The exercises will take place in room G40 in Mühlenpfordtstrasse 23. Your y-account is sufficient to login and access all tools. Ctrl+Alt+T gives you a terminal and g++ is your GNU C++ compiler.

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.

Each week you must complete the assignments and hand in your commented source code for the practical tasks, as well as your solutions to the theoretical tasks (with drawings/formulas). Please use different colors in your drawings and also make sure that formulas are recognizable in your source code. Your group must present the completed assignments on each Friday, 9:45.

4.1 Phong Shader (20 Points)

You already learned about computing the diffuse color for Lambertian surfaces that are indirectly illuminated. More advanced, the phong shading model uses light sources to increase the rendering realism and give objects a plastic like appearance. Take a look at shader/phongshader.h and shader/phongshader.cpp and implement the phong illumination model. The value \( L_r \) returned by the PhongShader should be calculated according to

\[
L_r = k_a c_a L_a + k_d c_d \sum_{l=0}^{n-1} L_l (\vec{d}_l \cdot \vec{n}) + k_s c_s \sum_{l=0}^{n-1} L_l (\vec{d}_l \cdot \vec{d}_m)^{k_e}
\]

- **The value** \( L_r \) **returned by PhongShader::shadeRay()** should be calculated according to

\[
L_r = k_a c_a L_a + k_d c_d \sum_{l=0}^{n-1} L_l (\vec{d}_l \cdot \vec{n}) + k_s c_s \sum_{l=0}^{n-1} L_l (\vec{d}_l \cdot \vec{d}_m)^{k_e}
\]

- **\( c_a \): Ambient color**
- **\( c_d \): Diffuse color**
- **\( c_s \): Specular color** (use \( c_s = (1, 1, 1) \) for white high lights)
- **\( k_a \): Ambient coefficient**
- **\( k_d \): Diffuse coefficient**
- **\( k_s \): Specular coefficient**
- **\( k_e \): Exponent (shine parameter)**
- **\( L_a \): Ambient radiance**
- **\( L_l \): Radiance arriving from light source \( l \)**
- **\( \vec{n} \): Shading normal**
- **\( \vec{d}_l \): Direction to light source \( l \)**
- **\( \vec{d}_m \): Reflected incident ray direction (must point away from the surface)**
- **\( n \): Number of lights sources**

Remember, that we already have ambient light sources implemented elsewhere. So you must not add any ambient terms from this formula.
4.2 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 an 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 \( \theta_1, \theta_2, \phi \) which are necessary for a BRDF lookup.

Have a look at `BrdfFShader::shade(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: It is less obvious how to derive \( \phi \) from the three vectors. Proceed as follows: Derive a local coordinate system for the infinitesimal patch using cross products. Project \( \vec{d} \) and \( \vec{d}_l \) into that system. Assume \( \phi_1 = 0^\circ \) 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 [http://people.csail.mit.edu/wojciech/BRDFDatabase/brdfs/](http://people.csail.mit.edu/wojciech/BRDFDatabase/brdfs/) 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 your implementation use green-acrylic and gold-paint. If you implemented everything correctly, your image should look like this:
4.3 Post-Effects (Desaturation) (20 Points)

Implement a new renderer in `renderer/desaturationrenderer.cpp` that renders a desaturated image. Desaturation is simply a linear interpolation between the raytraced color and the grayscale value of that color using a given intensity as interpolation factor. The grayscale value is simply the average over all color channels.

4.4 Post-Effects (Haze) (25 Points)

Implement a new renderer in `renderer/hazerenderer.cpp` that simulates atmospheric haze. The farther away a point is in the scene the more obscured it will be. The renderer has a haze color $c$ and a falloff $f$. First calculate the haze density using the ray length $t$:

$$
 d = e^{-tf}, \quad d \in [0, 1]
$$

where $f$ is the given falloff value. Your final color will then be a linear interpolation between the ray traced color and your hazeColor using $d$ as interpolation factor.

4.5 Post-Effects (Depth Map) (25 Points)

Implement a new renderer in `renderer/depthrenderer.cpp` that renders a normalized depth map of the scene. A depth map is a grayscale image in which the closest visible point is represented by the color white and the furthest visible point is black. Take into account, that the background is infinitely far away and must not be considered for the normalization (the background is always black).

Hint: You will have to do two passes. First, store the unclamped depth values in the image and then normalize these values in a second pass.

If you implemented your post-effects correctly, you should end up with these three images: