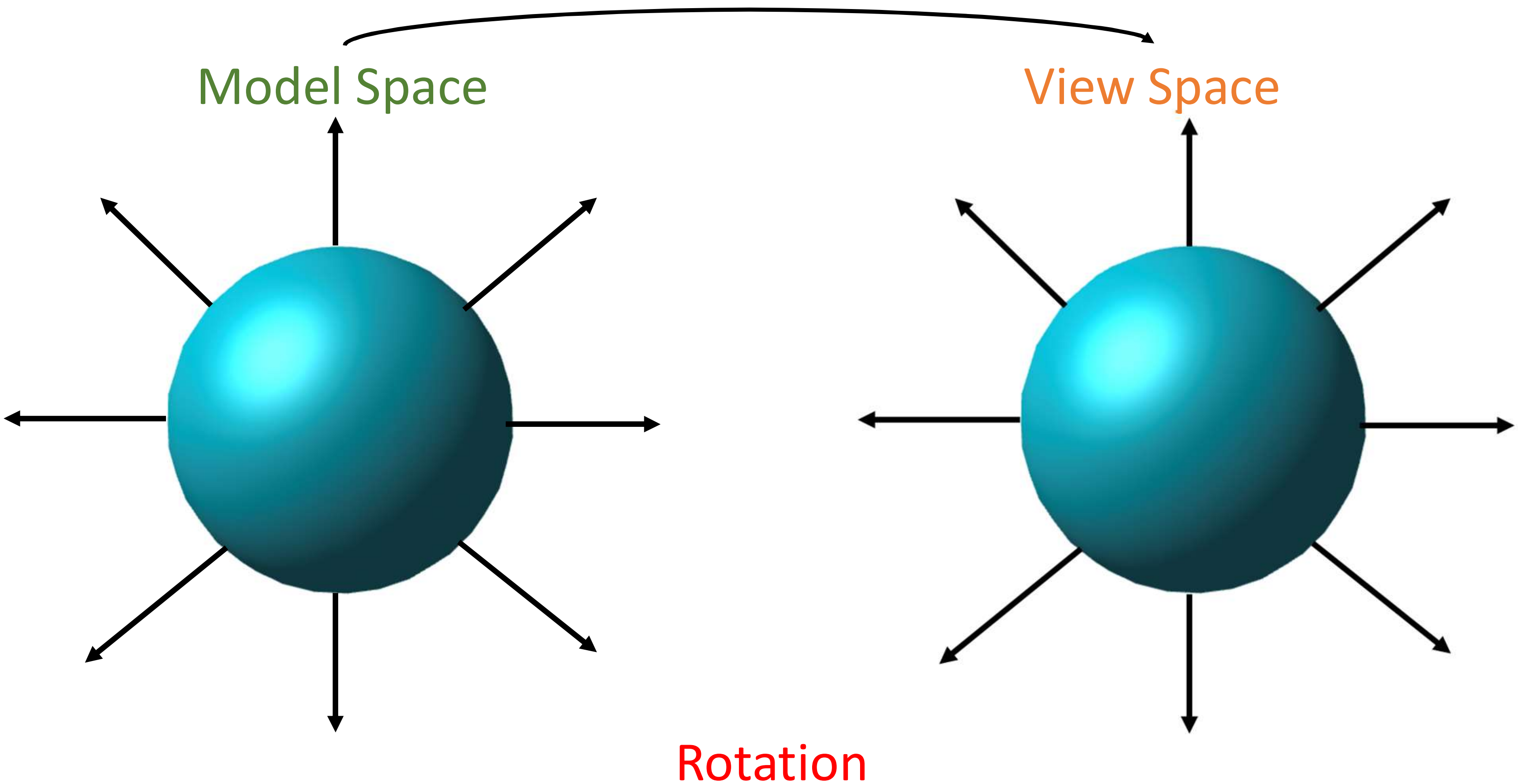
Model Space

View Space

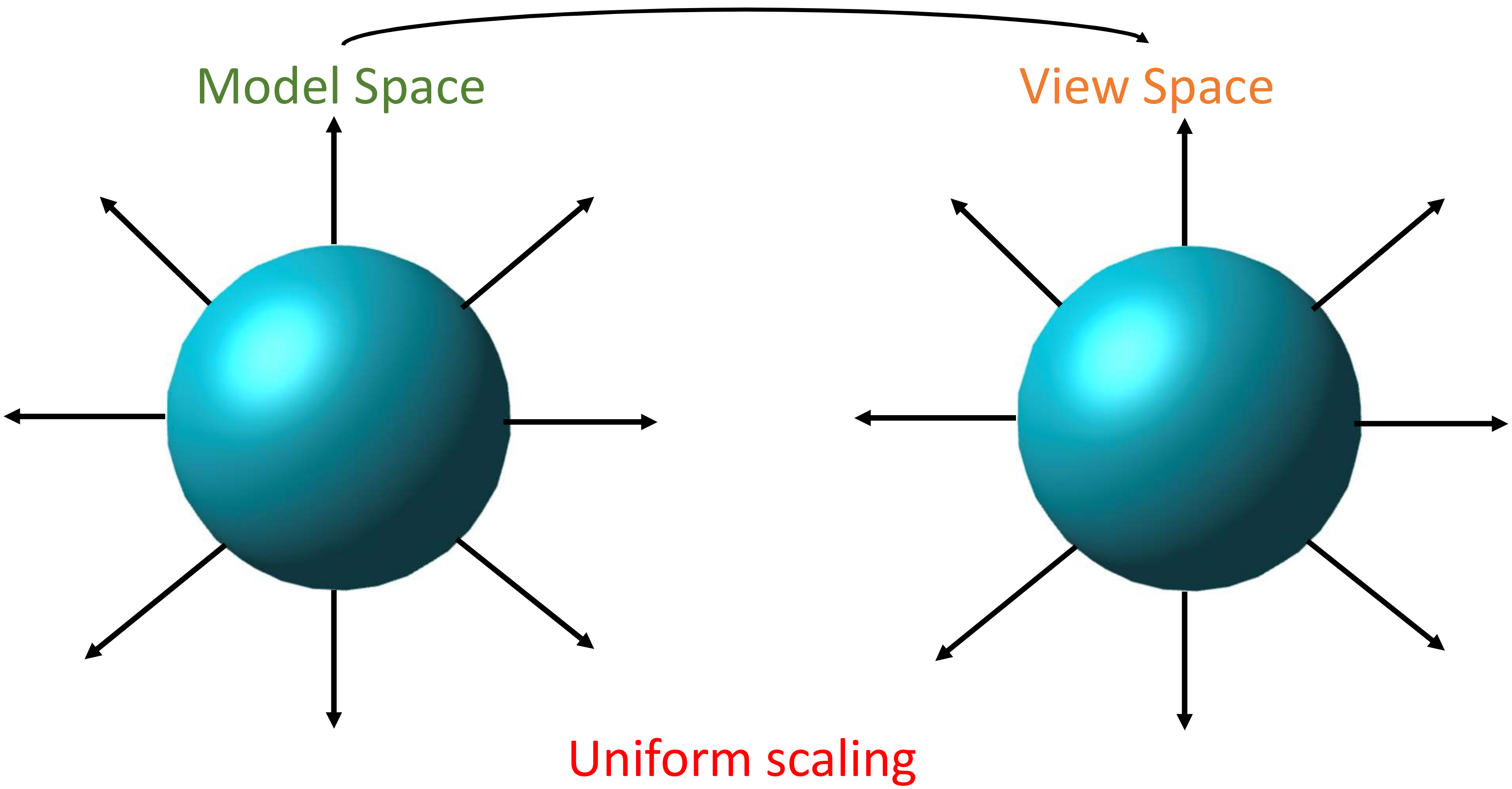


Translation

Shading Transformations: Normals



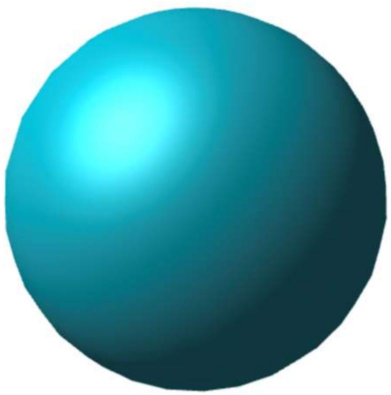
Shading Transformations: Normals

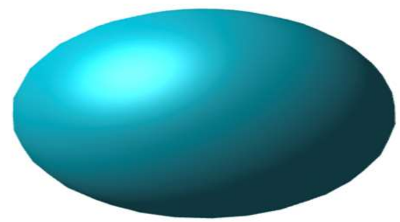


Shading Transformations: Positions

Model Space

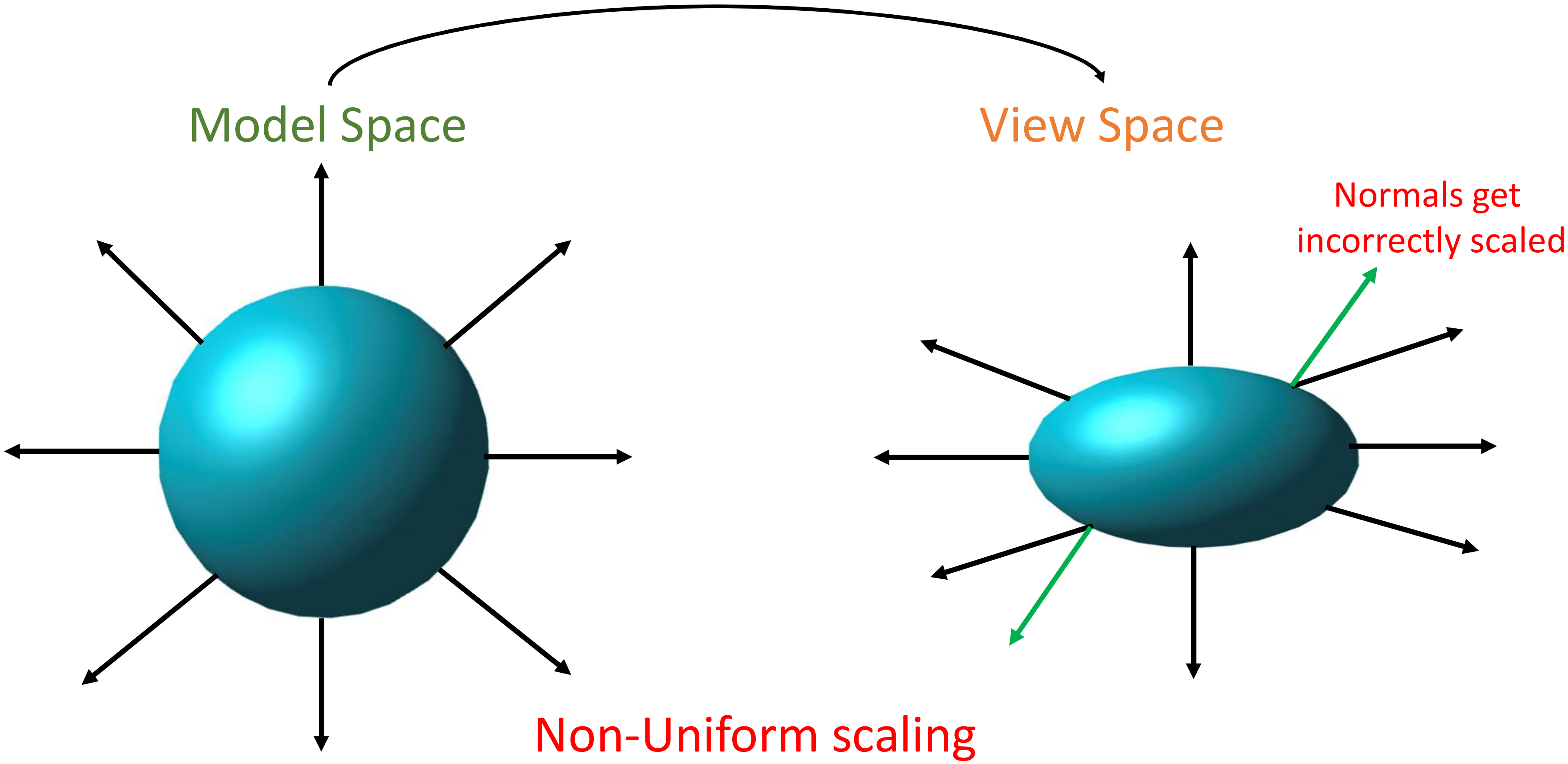
View Space





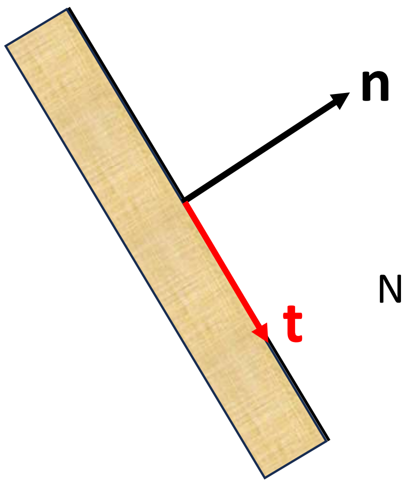
Non-Uniform scaling

Shading Transformations: Normals



Normal Transformation Matrix

- What we need to fix it is the inverse of the Scaling matrix!
- Let's see why that makes sense



Inverse of scale $\Longrightarrow$ $\mathbf{n'} = \begin{bmatrix} n_x/a \\ n_y/b \\ n_z/c \end{bmatrix}$

Non-uniform scale $\Longrightarrow$ $\mathbf{t'} = \begin{bmatrix} at_x \\ bt_y \\ ct_z \end{bmatrix}$

$$\boldsymbol{t'}.\boldsymbol{n'} = atxnx/a + btyny/b + ctznz/c$$
$$= t_x n_x + t_y n_y + t_z n_z = 0$$

$$\boldsymbol{t}.\boldsymbol{n} = t_x n_x + t_y n_y + t_z n_z = 0$$

Normal Transformation Matrix

View Space ⇐ Model Space

$p' = Mp$ ⇐ Positions

$n' = ?n$ ⇐ Normals

M = ModelView Matrix

$= \boldsymbol{ViewMat * ModelMat}$

Normal Transformation Matrix

View Space ⇐ Model Space

$$p' = M \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$ ← Position

M = ModelView Matrix

$$n' = M_{N(3X3)} \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix}$$ ← Normal

$M_{(3X3)}$= 3x3 normal transformation matrix

Normal Transformation Matrix

- Any matrix can be decomposed into two rotation matrices and a diagonal matrix using SVD

$$M = U * \Sigma * V^T$$

Rotation matrix

Scale matrix

Rotation matrix

Normal Transformation Matrix

- Any 3x3 matrix can be decomposed into two rotation matrices and a diagonal matrix using SVD

$$M_N = R_2 S^{-1} R_1$$ (This is what we want, the normal transformation matrix)

$$M_{3x3} = R_2 S R_1$$

$$M_{3x3}^{-1} = (R_2 S R_1)^{-1}$$

$$M_{3x3}^{-1} = R_1^{-1} S^{-1} R_2^{-1}$$

$$(M_{3x3}^{-1})^T = (R_1^{-1} S^{-1} R_2^{-1})^T$$

$$(M_{3x3}^{-1})^T = (R_1^T S^{-1} R_2^T)^T$$

$$(M_{3x3}^{-1})^T = R_2 (S^{-1})^T R_1$$

$$(M_{3x3}^{-1})^T = R_2 S^{-1} R_1$$

So, we get:

$$M_N = R_2 S^{-1} R_1 = (M_{3x3}^{-1})^T$$

Shading Transformation Matrices

View Space ⟵ Model Space

$$\boldsymbol{p'} = \boldsymbol{M}\boldsymbol{p}$$ ⟵ Position

$$\boldsymbol{n'} = (\boldsymbol{M}_{3x3}^{-1})^{\boldsymbol{T}}\,\boldsymbol{n}$$ ⟵ Normal

$\boldsymbol{M}$ = ModelView Matrix

Implementation Details: Flat Shading

IITK CS360A

- Vertex Shader:
  - Compute vertex position in clip space
  - Compute vertex position in eye space (posInEyeSpace)and send to fragment shader
  - Send View Matrix to fragment shader

Implementation Details: Flat Shading

- Fragment Shader:
  - Compute face normal and normalize it
    - normal = normalize(cross(dFdx(posInEyeSpace), dFdy(posInEyeSpace)));
  - Convert light to eye space from world space
    - Compute light vector (**L**) (from vertex position to light position) and normalize
    - Compute reflection vector (**R**) and normalize
      - R = normalize(-reflect(L,normal));
    - Compute view vector to camera from vertex position
      - V = normalize(-posInEyeSpace);
  - Finally compute the Phong shading lighting
    - Ambient + Diffuse + Specular
    - fragColor += Iamb + Idiff + Ispec;

Implementation Details: Gouraud Shading

- Vertex Shader:
  - Receive vertex positions and vertex normal sent from JavaScript code
  - Receive model, view, projection, and normal transformation matrices sent from JavaScript code
  - Compute vertex position in clip space
  - Compute vertex position in eye space (posInEyeSpace)and send to fragment shader
  - Transform normal to eye space by multiplying normal with normal transformation matrix and then normalize

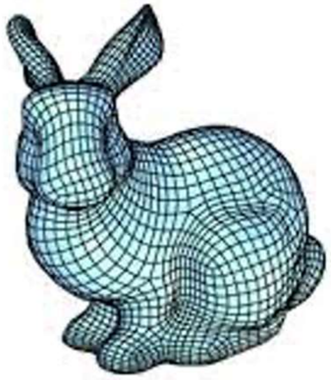
Implementation Details: Gouraud Shading

- Vertex Shader:
  - Convert light to eye space from world space
    - Compute light vector (**L**) (from vertex position to light position) and normalize
    - Compute reflection vector (**R**) and normalize
      - R = normalize(-reflect(L,normal));
    - Compute view vector to camera from vertex position
      - V = normalize(-posInEyeSpace);
  - Finally compute the Phong shading lighting
    - Ambient + Diffuse + Specular
    - vertexColor += Iamb + Idiff + Ispec;
    - Pass the vertexColor to fragment Shader where it will get interpolated automatically
- Fragment Shader:
  - Assign color to fragment that was sent from vertex shader

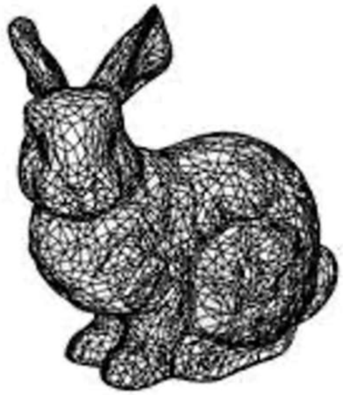
Triangular Meshes

Triangular Meshes

- In Graphics, when we import external objects, we typically load polygonal meshes
  - Quad mesh
  - Triangular mesh
- Before rendering, everything is converted to a triangular mesh
  - Triangles always form planer surface
  - Helps in rendering process



