UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
DIM0451 - COMPUTAÇÃO GRÁFICA

PROF. BRUNO MOTTA DE CARVALHO

Projeto Ray Tracer - 5ª Etapa - **Trabalho em grupo com 2 participantes**
Entrega: 07/07/2022 Horário: 23:59 Local: Entrega eletrônica via SIGAA

Instruções para entrega do trabalho: Os arquivos contendo o código do projeto devem ser salvos em um diretório com o nome `RT3`. Faça o *upload* no SIGAA antes do horário limite indicado acima.

Ray Tracer - Phase 5 - The Blinn-Phong Reflection Model

1 Introduction

After learning how to setup a virtual scene with several objects and generate viewing rays from a virtual camera, it is time to understand how to **shade** or assign a color to the closest ray-object intersection points. Proper color assignemnt creates an image that looks three dimensional (3D), with hidden surfaces and a sense of depth. The presence of shadows and the color variation the object's surface caused by the existence of light sources in the scene will also help in creating the desired 3D look-and-feel for your rendered images.

In this programming project you will have to create a new integrator, called `BlinnPhongIntegrator`, which implements the Blinn-Phong Reflection Model (BPRM). This empirical model of local illumination tries to (roughly) approximate how a surface reflects light as a combination of only two components: diffuse reflection, commonly found in rough surfaces such as wood or clay, and specular reflection, commonly found in shiny surfaces such as polished metal. Put simply, a reflection model is just one way of shading pixels on images based on factors such as light sources, and material properties. For that to happen

you need to add a **new material** called `BlinnMaterial` and create a new class to represent the entity **light**. To increase the scene realism, your integrator should cast **hard shadows**.

2 New Entities Your Ray Tracing Needs to Support

The first step you should do is to read all the suggested material that explain the technical aspects of the BPRM. From that reading you should realize that your ray tracing system needs to create and support new entities, shuch as light sources, and material properties.

This section describes these entities first, whereas the following section describes the procedure you should follow to shade objects according to the BPRM.

2.1 Support Light Sources

You should implement the following light types:

- **Point light**: It is an infinitely small sphere of light with a 3D location in space that emits light rays in all directions. In this case the light ray strikes a surface in a specific direction, determined by a vector from the strike location on the surface to the point light location in the world space.
- **Directional light**: It is light source located so far away that all incoming light rays that reach a surface are parallel (think of the sun) and *follow a single direction.*
- **Ambient light**: This is an "fake" light created only to simulate the amount of light diffusely reflected in a scene. This means that all objects are affected by this light, regardless of their location in the scene. Therefore, we only need to set a single ambient light for the entire scene; this light contribution should be added to all light calculations. We usually set this value to a small amount, such as `[0.1, 0.1, 0.1]`.
- **Spot light**: It is a particular type of point light that has a cone of influence: only objects inside this cone receive light rays. A flash light or a desk

lamp (think of the Pixar's Luxo Jr.), for instance, are classic examples of this type of light in the real world. We may have a hard-edge spotlight or a soft-edge spotlight. Hard-edge means that the light intensity inside the cone is the same, whereas in a soft-edge spotlight we have a transition zone (a cone within a cone) where the light intensity is linearly reduced from its normal value to zero as it goes through the transition zone. The effect in the later case is that the edge of the spotlight effect is blurred.

The Figures 1, 2 and 3 show how point, directional and spot lights interact with the world objects.

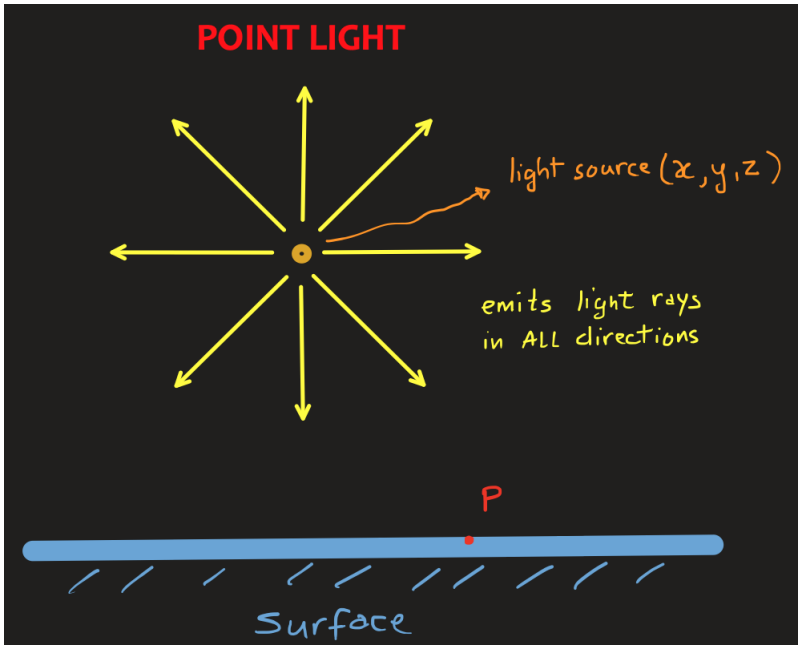


Figura 1: Depiction of how a point light interacts with the world.

I recommend that everyone check this OpenGL tutorial out to get a better understanding of these four light types. Other alternative explanations may be found here.

All types of lights introduced here have an intensity field represented as a 3D vector with values in range `[0,1]` that corresponds to the emitted photon intensity in each of the three color channels, `(R, G, B)`. By setting the intensity value properly, you might simulate colored lights.

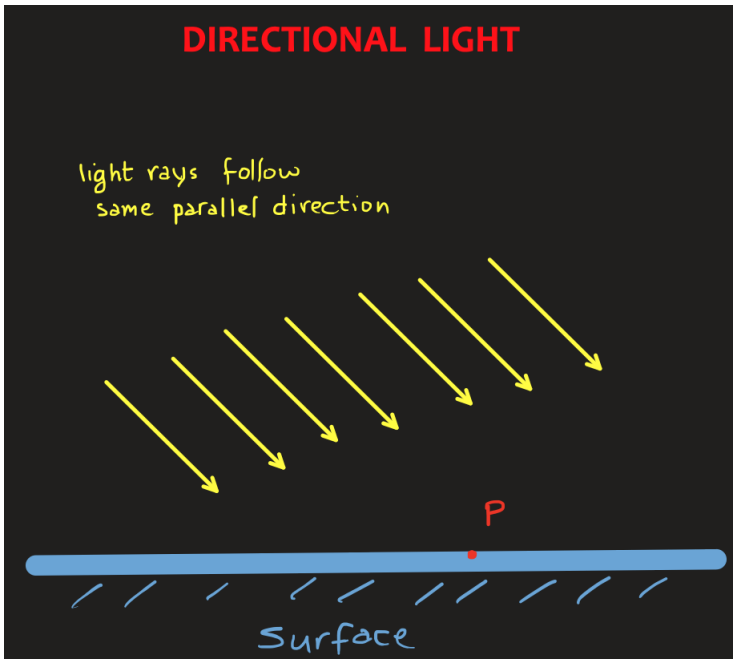


Figura 2: Depiction of how a directional light interacts with the world.

The code below demonstrate how lights of these tree types of light source may be coded in a scene file. These light tags should appear within the tags `<world_begin/>...<world_end/>`. The scale attribute is just another way to modulate the light intensity. In the examples below, all have scale set to 1, which means it is not affecting (reducing) the original light intensity. It is worth mentioning that it only makes sense to create a single ambient light source for the entire scene, since it tries to emulate and overall light reflection that might reach all objects in a scene. For the other light sources, though, you may create as many as you like.

```
<light_source type="ambient" L="0.2 0.2 0.2" />
<light_source type="directional" L="0.5 0.5 0.6" scale="1 1 1"
from="0 25 -14" to="0 0 1" />
<light_source type="point" I="0.3 0.3 0.1" scale="1 1 1"
from="0 1.3 -1.7" />
<light_source type="spot" I="0.5 0.5 0.4" scale="1 1 1"
from="1.5 5 -8" to="1.5 -2 -8" cutoff="30" falloff="15" />
```

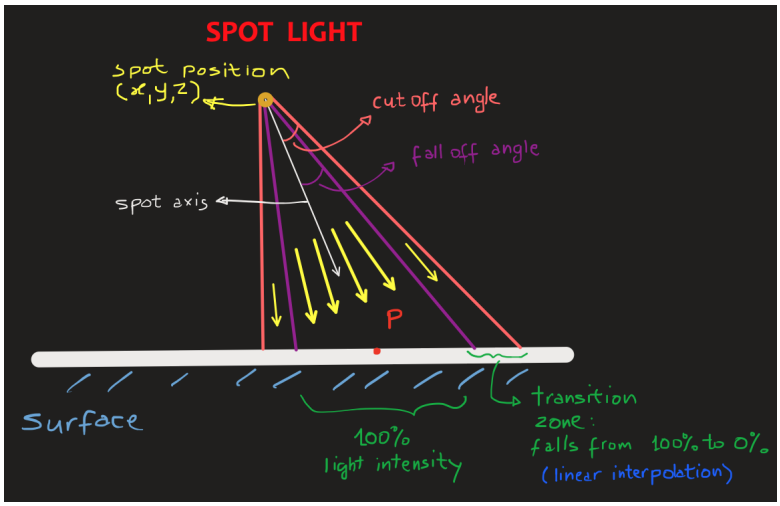


Figura 3: Depiction of how a spot light interacts with the world.

Here is a suggestion for a base class that represents a light source.

```
enum class light_flag_e : int {
        point = 1,
        directional = 2,
        area = 4,
        ambient = 8,
        spot = 16
};

bool is_ambient( int flag ) {
        return flag & (int) light_flag_e::ambient;
}

// Verifica se há oclusão entre dois pontos de contato.
class VisibilityTester {
        public:
        VisbilityTester()=default;
        VisbilityTester( const Surfel&, const Surfel& );
        bool unoccluded( const Scene& scene );
        public:
```

```
        Surfel p0, p1;
};

class Light {
        public:
        light_flag_e flags;
        public:
        virtual ~Light(){};
        Light( int flags );
        /// Retorna a intensidade da luz, direção e o teste oclusão.
        vitual ColorXYZ sample_Li( const Surfel&   hit  /*in*/,
        Vector3f        *wi  /*out*/,
        VisbilityTester *vis /*out*/ ) = 0;
        virtual void preprocess( const Scene & ) {};
};
```

2.2 Support New Material

The `BlinnPhongMaterial` should have the following data members:

- An **ambient** coefficient, k_a: It is a 3D vector with each individual values in `[0,1]`, that represents how much the incoming light is reflected. For instance, `[0.4, 0.4, 0.4]` means that 40% of the incoming ambient light is reflected back to the image.
- A **diffuse** coefficient, k_d: It is a 3D vector that indicates how much diffuse color is reflected. This is the component that defines the overall "color" of the object. For instance, `[0.9, 0.2, 0.2]` represents a predominantly red object.
- A **specular** coefficient, k_s: It is a 3D vector that represents the color of the specular highlight. Because we often want to imitate the highlights found in the real world, the k_s is usually set to be a white color, although colored highlights are also interesting.
- A **glossiness** exponent: This is a real (float) value that controls how "narrowed" is the specular highlight in the scene. The larger the value,

the smaller the region of the specular effect: this would correspond, for instance, to highly specular object such as a polished car surface.

To better understand the interplay among these components and the Blinn-Phong Reflection Model, I recommend to read this chapter.

The code below hopefully will help you understand how these coefficients should be associated with a material in a scene. In this case we are creating a named material, which means this material information should be stored in the library material (while parsing is still happening) for later use in the scene file. The material corresponds to a gold-looking surface, that has a highly specular coefficient.

```
<make_named_material type="blinn" name="gold" ambient="0.2 0.2 0.2"
diffuse="1 0.65 0.0" specular="0.8 0.6 0.2" glossiness="256"/>
```

2.3 Implement BlinnPhongIntegrator

If you have followed the RT architecture suggested in the previous projects, you need to create a new integrator `BlinnPhongIntegrator` that is derived from `SamplerIntegrator` and override the `Li(...)` method. All the calculations from the Blinn-Phong Reflection Model should be coded inside the `Li()` method. In the next section you will find a brief description of the theoretical background necessary to implemente the BPRM inside the `Li()` method. Below you have a top level sequence of steps for the `Li()` method.

```
ColorXYZ BlinnPhongIntegrator::Li( const Ray &ray, const Scene &scene,
ColorXYZ bkg, int depth )
{
        // [0] FIRST STEP TO INITIATE `L` WITH THE COLOR VALUE TO
        // BE RETURNED.
        // [1] FIND CLOSEST RAY INTERSECTION OR RETURN BACKGROUND
        // RADIANCE
        // [2] SPECIAL SITUATION: IF THE RAY HITS THE SURFACE FROM
        // "BEHIND" (INSIDE), WE DO NOT COLOR.
        // [3] GET THE MATERIAL ASSOCIATED WITH THE HIT SURFACE
        // [4] INITIALIZE COMMON VARIABLES FOR BLINNPHONG INTEGRATOR
```

```
        // (COEFFICIENTS AND VECTORS L, N, V, ETC.)
        // [5] CALCULATE & ADD CONTRIBUTION FROM EACH LIGHT SOURCE
        // [6] ADD AMBIENT CONTRIBUTION HERE (if there is any).
        // [7] ADD MIRROR REFLECTION CONTRIBUTION
        return L;
}
```

3 The Blinn-Phong Reflection Model

To determine the final color of a pixel associated with a single camera ray, the BPRM only calculates and adds three color contributions: diffuse, specular, and ambient. These contributions are computationally cheap to calculate and allow the RT to portrait matte objects (like clay), shiny objects (like a polished metal surface), or the combination of both.

Let us investigate each of these three reflections.

3.1 The Lambertian Reflection (aka diffuse contribution)

The Lambertian reflection models a perfect diffuse surface that scatters incident illumination equally in all directions. Although this reflection model is not physically plausible, it is a reasonable approximation to many real-world surfaces such as matte paint.

One important point about the Lambertian reflection is that it is **view independent**, which means regardless of the camera position, the amount of light reflected is the same. The only factor that influences how much light is reflected back to the scene is the angle formed between the surface's normal, $\hat{\mathbf{n}}$, and a vector towards the light direction $\overrightarrow{\mathbf{l}}$. More precisely:

"The amount of energy from a light source that falls on an area of surface depends on the angle of the surface to the light."

To calculate the amount of light contribution towards a given view direction (i.e. the camera ray), first we need to calculate the cosine of the angle θ formed between the light source and the surface's normal and use this value to modulate the light source intensity and with the material's properties of reflection (or absorption) of certain wavelengths from the visible light.

The image below depicts the interplay between these components.

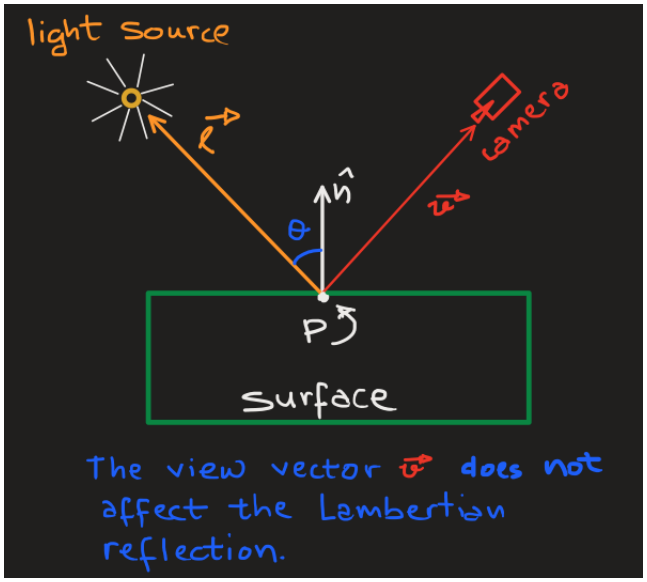


Figura 4: Interaction between light and an object with a Lambertian surface material.

In terms of your RT system the Lambertian reflection receives as an input the following elements:

1. The camera ray: this corresponds to the view direction $\overrightarrow{\mathbf{v}}$;
2. The Surfel struct: this contains both the hit point on the surface `P`, and the surface's normal $\hat{\mathbf{n}}$ at that location.
3. The light source: this yields the light direction $\overrightarrow{\mathbf{l}}$.
    - If we have a spot or point light source, we find the light direction vector by subtracting the 3D light location (`Point3f`) from the hit point `P` to get the vector $\overrightarrow{\mathbf{l}}$.
    - If we have a directional we do not need to do anything, because the light direction was explicitly informed when the light source was parsed from the input scene file.

Once we have all these 3 elements, $\overrightarrow{\mathbf{v}}$, $\hat{\mathbf{n}}$, and, $\overrightarrow{\mathbf{l}}$, we are ready to compute

the Lambertian reflection contribution and store it in, say, a L variable, where

$$L = k_d * I * max(0, \hat{\mathbf{n}} \cdot \overrightarrow{\mathbf{l}}).$$

Important: do not forget to normalize the light direction $\overrightarrow{\mathbf{l}}$ because to get value of $cos(\theta)$ using dot product both vectors must be normalized.

The L value would be the color of the pixel in the final image, if we were to consider only the Lambertian reflection. However, the calculations are not over just yet. Weed to add other contributions to the L variable, namely the specular reflection and ambient contributions.

3.2 The Specular Reflection (aka specular contribution)

The Spectular reflecton tries to simulate a fuzzy spot of light on a surface with the same color as the light hitting the surface. This effect can be seen when you look at a shinny surface, such as a well polished hood of a car. The center of the specular spot of light should be located where the direction of your eye (or the camera, in the RT case) lines up with a perfect reflection of the incoming light direction vector about the surface's normal.

In the image below, the light direction is $\overrightarrow{\mathbf{l}}$, the hit point is P, the normal at the hotpoint is $\hat{\mathbf{n}}$, the view direction is $\overrightarrow{\mathbf{v}}$, and $\overrightarrow{\mathbf{r}}$ indicates how the light ray $\overrightarrow{\mathbf{l}}$ would perfectly reflect about $\hat{\mathbf{n}}$. The yellow elipses represents the intensity and size of the specular spot. Notice that the view direction is not aligned with the light reflection vector $\overrightarrow{\mathbf{r}}$ and, therefore, barely touches the yellow elipses; consequently for this particular ray/pixel, the specular contribution that should be added to the diffuse reflection is minimal. In practice the location of P in the final image would be somewhere in the outer rim of the specular spot.

One way to find out whether the viewing direction $\overrightarrow{\mathbf{v}}$ lines up with the light reflection vector $\overrightarrow{\mathbf{r}}$ without having to actually calculate this vector is to consider a normalized half vector $\hat{\mathbf{h}}$ as the bisector of the smallest angle formed between $\overrightarrow{\mathbf{l}}$ and $\overrightarrow{\mathbf{v}}$.

Calculating $\hat{\mathbf{h}}$ is cheaper than calculating $\overrightarrow{\mathbf{r}}$ directly, because it requires one addition and one division,

$$\hat{\mathbf{h}} = \frac{\overrightarrow{\mathbf{v}} + \overrightarrow{\mathbf{l}}}{||\overrightarrow{\mathbf{v}} + \overrightarrow{\mathbf{l}}||},$$

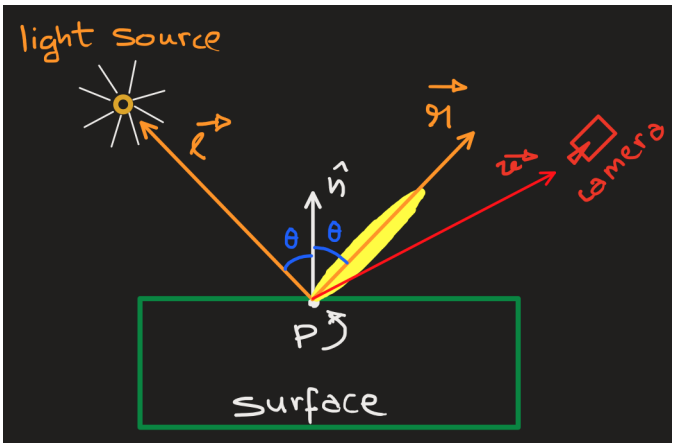


Figura 5: Interaction between light and an object with a specular surface material.

whereas the other requires the normalization of $\overrightarrow{\mathbf{l}}$, one dot product, one vector multiplication and one vector substraction.

$$\hat{\mathbf{r}} = \hat{\mathbf{l}} - 2(\hat{\mathbf{l}} \cdot \hat{\mathbf{n}}) * \hat{\mathbf{n}}$$

Once you determined the half vector $\hat{\mathbf{h}}$ you need to measure how close this vector is to the surface normal $\hat{\mathbf{n}}$ by calculating the cosine of the angle they form. If we use the cosine measurement alone to module the specular contribution, we would end up with very large spots (the yellow ellipse would be "fatter"). To make this contribution decrease rapidly, the Phong reflection model introduces the Phong exponent: an exponentiation factor applied to the cosine, so that small changes in the cosine would reduce greatly the specular contribution, if the phong exponent is high.

Put simply the Phong exponent is just a math "trick" introduced to allow us to control how small or large the specular spot is on a surface. In our formulation, let us call the Phong exponent g, to associate with the term glossiness. With the addition of the specular contribution, our shading equation becomes

$$L = k_d * I * max(0, \hat{\mathbf{n}} \cdot \overrightarrow{\mathbf{l}}) + k_s * I * max(0, \hat{\mathbf{n}} \cdot \hat{\mathbf{h}})^g,$$

where k_s is the specular coeficiente of the material assigned to the surface.

Figure 6 demonstrates the effect of the specular reflection on several grey spheres. The left most sphere has no specular reflection, only diffuse contribution. The following spheres, from left to right, has assigned a power of two value to the `g` exponent, starting at 2, then 4, 8, etc., up to 512. Notice how the specular spot shrinks rapidly as the g values increases, to become a very small yellowish spot in the right most sphere. In this case, the light source is a directional yellowish light, and the camera is orthogonal.

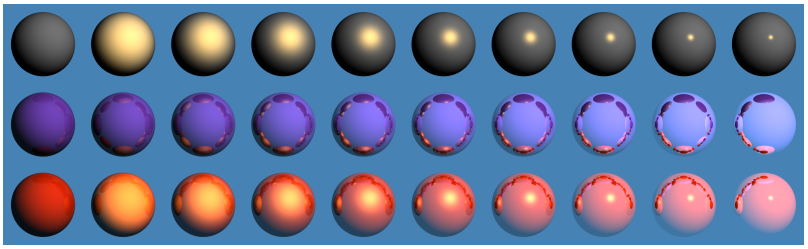


Figura 6: Interaction between light and an object with a specular surface material.

3.3 The Last Component: the Ambient Contribution

Depending on how you set up the lights for a given scene, some objects may appear completely dark in the final image, if they do not receive any contribution from diffuse and specular reflections. This could happen, for instance, if we create a scene with only spot lights. In the real world, however, objects in a scene often receive, at least, a small amount of lighting coming from outside or even reflected off some wall of other reflective surfaces present in a scene. This is called indirect illumination and, in reality, this phenomenom is responsible for most of the light contribution we see in a real world scene. However, simulating such visual effects is extremely complex and computer intensive. Indirect Illumination is normally done with sophisticated physically based rendering algorithms implemented with Monte Carlo integration, such as path tracing.

Not so long ago computer graphics systems and computers could not handle such computer-intensive algorithms because of the computer processing power constraints of the time. So, earlier computer graphics scientists were forced to be very "creative" in designing algorithms that were not necessarily physically

correct but produced visually pleasant images that would approximate the real ones. The ambient contribution is classic example of such ingenuity. This component in the BPRM is an arbitrary way of simulating the indirect illumination described above.

In this project, ambient lighting is simply calculated as the product of a the material color, k_a associated with a surface and the overall ambient light intensity I_a. Because we may have individual k_a coefficients associated with a material, is possible to control how individual materials are affected by this overall ambient light.

$$L = k_a * I_a + k_d * I * max(0, \hat{\mathbf{n}} \cdot \overrightarrow{\mathbf{l}}) + k_s * I * max(0, \hat{\mathbf{n}} \cdot \overrightarrow{\mathbf{h}})^s$$

3.4 Multiple Lights and the Final BPRM Equation

Because the light contribution in a scene with multiple light sources is cumulative, we can simply add the contribution of individual lights together to produce the overall lighting effect we desire. Therefore, if a scene has a set of k light sources defined, each with individual intensity I_i, we would apply the full BPRM equation below to obtain the color of a single pixel L:

$$L = k_a * I_a + \sum_{i=1}^{k} [k_d * I_i * max(0, \hat{\mathbf{n}} \cdot \overrightarrow{\mathbf{l}}) + k_s * I_i * max(0, \hat{\mathbf{n}} \cdot \overrightarrow{\mathbf{h}})^g],$$

where

- I_i `[Vector3f]`: is the intensity of the i-th light source (we may have several of these in a scene),
- I_a `[Vector3f]`: is the single ambient light intensity (just one for the entire scene),
- k_a `[Vector3f]`: is the ambient coefficient of the material,
- k_d `[Vector3f]`: is the diffuse coefficient of the material,
- k_s `[Vector3f]`: is the specular coefficient of the material,
- g `[float]`: is the specular exponent of the material (glossiness),

- $\hat{\mathbf{n}}$ `[Normal3f]`: is the normal vector at the location where the ray hit the surface,
- $\overrightarrow{\mathbf{h}}$ `[Vector3f]`: is the half vector, or the bisector of the angle between the view vector $\overrightarrow{\mathbf{v}}$ and the light direction vector $\overrightarrow{\mathbf{l}}$,
- $\hat{\mathbf{n}} \cdot \overrightarrow{\mathbf{l}}$ `[float]`: corresponds to the $\cos(\theta)$, the angle between $\hat{\mathbf{n}}$ and $\overrightarrow{\mathbf{l}}$,
- $\hat{\mathbf{n}} \cdot \overrightarrow{\mathbf{h}}$ `[float]`: corresponds to the $cos(\alpha)$, the angle between $\hat{\mathbf{n}}$ and $\overrightarrow{\mathbf{h}}$.

4 Casting Shadows

To cast hard shadows, you must do the following: every time a ray hits a surface at, say, P, your integrator must shoot another ray, called **shadow ray**, from P towards the light source position in space. The origin of the ray corresponds to the ray parameter $t = 0$, while the light position location corresponds to $t = 1$. Then you ask the scene to see if this new shadow ray intersects any object: a positive answer would mean that there is "something" between the original hit point P and the light's location in space blocking the light path. Therefore, the color at P should be black (the shadow color) or the ambient contribution if one is set.

Notice that we do not need to determine any intersection information of the shadow ray besides the `true` or `false` indicating that an intersection happened. So this is the perfect opportunity to call the simpler `intersect_p()` version of the intersection methods defined by the `Primitive` class.

In the case of **directional** lights we may have a problem while casting shadows. Let us see why. We have the surface hit point `P` (the shadow ray origin), we have the shadow ray `direction` (from the directional light data) but we do not have the `location`, or the end-point of the shadow ray to test for intersection.

Put simply, we want to answer: *How far along the shadow ray should I keep testing intersection for?* The answer is: we stop the shadow ray when it goes beyond the scene's limits. This limit would correspond to a giant sphere or box that encompasses all objects in the scene. So, to determine the scene's limits or bounds, we need to create a bounding box around each primitive in the scene. In

this way, the entire scene's bounding box is the union of all (smaller) bounding boxes. The maximum extent the shadow rays is valid for corresponds to the diagonal of that (giant) bounding box.

Therefore, to support shadows with directional lights, we need to create a bounding box for every primitive. During the `preprocess()` of the rendering, we call the `preprocess()` of the lights. That method should determine the maximum extent or length for the shadow ray, based on all bounding boxes associated with the primitives. The implementation of the bounding boxes for primitives will also pay off later on, when we implement the *Bounding Volume Hierarchy* data structure that enables the **acceleration** of the intersection test rays-scene.

5 Mirror Effect

One nice addition to the RT is the mirror effect. It is simple to code and and might add a dramatic effect to your scenes. Coding mirror effect becomes easy because all we need to achieve that is already inside the RT. We have rays, vectors, the scene, and the method `Li()` that has acces to all these elements and determines the color for a single pixel.

So, the first step is do add and extra attribute to the Blinn-Phong material tag, introduced earlier in this document. The attribute should be called `mirror` and defines how much light should the mirror material reflect into the scene. The code below describes a material that reflects only 40% of the incident light.

```
<make_named_material type="blinn" name="gold" ambient="0.2 0.2 0.2"
 diffuse="1 0.65 0.0" specular="0.8 0.6 0.2" glossiness="256"
 mirror="0.4 0.4 0.4"/>
```

Although there are several types of material that reflect light in one way or another, we are interested, at this point, in the so called **perfect mirror**. This means that the incoming light ray must reflect at the perfect reflection angle, which is the reflection of the incoming angle about the surface normal.

To determine the color at the surface point of contact we have to follow the reflected ray $\hat{\mathbf{r}}$ into the scene. This means making a recursive call to the `Li()`

function that does exactly that: follows rays into the scene and determines the color for it.

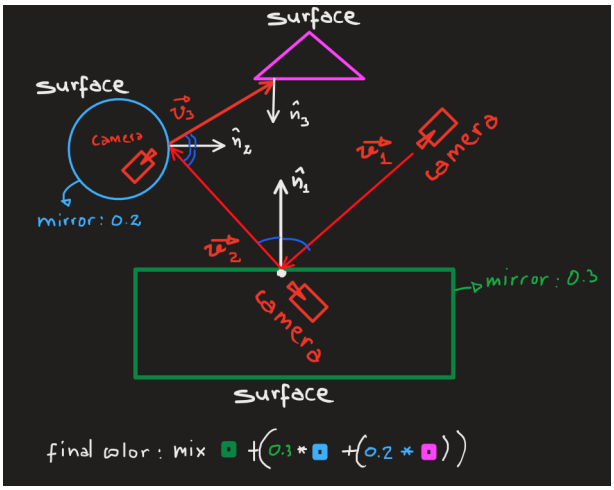


Figura 7: Scheme showing the interaction of objects with materials which are reflective.

Thus, the color at the first point of contact is given by:

```
// [1] Determine color L based on the Blinn-Phong model
// [2] Find new ray, based on perfect reflection about surface
// normal.
Ray reflected_ray = ray - 2(dot(ray,n))*n;
// [3] Offset reflect_ray by an epsilon, to avoid self-intersection
// caused by rounding error.


// [4] Recursive call of Li() with new reflected ray.
if ( depth < max_depth )
L = L + km * Li(refected_ray, scene, depth+1);
```

where `km` (k_m) is the mirror coefficient we read earlier from the scene description file. The `if` is necessary to make our RT follow the light ray up to a certain recursion depth, otherwise we might be caught in an infinite loop of the scene was full of mirror surfaces, for example. The `max_depth` parameter is passed to the RT as a field (tag) of the `integrator` part of the scene. Figure 8 shows an image created with just one level of recursion.

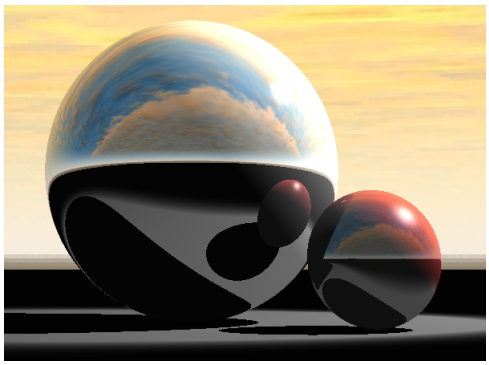


Figura 8: Image generated using one level of recursion.

Note that the same reason that made us offset the shadow ray to avoid self-shadowing, will require us to offset the mirror-ray to avoid a premature termination of the color calculation, since the mirror-ray might hit the very surface it is spawning from.

The result of incorporating mirrors into your scene can be seen in Figure 9. We have a sequence of 5 images, where the first one has no mirroring. The following images add progressive depth levels of mirror-ray reflections. Note that as the level of recursion increases we begin to have reflections of reflections of reflections, emulating an effect similar to what happens when you put a mirror in front of another.

6 The Scene File Format

You will find examples of scene file format with the new tags introduced in this project and the corresponding images on the couse's SIGAA webpage.

Figure 10 shows an image generated with only the ambient contribution, while Figure 11 shows an image generated using two directional lights. Figure 12 shows the same scene rendered using a single point light and Figure 13 shows the scene rendered using four spot lights. Finally, Figure 14 shows an image of the scene rendered using several lights of different types.

7 Extra Feature: light attenuation

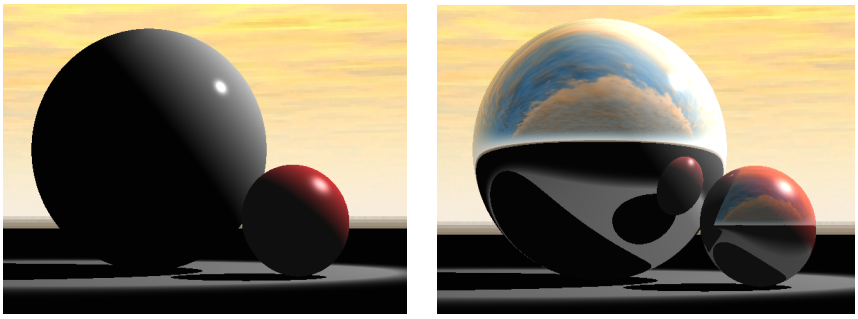
Make your Blinn-Phong integrator support light attenuation. Light is preserved while traveling in the space (vacuum). However, while traveling through a medium such as the air, water or a room full of dust, its intensity should be reduced (attenuated) at a rate inversely proportional to the squared distance between the light source and the point that receives the light.

To support this feature I recommend reading the section **Attenuation** here. To implement the suggested model, you will need to add extra constants to your world description. These constants tries to model the attenuation effect of the medium that fills up the empty "spaces" in the scene. These values would probably need a new tag somewhere between the `<world_begin>...<world_end>` tags, perhaps named `attenuation` tag.

```
<!--- Attenuation -->
<attenuation kc="1.0" kl="0.22" kq="0.20"/>
```

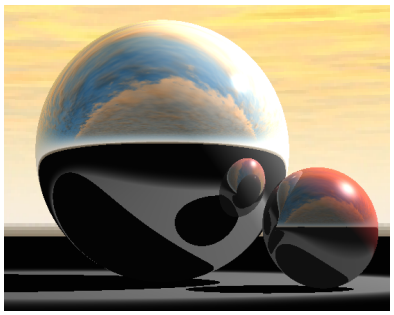
Alternatively, you might want to associate this tag to each individual light. Practically it makes sense, since the attenuation affects how the light intensity from a light source is reduced. Physically it does not make sense, since the attenuation is related to the medium through which the light travel, so it would be associated with the scene not with individual light sources.

But, in the end, it is up to you to decide where to put that information.

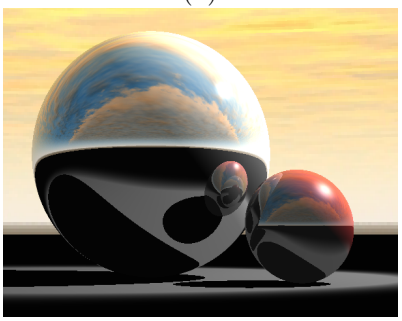


(a)

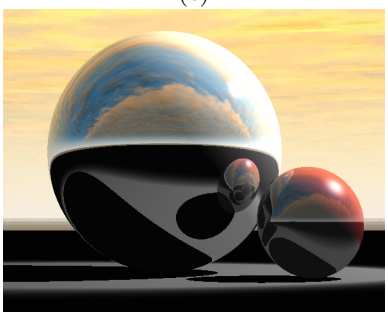
(b)



(c)



(d)



(e)

Figura 9: Spheres with mirror material generated using different maximum depth levels: 0 (a), 1 (b), 2 (c), 3 (d) and 4 (e).

Figura 10: Image generated using only ambient illumination.

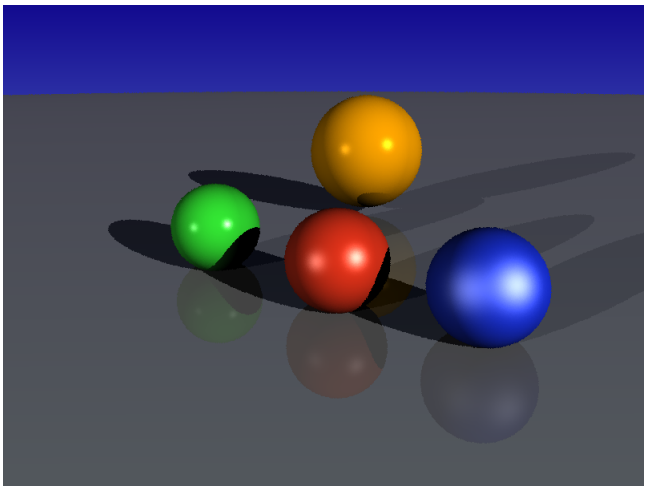


Figura 11: Image generated using two directional lights.

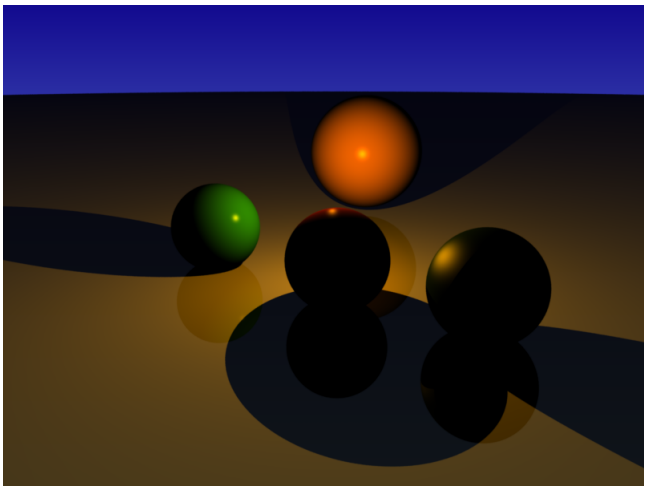


Figura 12: Image generated using a point light.

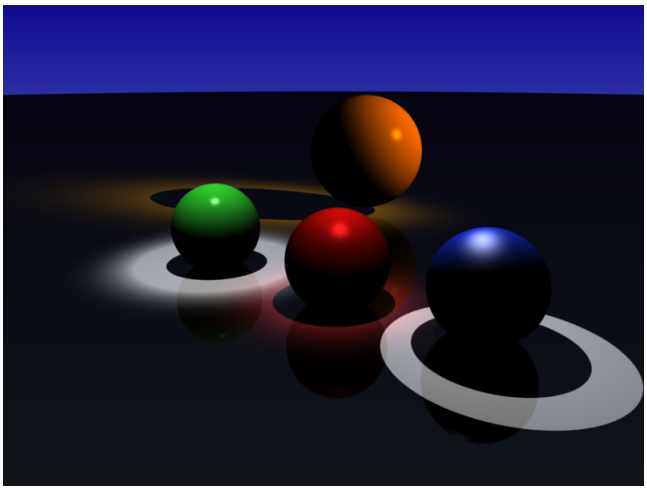


Figura 13: Image generated using four spot lights.

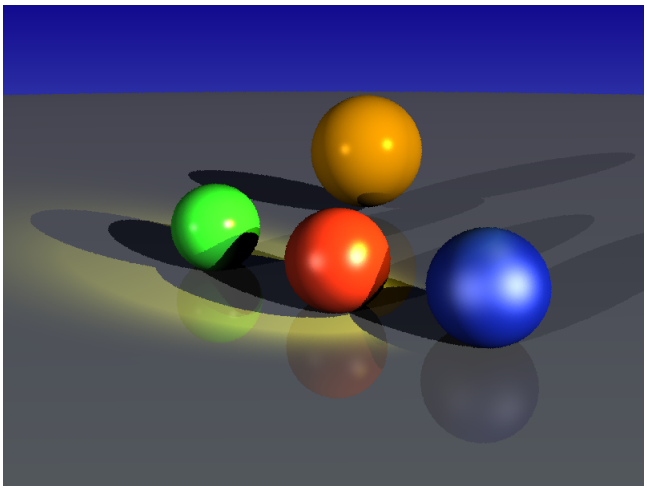


Figura 14: Image generated using several lights.