Working with lights and shadows – Part I: Phong reflection model

JULY 6, 2017

Some time ago I promised to write a bit more about how **shadow mapping** works. It has taken me a while to bring myself to actually deliver on that front, but I have finally decided to put together some posts on this topic, this being the first one. However, before we cover shadow mapping itself we need to cover some **lighting basics** first. After all, without light there can't be shadows, right?

This post will introdcuce the popular **Phong reflection model** as the basis for our lighting model. A lighting model provides a simplified representation of how light works in the natural world that allows us to simulate light in virtual scenes at reasonable computing costs. So let's dive into it:

Light in the natural world

In the real world, the light that reaches an object is a combination of both **direct and indirect light**. Direct light is that which comes straight from a light source, while indirect light is the result of light rays hitting other surfaces in the scene, bouncing off of them and eventually reaching the object as a result, maybe after multiple reflections from other objects. Because each time a ray of light hits a surface it loses part of its energy, indirect light reflection is less bright than direct light reflection and its color might have been altered. The contrast between surfaces that are directly hit by the light source and surfaces that only receive indirect light is what creates shadows. A shadow isn't but the part of a scene that doesn't receive direct light but might still receive some amount of (less intense) indirect light.

Recent Posts

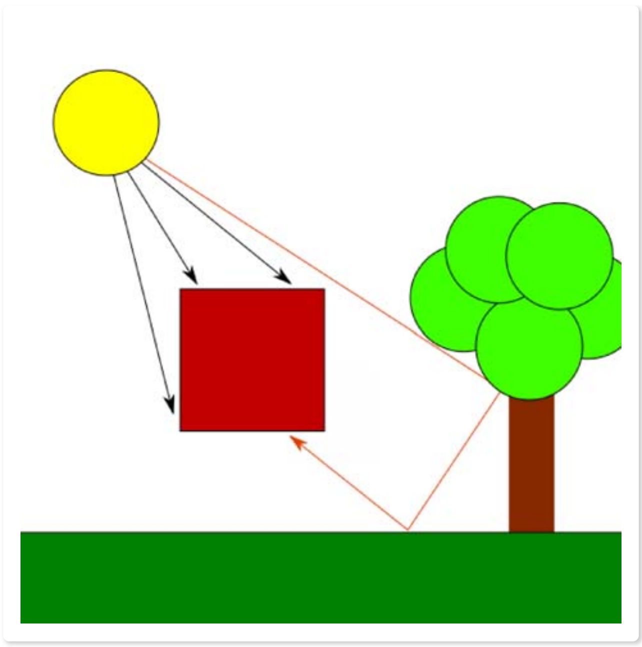
VK_KHR_shader_float16_int8 on Anvil
Intel Mesa Vulkan driver now supports shaderInt16
Frame analysis of a rendering of the Sponza model
Improving shader performance with Vulkan's specialization constants
Intel Mesa driver for Linux is now Vulkan 1.1 conformant

Recent Comments

Hans-Kristian on VK_KHR_shader_float16_int8 on Anvil
Iago Toral on Intel Mesa Vulkan driver now supports shaderInt16
oscar on Intel Mesa Vulkan driver now supports shaderInt16
J.A. on Intel Mesa driver for Linux is now OpenGL 4.6 conformant
J.A. on OpenGL terrain renderer: rendering the terrain mesh

Archives

December 2018
May 2018
April 2018
March 2018
February 2018
October 2017
July 2017
January 2017
November 2016


Direct vs Indirect light

Light in the digital world

Unfortunately, implementing realistic light behavior like that is too expensive, specially for real-time applications, so instead we use simplifications that can produce similar results with much lower computing requirements. The *Phong reflection model* in particular describes the light reflected from surfaces or emitted by light sources as the combination of 3 components: **diffuse, ambient and specular**. The model also requires information about the direction in which a particular surface is facing, provided via vectors called **surface normals**. Let's introduce each of these concepts:

Surface normals

When we study the behavior of light, we notice that the direction in which surfaces reflect incoming light affects our perception of the surface. For example, if we lit a shiny surface (such as a piece of metal) using a strong light shource so that incoming light is reflected off the surface in the exact opposite direction in which we are looking at it, we will see a strong reflection in the form of highlights. If we move around so that we look at the same surface from a different angle, then we will see the reflection get dimmer and the highlights will eventually disappear. In order to model this behavior we need to know the direction in which the surfaces we render reflect incoming light. The way to do this is by associating vectors called *normals* with the surfaces we render so that shaders can use that information to produce lighting calculations akin to what we see in the natural world.

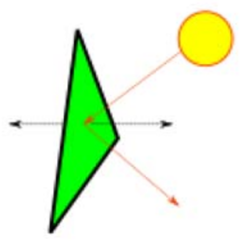
Usually, modeling programs can compute normal vectors for us, even model loading libraries can do this work automatically, but some times, for example when we define vertex meshes programatically, we need to define them manually. I won't covere here how to do this in the general, you can see this article from Khronos if you're interested in specific algorithms, but I'll point out something relevant: given a plane, we can compute normal vectors in two opposite directions, one is correct for the front face of the plane/polygon and

October 2016
August 2016
July 2016
July 2015
May 2015
April 2015
March 2015
November 2014
September 2014
August 2014
July 2014
February 2014
January 2014
December 2013
June 2013
May 2013
June 2011
May 2011
August 2010
June 2010
May 2010
April 2010
March 2010
February 2010
October 2009
July 2009
June 2009
March 2009
June 2008
April 2008
October 2007
July 2007
June 2007
May 2007
March 2007
February 2007
November 2006
October 2006
September 2006
August 2006
July 2006
June 2006
March 2006

Categories

graphics
Maemo
Uncategorized
webkit

the other one is correct for the back face, so make sure that if you compute normals manually, you use the correct direction for each face, otherwise you won't be reflecting light in the correct direction and results won't be as you expect.



Light reflected using correct normal vector for the front face of the triangle

In most scenarios, we only render the front faces of the polygons (by enabling back face culling) and thus, we only care about one of the normal vectors (the one for the front face).

Another thing to notice about normal vectors is that they need to be transformed with the model to be correct for transformed models: if we rotate a model we need to rotate the normals too, since the faces they represent are now rotated and thus, their normal directions have rotated too. Scaling also affects normals, specifically if we don't use regular scaling, since in that case the orientation of the surfaces may change and affect the direction of the normal vector. Because normal vectors represent directions, their position in world space is irrelevant, so for the purpose of lighting calculations, a normal vector such as (1, 0, 0) defined for a surface placed at (0, 0, 0) is still valid to represent the same surface at any other position in the world; in other words, we do not need to apply translation transforms to our normal vectors.

In practice, the above means that we want to apply the rotation and scale transforms from our models to their normal vectors, but we can skip the translation transform. The matrix representing these transforms is usually called the *normal matrix*. We can compute the normal matrix from our model matrix by computing the transpose of the inverse of the 3×3 submatrix of the model matrix. Usually, we'd want to compute this matrix in the application and feed it to our vertex shader like we do with our model matrix, but for reference, here is how this can be achieved in the shader code itself, plus how to use this matrix to transform the original normal vectors:

```
mat3 NormalMatrix = transpose(inverse(mat3(ModelMatrix)))
vec3 out_normal = normalize(NormalMatrix * in_normal);
```

Notice that the code above normalizes the resulting normal before it is fed to the fragment shader. This is important because the rasterizer will compute normals for all fragments in the surface automatically, and for that it will interpolate between the normals for each vertex we emit. For the interpolated normals to be correct, all vertex normals we output in the vertex shader must have the same length, otherwise the larger normals will deviate the direction of the interpolated vectors towards them because their larger size will increase their *weight* in the interpolation computations.