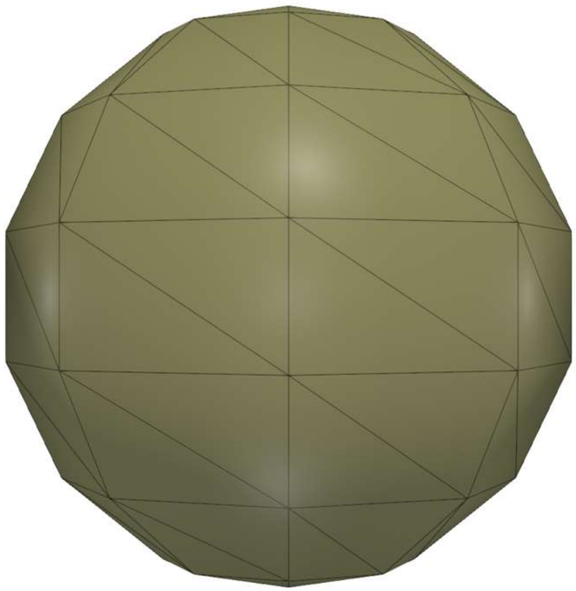
Quad mesh



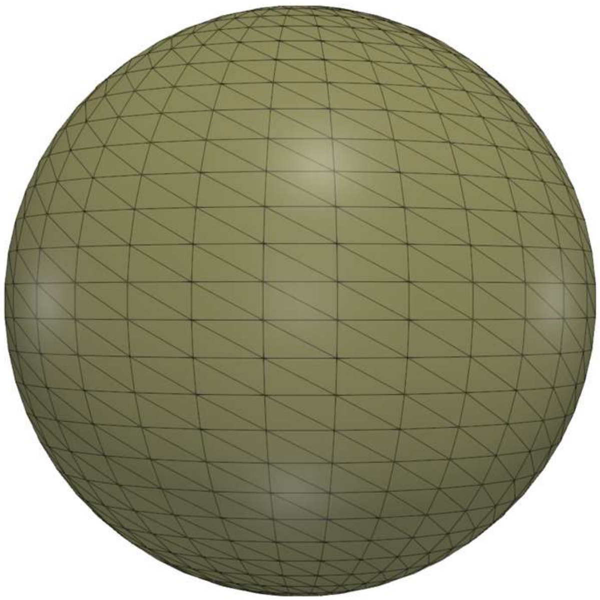
Triangular mesh

http://docs.mcneel.com

Triangular Meshes



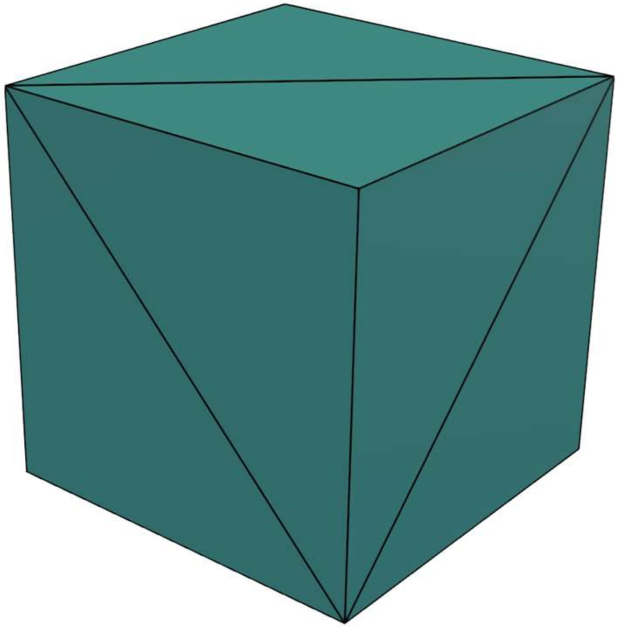
Low resolution polygonal mesh of a sphere



High resolution polygonal mesh of a sphere

Triangular Meshes

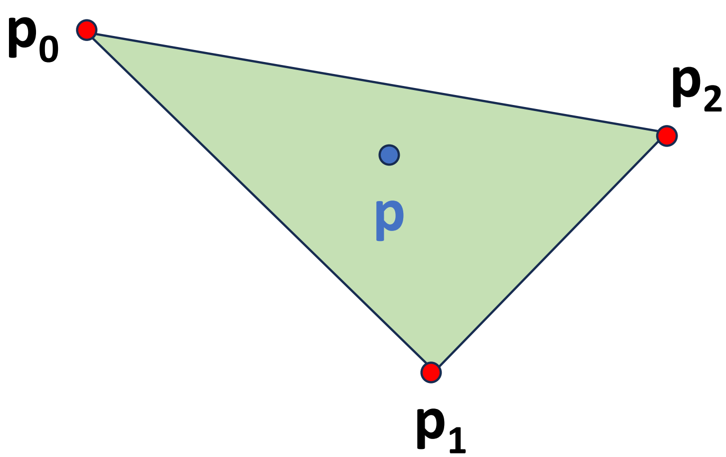
A Hollow Cylinder



A Cube

Triangles

- Defining a triangle is easy, we need three points in space
- How about a point inside the triangle?

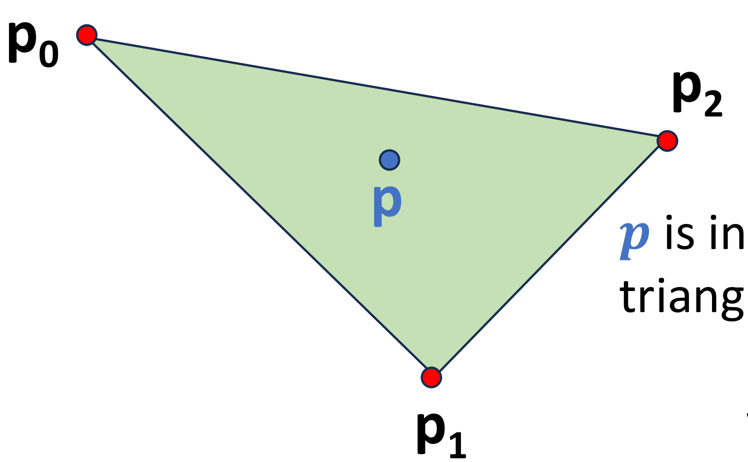


$$p = \alpha \boldsymbol{p_0} + \beta \boldsymbol{p_1} + \gamma \boldsymbol{p_2}$$

Linear combination of three vertices

Barycentric Coordinates of Triangles

- (α, β, γ) are called Barycentric Coordinates



$$\boldsymbol{p} = \alpha \boldsymbol{p_0} + \beta \boldsymbol{p_1} + \gamma \boldsymbol{p_2}$$

$$\alpha + \beta + \gamma = 1$$

$\boldsymbol{p}$ is inside the triangle when $0 \leq \alpha \leq 1 \quad 0 \leq \beta \leq 1 \quad 0 \leq \gamma \leq 1$

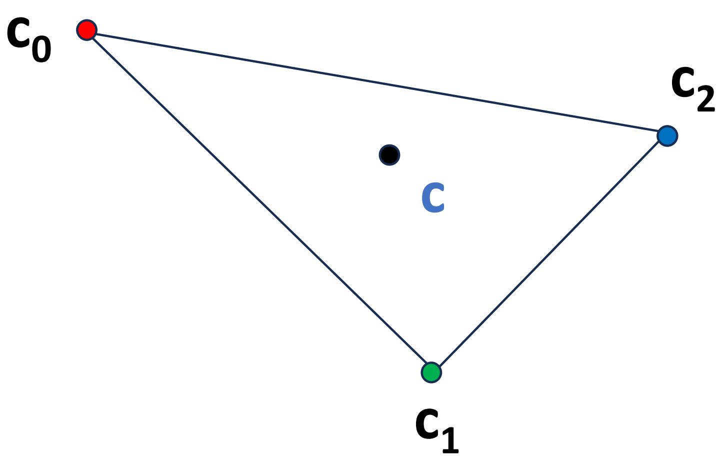
We can use these coordinates to interpolate values or find points inside a triangle easily

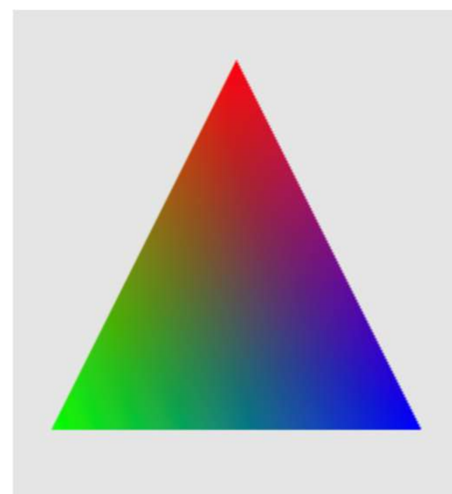
Barycentric Coordinates of Triangles

- (α, β, γ) are called Barycentric Coordinates

We can use these coordinates to interpolate values or find points inside a triangle easily

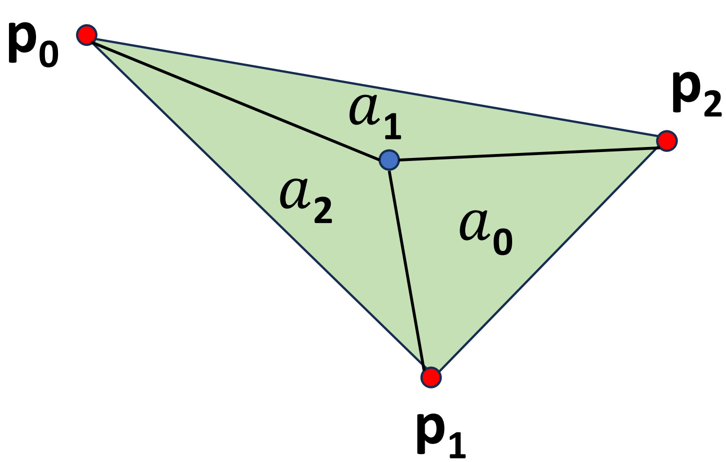
$$c = \alpha c_0 + \beta c_1 + \gamma c_2$$





Barycentric Coordinates of Triangles

- (α, β, γ) are called Barycentric Coordinates
- How do we compute these coordinates?
- Area of the triangle $= a$



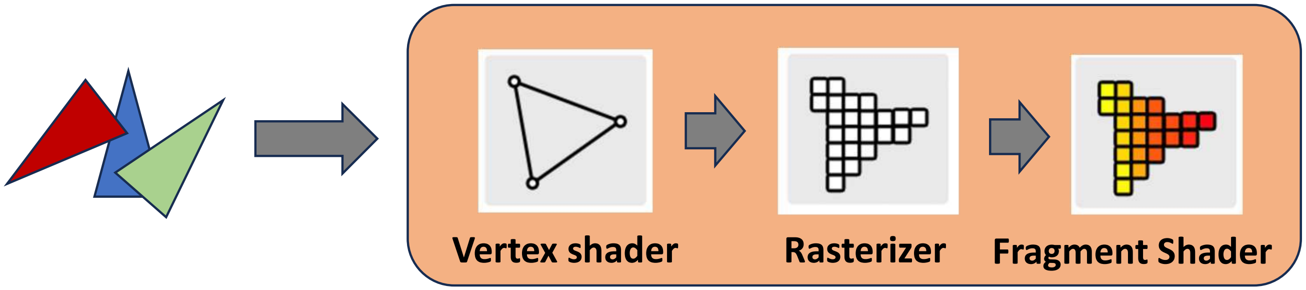
$$\boldsymbol{p} = \alpha \boldsymbol{p_0} + \beta \boldsymbol{p_1} + \gamma \boldsymbol{p_2}$$

$$\alpha + \beta + \gamma = 1$$

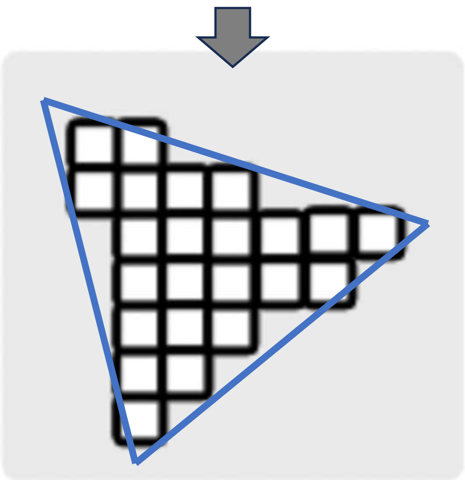
$$\alpha = \frac{a_0}{a} \quad \beta = \frac{a_1}{a} \quad \gamma = \frac{a_2}{a}$$

$$a = a_0 + a_1 + a_2$$

Revisit GPU Pipeline



Given vertices of triangles, GPU rasterizer will compute the barycentric coordinates of the points/fragments inside each triangle for us in the canonical view volume or Clip space!



GPU will also interpolate colors for those fragments inside each triangle by interpolating color values from the three vertices

WebGL: Color Interpolation in Shader

```
const vertexShaderCode = `#version 300 es
in vec2 aPosition;
in vec3 aColor;
out vec3 fColor;

void main() {
  fColor = aColor;
  gl_Position = vec4(aPosition,0.0,1.0);
}`;
```

Vertex Shader Code

```
const fragShaderCode = `#version 300 es
precision mediump float;
out vec4 fragColor;
in vec3 fColor;

void main() {
  fragColor = vec4(fColor, 1.0);
}`;
```

Fragment Shader Code

Triangular Meshes: Attributes

- **Vertex position**
- **Vertex Normal**
- **Texture Coordinate**
- **Color**

Vertex pos X	Vertex pos Y	Vertex pos Z
V_{x1}	V_{y2}	V_{z3}
V_{x2}	V_{y2}	V_{z2}
V_{x3}	V_{y3}	V_{z3}

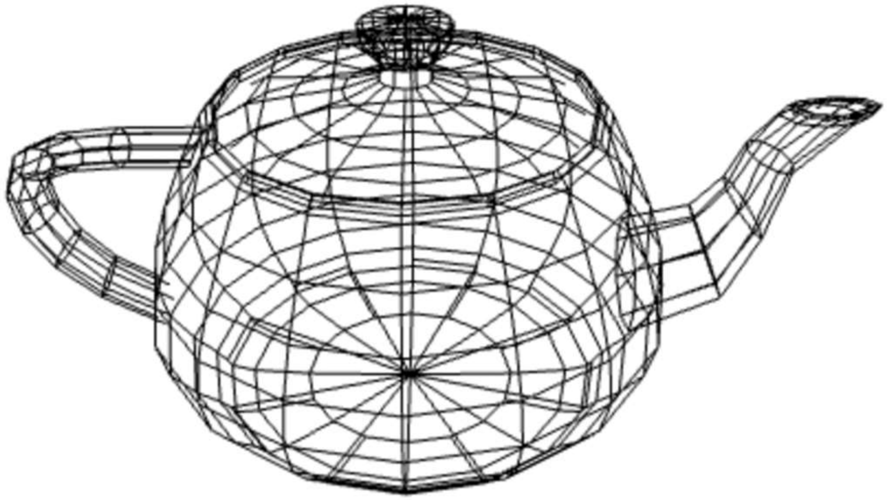
Vertex normal X	Vertex normal Y	Vertex normal Z
N_{x1}	N_{y2}	N_{z3}
N_{x2}	N_{y2}	N_{z2}
N_{x3}	N_{y3}	N_{z3}

Vertex Color R	Vertex Color G	Vertex Color B
C_r	C_g	C_b
C_r	C_g	C_b
C_r	C_g	C_b

Vertex texture U	Vertex normal V
T_{x1}	T_{y2}
T_{x2}	T_{y2}
T_{x3}	T_{y3}

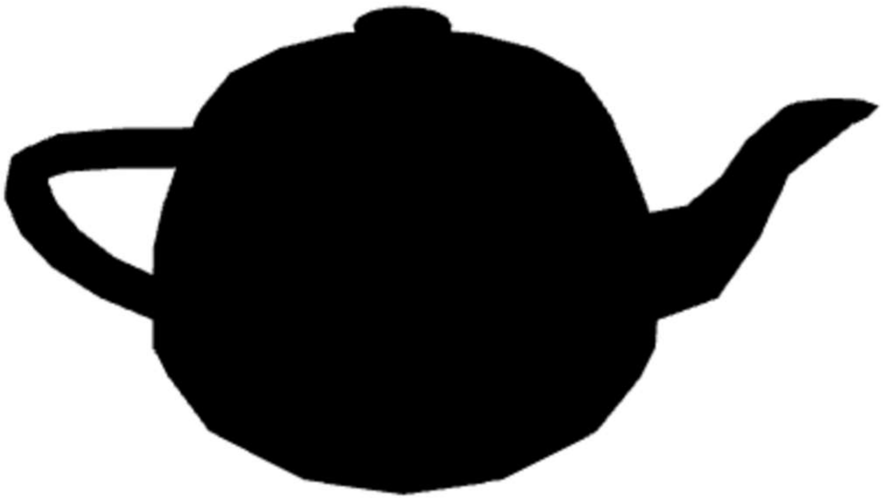
Loading Triangular Meshes

Point Rendering

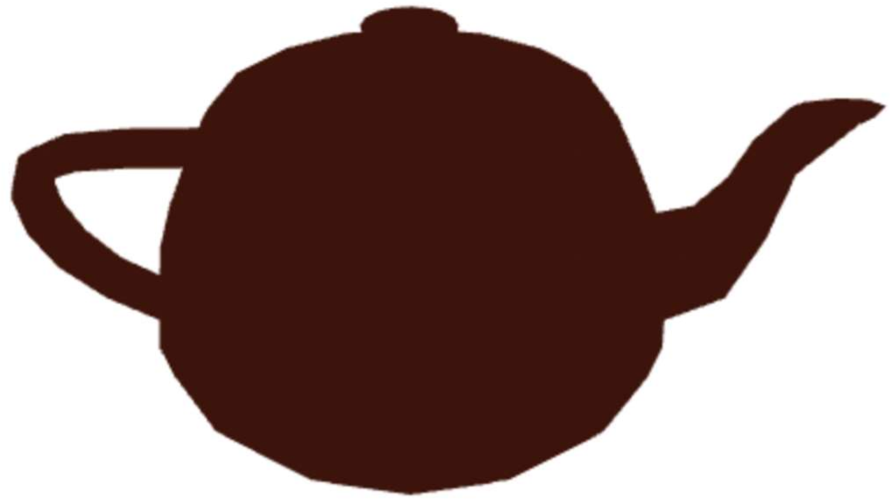


Line Rendering

Loading Triangular Meshes

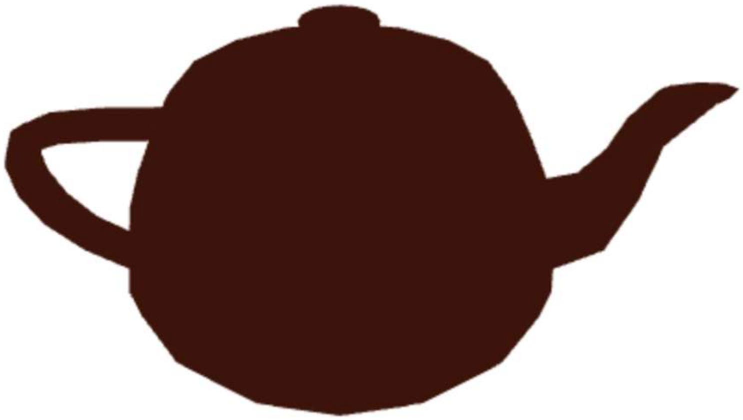


No Color and Shading



Ambient Color

Loading Triangular Meshes



Ambient Color

Ambient + Diffuse Color

Ambient + Diffuse + Specular Color

Loading Triangular Meshes

- We will use JSON files representing polygonal (triangular) mesh objects for rendering
- Rendering technique of such objects is exactly same as rendering a sphere or a cube or any other polygonal mesh

Examples: Different Types of Materials

Shiny Gold Statue

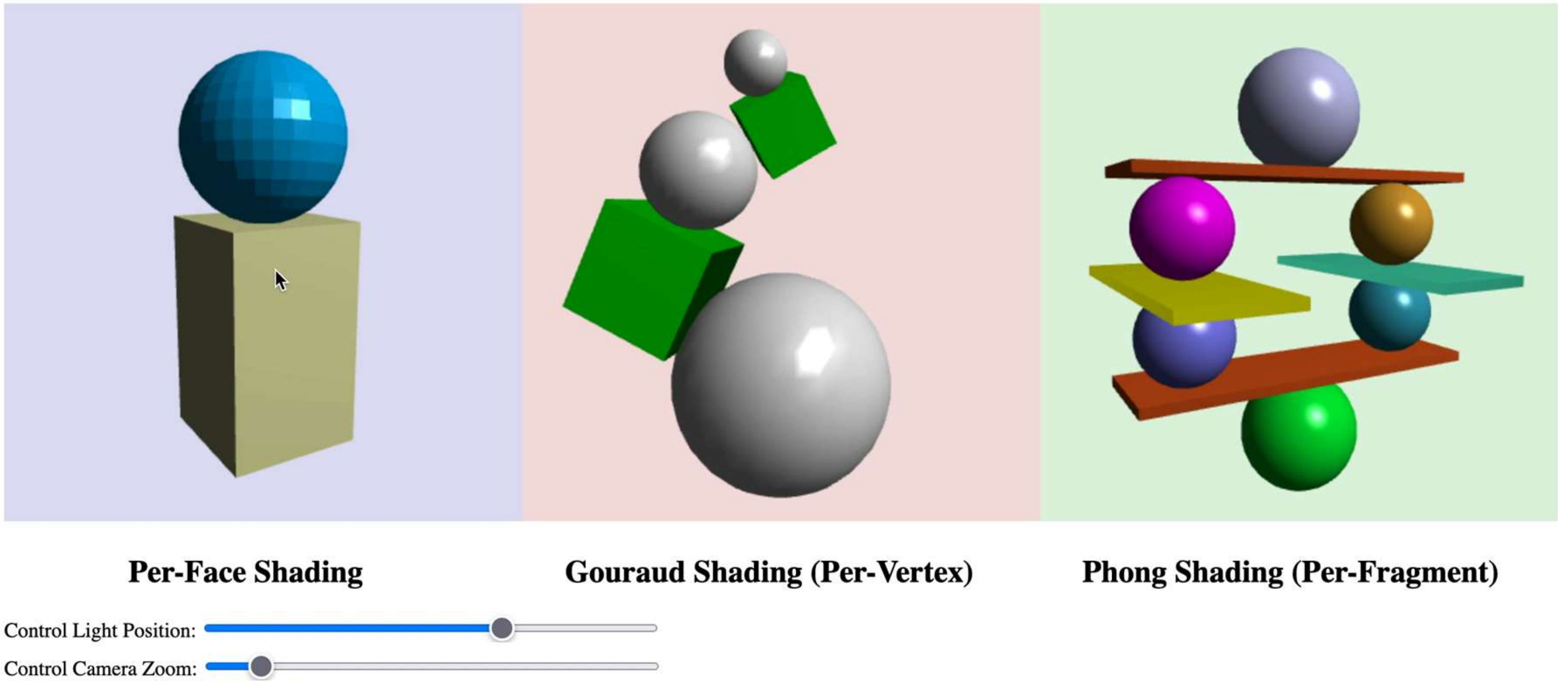
Rough Bronze Statue

Demo Code

- Demonstrate Code for Loading JSON Object File
  - simpleLoadObjMesh.js, simpleLoadObjMesh.html

Assignment 2: Due: Sept 9th 11:59pm

- Simple 3D Object rendering with 3 different shading models
  - Flat Shading, Gouraud Shading, Phong Shading
- Handling 3 viewports and allowing exclusive interactions on them
- Using sliders allow light movement and camera zooming



Structure of Your drawScene() Function

- **Setup viewport 1**
- shaderProgram = flatShaderProgram;
- gl.useProgram(shaderProgram);
- Now setup all shader variables, attributes, enable attributes, and setup uniforms and then draw scene

- **Setup viewport 2**
- shaderProgram = perVertshaderProgram;
- gl.useProgram(shaderProgram);
- Now setup all shader variables, attributes, enable attributes, and setup uniforms and then draw scene

- **Setup viewport 3**
- shaderProgram = perFragshaderProgram;
- gl.useProgram(shaderProgram);
- Now setup all shader variables, attributes, enable attributes, and setup uniforms and then draw scene