# Exercise 2

Kevin Serrano, Gianni Scarnera

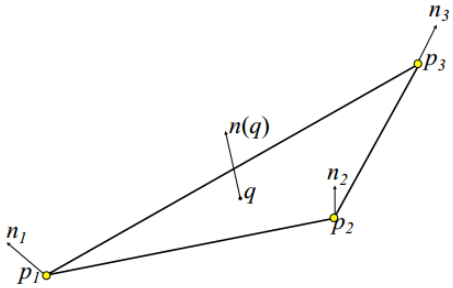
October 16, 2012

## 2.1.1 Normals and Materials

To get the material and the normal at the intersection point with the plane is very easy because the plan is a flat and infinite so the normal is the same everywhere on the plane. So to get the normal at the intersection we can use $iData \rightarrow normal = getNormal()$ and for the material : $iData \rightarrow material = getMaterial()$.

For the triangle is a little bit complicated because we have to interpolate the normals of vertices to get the normal at the intersection.

Figure 1: Normals of vertices and the interpolation at point q




It's easy to interpolate the normal $n(q)$at a point $q$ inside the triangle with barycentric coordinates. We can take our 3 values $s_1, s_2, s_3$ at point $q$ with the formulas of precedent exercise to compute the normal at point $q$ and normalized.

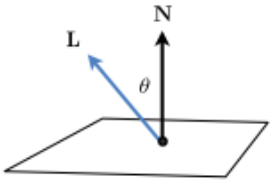
$$n(q) = \frac{s_1\mathbf{n_1} + s_2\mathbf{n_2} + s_3\mathbf{n_3}}{||s_1\mathbf{n_1} + s_2\mathbf{n_2} + s_3\mathbf{n_3}||}$$

where $\mathbf{n_1}, \mathbf{n_2}, \mathbf{n_3}$ are the normal at each vertices. Now we can get the normal and the material at the intersection of the triangle with $iData \rightarrow normal = n(q)$ and for the material : $iData \rightarrow material = getMaterial()$

## 2.1.2 Diffuse Lighting Model

The diffuse reflection depends on surface orientation and light position but is independent of camera position. The brightness depends of the angle between the normal at intersection point and the source light. Now for each light in

Figure 2: angle between the normal (N) and the vector from intersection point to source light (L)



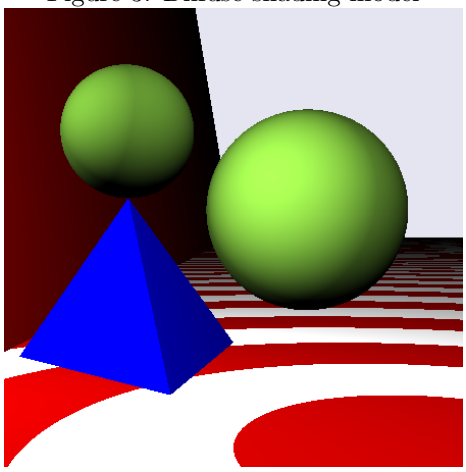

the scene, we have to compute the intensity at the intersection point with the

diffuse formula

$$I = I_p k_d cos(\theta) = I_p k_d (\mathbf{N} \cdot \mathbf{L})$$

where $I_p$ is the color of the light source and $k_d$ the diffuse material color. To implement this formula, we have to take the vector of source light in the object $scene \rightarrow getLight()$ and then use a iterator to calculate the intensity $I$ of each light sources and then the resulting of different light sources are added up and clamped to [0,1] at the end. All vectors are normalized. Then after calculated and normalise the vector $L$ for each light source, we have to check if the light source and the normal point in the same direction, i.e $(\mathbf{N} \cdot \mathbf{L} > 0)$ otherwise the light does not contribute to diffuse shading. Then it's possible to calculate the intensity $I$ if we use the element-wise multiplication with $I_p$ and $k_d$ because they are Vector4. Then $I_p = lightsource \rightarrow getColor()$ and $k_d = iData \rightarrow material \rightarrow diffuse$

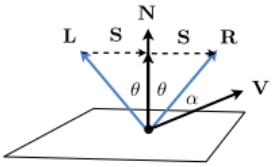
Figure 3: Diffuse shading model



## 2.1.3 Phong Lighting Model

The phong lighting model is the combination of the ambient, diffuse and specular reflection. $\mathbf{L}, \mathbf{N}$ are the same vector than before and $\mathbf{R}$ is the reflected vector

Figure 4: Specular reflection




from the light $\mathbf{L}$ with $\mathbf{N}$ with the same angle $\theta$, and $\mathbf{V}$ is the normalised vector from the intersection to the camera position. So for each light source, we can calculate the vector $\mathbf{R}$ with the formula

$$\mathbf{R} = 2\mathbf{N}cos(\theta) - \mathbf{L} = (2\mathbf{N}(\mathbf{N} \cdot \mathbf{L}) - \mathbf{L})$$

Then we can calculate the intensity of specular reflection for each light sources with the formula

$$I_p k_s cos(\alpha)^n = I_p k_s (\mathbf{R} \cdot \mathbf{V})^n$$

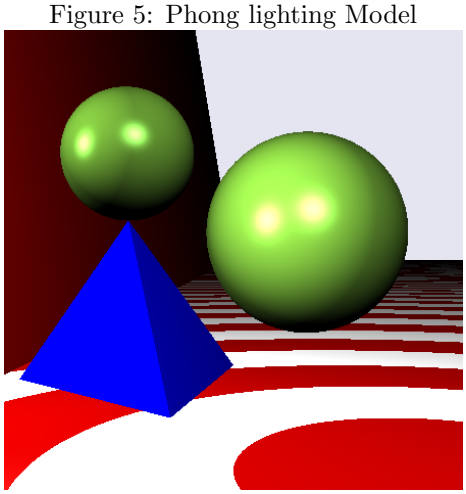
where $I_p$ is the color of the light, $k_s$ the specular color of the material, and $n$ the shininess of the material. So

- $I_p = sourcelight \rightarrow getColor()$
- $k_s = iData \rightarrow material \rightarrow specular$
- $n = iData \rightarrow material \rightarrow shininess$

The intensity of the ambient is $I_{ambient} = I_a k_a$ where $I_a$ is the scene ambient light color, i.e $I_a = scene \rightarrow getAmbient()$ and $k_a$ is the ambient reflection color of the intersected objects material, i.e $k_a = iData \rightarrow material \rightarrow ambient$. The intensity of the diffuse shading is $I_{diffuse}$ and it's the same formula as the part before. Then we can add the intensity of the ambient, diffuse and specular reflection to get the Phong lighting Model

$$I = I_{ambient} + I_{diffuse} + I_{specular}$$

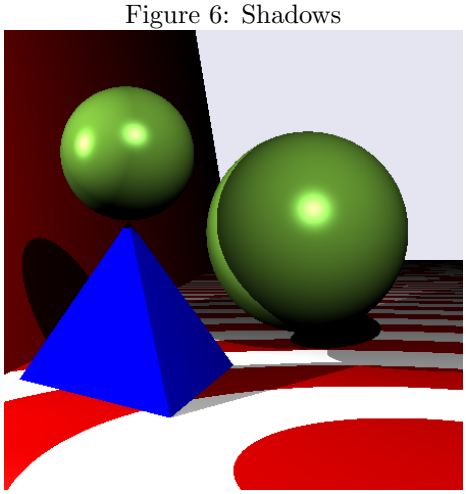
Figure 5: Phong lighting Model



### 2.2.1 Shadows

To create the shadows, we need to figure out for a point, which light are occluded and which are not. In the method getNonOccludedLights of Scene we retrieve the list of Lights not occluded for a selected point in the scene. To do so, for each Lights in the scene, we generate a ray from the point to the Light and we figure out with the method fastIntersect if this ray has an intersection in the scene or not. If not, then this Light is not occluded for the point and is added to the list. We then change the shade methods to use this method. Generateray sets maxt to avoid having a collision with the border of the point.

Figure 6: Shadows



### 2.2.2 Additional Intersection Information for Recursion

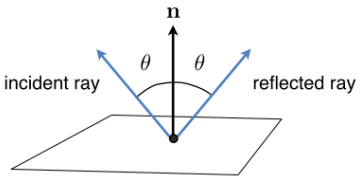
We just have to store the value of the reflection percentage in the field of the object $IntersectionData$ with $iData \rightarrow reflectionPercentage = getReflectionPercentage()$; for each geometrics when a intersection occurred.

### 2.2.3 Reflection

The principle of recursion is easy : each incident ray is reflected with the same angle with the normal of the surface when a collision occurred, and then this ray is just a new ray with an another direction and we can do a recursion with a pre-definite recursion depth to generate the image. We add the ray's constant value epsilon_t to min_t in order to avoid an intersection at the start of the ray with the object due to imprecisions. If we don't use this, then incorrect intersections will be added, and some legit intersections will be neglected due to the previous incorrect one.

