TU BRAUNSCHWEIG
Prof. Dr.-Ing. Marcus Magnor
Institut für Computergraphik
Contact: cgg@cg.cs.tu-bs.de

November 24, 2023

Computer Graphics WS 23/24
Assignment 4

Throughout the course you will implement your own minimal raytracer. In each exercise you will extend your raytracer a little further. To make the task easier, you are provided with a basic raytracing framework so that you just have to **fill in** the missing core parts.

Please use different colors in your drawings and also make sure that formulas are recognizable in your source code. Be prepared to present the completed assignments on **Friday, 9:45** in your given time slot. To keep presentation time short, make sure that the last commit contains the original scene file which generates the results shown below.

4.1 Lambert Shader (10 Points)

Create a new `LambertShader` using what you have learned in the lecture. Take a look at `shader/lambertshader.h` and `shader/lambertshader.cpp` and implement the missing part.

4.2 Phong Shader (20 Points)

Implement a Phong shader in `shader/phongshader.cpp`. Use the definition of the original model, not its physically plausible extension from the lecture.

4.3 Cook-Torrance Shader (30 Points)

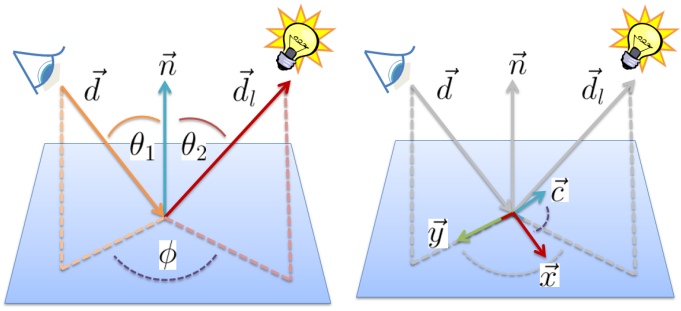
Implement the Cook-Torrance shader using what you have learned in the lecture. Look at `shader/cooktorrance.cpp` and implement the missing parts.
Use Schlick's approximation for the Fresnel term and Beckmann's distribution as the distribution function.

4.4 BRDF Shader (30 Points)

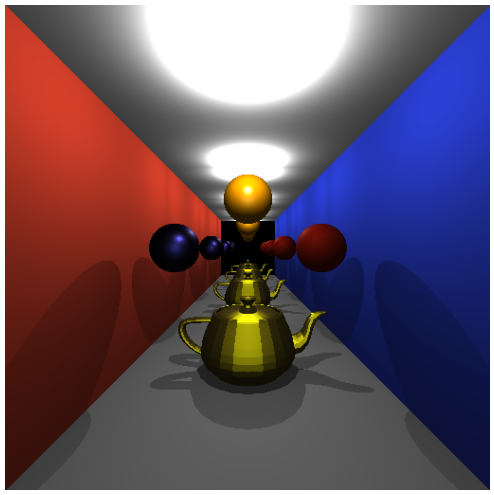
In the lecture you have learned about the BRDF and its functionality. It specifies the amount of reradiated light for an incoming light direction and and outgoing view direction and it is usually implemented as a lookup table or a mathematical function. You have also learned about isotropic BRDFs, for which the surface is invariant to rotation around its surface normal. In this case you only consider the difference between the azimuthal angles of the two vectors.
Given a ray $r(t) = \vec{o} + t \cdot \vec{d}$ which hits a reflective surface at $t = t_{hit}$, the surface has the normal $\vec{n}$ at the hit point. A secondary ray to a hypothetical light source has direction $\vec{d_l}$. Assume that all vectors are normalized. Draw an illustration based on the figure below and derive the angles θ_1, θ_2, ϕ which are necessary for a BRDF lookup.

Have a look at `BrdfShader::shade(Scene const & scene, Ray & ray)` and fill in the missing parts, use your illustration for the lookup. A look at `common/BRDFReader` might help. It specifies a loading routine and an interface that provides intensity values for given input and output angles via `BRDFRead::lookupBrdfValues()`.

Hint: Its is less obvious how to derive ϕ from the three vectors. Procceed as follows: Derive a local coordinate system for the inifinitesimal patch using cross products. Project $\vec{d}$ and $\vec{d}_l$ into that system. Assume $\phi_1 = 0°$ and $\phi_2 = \phi$. Draw a 2D illustration of the vectors in the plane, determine a right-angled triangle and use atan2 (opposite over adjacent) instead of acos (adjacent over hypothenuse) to make the calculation numerically more stable.



Now go to `https://cdfg.csail.mit.edu/wojciech/brdfdatabase` where the MERL lab has captured BRDF files from physical objects, free for educational, research and non-profit purposes.
Browse to the folder brdfs and download some BRDF files. Assign your scene objects with those to create an interesting appearance. To test you implementation use blue-metallic-paint and dark-red-paint (already provided by us in the *data.zip*). If you implemented everything correctly, bulding `tracey_ex4` should result in this image:



4.5 Smooth Triangles (10 Points)

You may have noticed that each vertex in `Scene::addObj` gets assigned a normal vector, if it is present in the `.obj`-file. This normal can be used for *smooth shading*. This removes the faceted look from our models.
Instead of returning the plane normal of a triangle return the interpolated normal vector of the intersection point in `Triangle::intersect`. Default back to the usual behavior, if one of the normal vectors of the triangle is not set (i.e is the zero vector). Use the barycentric coordinates u and v for interpolation and remember that an interpolated normal is not necessarily normalized.
If everything is implemented correctly, your result should look like this: