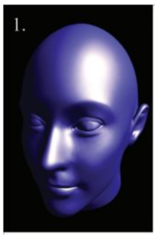
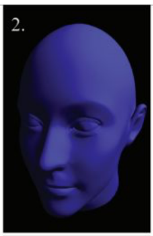
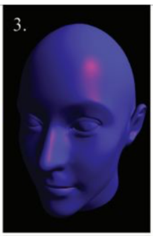
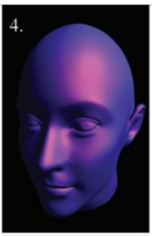
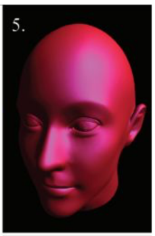
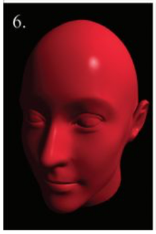
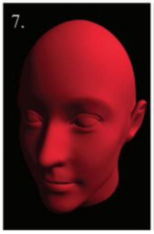
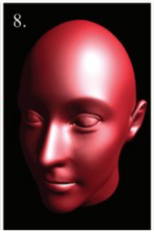
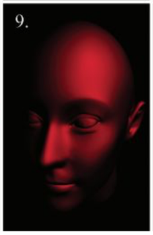
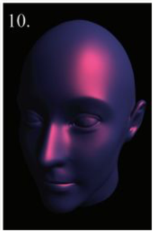
Homework 4 - Lighting
Solution

1. **Define the three components of light used in the Phong lighting model: diffuse, specular, and ambient.**

   Diffuse: The "Lambertian Reflectance" the matte shading on a model $I * (K_d * \max(0, n * v_L))$
   Specular: "Shininess", mirror like reflections of an object $I * (K_s * \max(0, n * v_H)^n)$
   Ambient: Constant lighting value to simulate ambient light bouncing off all objects in a scene. $k_a * I$

2. **Match the images below to the properties defined in the table. Write the image number in the correspondent table row.**


1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

	Diffuse Color	Specular Color	Specular Exponent (giving the size of the highlight)	Image Number
(a)	Red	Black	Any	7
(b)	Blue	Red	Small	3
(c)	Blue	White	Big	1
(d)	Darker Blue	Red	Medium-Big	10
(e)	Blue	Red	Big	4
(f)	Blue	Black	Any	2
(g)	Black	Red	Big Red	9
(h)	Red	White	Small	6
(i)	Red	White	Big	8
(j)	Red	Blue	Big	5

3. **Consider a scene with a single sphere of radius 1 centered at the origin (0,0,0). The sphere's ambient coefficient is k_a = RGB(0.1,0.1,0.1), diffuse coefficient k_d = RGB(0.5,0.5,0.5), and specular coefficient k_s = RGB(0.5,0.5,0.5), with specular exponent s = 2. There is a light at position p_1 = (0,2,0) with color c_1 = RGB(1.0, 0.0, 0.0). The camera is located at position p_2 = (3,1,0). Calculate the Phong Illumination for the sphere fragment located at position p_3 = (0,1,0).**

The ambient, diffuse, and specular coefficients are given as RGB colors. Sometimes they are given as scaler values and sometimes RGB colors, don't be confused when you see it differently. The light color is given as c_1, while in question 1 above it is stated as I. The normal of the sphere at the fragment is $v_N = p3 - origin = (0, 1, 0)$. The light vector is $v_L = p_1 - p_3 = (0, 1, 0)$. Since the light and normal are aligned, the reflection vector is $v_R = (0, 1, 0)$. The eye/camera vector is $v_E = p_2 - p_3 = (3, 0, 0)$. All these vectors need to be normalized to unit length before using in the dot products below.

(a) **Calculate the ambient portion of the color of this fragment.**
$= k_a * i$
= RGB(0.1, 0.0, 0.0)

(b) **Calculate the diffuse portion of the color of this fragment.**
$= k_d * i * \max(0, v_L * v_N)$
$= k_d * i * \max(0, <0,1,0> * <0,1,0>)$
$= k_d * i * 1$
= RGB(0.5, 0.0, 0.0)

(c) **Calculate the specular portion of the color of this fragment.**
$= k_s * i * max(0, v_R * v_E)$
$= k_s * i * max(0, <0,1,0> * <1,0,0>)$
$= k_s * i * 0$
= RGB(0.0, 0.0, 0.0)

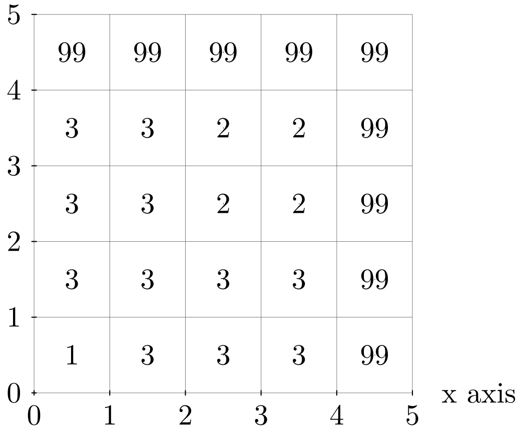
(d) **Calculate the total color of this fragment.**
= RGB(0.1, 0.0, 0.0) + RGB(0.5, 0.0, 0.0) + RGB(0.0, 0.0, 0.0)
= RGB(0.6, 0.0, 0.0)

