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. 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.

At the start, your raytracer consists of only three simple parts:

- **Primary Ray Generation** for generating the rays to be cast from a virtual camera into the scene.
- **Ray Tracing** for finding the (closest) intersection of a ray with the scene to be rendered.
- **Shading** for calculating the color of the ray.

To begin, download the raytracing framework from our course web site:

http://www.cg.cs.tu-bs.de/teaching/lectures/ws1617/CGI/exercise/tracey.zip

Have a look at the ray tracing framework and its C++ classes:

- The framework is structured into the components Camera, Light, Primitive, Renderer, Scene, and Shader. Each has a base class of the same name, as well as multiple child classes that we will be developing over the course of this semester.
- A Ray is defined by its origin (Vector3d), direction (Vector3d), and length (float).
- The Scene holds all the geometry in the form of Primitives. Each type of Primitive has a virtual method Primitive::intersect(Ray * ray), which has to be implemented by you.
- The abstract base class Camera handles camera parameters. For each derived class, e.g. a perspective or orthogonal camera, the pure virtual method Camera::castRay(float x, float y) has to be implemented. Here, x and y specify the relative position in the camera frustum.
- In the class SimpleRenderer you will have to implement the function SimpleRenderer::renderImage(Scene const& scene, Camera const& camera, int width, in height). This function calculates the images aspect ratio and casts a ray for each pixel.

Before implementing anything read through the presented classes and main.cpp.

We will use the Qt library during this course. While the current framework does not use it a lot yet. It
will be used more throughout the semester. The framework comes with a tracey.pro project file for Qt.
While it is not mandatory, I suggest that you use the Qt Creator for the exercises.
1.1 Primary Ray-Generation for a Perspective Camera Model (30 Points)

Have a look at `camera/perspectivecamera.cpp` and fill out the missing section. A Perspective Camera Model can be defined by the following parameters:

- Camera origin (center of projection) `position`
- Viewing direction `forwardDirection`
- (Vertical) full opening angle `angle` of the viewing frustum (in degrees) `fovAngle`
- Up-vector `upDirection`

Given the above camera description, derive the ray direction from the camera to a relative screen coordinate $x, y \in [-1, +1]$. The projection plane is perpendicular to the camera `forwardDirection`. You will have to incorporate the focus (distance from camera position to image plane along the `forwardDirection`).

Please incorporate the aspect ratio as part of the `SimpleRenderer` class in `renderer/simplerenderer.cpp`. You can achieve different aspect ratios by not using the entire ranges for $x$ and $y$ in the rendering function. For example a 16:9 image would use $x \in [-1, +1]$ and $y \in [-\frac{9}{16}, +\frac{9}{16}]$.

1.2 Ray-Surface Intersection (60 Points)

Given a ray $r(t) = o + td$ with origin $o = (o_x, o_y, o_z)$ and direction $d = (d_x, d_y, d_z)$, derive the equations to compare the parameter $t$ for the intersection point(s) of the ray and the following implicitly represented surfaces:

a) An infinite plane $(p - a) \cdot n = 0$ through point $a = (a_x, a_y, a_z)$ with surface normal $n = (n_x, n_y, n_z)$, where any point $p = (x, y, z)$ that satisfies the equation lies on the surface. Use this to fill the missing section in `primitive/infiniteplane.cpp`.

b) Consider a triangle with vertices $V_0, V_1$ and $V_2$. Fill the missing section in `primitive/triangle.cpp` using what you have learned in the lecture.

c) A sphere $(p_x - C_x)^2 + (p_y - C_y)^2 + (p_z - C_z)^2 = r^2 \iff (p - C)^2 = R^2$ with center $C = (C_x, C_y, C_z)$, radius $r \in \mathbb{R}$ and point $p = (p_x, p_y, p_z) \in \mathbb{R}^3$ on its surface. Compute the values of $t$ for which the ray intersects the sphere. Use this to fill the missing section in `primitive/sphere.cpp`.

1.3 Ray Tracing (10 Points)

Have a look at `renderer/simplerenderer.cpp` and fill out the missing section. Generate a ray and intersect all objects of your scene with it; assign unique colors of your choice to the objects. The program should generate an image which should look like the one on the left. By moving the camera you can create more interesting perspectives. Try to create your own scenes, by manipulating the `main.cpp`. 

![Image](image_url)
1.4 Useful Stuff

Have a look at the following links. They may help you solving the single Tasks.

- Qt Creator https://www.qt.io/ide/  
- http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtrace0.htm gives you useful information on ray-object intersection.
- Realistic Ray Tracing, Peter Shirly.