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. This project offers a `CMake` configuration to generate and executable.

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. You may use your own computer to solve the exercises, but your final program must run on the machines in the CIP pool.

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.

To keep presentation time short, keep (a copy of) the original scene to generate the result shown below.

### 4.1 Lambert Shader (25 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 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 `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:** 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 hypotenuse) 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 blue-metallic-paint, dark-red-paint and gold-paint. If you implemented everything (including the Lambert shader) correctly, building `tracey_ex4_1` should result in this image:

![Image](image.png)

### 4.3 Discretization Shader (25 Points)

The discretization shader can alter the look of every other shader to achieve a toon-like look, although it is not what can be found as "toon-shader".

This shader works by modifying the color value of its sub-shader. Implement the missing part in `discretizeshader.cpp`. Use the method described here [https://de.wikipedia.org/wiki/HSV-Farbraum](https://de.wikipedia.org/wiki/HSV-Farbraum) to convert RGB values to HSV. Then map the "V" part of the HSV color to nearest value of the lookup table and return the RGB value of this modified color.

Build `tracey_ex4_2`. If you have implemented everything correctly, your image should look like this:
4.4 Post-Effects Haze and Desaturation (20 Points)

Implement a new renderer in renderer/hazerenderer.cpp that simulates atmospheric haze. The further 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. Build tracey_ex4_3. Your image should look like the left image.

Implement a second 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. Build tracey_ex4_4. Your image should look like the right image.