Getting to know the Phong reflection model

A simple implementation with a ray tracer in C++, with SDL2 and GLM

DH2323 Computer Graphics and Interaction

Josephine Kvarnberg
jkv@kth.se

20/05/2022

Abstract
The aim of this project was to get to know the Phong Reflection model through a simple implementation in working with ray tracing in C++. The work was done as an extension of the code written for Lab 2 in the course DH2323, Computer Graphics and Interaction. The Phong Reflection model generally consists of three components: a constant ambient term, a diffuse term and a specular term. In some cases there will be a fourth component of emissive light, but this is usually omitted from the model. The implementation was done in steps and is documented in this report. The result is then discussed, and questions and concerns that came up during the implementation are handled. Suggestions on further implementations and future work will then be mentioned.

The progress of the project is documented on the following blog:
https://lillekvarre.wixsite.com/website

1. Introduction

Ray tracing is a way of rendering an arrangement of 3D objects to a scene represented by a 2D image, shown from a specific viewpoint. It works by computing one pixel at a time, and the output will be an array of pixels to be put on the screen. The ray tracing method of rendering can be described in three steps. First, it has to generate a ray and compute what ray (or vector) will point from the viewer to the pixel in question. This will be based on the placement of the camera, or viewer. Secondly, an intersection will be calculated for the closest object that will intersect the viewing ray. Lastly, shading will be done to compute what color the pixel will have based on the intersection that occurred [1].

There are different ways of rendering what colors the pixels will have. A straightforward way is assigning the pixels the color of the object that is intersected. This will give a flat but okay looking scene. However, there are ways of rendering reflections and light to give the scene a sense of life and dynamics. Adding a light source will give the opportunity to calculate shadows, reflections and other things that light rays might affect.

There are two different types of illumination at any given point in a scene, direct and indirect [3]. When working with illumination, you can therefore choose what you want to include in your computations, maybe based on computational efficiency or something else.

A model can be global and consider both direct and indirect light. A model can also be local and only consider direct light. As an example, in lab 2 of this course only direct light is considered as calculating bouncing rays would be computationally heavy. Instead, indirect light is lightly simulated by adding a constant vector.

In this project, the Phong reflection model will be in the center of attention. It is a local model, working with direct light for creating illumination in scenes. Light is approximated to consist of three components, the RGB values, and in previous labs we have worked with vec3 vectors to simulate light color [3].

When going deeper into light, computer graphics and reflections, one will come across the BRDF model. The bidirectional reflectance distribution function is a function that tells us how much light is reflected at an opaque surface. There are several models that can be generalized as BRDFs, like the Lambertian model, the Cook-Torrance model, and of course the Phong reflection model, to name a few.

The function is used in computer graphics algorithms. The light reflected depends on what material the surface is of. It takes the incoming light direction and outgoing direction, and returns the ratio of reflected radiance that goes in the outgoing direction. In short, it is a model that describes how an object will scatter or reflect light. For such a model to be physically realistic, it has to fulfill some properties. These are the functions being positive, it has to be reciprocal (e.g if the incoming and outgoing directions are swapped, the function should be the same), and it should conserve energy and not create more light than received. This last property is related to the emissive component mentioned before. The surface of an object will not create more light unless it is glowing or emitting light, like a sun, but that is not the case in this project [1,4,5,6].

The models that are BRDFs can be more or less adhering to the properties of being physically realistic. The Phong reflection model will produce rather plastic-looking surfaces, based on how it handles specularity. It is a computationally efficient model, but it is not physically correct.

The Phong reflection model, or illumination model, was introduced by Phong and later on modified by Blinn. The model will work with illuminating locally using direct light and do not take bouncing secondary light rays into account. The components of it will be described in the next part of the report.

2. Background

The Phong reflection model generally consists of three components, being ambient light, diffuse light, and specular light. In some cases there will be a fourth component of emissive light, but this is usually omitted from the model.

All the formulas in section 2.1 and 2.2 are screenshotted from the book by Angel and Shreiner [2].

2.1 Components

The ambient component is an additive one. Ambient lighting is achieved when there is a uniform light level throughout a space. The intensity of the illumination is identical at every point in the space or scene.

$$I_{\mathrm{a}} = k_{\mathrm{a}} L_{\mathrm{a}}.$$

La is the ambient light intensity, and can be set to the global ambient term. Ka is the ambient surface constant, and should be a number between 0 and 1.

Diffuse lighting will occur when the incident light is reflected equally in all directions from the surface. The amount of light that is reflected depends on the incident angle in relation to the surface normal.

$$I_{\mathrm{d}} = k_{\mathrm{d}}(\mathbf{l} \cdot \mathbf{n})L_{\mathrm{d}}.$$

Ld is the diffuse light intensity. Kd is the diffuse surface constant. This last part is the dot product between the surface normal n and the light direction l. In practice, in can be good to use max((l*n)Ld, 0) to make sure the expression is not negative if the light source lies below the horizon. A distance term can also be added to the diffuse component, if we wish to take into account that light weakens by the distance d that the light travels.

The specular component adds highlights to surfaces. In nature, a smooth surface will have a bigger reflection than a rough surface. The light that is reflected depends on the incident angle. "Specular surfaces appear shiny because most of the light that is reflected or scattered is in a narrow range of angles close to the angle of reflection." - Angel and Shreiner [2], p.26.

$$I_{\mathrm{s}} = k_{\mathrm{s}}L_{\mathrm{s}}\max((\mathbf{r} \cdot \mathbf{v})^{\alpha}, 0).$$

Ks is the specular surface constant. Ls is the specular light intensity. The angle between r, the direction of a perfect reflector, and v, the direction of the viewer, determines the amount of light the viewer will see. The size of the specular highlights is regulated by a shininess constant α, based on the surface material properties. Again, we make use of the function max to make sure the expression is not negative.

The emissive component is purely additive and can be disregarded for most materials of surfaces. Emission will only occur in a surface that emits light, e.g objects that glow or shine bright. The emissive component is usually omitted from simple models like Phong, if you are not modeling objects that glow.

2.2 The total model
All the components are added together into one formula, which is computed for each light source (if there are more than one). In the case of this project, there will be only one light source.

$$I = \frac{1}{a + bd + cd^2}(k_{\mathrm{d}}L_{\mathrm{d}}\max(\mathbf{l} \cdot \mathbf{n}, 0) + k_{\mathrm{s}}L_{\mathrm{s}}\max((\mathbf{r} \cdot \mathbf{v})^{\alpha}, 0)) + k_{\mathrm{a}}L_{\mathrm{a}}.$$

In the formula above, there is also a distance term added to account for how light weakens as it travels a distance. This was not added in this project, and is rather a further implementation to be tried later on.

3. Method

The project was done using the language C++ and building onto a previous computer laboration in the course DH2323 Computer Graphics and Interaction. The previous work is lab 2 which revolves around ray tracing and implementing illumination and shading. It makes use of the libraries SDL2 and GLM, which is the OpenGL library for mathematics.
The code was written using Visual Studio Code, VSC, and the whole project in code was made into an executable using Cmake.

A code skeleton is used as a basis for this project. It was initially based on the skeleton by user Lemonad on GitHub, and expanded as part of lab 2 in the course DH2323. As the laboration focuses on implementing ray tracing and some illumination and shadows, it is suitable to have it as a base and extend with the Phong reflection model. Variables, structures and functions declared in lab 2 were reused and altered. This is commented on in the code.

In lab 2, Ray tracing, in the course DH2323, the implementation of the function DirectLight for direct illumination is done by using a glm::vec3 vector component. It is implemented as a function and takes an intersection as an argument. That enables the function to handle the position of the direct illumination, the index of the triangle that is illuminated, as well as the normal of the surface. The function then returns the direct illumination as a vec3 vector. To make shadows appear in the scene, it also calls on ClosestIntersection to

vec3 DirectLight(const Intersection& i);

The base code also has a component for indirect illumination. As the instructions for the lab 2 describes, it is very computationally expensive to simulate an accurate computation of indirect illumination, where light bounces multiple times. Instead of doing that, the indirect illumination is done by using a constant approximation. This will not be used in this project.

vec3 indirectLight = 0.5f*vec3(1, 1, 1);

To use DirectLight, the function is called upon in the Draw function, which uses two for-loops to loop through all pixels and compute the corresponding ray direction. The function ClosestIntersection is called in the two for-loops to compute the closest intersection in that direction, and will return the boolean true if an intersection occurs. It will take the arguments of the starting point of the ray, direction of the ray, and a vector of triangles. It will also take an argument of the struct Intersection, which will be updated with the 3D position of the closest intersection.

Implementation of the Phong reflection model will be done component by component, to see what each of them add to the scene.

4. Results

4.1 Implementation

The first part of this project was making sure the setup and base code is working. I used my final version of lab 2, Ray tracing, in DH2323, based on the skeleton by Lemonad on Github [7]. This gave me a working ray tracer, a rendered scene and direct and simple indirect light.

The next part was making the outline of the Phong reflection model. I tried making void() functions for each of the components but VSC gave an error when trying to return a vec3 from it. Instead, I had to change it to vec3() functions, as in lab 2 with DirectLight which is also a vec3 function.

Moving onto the first coding, I disabled the old code from lab 2 that would give illumination in the scene. That left the rendered scene very flat, see figure 1. I worked with the implementation in the same way as DirectLight was working before, meaning the function returns the illumination and this is later multiplied with the color for each pixel that will be drawn in the Draw function.

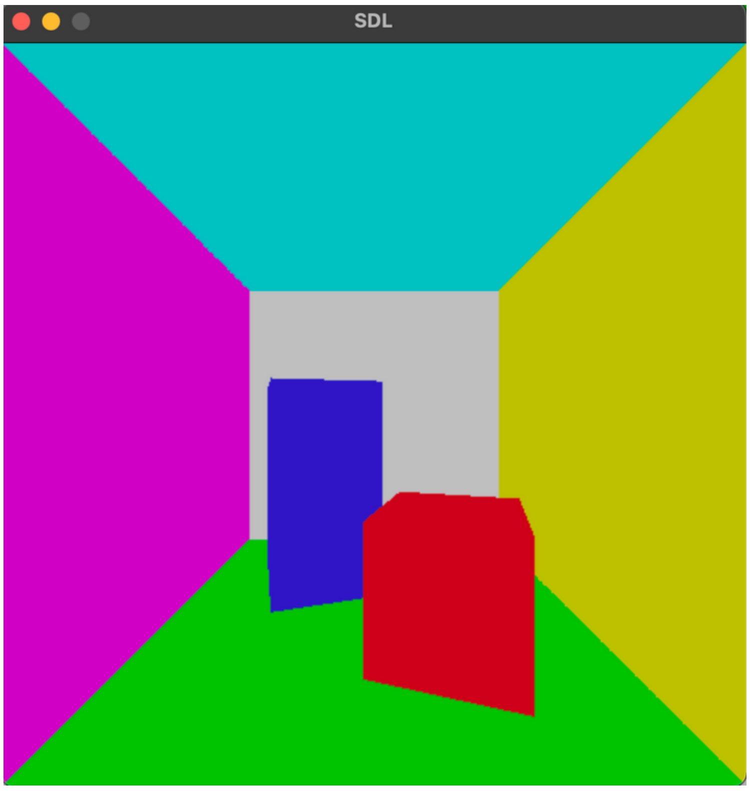


Figure 1: A first render of the scene without any illumination

I went with the ambient lighting as a trial. This component of the model is probably the easiest one. I found a comprehensive table of different values for surface constants and chose a random one that was supposed to be for silver as a material [8]. The rendered scene in figure 2 differs from the one in figure 1, which is an indication that something is happening at least. The light level seems to be uniform throughout the scene and on its objects.

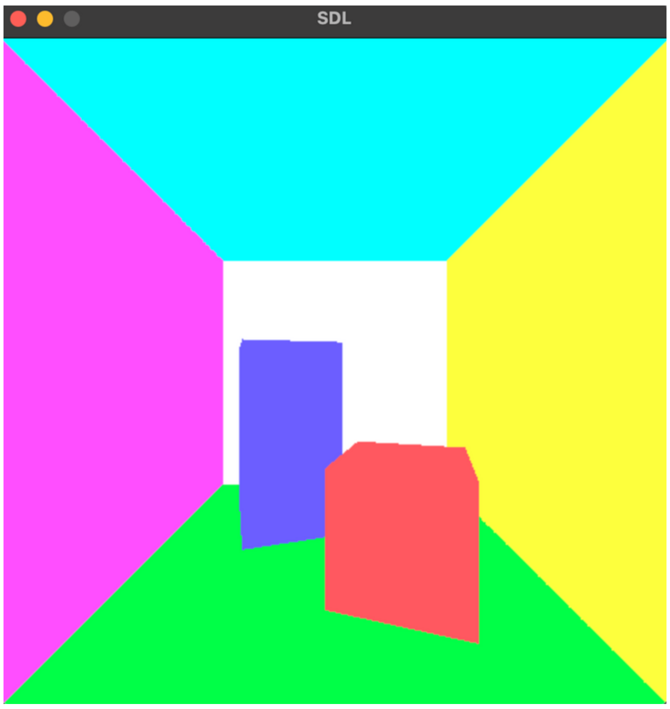
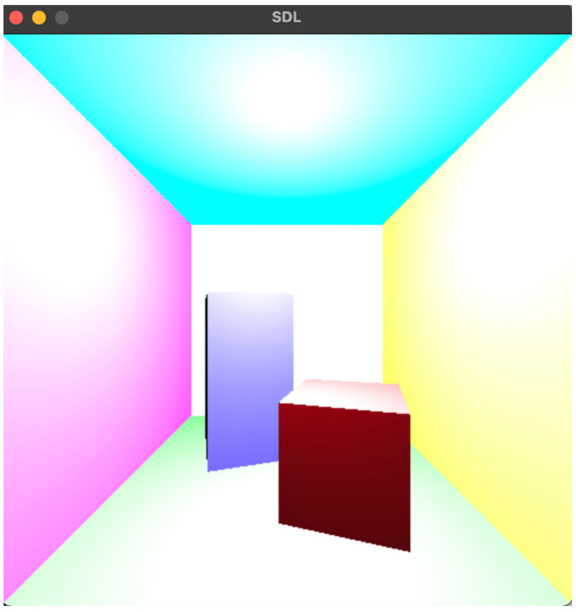


Figure 2: Rendered scene with the ambient component added

The diffuse component is not as easy as the ambient one. Still, the formula is not that difficult to understand. The vectors that are supposed to be used in the formula were easy to get a hold of, since they were used in the DirectLight function in lab2. As described in the book by Angel and Shreiner [2], there could be a potential problem with the expression for the component because (**l** . **n**)*Ld* will be negative if the light source is below the horizon. In this case, we want to use zero rather than a negative value. Therefore, I used max((**l** . **n**)*Ld* , 0).

I implemented the first try of the code straight forward, as in the ambient part. This can be seen in figure 3. It is very bright, almost hard to look at, so I tried one render with a lower lightColor intensity (diffuse light intensity). The first one is set to 14.f*vec3(1,1,1) and the second to 2.f*vec3(1,1,1), which is seen in figure 4..



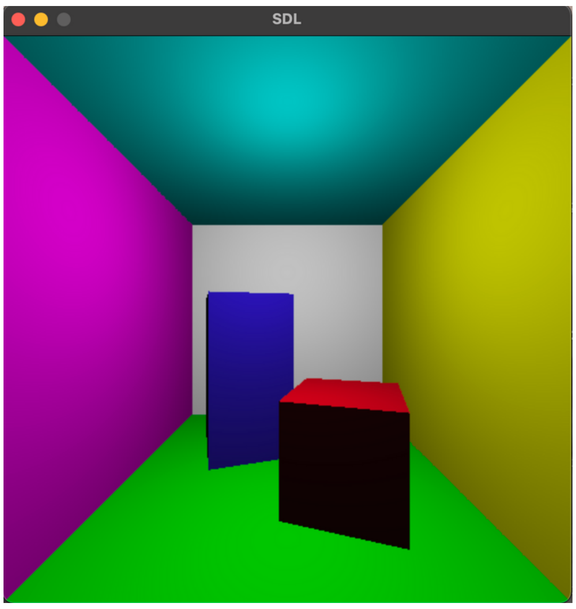
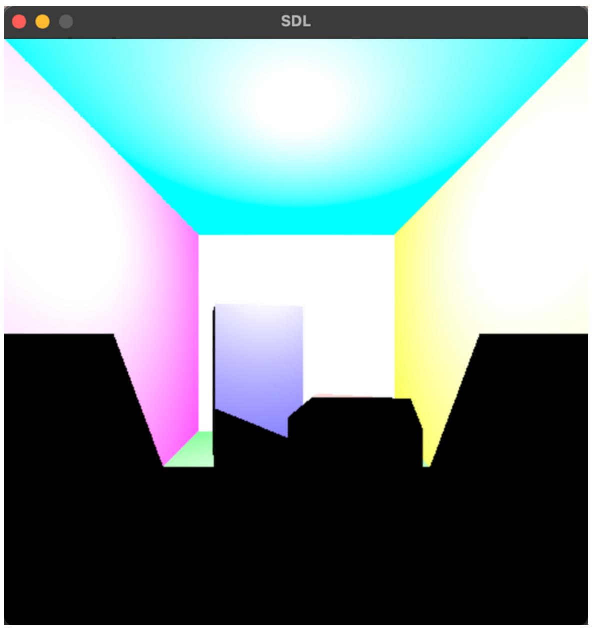


Figure 3 (left): Diffuse lighting with lightColor intensity (diffuse light intensity) of 14.f*vec3(1,1,1)
Figure 4 (right): Diffuse lighting with intensity of 2.f*vec3(1,1,1)

I think the diffuse lighting looks very similar to the illumination done in lab 2, with the spherical spreading of the light on the walls. I noted that there are no shadows in this, which is reasonable since I have only put the formula for diffuse lighting in the code, and not tried to work with putting any of the pixels to black if not intersected with light rays.

In lab2, shadows were created in DirectLight using ClosestIntersection. In the instructions from lab2, it is described like this: "To simulate this we can use the ClosestIntersection function that we use to cast rays. When illuminating a surface point we cast another ray from it to the light source. Then we check the distance to the closest intersecting surface. If that is closer than the light source the surface is in shadow and does not receive any direct illumination from the light source. "

In short, we do the same check for the closest intersections as in the Draw function (the main way of working with the ray tracer) and in DirectLight, to see whether the distance to the closest intersecting surface is closer than the light source. I copied what I had done for this in DirectLight and put this in the Diffuse function, see figure 5. There were some errors. I also tried it with less light intensity, see figure 6.



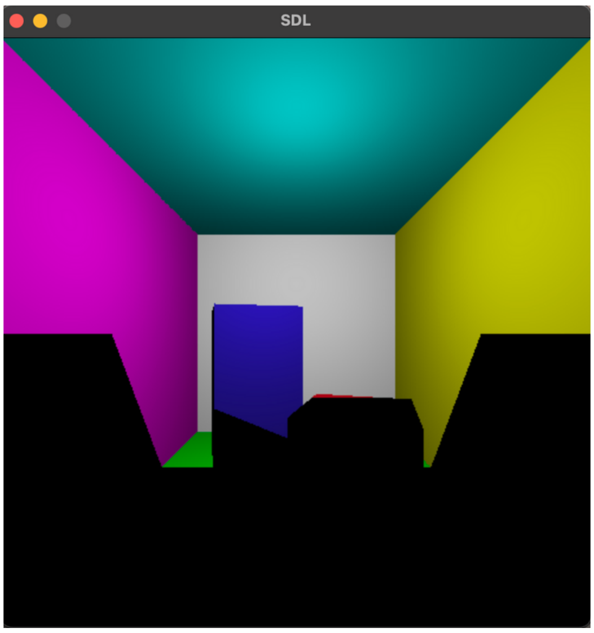


Figure 5 (left): Diffuse with shadows, and errors.
Figure 6 (right): same as in figure 5, but with less diffuse light intensity.

After this, I tested out adding together the Ambient and Diffuse lighting, in the Draw function. This gave me a kind of nice result, but still with faulty shadows. See figure 7.

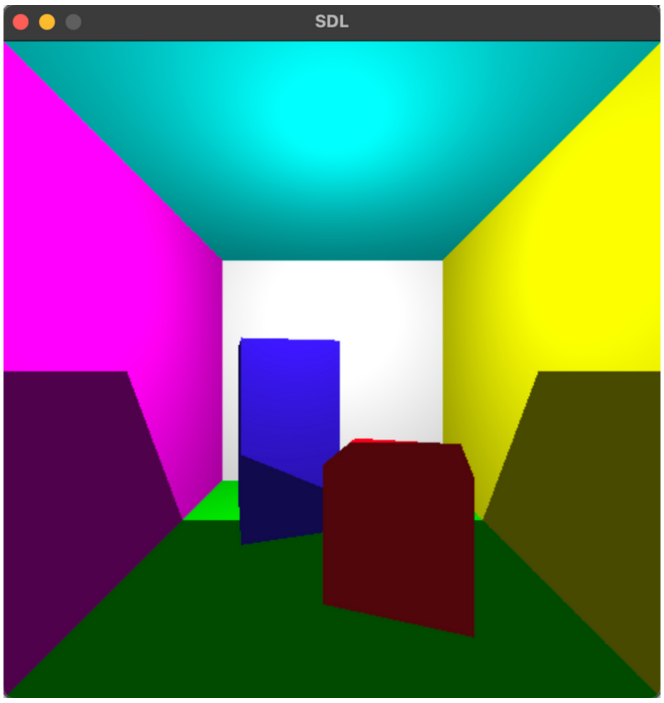


Figure 7: ambient + diffuse, faulty shadows.

I recognized the shape of the faulty shadows and looked at the screenshots I had saved of all previous lab errors. I realized I changed something in the code for the ClosestIntersection call when that is done in DirectLight, so I did a comparison and found that I had forgotten to normalize a vector and changed a sign. This solved the problem and the output of ambient and diffuse lighting now looks like in figure 8.

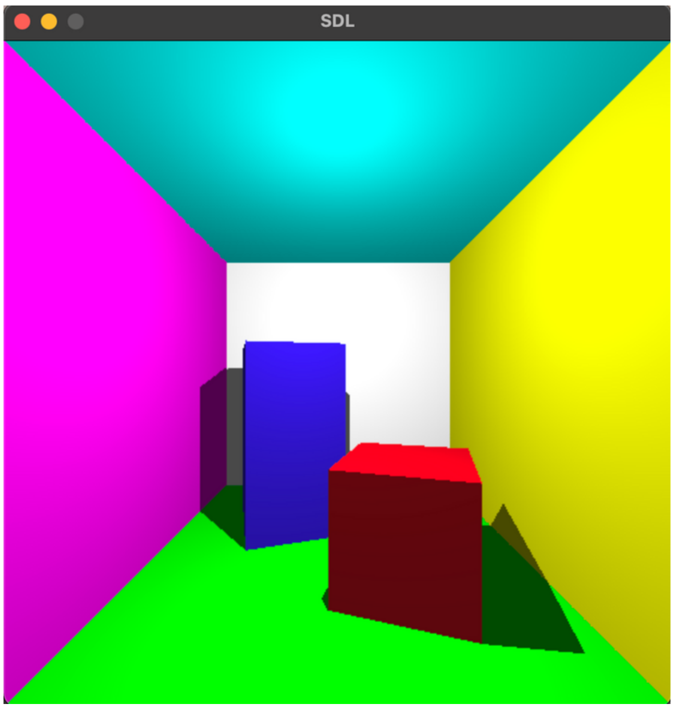


Figure 8: ambient and diffuse lighting with correct shadow.

I played around with the position of the light but ended up going back to the position that I used throughout the labs, which is (0,-0.5,-0.7). It looks good that way.

I took the same approach as with the previous parts with the specular component, and just added the formula in the Specular function and then returned the resulting vector. I added the specular lighting to the color in the Draw function and the executable looked interesting, see figure 9.

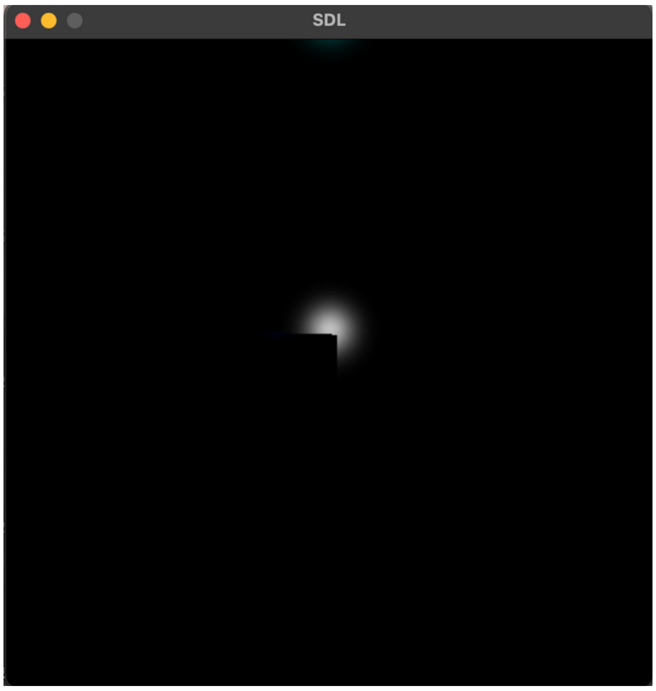


Figure 9: specular illumination of the scene.

I chose Ks, the specular surface constant, to represent silver again. This was just to have some kind of number for all the constants, rather than putting a random one. The size of the specular highlights is regulated by a shininess constant α, based on the surface material properties. This constant can be altered based on Ks, to get the wanted highlight on the surface. These were tried to get a nice looking light, as well as moving the position of the light to get a good view of the highlight. The shininess constant was put to 50, after looking at a picture of specular results from different constants used in the Scratchapixel article [6].

All the components were then added together in the Draw function. In the current position of the light source, there was not much of a difference to the lighting of the scene when the specular component was added, see figure 10. The light intensity was again set to 2.f*vec3(1,1,1) to not be too bright.

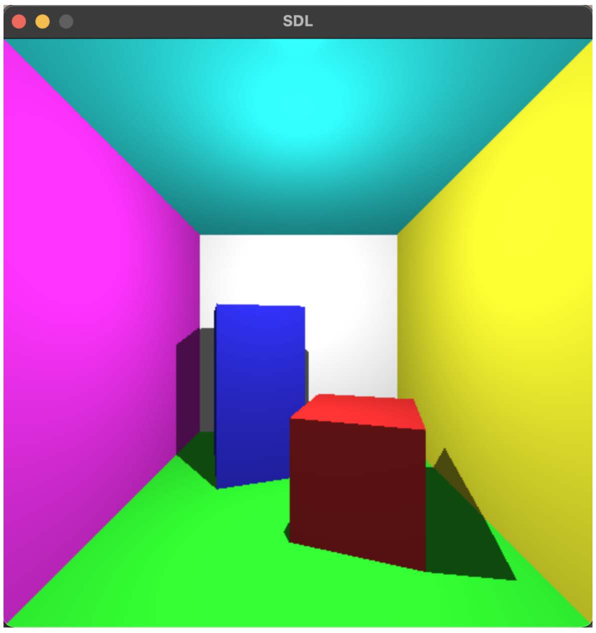
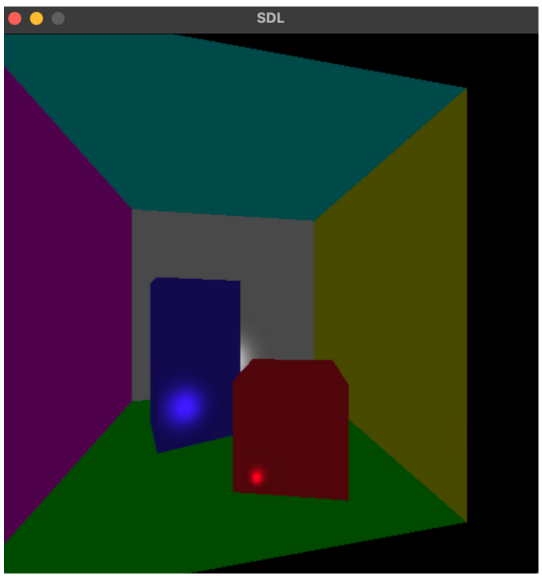


Figure 10: The total Phong illumination of the scene.

To see what actually happens at different places of the scene, the position of the light source was moved from (0, -0.5, -0.7) to something else. It was actually very interesting seeing how this was expressed through the different ways the scene was illuminated. See figure 11 and 12, which shows the light being positioned a bit differently.



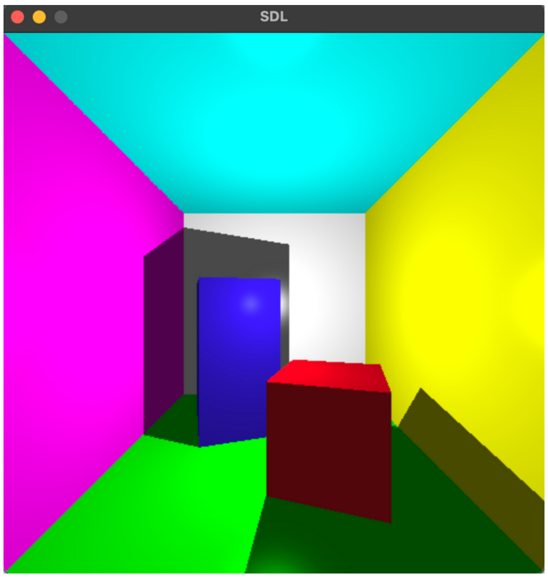


Figure 11 (left): moving of light and camera positions.
Figure 12 (right): another moving of light position.

5. Discussion and further testing

The motivation for looking into the Phong reflection model and its implementation in ray tracing, arose from a previous course where the topic was touched upon. The aim of the project was to do a simple implementation of the Phong reflection model in C++, building onto previous code written in the course. By doing so, the scope is smaller than it would have otherwise been if the model would have been implemented in a ray tracer or other rendering method built from scratch. Overall, the step of going from the labs to the project was not too big and the set up of the project was relatively easy. The Phong reflection model is a basic model that has a lot of documentation and usage online, making it fairly simple to research and get to know.

A question that arose during testing of the complete model was whether some of the light should be distributed as in lab 2 and DirectLight, in a spherical manner. This was tested with the diffuse component being divided by the spherical equation used in DirectLight, but there were no visual differences to the scene.

Another question that comes from reading about the Phong reflection model and whether the light intensities for all three components should be the same ligthColor global variable. According to a presentation on “Rendering Light Reflection Models Direct Illumination” by Donald P. Greenberg [9], the phong equation uses both the light color and the surface color as the light intensity values. In that case, the ambient and diffuse terms would make use of the surface color instead of the light color, while the specular term would still use the light color. This was not something I found documented elsewhere, but it was tested and gave the results seen in figure 13. Subjectively, this looks a lot better than the previous results of the model seen in figure 10, where the specular light is not really visible. Another thing that Greenberg states in his presentation is that the sum of the three surface constants should equal one. This is not the case in this project, where the three constants have been chosen to represent silver [8].

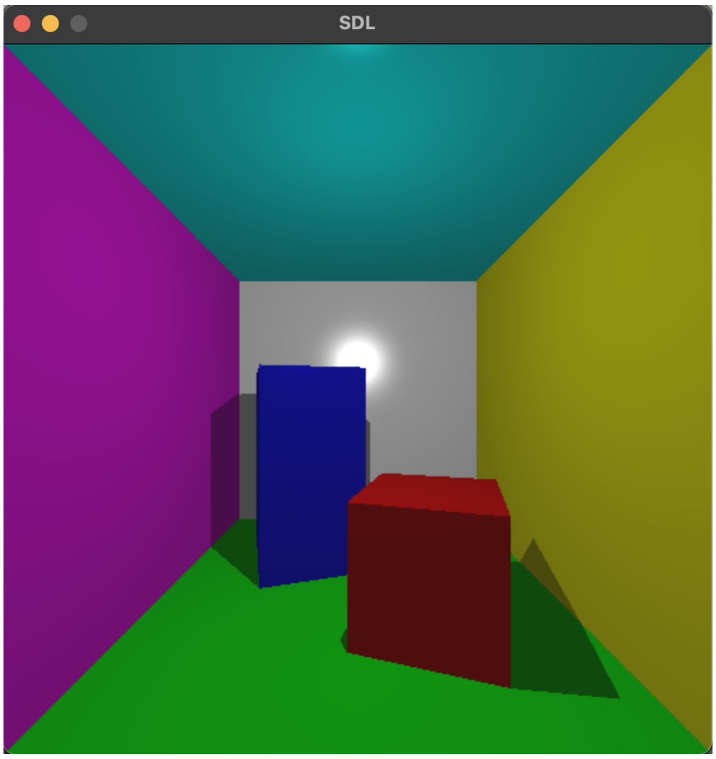


Figure 13: surface constants Ka and Kd as the color of the surface instead of the global lightColor variable.

Working with the model, a choice of how to show shadows could be made. In this project, I chose to show shadows in the scene and made that visible in the Diffuse function. A question that occured was whether this would also be represented in the Specular function, or if that kind of lighting does not interfere with shadows. A test was made by implementing the same handling of whether the distance to the closest intersecting surface is closer than the light source. The output looked exactly the same as in figure 10.

The rendering of the scene is quite efficient and takes about 350 ms to do, with numbers varying from 250 to about 380. However, there are some higher numbers in there that reach about 500 ms or over.

The looks of the scene are aesthetically and subjectively okay, although not being physically correct as described earlier. This was to be expected.

Future work

As described in the book by Angel and Shreiner, a distance term can be added to the model to account for how light will weaken when traveling a distance. This was not taken into account in this project, but it would have been an interesting test to see what this would do to the output.

The modified Phong model, Blinn-Phong, makes a change to the specular component of the model. In the formula for the specular reflection, the dot product is recalculated at every point on the surface. Instead, an approximation can be done using the unit vector halfway between the vector v, the direction of the viewer, and the vector r, the direction of a perfect reflector. Instead, we calculate the cosine angle between the normal vector n and the halfway vector h, to avoid the recalculation. This is an approach that would be interesting to try further on, and see what changes there are to the efficiency and output of the model.

Another future approach would be to add textures to the objects in the scene. By doing that, the output of the reflection model could potentially be made to look more realistic. For example, it could have been interesting to make the objects in the scene have a realistic surface of silver or any other material that might give a nice specular highlight. As for now, only the surface constants of silver is used but the material is not represented in the scene.

It would also have been fun to render other objects than the rectangular shapes that are now part of the Cornell box used in the scene. Spheres seem to be a popular part of implementations of the box, as well as adding other shapes such as teapots, animals and fun textures. A study on the perception of how different materials are illuminated and whether they are perceived as realistic or not, could be a part of a bigger expansion on the topic.

Critical evaluation

The process of this project was pretty straightforward and the implementation was done without bigger problems. Having a blog to document the process and progress was a good way of collecting findings for the report, as well as keeping up with what is next to do.

Since there are a lot of resources on the Phong reflection model and the implementation was simple, I was not expecting the project to be of a higher grade since a lot of previous code is reused and extended. However, it could be interesting to work with a more novel topic and challenge my mathematical and coding abilities. This would have been possible with more time. It could be an extension.

In retrospect, I would have liked to further read up on other reflection models and do a bigger test of how the Phong reflection model holds up to other similar models. However, that would have been a very big scope of a project and would not have been done in time for the deadline.

References

1. Shirley, P. and Marschner, S. *Fundamentals of Computer Graphics, Third Edition*. A K Peters, Natick, Massachusetts, 2009.
2. Angel, E. and Shreiner, D. *Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL (6th Edition)*. Addison Wesley, 2011.
3. Duroni, F. Phong reflection model. *Medium*, 2017. https://chicio.medium.com/phong-reflection-model-764f8065f19a.
4. What is BRDF model? – Gzipwtf.com. *Gzipwtf.com*, 2020. https://gzipwtf.com/what-is-brdf-model/.
5. Bidirectional reflectance distribution function - Wikipedia. *En.wikipedia.org*, 2021. https://en.wikipedia.org/wiki/Bidirectional_reflectance_distribution_function.
6. The Phong Model, Introduction to the Concepts of Shader, Reflection Models and BRDF. *Scratchapixel*, 2022. https://www.scratchapixel.com/lessons/3d-basic-rendering/phong-shader-BRDF.
7. GitHub - lemonad/DH2323-Skeleton: SDL2 skeleton for lab assignments 1–3 of the KTH course DH2323, Computer Graphics and Interaction. *GitHub*, 2016. https://github.com/lemonad/DH2323-Skeleton.
8. OpenGL/VRML Materials. *Devernay.free.fr*, 2022. http://devernay.free.fr/cours/opengl/materials.html.
9. Greenberg, D. Rendering Light Reflection Models Direct Illumination. *Graphics.cornell.edu*, 2020. http://www.graphics.cornell.edu/academic/art2907/secure/14_LECTURE_Light_Reflection_Models.pdf.