Introduction to Shading

Contents

Structure of this Chapter

- Learn how to simulate spherical lights.
- Learn how to cast shadows with spherical lights.

Spherical Lights

We already introduced the concept of directional light sources which can be used to simulate distant light sources such as the sun. Directional sources are light sources which are so far away from the scene that the light rays they emit can be looked at as parallel to each other. In other words, a distant light source emit rays in one single direction.

Spherical light sources are different. They are not distant like directional light sources but local. A spherical light can be represented as a single point of light in 3D space from which light is emitted from the point of emission outwards. For this reason, they are also frequently calls **point light sources**. Point light sources can be used to simulate things such the flame of a candle, a light bulb, etc. Though as mentioned in the chapter on directional light sources, spherical or point light sources are also considered as light sources with no size. In other words, we model them as an ideal point in space from which light is emitted radially, but such object in nature doesn't exist. The flame of a candle or a light bulb may as well emit light radially, though they have a size. This is not a problem right now, but keep in mind that so far, our lights are considered to be infinitesimally small in size. Such light sources are often called **delta lights** (as explained in the introductory chapter on lights. When spherical lights emit light equally in all direction we say they are **isotropic**.

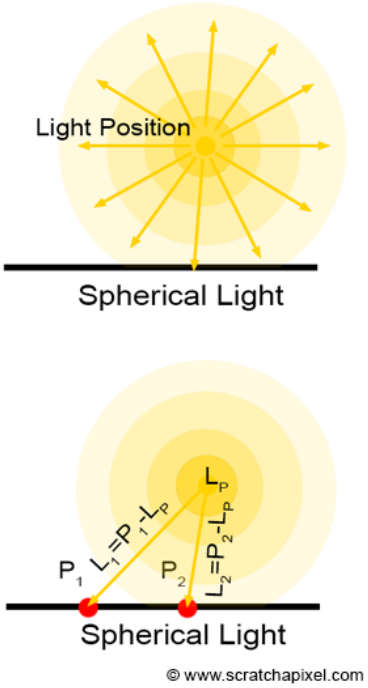


Figure 1: compute the light direction for point light sources.

How do we simulate point light sources? First we need to consider the way light is emitted: radially. From a coding point of view, this can be simulated by simply tracing a line from the point that is being shaded P to the spherical light position. This line will simply indicates the direction of the light ray emitted by the point light source in the direction of P as shown in figure 1.

From a coding point of view, all we need to do to define a point light source, is to add a member variable to the `Light` base class to keep track of its position in space. We will assume that the point light source is originally created at the origin of the world coordinate system. To modify its position in 3D space, we will use the light-to-world transformation matrix (as we did to modify the directional light source light direction):

```
class PointLight : public Light
{
public:
    PointLight(const Matrix44f &l2w, const Vec3f &c = 1, const float &i = 1) : Light(l2w, c, i
    { l2w.multVecMatrix(Vec3f(0), pos); }
    Vec3f pos;
};
```

Since the light direction depends on the position of P and the position of the light source in 3D space, we will add a `getDirection()` function to the `Light` base class. This function will take the point P as an argument and return a normalized light direction. For directional light source, this direction is constant. For spherical light sources, it will need to be computed explicitly:

```
class PointLight : public Light
{
    Vec3f pos;
public:
    PointLight(const Matrix44f &l2w, const Vec3f &c = 1, const float &i = 1) : Light(l2w, c, i
    { l2w.multVecMatrix(Vec3f(0), pos); }
    // P: is the shaded point
    Vec3f getDirection(const Vec3f &P) const { return (pos - P).normalize(); }
};
```

Spherical light sources differ from directional light sources on one more point. Let's consider the case of point light sources first. As mentioned earlier in this chapter, spherical light sources are sources that emit light from a single point in space. This light energy emitted from that point is distributed across the surface of a sphere as shown in figure 2. Without getting into the details (this would involved to know more about radiometry), let's just say for now that the entire light energy or power of the point light source is redistributed over that sphere. As with the case of diffuse surfaces, the amount of light arriving on a small differential area dA is somehow proportional to the amount of light contained within a small patch of the sphere with radius r_1 roughly equal in area to dA as shown in the illustration below. However as you can see in the same illustration, as the sphere keeps expanding, the area of the differential solid angle $d\omega$ that matched the area of the differential area dA also keeps increasing. In other words, the energy that was contained over $d\omega$ is now spread across a much larger area. As a consequence, P_2 receives less light than P_1 or to say it differently, P_2 will be darker than P_1 (the amount of light P_2 receives is proportional to the green solid angle over the red solid angle).

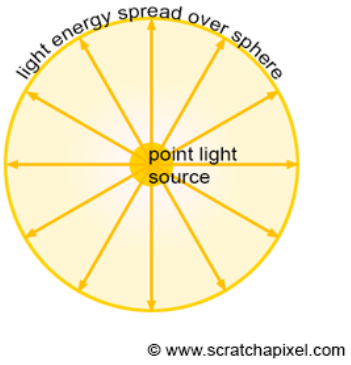
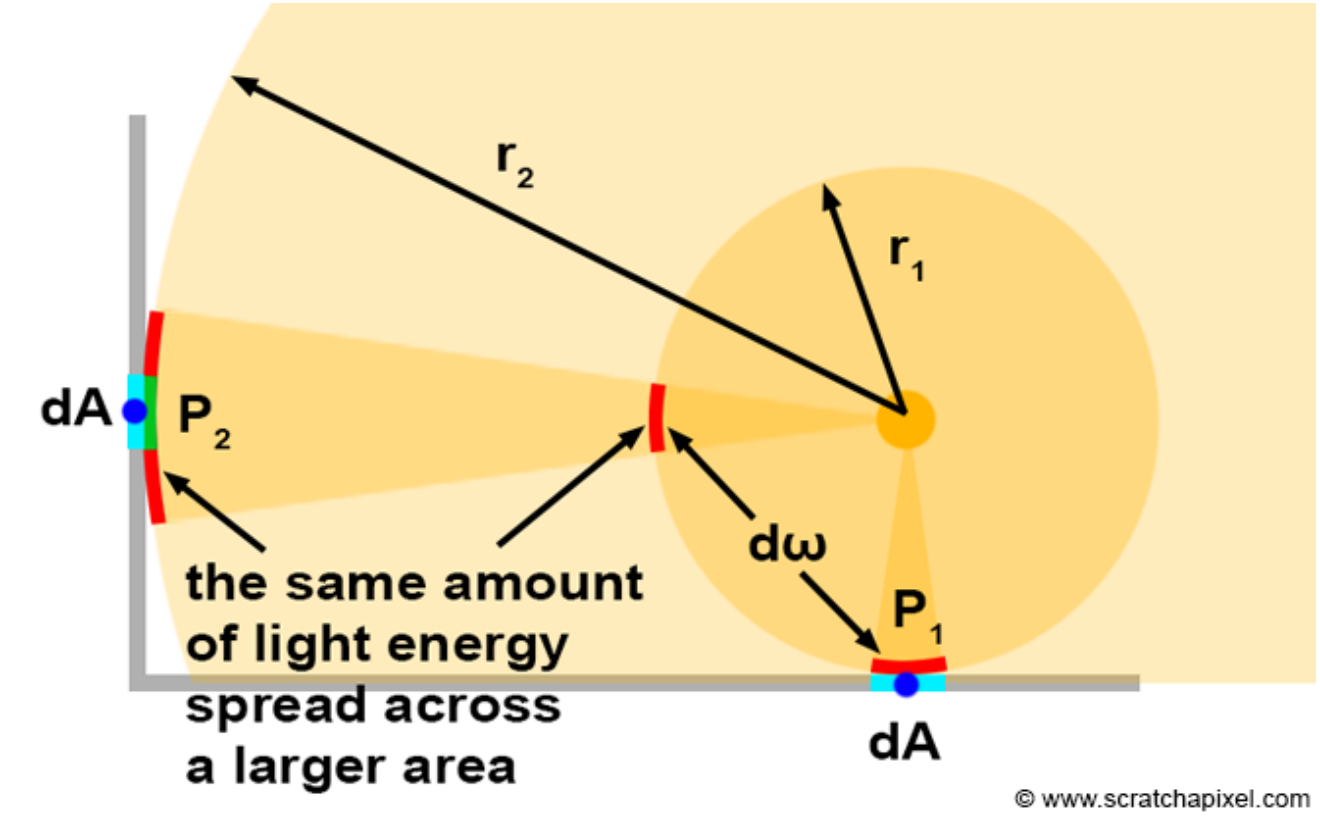


Figure 2: light energy emitted from a point light source is distributed across the surface of a sphere of radius r. This is of course an infinity of such spheres centred around the point light position.

This effect can easily be observed with any real point light sources such as a light bulb. Objects which are closer to the source are brighter. The contribution of the light somehow seems to decrease as the distance from the light source to the objects increases. The question now, is to find out what the rule of that **falloff** is.

In fact, the amount of light energy arriving on a point in the scene from a point light source, depends on the area of the "sphere" which itself depends on the sphere radius. Note that this sphere doesn't really exist. We just use this image to help you visualising the process. When we speak of sphere what we actually mean is some sort of virtual sphere centred around the point light source origin and whose radius is the distance from the point light source to P a point in the scene that we wish to shade. What we are trying to find is how much light arrives at P from that point light source. As suggested, the contribution of the point light source depends on the area of the sphere of radius r:



$$A = 4\pi r^2.$$

The intensity of the light arriving at P is in fact inversely proportional to the sphere area. In other words:

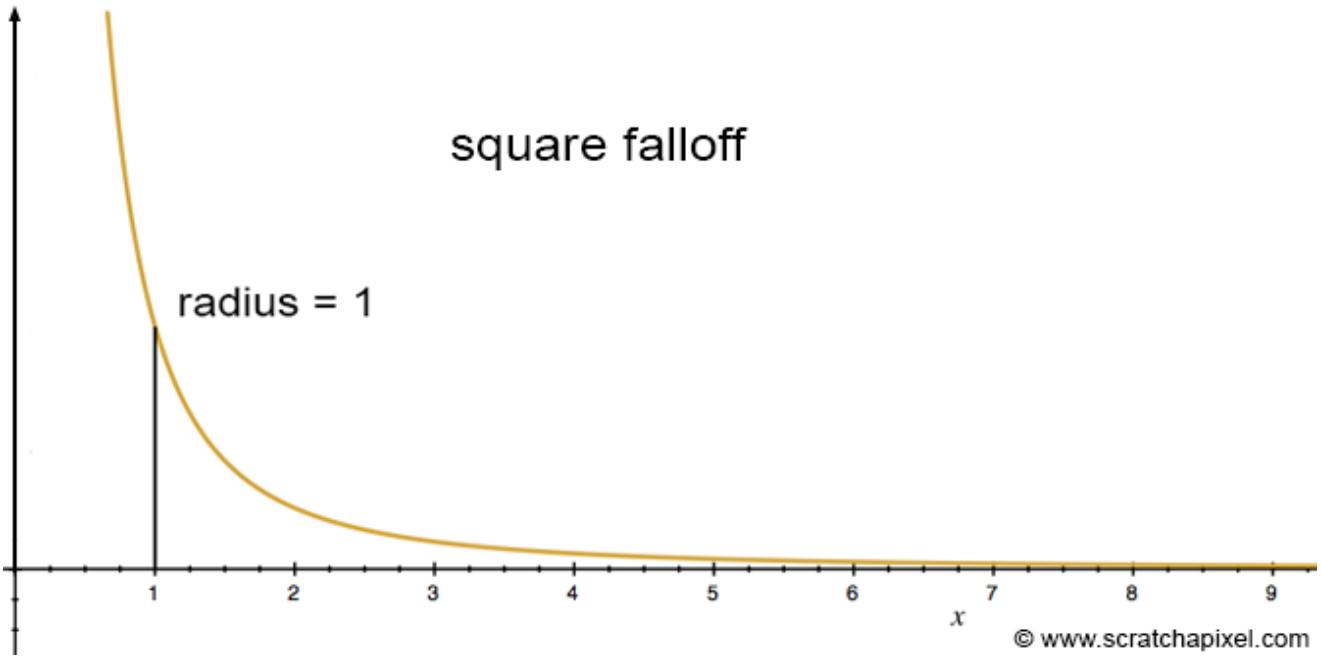
$$L_i = \frac{\text{light intensity} * \text{light color}}{4\pi r^2}.$$

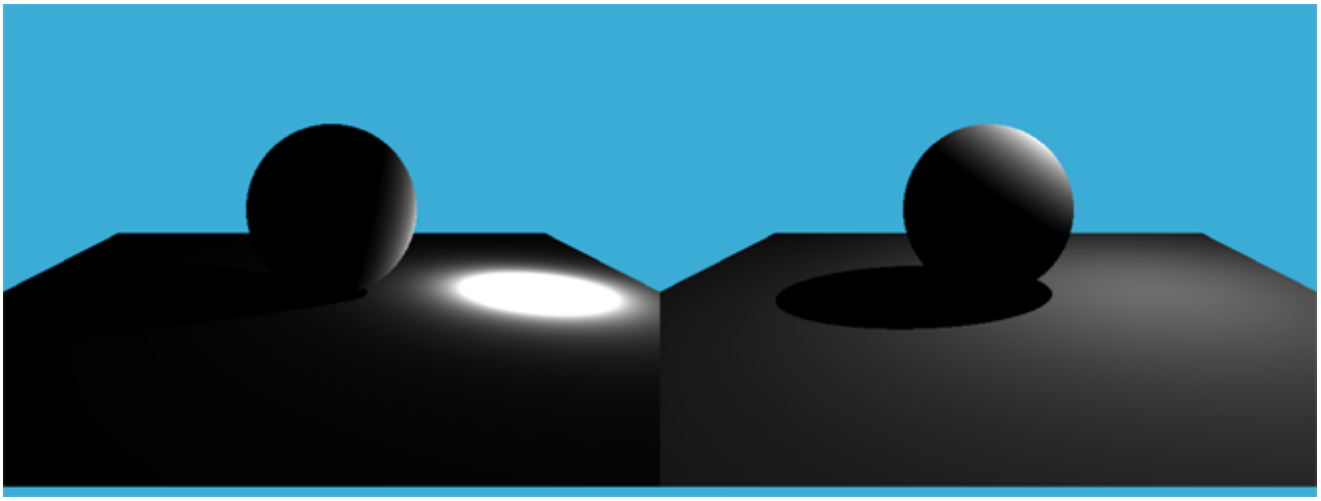
For a more format explanation, please refer to the lesson on radiometry [link].

Where r is equal to the distance between the light position and P. This falloff is known in CG as the **square falloff**. Note that when the radius of the sphere doubles (if the distance between the point light source and P doubles), the area of the sphere is multiplied by 4 and thus the light contribution is 4 times smaller. More generally, this kind of light attenuation follows what we call the **inverse -square law**. The profile of this falloff can be seen in the following figure.

As you can see by just looking at this graph, the contribution of a point light source decreases very rapidly with distance.

To implement this square falloff we will just make a small modification to the `getDirection` method which we will rename `getDirectionAndIntensity`. As you may have guessed, this is also where we will compute the light intensity for a given P. We do so because we already compute in the method the vector connecting the light to P in order to compute the light direction. We can use the square of this vector length to attenuate the light intensity according to the inverse square law (line 13):





```
001 class PointLight : public Light
002 {
003     Vec3f pos;
004 public:
005     PointLight(const Matrix44f &l2w, const Vec3f &c = 1, const float &i = 1) :
006     Light(l2w, c, i)
007     { l2w.multVecMatrix(Vec3f(0), pos); }
008     // P: is the shaded point
009     void getDirectionAndIntensity(const Vec3f &P, Vec3f &lightDir, Vec3f &lightIntensity)
010      const
011     {
012         lightDir = pos - P; // compute light direction
013         float r2 = lightDir.norm();
014         lightDir.normalize();
015         lightIntensity = intensity * color / (4 * M_PI * r2);
    }
};
```

Spherical Lights & Shadows

Computing shadows with spherical lights is very simple. We can use the same techniques than the one we used for distant light. We just need to compute the light direction and trace a ray in the opposite direction. Computing the light direction is simple. We just need to trace a line from the light source to P , the shaded point. By normalizing the resulting vector, you get the light direction:

```
class PointLight : public Light
{
    Vec3f pos;
public:
    PointLight(const Matrix44f &l2w, const Vec3f &c = 1, const float &i = 1) :
        Light(l2w, c, i)
    { l2w.multVecMatrix(Vec3f(0), pos); }
    // P: is the shaded point
    void getDirectionAndIntensity(const Vec3f &P, Vec3f &lightDir, Vec3f &lightIntensity)
          const
    {
        lightDir = pos - P;
        float r2 = lightDir.norm();
        lightDir.normalize();
        lightIntensity = intensity * color / (4 * M_PI * r2);
    }
};
```

Though there is a trap. Directional lights are by definition extremely far away, in fact they are considered to be at infinity. This is not the case of spherical lights though. If we trace a shadow ray int the direction of the light, this ray could very well overshoot the point light source and intersects an object further away than the source. This would lead us to think that the point is in shadow when in fact, P and the light are visible to each other (figure 3). The solution to this problem is to set the ray max length or $tNear$ to the distance between P and the light position (the distance we denoted r). That way, the `trace()` function will never accept an intersection whose distance is greater than r. In other words, if an intersection is found but that the intersection distance is further away than the distance between P and the light, then it will be ignored. Since we already computed the square of that distance in the `getDirectionAndIntensity()` function, we will just add an additional variable to the method that will be set with to the distance between P and the light:

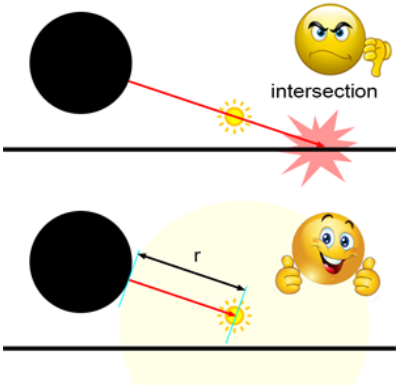


Figure 3: any intersection whose distance is greater than the distance between P and the light can be ignored.

```
class PointLight : public Light
{
    Vec3f pos;
public:
    PointLight(const Matrix44f &l2w, const Vec3f &c = 1, const float &i = 1) :
        Light(l2w, c, i)
    { l2w.multVecMatrix(Vec3f(0), pos); }
    // P: is the shaded point
    void getShadingInfo(const Vec3f &P, Vec3f &lightDir, Vec3f &lightIntensity , float &dist)
    const
    {
        lightDir = pos - P;
        // compute the square distance
        float r2 = lightDir.norm();
        dist = sqrtf(r2);
        // normalize the incident light ray direction
        lightDir.x /= dist, lightDir.y /= dist, lightDir.z /= dist;
        // apply square falloff
        lightIntensity = intensity * color / (4 * M_PI * r2);
    }
};
```

Then we will set the shadow ray $tNear$ variable to the result of `dist`. In fact this can be done directly if we pass the `isectShad.tNear` variable directly to the `getShadingInfo()` method of the light (we can change the method's name in the process to make it more generic).

```
IsectInfo isectShad;
light->getShadingInfo(hitPoint, lightDir, lightIntensity, isectShad.tNear);
bool vis = !trace(hitPoint + hitNormal * options.bias,
        -lightDir, objects, isectShad, kShadowRay);
```

What's Next?

We can now simulate diffuse surfaces, and the effect of directional and point light sources. Though, so far, we have only rendered scenes containing one light source. How we can extend the technique to multiple light sources is the topic of the next chapter.