Prof. G. Zachmann
H. Meißenhelter
T. Hudcovic

Summer Semester 2025

Assignment on Advanced Computer Graphics - Sheet 1

Due Date 08.05.2025

Exercise 1 (Raytracing Basics)

n this exercise, you have to complete the *RaytracingFramework*, which you can download from our CGVR website. To set up the development environment so that you can work on the project, follow the instructions in the slides of the first tutorial, also available on the CGVR website.
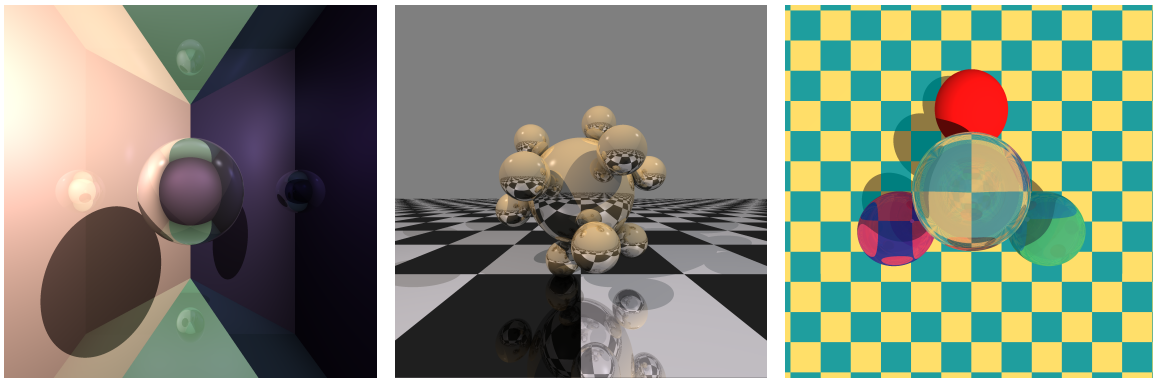

Figure 1: Raytraced images of the *A Sphere and Planes*, the *Metal Spheres* and the *Glass Spheres* scene when implemented correctly.

Overview of the framework

Let's start with the important math class **Vec4f**, that is used in many places in the framework. The class stores four float values x, y, z and w and implements basic vector operations like addition, subtraction, scaling, but also the dot product and cross product. This class is used to store points, vectors as well as colors as a variable:

- **Point** or **Vector**: For points and vectors, the x, y and z values store the coordinates. In case of a point, the w value is `w = 1.0` and for vectors, the w value is `w = 0.0`. Also note this when creating a vector or point and set the w value accordingly. Please refer to the documentation in `Vec4f.h` (e.g. the constructor with three parameters sets the fourth w-value to 1.0 and thus creates a point).
- **Colors**: For colors, the x, y, and z values store the red (x), green (y), and blue (z) values.

The core class is called **Raytracer**, which contains the basic functionality for raytracing. The important functions are:

- `render()`: generates a ray for each pixel of the image. This ray is then traced using the `traceRay` function.
- `traceRay()`: traces the ray through the scene and recursively calls `traceRay` for secondary rays.
- `shade()`: This function computes the local (Phong) shading for a point in the scene. Moreover, it should test whether or not the point is in the shadow.

The **Camera** class implements a simple pinhole camera. The function `generateRay()` generates a ray from the eye point through the pixel (`x,y`) in the image plane.

The **Ray** class represents a ray that is defined by an origin point and a direction vector. Moreover, the class includes functions to compute a reflected and refracted ray.

The **Surface** class is the base class for every geometric object. All derived classes implement their own `intersect()` function that computes the intersection between the objects and a ray. The derived classes `Plane`, `Sphere` and `Checkerboard` are fully implemented.

The **Shader** class implements a simple Phong Shader that is fully implemented and can be used to calculate the color of an intersection point by using the `shade` function.

The **PointLight** class defines a simple point light and is fully implemented.

Your tasks are:

a) Implement the function `Plane::intersect()` in the file `Plane.cpp`. For this purpose check if there is an intersection between the given ray and the plane and return it.

b) Implement the function `Raytracer::shade()` in the file `Raytracer.cpp`. In this function, you should apply phong shading for every light source if the point lies not in the shadow (s. "Raytracing" lecture, "A Local Lighting Model").

c) In order to realize a real raytracer, we have to trace also reflected rays. To do that, you should implement the function `Ray::reflectedRay()` in `Ray.cpp` (s. "Raytracing" lecture, "Generation of Secondary Rays" & "Specular Reflection").
Moreover, you have to use this function in `Raytracer::traceRay()` to recursively compute the reflected color (that has to be added to the current color value).

d) Finally, we also want to include refractions into our raytracer. Therefore, you have to implement the function `Ray::refractedRay()` in `Ray.cpp`. It should, depending on the refraction parameter of the material, return the refracted ray (s. "Raytracing" lecture, "The Refracted Ray").
As in the previous part, you also have to trace the refracted ray recursively through the scene in `Raytracer::traceRay()` to compute the appropriate amount of color.

Note: It's normal that the *Metaball* scenes don't show anything.