So we get the normal **N** in the object $IntersectionData$ and calculate the

Figure 7: Reflection




new ray

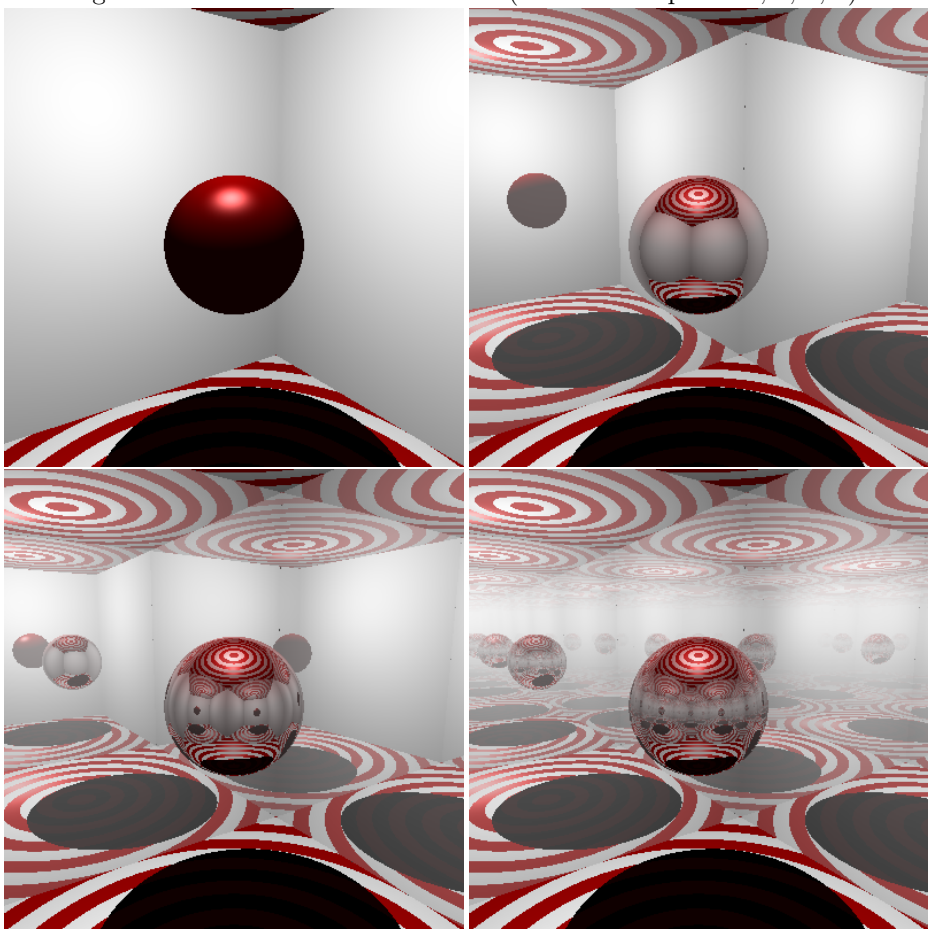
$$\mathbf{R_{reflected}} = \mathbf{2N} \cdot (\mathbf{N} \cdot \mathbf{R_{incident}}) - \mathbf{R_{incident}}$$

where $\mathbf{R_{incident}}$ is the direction of the incoming ray in opposite direction and it's given in the object *Ray* by $ray.direction$. Then just change the sign, i.e $-ray.direction$, and replace in the formula to get the new ray. So we created an new object *Ray* from the intersection point and in the direction of $\mathbf{R_{reflected}}$ and then use a recursion to compute the color of the final point with the formula

$$reflectionPercentage * reflectedShading + (1 - reflectionPercentage) * objectShading$$

where $reflectedShading$ is the recursion step and $objectShading$ is the m_shader.

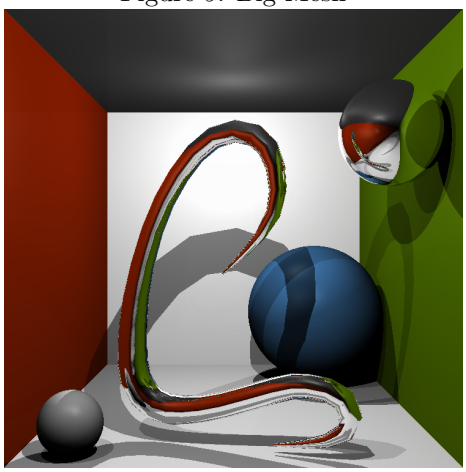
Figure 8: Recursion in Infinite Room (Recursion depth = 0, 1, 2, 8)



## 2.3 Final Rendering

To render the big Mesh, we just change the scene in *Raytracer.cpp* and as we can see the shape in 588 triangles is correctly interpolated and reflected, as the sphere at top right.

Figure 9: Big Mesh



This take 1129 sec on a laptop and 622 sec on the other laptop with a recursion depth of 8 and Release mode. So it's very long.