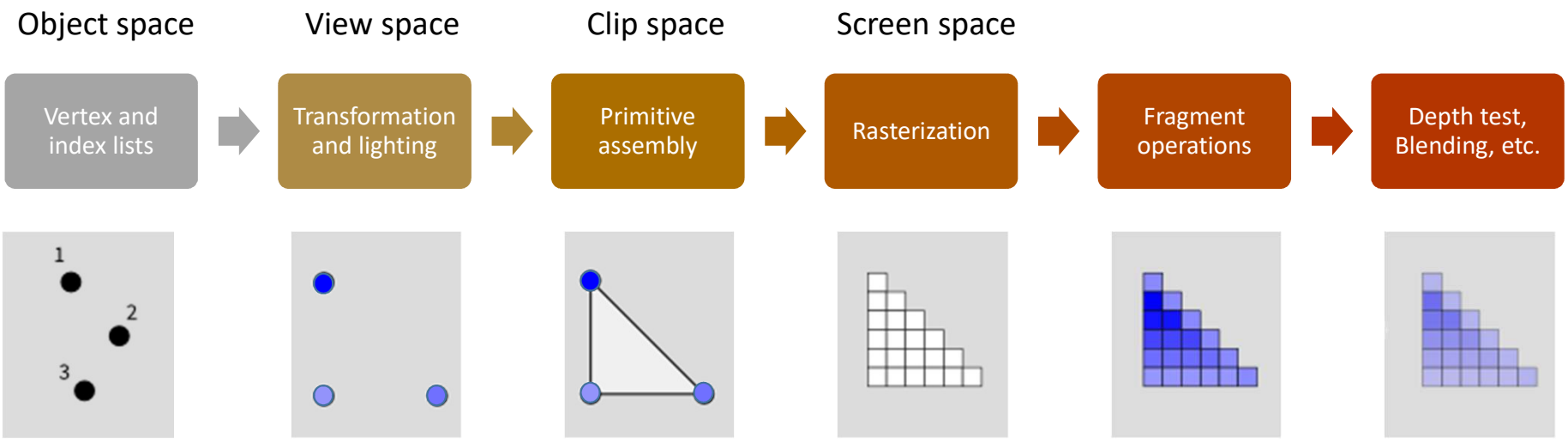
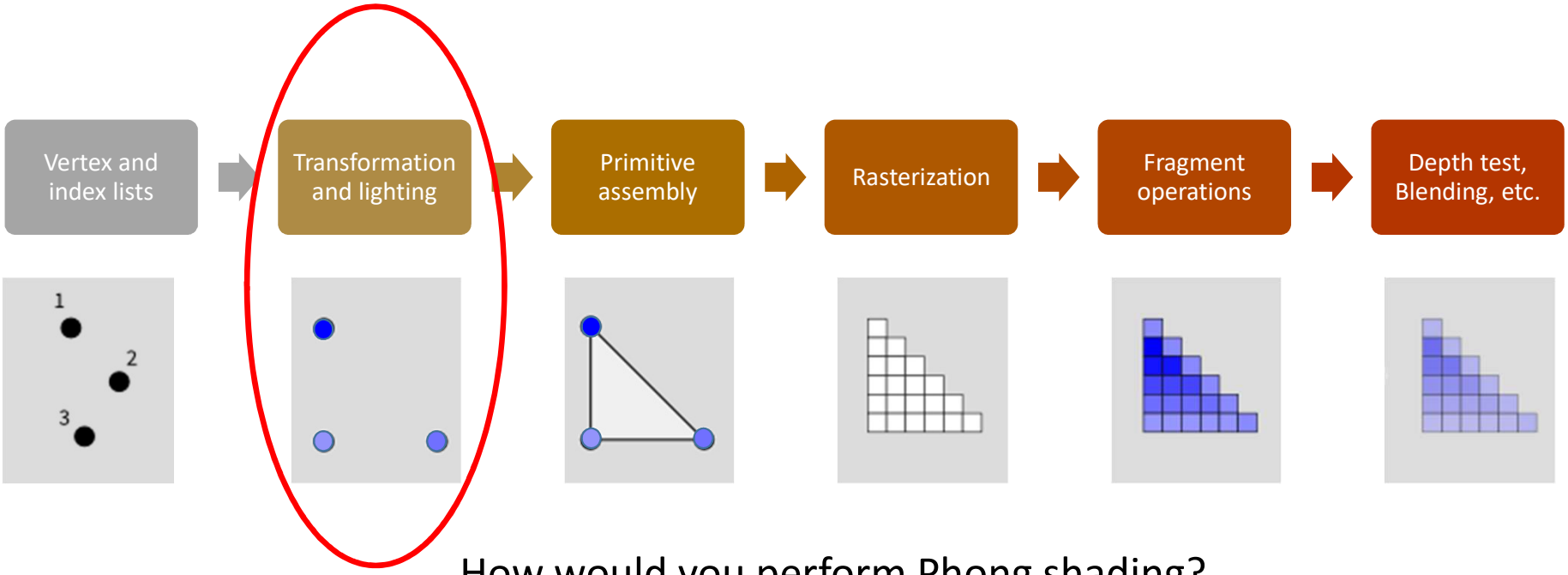
The OpenGL Shading Language

Rahul Arora

The Fixed Functionality Rendering Pipeline



The Fixed Functionality Rendering Pipeline

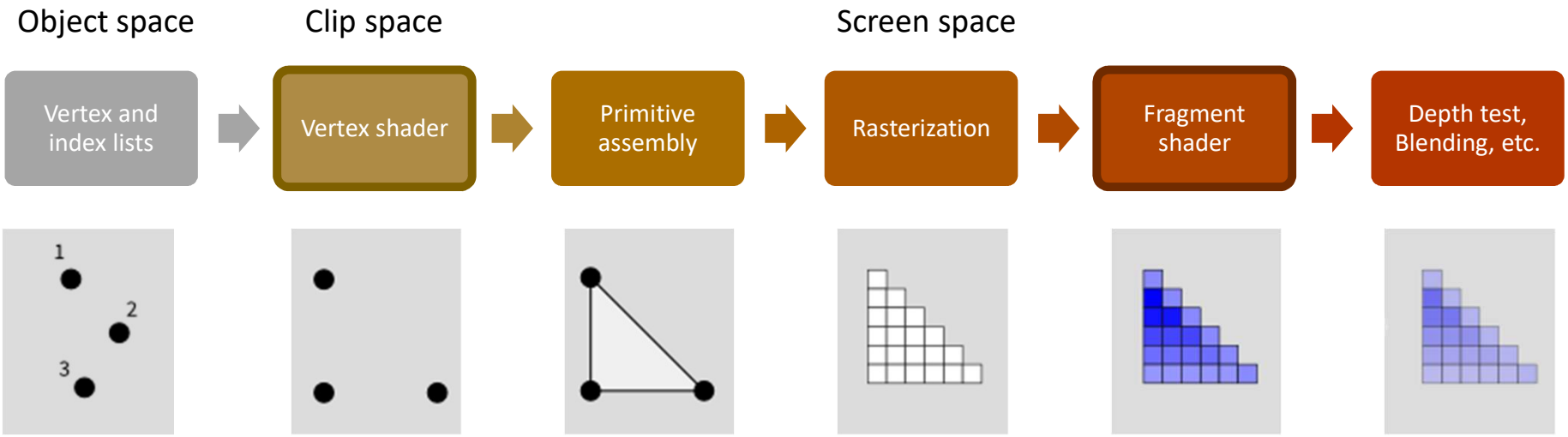


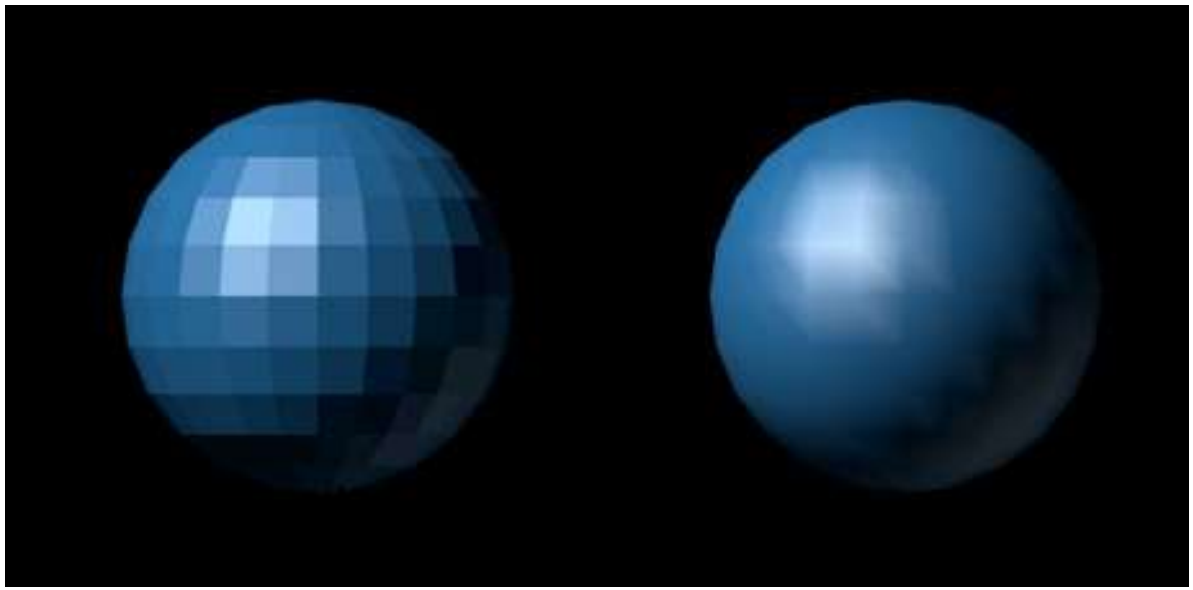
How would you perform Phong shading?

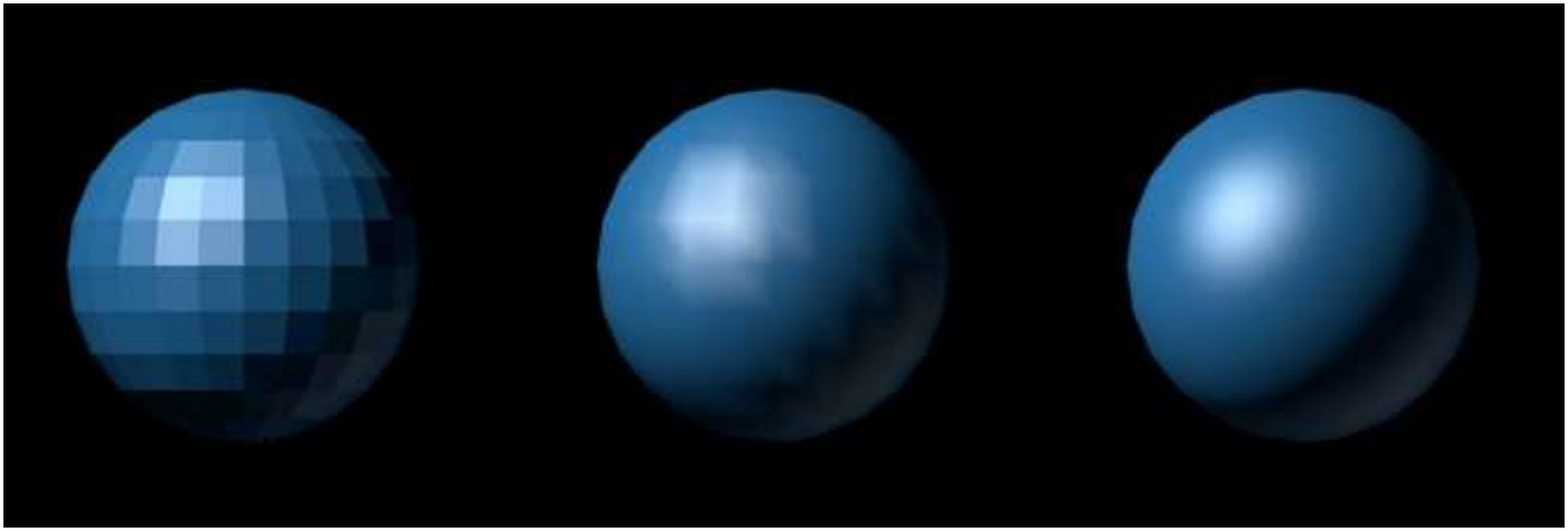
Problems?

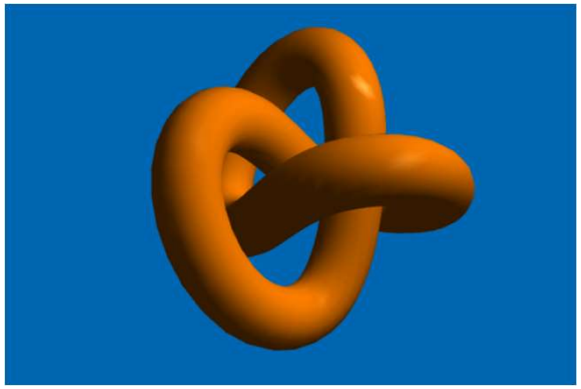
- Only Phong illumination model supported.
- Only a few *pre-programmed* shading models supported.
  - Flat shading
  - Gouraud shading
- No per-fragment lighting.
- No screen space shading.

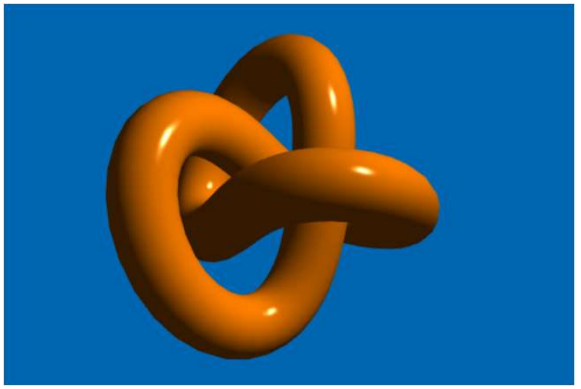
The Programmable Pipeline

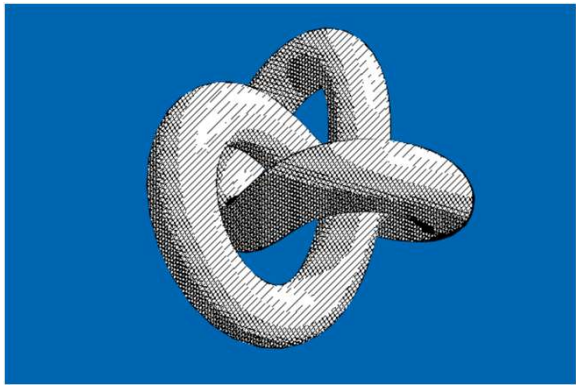


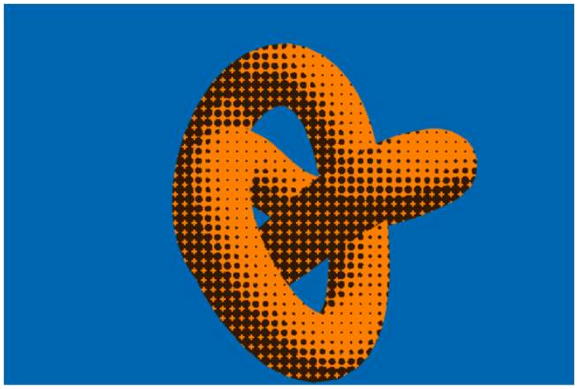