(e) **Now suppose the camera moves to p_4 = (0,3,0). What is the total color of the fragment?**
RGB(0.1, 0.0, 0.0) + RGB(0.5, 0.0, 0.0) + $k_s * i * max(0, v_R * v_E)$
= RGB(0.6, 0.0, 0.0) + $k_s * i * max(0, <0,1,0> * <0,1,0>)$
= RGB(0.6, 0.0, 0.0) +$k_s * i * 1$
= RGB(0.6, 0.0, 0.0) + RGB(0.5, 0.0, 0.0)
= RGB(1.1, 0.0, 0.0) (Which is really RGB(1.0, 0.0, 0.0) as 1.0 is the maximum value)

4. **Consider an orthographic scene with objects listed below being rendered at 5 x 5 pixels with with the view plane at z=0 (near=0, far=99). Show the state of the z-buffer (with numbers) once these squares have been rendered (without using anti-aliasing or sub-sampling). Note that the pixels are drawn such that the z-buffer values are stored halfway between integers (look at the labels below the z-buffer).**

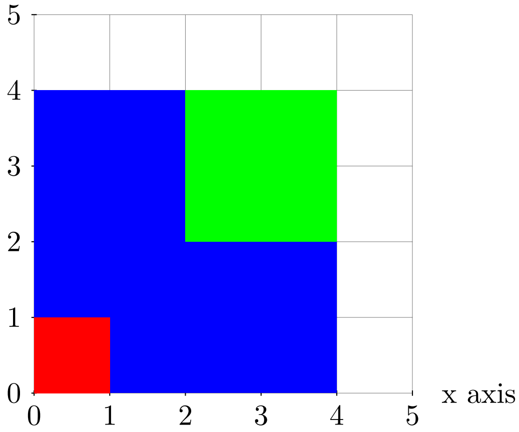
   Red square with vertices: (0,0,1), (1,0,1), (1,1,1), (0,1,1)
   Green square with vertices: (2,2,2), (4,2,2), (4,4,2), (2,4,2)
   Blue square with vertices: (0,0,3), (4,0,3), (4,4,3), (0,4,3)

   Z Buffer:


5
4
3
2
1
0
0
1
2
3
4
5
x axis

99	99	99	99	99
3	3	2	2	99
3	3	2	2	99
3	3	3	3	99
1	3	3	3	99

   Color Buffer:



5. Consider the following vertex and fragment shaders:

```
// Vertex Shader
uniform mat4 u_NormalMatrix;
uniform mat4 u_ModelMatrix;
uniform mat4 u_ViewMatrix;
uniform mat4 u_ProjectionMatrix;

attribute vec4 a_Position;
varying   vec3 v_Position;
attribute vec4 a_Normal;
varying   vec3 v_Normal;

void main() {
    v_Normal = normalize(vec3(u_NormalMatrix * a_Normal));
    v_Position = vec3(u_ModelMatrix * a_Position);
    gl_Position = u_ProjectionMatrix * u_ViewMatrix * u_ModelMatrix * a_Position;
}`;

// Fragment Shader
varying vec3 v_Position;
varying vec3 v_Normal;
uniform vec3 u_LightPos;
uniform vec3 u_LightColor;

void main() {
    vec3 l = normalize(u_LightPos - v_Position);
    float nDotL = max(dot(l, v_Normal), 0.0);
    gl_FragColor = vec4(u_LightColor * nDotL);
}
```

(a) **In the vertex shader, what is the normal matrix and why do we need to multiply it by the normal vector?**

The normal matrix is the inverse transpose of the model matrix. Analogous to using the ModelMatrix to transform vertex positions, this is the matrix we multiply by the normal vector to get the final normal of the model. It might seem we could just use the ModelMatrix for this, but thats not how the math works out. It might also seem we could just calculate this in the vertex shader rather than passing it from javascript, but that would be inefficient since we would have to recalculate it on every vertex. For a large model this would slow things down. So its not really about javascript versus GLSL, its just that we only want to calculate it once.

(b) **In the vertex shader, why do we multiply the vertex position by the model matrix?**

The positions stored in the a_Position attribute are in object coordinates. The light is specified in world coordinates. Thus we want to transform from object to world coordinates, so that we know where the object position is in the same coordinate frame as the lighting. We pass this value to the fragment shader with v_Position, since thats where lighting is calculated in this example. Its not necessary to choose World Coordinates. We could instead move the light into Object Coordinates, but we dont have that matrix handy. Or we could move them both into Eye Coordinates. However World Coordinates is the easiest to think about and most often used.

(c) **What is the fragment shader computing?**

The fragment shader is displaying the lambertian shading, or diffuse component of the Phong light model.