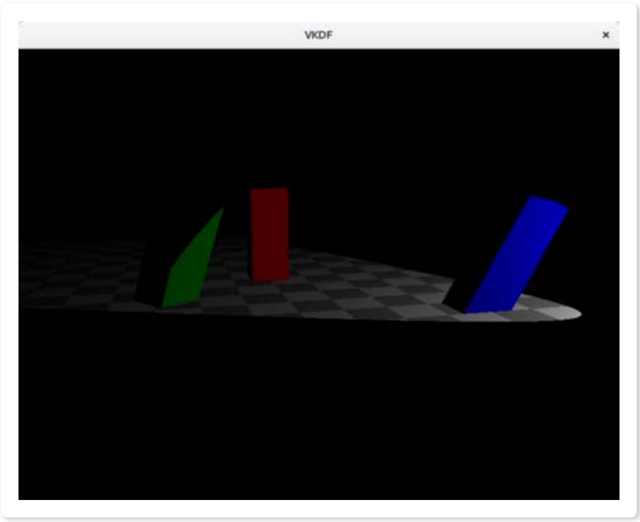
Meta

Log in
Entries RSS
Comments RSS
WordPress.org

Finally, even if we emit normalized vectors in the vertex shader stage, we should note that the interpolated vectors that arrive to the fragment shader are not guaranteed to be normalized. Think for example of the normal vectors (1, 0, 0) and (0, 1, 0) being assigned to the two vertices in a line primitive. At the half-way point in between these two vertices, the interpolator will compute a normal vector of (0.5, 0.5, 0), which is not unit-sized. This means that in the general case, input normals in the fragment shader will need to be normalized again even if have normalized vertex normals at the vertex shader stage.

Diffuse reflection

The diffuse component represents the reflection produced from direct light. It is important to notice that the intensity of the diffuse reflection is affected by the angle between the light coming from the source and the normal of the surface that receives the light. This makes a surface looking straight at the light source be the brightest, with reflection intensity dropping as the angle increases:



Diffuse light (spotlight source)

In order to compute the diffuse component for a fragment we need its normal vector (the direction in which the surface is facing), the vector from the fragment's position to the light source, the diffuse component of the light and the diffuse reflection of the fragment's material:

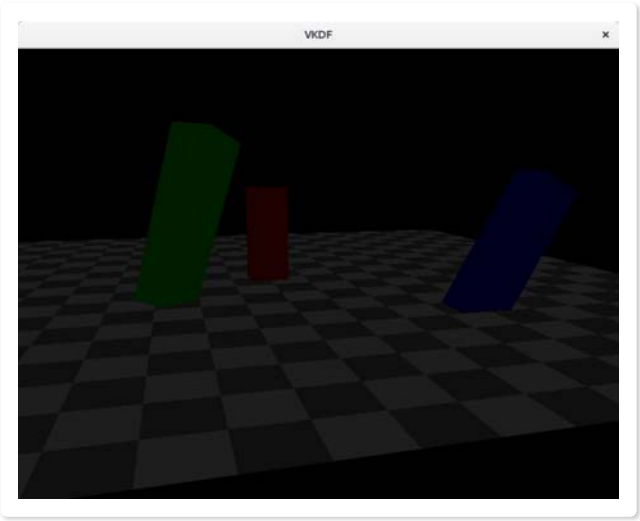
```
vec3 normal = normalize(surface_normal);
vec3 pos_to_light_norm = normalize(pos_to_light);
float dp_reflection = max(0.0, dot(normal, pos_to_light_n
vec3 diffuse = material.diffuse * light.diffuse * dp_refl
```

Basically, we multiply the diffuse component of the incoming light with the diffuse reflection of the fragment's material to produce the diffuse component of the light reflected by the fragment. The diffuse component of the surface tells how the object absorbs and reflects incoming light. For example, a pure yellow object (diffuse material vec3(1,1,0)) would absorb the blue component and reflect 100% of the red and green components of the incoming light. If the light is a pure white light (diffuse vec3(1,1,1)), then the observer would see a yellow object. However, if we are using a red light instead (diffuse vec3(1,0,0)), then the light reflected from the surface of the object would only contain the red component (since the light isn't emitting a green component at all) and we would see it red.

As we said before though, the intensity of the reflection depends on the angle between the incoming light and the direction of the reflection. We account for this with the dot product between the normal at the fragment (surface_normal) and the direction of the light (or rather, the vector pointing from the fragment to the light source). Notice that because the vectors that we use to compute the dot product are normalized, *dp_reflection* is exactly the cosine of the angle between these two vectors. At an angle of 0º the surface is facing straight at the light source, and the intensity of the diffuse reflection is at its peak, since cosine(0º)=1. At an angle of 90º (or larger) the cosine will be 0 or smaller and will be clamped to 0, meaning that no light is effectively being reflected by the surface (the computed diffuse component will be 0).

Ambient reflection

Computing all possible reflections and bounces of all rays of light from each light source in a scene is way too expensive. Instead, the Phong model approximates this by making indirect reflection from a light source constant across the scene. In other words: it assumes that the amount of indirect light received by any surface in the scene is the same. This eliminates all the complexity while still producing reasonable results in most scenarios. We call this constant factor ambient light.



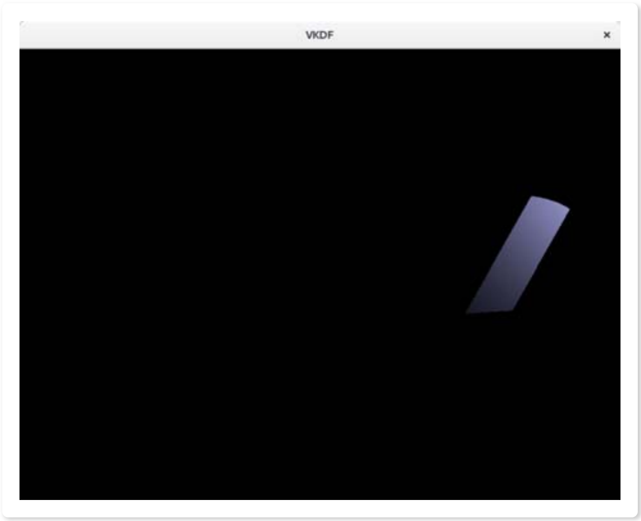
Ambient light

Adding ambient light to the fragment is then as simple as multiplying the light source's ambient light by the material's ambient reflection. The meaning of this product is exactly the same as in the case of the diffuse light, only that it affects the indirect light received by the fragment:

```
vec3 ambient = material.ambient * light.ambient;
```

Specular reflection

Very sharp, smooth surfaces such as metal are known to produce specular highlights, which are those bright spots that we can see on shiny objects. Specular reflection depends on the angle between the observer's view direction and the direction in which the light is reflected off the surface. Specifically, the specular reflection is strongest when the observer is facing exactly in the opposite direction in which the light is reflected. Depending on the properties of the surface, the specular reflection can be more or less focused, affecting how the specular component scatters after being reflected. This property of the material is usually referred to as its shininess.



Specular light

Implementing specular reflection requires a bit more of work:

```
vec3 specular = vec3(0);
vec3 light_dir_norm = normalize(vec3(light.direction));
if (dot(normal, -light_dir_norm) >= 0.0) {
   vec3 reflection_dir = reflect(light_dir_norm, normal);
   float shine_factor = dot(reflection_dir, normalize(in_
   specular = light.specular.xyz * material.specular.xyz
         pow(max(0.0, shine_factor), material.shininess.x
}
```

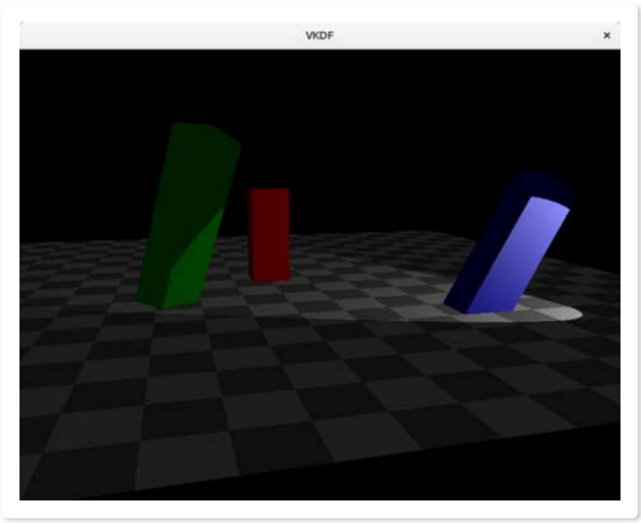
Basically, the code above checks if there is any specular reflection at all by computing the cosine of the angle between the fragment's normal and the direction of the light (notice that, once again, both vectors are normalized prio to using them in the call to dot()). If there is specular reflection, then we compute how shiny the reflection is perceived by the viewer based on the angle between the vector from this fragment to the observer (in_view_dir) and the direction of the light reflected off the fragment's surface (reflection_dir). The smaller the angle, the more parallel the directions are, meaning that the camera is receiving more reflection and the specular component received is stronger. Finally, we modulate the result based on the **shininess** of the fragment. We can compute *in_view_dir* in the vertex shader using the inverse of the View matrix like this:

```
mat4 ViewInv = inverse(View);
out_view_dir =
   normalize(vec3(ViewInv * vec4(0.0, 0.0, 0.0, 1.0) - wo
```

The code above takes advantage of the fact that camera transformations are an illusion created by applying the transforms to everything else we render. For example, if we want to create the illusion that the camera is moving to the right, we just apply a translation to everything we render so they show up a bit to the left. This is what our *View* matrix achieves. From the point of view of GL or Vulkan, the camera is always fixed at (0,0,0). Taking advantage of this, we can compute the position of the virtual observer (the camera) in world space coordinates by applying the inverse of our camera transform to its fixed location (0,0,0). This is what the code above does, where *world_pos* is the position of this vertex in world space and *View* is the camera's view matrix.

In order to produce the final look of the scene according to the Phong reflection model, we need to compute these 3 components for each fragment and add them together:

```
out_color  = vec4(diffuse + ambient + specular, 1.0)
```



Diffuse + Ambient + Specular (spotlight source)

Attenuation

In most scenarios, light intensity isn't constant across the scene. Instead, it is brightest at its source and gets dimmer with distance. We can easily model this by adding an attenuation factor that is multiplied by the distance from the fragment to the light source. Typically, the intensity of the light decreases quite fast with distance, so a linear attenuation factor alone may not produce the best results and a quadratic function is preferred:

```
float attenuation = 1.0 /
    (light.attenuation.constant +
     light.attenuation.linear * dist +
     light.attenuation.quadratic * dist * dist);

diffuse = diffuse * attenuation;
ambient = ambient * attenuation;
specular = specular * attenuation;
```

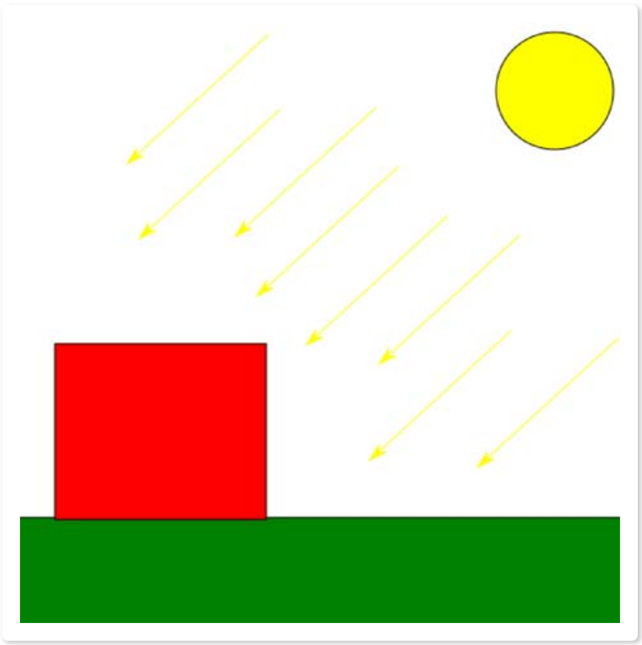
Of course, we may decide not to apply attenuation to the ambient component at all if we really want to make it look like it is constant across the scene, however, do notice that when multiple light sources are present, the ambient factors from each source will accumulate and may produce too much ambient light unless they are attenuated.

Types of lights

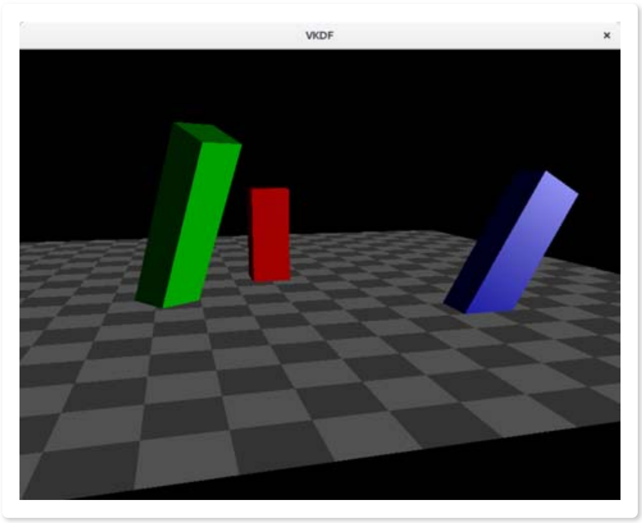
When we model a light source we also need to consider the kind of light we are manipulating:

Directional lights

These are light sources that emit rays that travel along a specific direction so that all are parallel to each other. We typically use this model to represent bright, distant light sources that produce constant light across the scene. An example would be the sun light. Because the distance to the light source is so large compared to distances in the scene, the attenuation factor is irrelevant and can be discarded. Another particularity of directional light sources is that because the light rays are parallel, shadows casted from them are regular (we will talk more about this once we cover shadow mapping in future posts).


Directional light

If we had used a directional light in the scene, it would look like this:

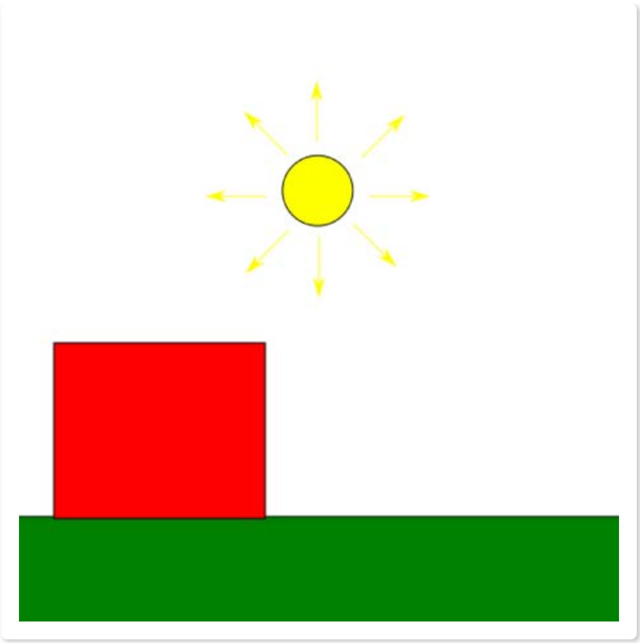


Scene with a directional light

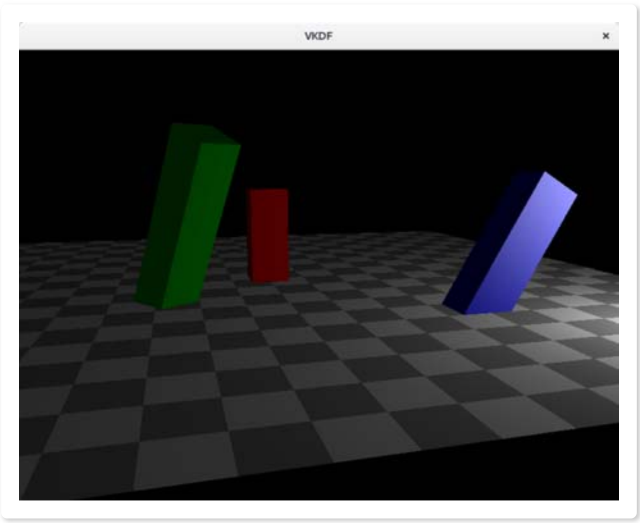
Notice how the brightness of the scene doesn't lower with the distance to the light source.

Point lights

These are light sources for which light originates at a specific position and spreads outwards in all directions. Shadows casted by point lights are not regular, instead they are projected. An example would be the light produced by a light bulb. The attenuation code I showed above would be appropriate to represent point lights.


Point light

Here is how the scene would look like with a point light:

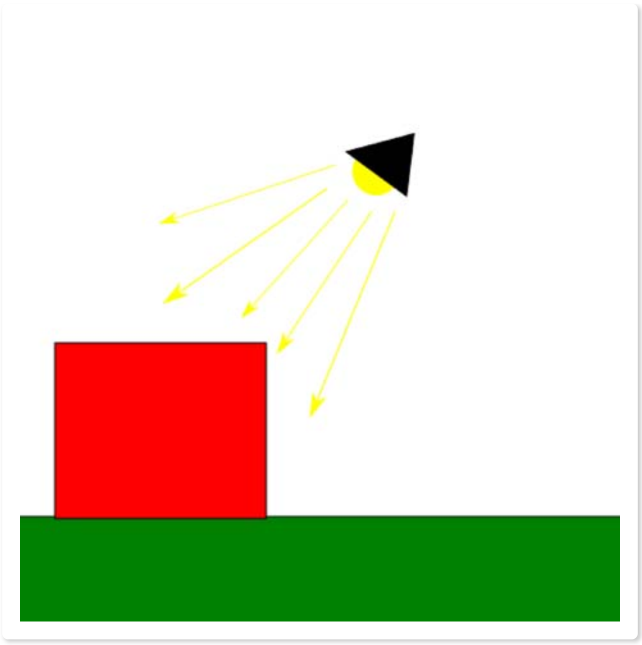


Scene with a point light

In this case, we can see how attenuation plays a factor and brightness lowers as we walk away from the light source (which is close to the blue cuboid).

Spotlights

This is the light source I used to illustrate the diffuse, ambient and specular components. They are similar to point lights, is the sense that light originates from a specific point in space and spreads outwards, however, instead of scattering in all directions, rays scatter forming a cone with the tip at the origin of the light. The angle formed by the lights's direction and the sides of the cone is usually called the cutoff angle, because not light is casted outside its limits. Flashlights are a good example of this type of light.


Spotlight

In order to create spotlights we need to consider the cutoff angle of the light and make sure that no diffuse or specular component is reflected by a fragment which is beyond the cutoff threshold:

```
vec3 light_to_pos_norm = -pos_to_light_norm;
float dp = dot(light_to_pos_norm, light_dir_norm);
if (dp <= light.cutoff) {
   diffuse = vec3(0);
   specular = vec3(0);
}
```

In the code above we compute the cosine of the angle between the light's direction and the vector from the light to the fragment (dp). Here, *light.cutoff* represents the cosine of the spotlight's cutoff angle too, so when *dp* is smaller it means that the fragment is outside the light cone emitted by the spotlight and we remove its diffuse and specular reflections completely.

Multiple lights

Handling multiple lights is easy enough: we only need to compute the color contribution for each light separately and then add all of them together for each fragment (pseudocode):

```
vec3 fragColor = vec3(0);
foreach light in lights
    fragColor += compute_color_for_light(light, ...);
...
```

Of course, light attenuation plays a vital role here to limit the area of influence of each light so that scenes where we have multiple lights don't get too bright.

An important thing to notice above the pseudocode above is that this process involves looping through costy per-fragment light computations for each light source, which can lead to important performance hits as the number of lights in the scene increases. This shading model, as described here, is called **forward rendering** and it has the benefit that it is very simple to implement but its downside is that we may incur in many costy lighting computations for fragments that, eventually, won't be visible in the screen (due to them being occluded by other fragments). This is particularly important when the number of lights in the scene is quite large and its complexity makes it so that there are many occluded fragments. Another technique that may be more suitable for these situations is called **deferred rendering**, which postpones costy shader computations to a later stage (hence the word *deferred*) in which we only evaluate them for fragments that are known to be visible, but that is a topic for another day, in this series we will focus on *forward rendering* only.

Lights and shadows

For the purpose of **shadow mapping** in particular we should note that objects that are directly lit by the light source reflect all 3 of the light components, while objects in the shadow only reflect the ambient component. Because objects that only reflect ambient light are less bright, they appear shadowed, in similar fashion as they would in the real world. We will see the details how this is done in the next post, but for the time being, keep this in mind.

Source code

The scene images in this post were obtained from a simple shadow mapping demo I wrote in **Vulkan**. The source code for that is available here, and it includes also the shadow mapping implementation that I'll cover in the next post. Specifically relevant to this post are the scene vertex and fragment shaders where lighting calculations take place.

Conclusions

In order to represent shadows we first need a means to represent light. In this post we discussed the Phong reflection model as a simple, yet effective way to model light reflection in a scene as the addition of three separate components: diffuse, ambient and specular. Once we have a representation of light we can start discussing shadows, which are parts of the scene that only receive ambient light because other objects occlude the diffuse and specular components of the light source.

BOOKMARK THE PERMALINK.

← GL_ARB_gpu_shader_fp64 / OpenGL 4.0 lands for Intel/Haswell. More gen7 support coming soon!

Working with lights and shadows – Part II: The shadow map →

Comments are closed.