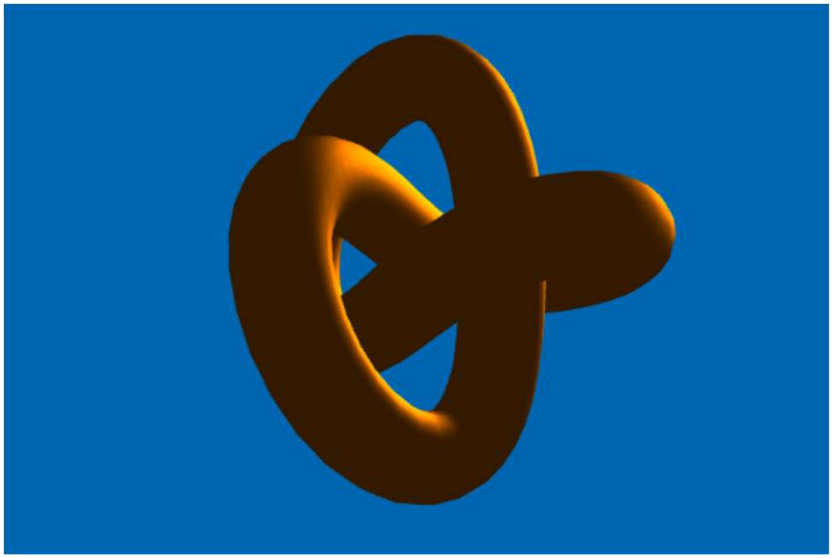


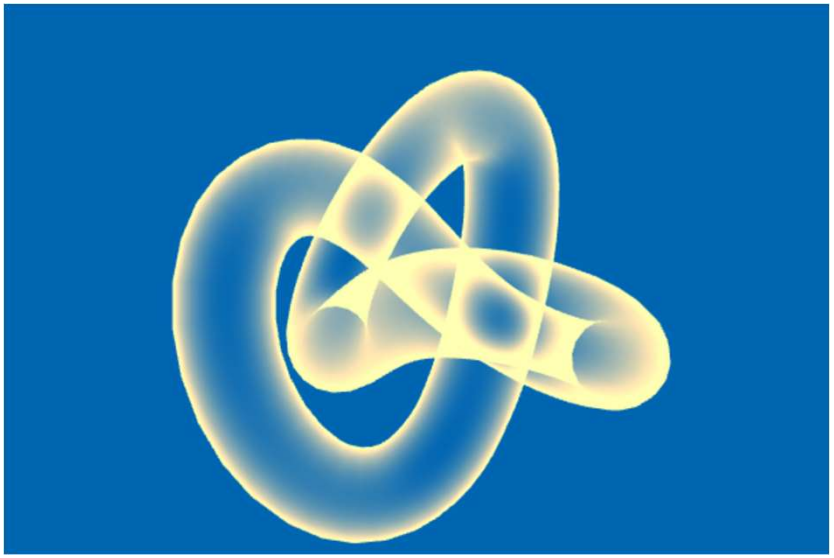






Or Use an Entirely Different Illumination Model!



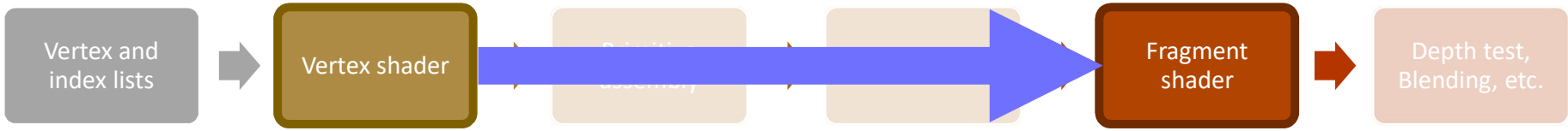


GLSL: The OpenGL Shading Language

- C-like programming language.
- Both vertex and fragment shaders are written in GLSL.
- OpenGL requires certain outputs from the shaders.
- But you can add additional ones for doing cool things.

GLSL Qualifiers

- `uniform`
  - Remains the same throughout the execution of the shader
- `attribute`
  - Per-vertex data
- `varying`
  - Per-fragment data
  - Automatically interpolated by fixed stages of the pipeline



GLSL: Data types

- Scalars
  `float, int, bool`
- Vectors
  Float: `vec2, vec3, vec4`
  Integer: `ivec[2|3|4]`
  Boolean: `bvec[2|3|4]`
  Accessing data: `vert[0], vert.x, vert.r, vert.xyz, vert.rgba`
- Matrices
  Floating point: `mat2, mat3, mat4`
- Textures
  `sampler1D, sampler2D, sampler3D`
  Accessing data: `texture(u, v)`

GLSL: Built-in Functions

- Trigonometric
  `cos, sin, tan`, etc.
- Exponentiation
  `exp, log, sqrt`, etc.
- Common floating-point
  `abs, floor, min, clamp`, etc.
- Geometric
  `length, dot, cross, normalize, reflect`, etc.

Built-in Inputs and Outputs

Vertex Shader

Input: _____

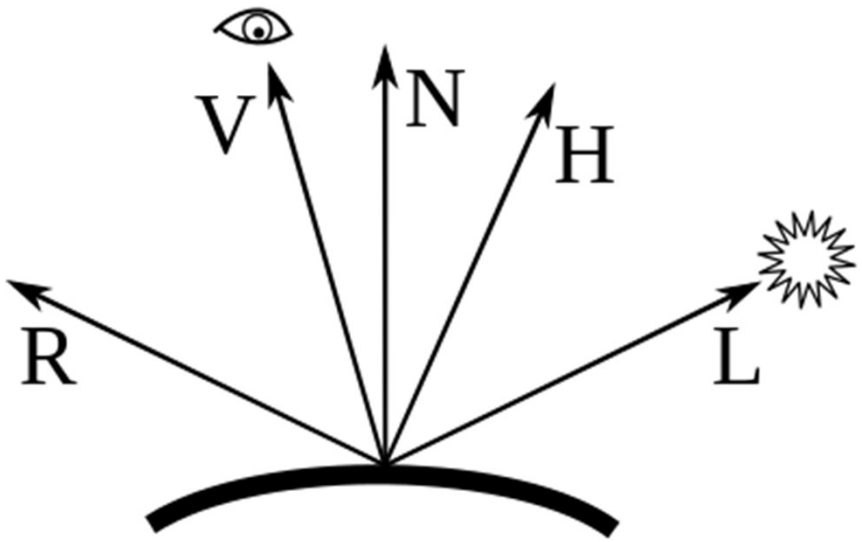
Output: `gl_Position (vec4)`

Fragment Shader

Input: `gl_FragCoord (vec4)`

Output: `gl_FragColor (vec4)`

Recall the Phong Illumination Model



shininess

Ambient term

$$I_{\mathrm{p}} = k_{\mathrm{a}} i_{\mathrm{a}} + \sum_{m \in \text{ lights}} \left(k_{\mathrm{d}} (\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + k_{\mathrm{s}} (\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,\mathrm{s}} \right)$$

Diffuse term

Specular term

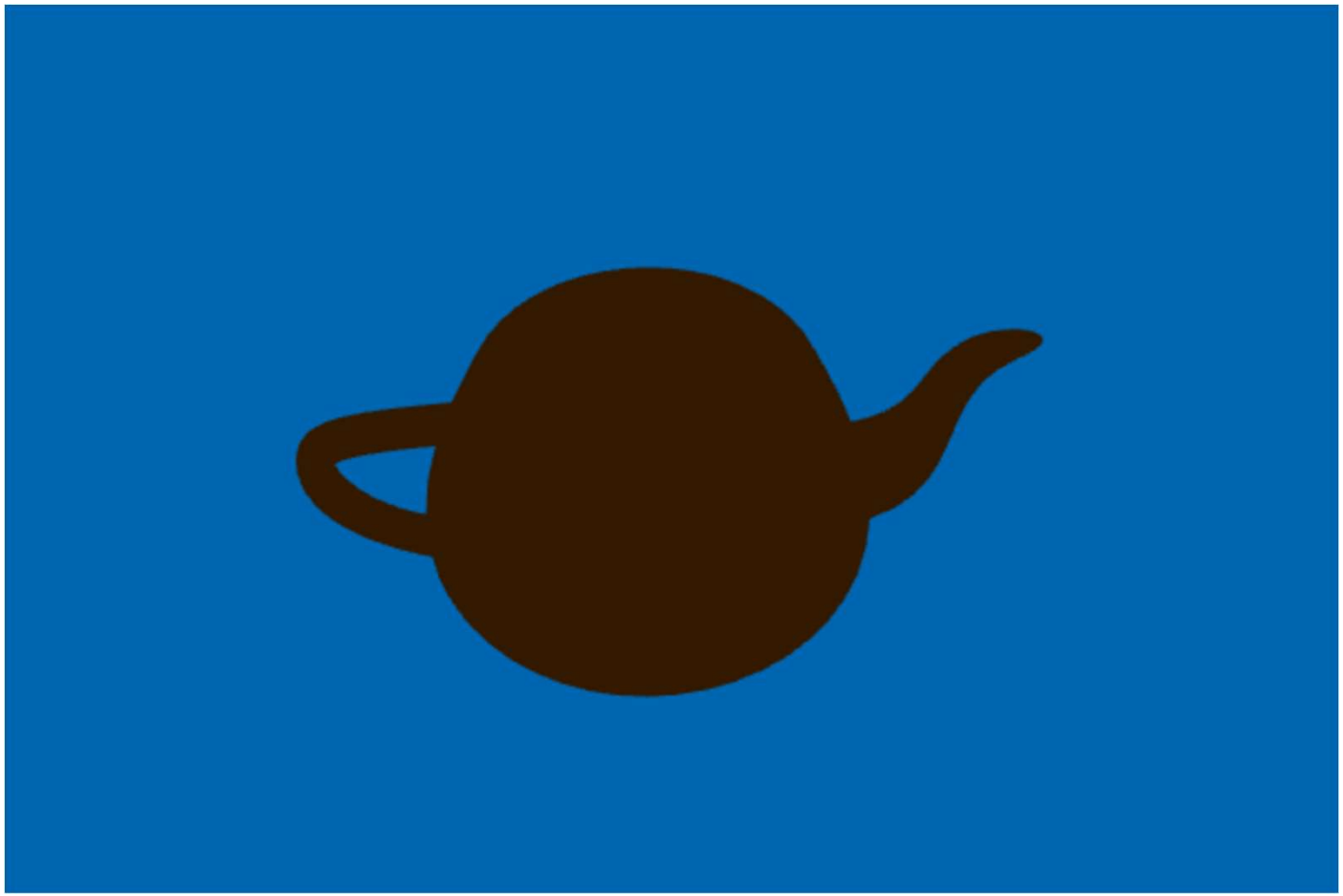
Example

```
// Vertex Shader
attribute highp vec4 vertex;
uniform mediump mat4 modelview;
uniform mediump mat4 projection;

void main()
{
   gl_Position =
   projection * modelview * vertex;
}
```

```
// Fragment Shader
uniform vec3 ambientColor;

void main()
{
   gl_FragColor =
   vec4(ambientColor, 1);
}
```



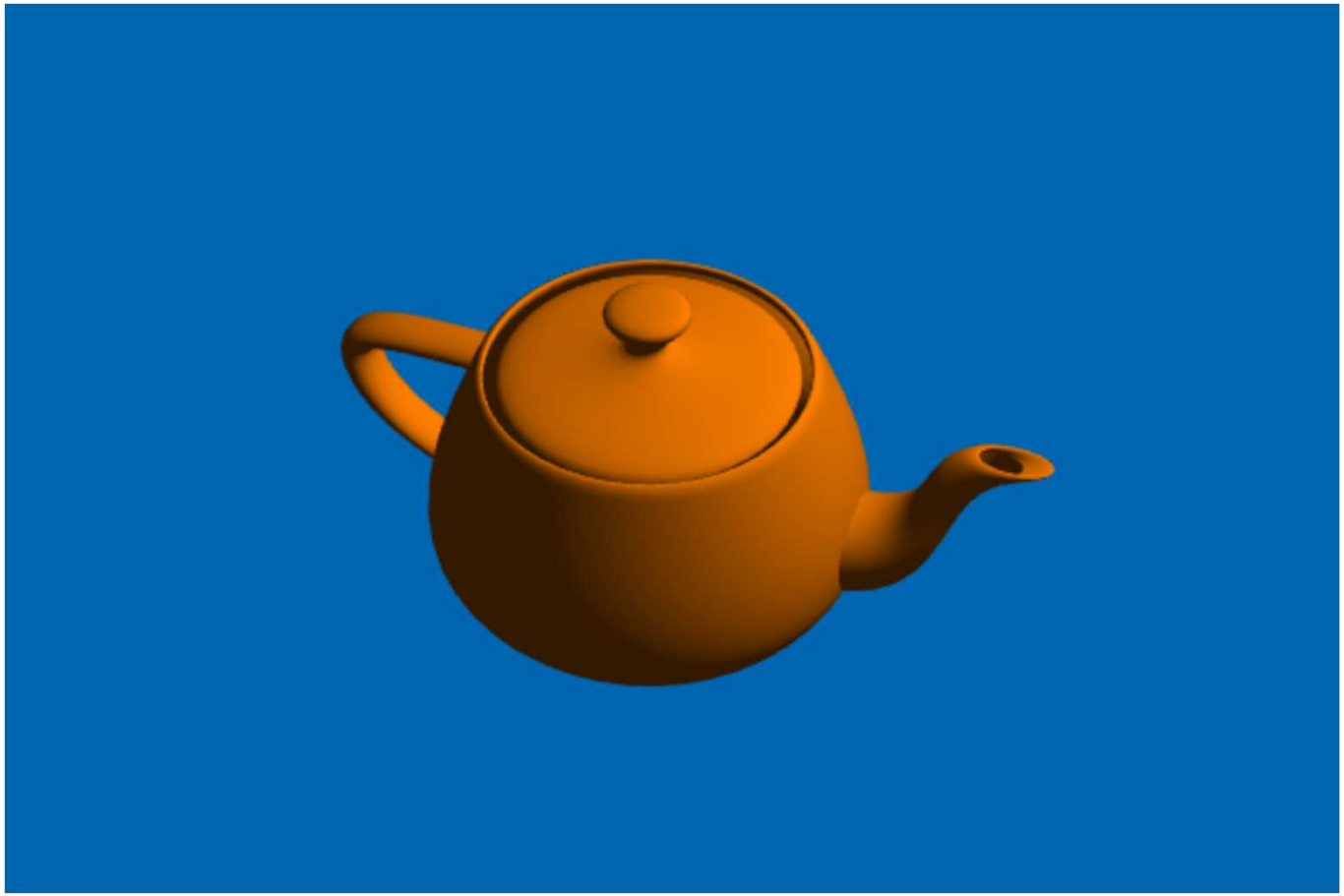
Example

```
// Vertex Shader
attribute highp vec4 vertex;
uniform mediump mat4 modelview;
uniform mediump mat4 projection;
varying vec3 normalInterp;

void main()
{
    gl_Position =
    projection * modelview * vertex;
    normalInterp = <your_code>;
}
```

```
// Fragment Shader
uniform vec3 ambientColor;
uniform vec3 diffuseColor;
varying vec3 normalInterp;
uniform vec3 lightPos;

void main()
{
    // normalize normalInterp first
    vec3 N = <your_code>;
    // get light direction
    vec3 L = <your_code>;
    float lambertian = <your_code>;
    gl_FragColor =
    vec4(ambientColor + lambertian * diffuseColor, 1);
}
```



Example

```
// Vertex Shader
attribute highp vec4 vertex;
uniform mediump mat4 modelview;
uniform mediump mat4 projection;
varying vec3 normalInterp;

void main()
{
    gl_Position =
    projection * modelview * vertex;
    normalInterp = <your_code>;
}
```

```
// Fragment Shader
...
uniform vec3 specularColor;
uniform float shininess;

void main()
{
    // find N, L, lambertian
    ...
    // use lambertian and shininess to find
    // specular intensity
    float specular = <your_code>;
    gl_FragColor = vec4(ambientColor + lambertian *
    diffuseColor + specular * specularColor, 1);
}
```

Questions?