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

October 23, 2019

Computer Graphics WS 19/20
Assignment 1

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. Be prepared to present the completed assignments on **Friday, 9:45**.
To keep presentation time short, make sure that the last commit contains the original scene file which generates the results shown below.

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.

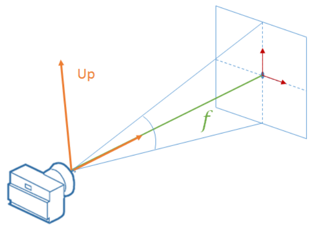
To begin, create an account on our git `git.cg.cs.tu-bs.de` and tell me your account name so I can give you access rights Have a look at the ray tracing framework in repository `WS1920` and its C++ classes:

- 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 `ex1.cpp`.

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**).

You will incorporate the *aspect ratio* as part of the `SimpleRenderer` class in exercise 1.3. 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 (50 Points)

Given a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ with origin $\mathbf{o} = (o_x, o_y, o_z)$ and direction $\mathbf{d} = (d_x, d_y, d_z)$, 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 $(\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$ through point $\mathbf{a} = (a_x, a_y, a_z)$ with surface normal $\mathbf{n} = (n_x, n_y, n_z)$, where any point $\mathbf{p} = (x, y, z)$ 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 $(p_x - C_x)^2 + (p_y - C_y)^2 + (p_z - C_z)^2 = r^2 \Leftrightarrow (p - C)^2 = r^2$ with center $C = (C_x, C_y, C_z)$, radius $r \in \mathcal{R}$ and point $p = (p_x, p_y, p_z) \in \mathcal{R}^3$ 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 (20 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 `ex1.cpp`. Compute the time for image generation using `std::chrono::steady_clock` and print it afterwards. Modify the `common/Ray.h` and `renderer/simplerenderer.cpp` to implement a simple ray counter, which can be used to calculate the number of processed rays per second. Print this number as well (*Hint: A* `static` *member variable for the* `Ray`*-struct is usually how such counting is handled*).

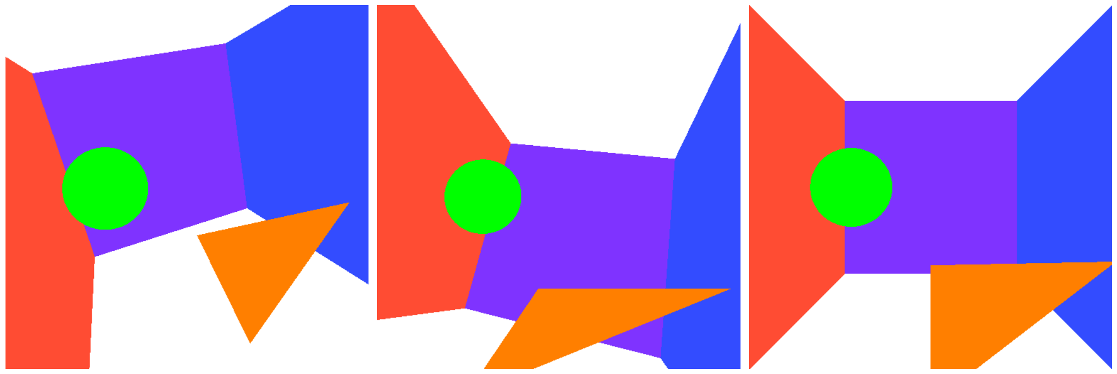


Figure 1: Output of the minimal ray tracing framework. The primitives (spheres, triangles, plane) are assigned a unique color. For each pixel the color of the closest object is assigned.

1.4 ***Useful Stuff***

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

- CMake `https://cmake.org/`
- The TU offers free student licenses for Visual Studio here `https://www.tu-braunschweig.de/it/downloads/software/rahmenvertraege/msdn-aa`
- `http://www.realtimerendering.com/intersections.html` gives you useful information on intersections.
- Realistic Ray Tracing, *Peter Shirly*.
- 3D Modelling: Blender `https://www.blender.org/` or 3ds Max Free Student Version `http://www.autodesk.com/education/free-software/3ds-max`