UNIVERSITÄT DES SAARLANDES
Prof. Dr.-Ing. Philipp Slusallek
Computer Graphics Group
Arsène Pérard-Gayot (perard@cg.uni-saarland.de)

24. October 2019

Introduction to Computer Graphics
Assignment 1

Submission deadline for the exercises: 31. October 2019

The paper copies for the theoretical parts of the assignments will be collected at the beginning of the lecture on the due date. The programming parts must instead be marked as release before the beginning of the lecture on the due date.
The code submitted for the programming part of the assignments is required to reproduce the provided reference images. The submission ought to tag the respective commit in your group's git repository as well as attach the mandatory generated images to the release. The submission should also contain a creative image show-casing all extra-credit features that have been implemented.
The projects are expected to compile and work out of the box. A successful build by Drone CI is a good indicator. If it fails to do so on our Windows or Linux computers, we will try to do so on the CIP-pool students' lab as a "fallback".
To pass the course you need for every assignment at least 50% of the points.

1.1 Vector Properties (1 + 2 + 2 + 1 + 5 Points)

Let u, v, w be vectors in $\mathbb{R}^3$. Prove or disprove the following statements:

- $(u \cdot u)$ is the square of the length of u.
- $(u \cdot v)^2 + (u \times v)^2 = |u|^2|v|^2$.
- $|u + v|^2 = |u|^2 + |v|^2 + 2(u \cdot v)$.
- $(u \cdot v)^2 = u^2 \cdot v^2$, where t^2, for vector t, is defined as $t \cdot t$.
- $u \times (v \times w) = (u \cdot w)v - (u \cdot v)w$.
- $(u \cdot v) \cdot w = u \cdot (v \cdot w)$.

1.2 Intrinsic Camera Parameters (5 + 3 + 5 + 5 Points)

You are given a full-frame camera, whose sensor size is 36mm in width x 24mm in height.

a) Compute the focal length allowing a rectangular 1.8m wide and 1.2m high subject at a distance of 4m from the pinhole to fill half of the frame area.

b) To what value should you set the focal length to have a 60-degree horizontal field of view?

c) At what distance should the camera stand from the same subject for it to still fill half of the frame?

d) Of what size should the sensor of an idealized orthographic camera be so that the subject appears to fill the frame by the same ratio as with a perspective camera with a 60-degree horizontal field of view at a distance of 4m?

1.3 Introduction to the Framework

You should have received credentials and the URL of your group's git repository. Please clone and familiarize yourself with the provided framework. We are providing merely an interface for the basic framework. Your job is to provide an implementation as well as to extend it to fit your needs.
New practical assignments will add new files to the framework, but the existing ones (with an exception of `main/main.cpp`) are not going to be changed. You are allowed to modify the framework in any way you see fit, but the set interface must remain the same. In particular:

- You may add and modify private members of the classes.
- You may add new public members to the classes, but the existing ones should not be removed.
- You may create new classes and add new files into the project.

The project is split into 3 directories:

- `core` — containing basic utility functions and classes. These functions are used heavily in the raytracer, but are not specific to ray tracing in general.
- `rt` — containing code specific to ray tracing.
- `main` — contains the starting point of the program, as well as code used for building up the scene and invoking the raytracer. This is where assignment testing code will appear, but you can put your own interesting scenes as well.

We are providing a `CMakeLists.txt` to allow you to use your favorite build system. The respective `assignment-xx.cmake` lists all files that were added with an assignment. You may add your additional files here too, but keep the original `CMakeLists.txt` unchanged. On Linux libpng-dev is required for the build and should be automatically detected.
By invoking:

- `mkdir build_debug && cd build_debug` — you create and enter a build directory for CMake.
- `cmake -DCMAKE_BUILD_TYPE=Debug ..` — you configure the project in debug mode.
- `cmake --build .` — you compile the project, producing executable `cgray`.

For release mode use `cmake -DCMAKE_BUILD_TYPE=Release ..` in a different `build_release` directory respectively.
On Windows using Visual Studio you can generate a single multi-configuration build directory.

Already provided functionality
The bare-bone framework already provides for you:

- Assertions (in `core/assert.h`) which can be used to debug the code.
  - `assert(cond)` macro function terminates the program and prints an error message if the boolean condition `cond` is not met. The error message indicates the file name and the line number, however the `assert` macro can be followed by any stream-like expression that will be printed in addition to the standard error message. For example:
    ```
    assert(x < w) << "Coordinate " << x << " is beyond the bound " << w;
    ```
    The assertion is ignored when compiling in release mode.
  - `UNREACHABLE` macro statement indicates a code that should never be reached.
  - `NOT_IMPLEMENTED` macro statement indicates that given functionality is not implemented.
- Simple 2-dimentional raster image library (in `core/image.h`). Apart from standard per-pixel operations, it allows you to store or load the image from a `.png` file format.
- Julia set generator (in `core/julia.h`). For the input complex number `v` (represented as a `Point` object), the function recursively computes $v' = v^2 + c$ and returns the number of iterations for the v to reach near-infinity. The return value is clamped to 512.

- Some low-level macros are provided in `core/macros.h` but we do not use those often in the framework.
  - `ALIGN(n)` in front of a class definition overrides the type memory alignment to specific value. It is used for `Point`, `Vector`, `RGBColor` and `Float4` classes to permit implicit SIMD optimizations in release mode.
  - `THREADLOCAL` in front of global variable declaration create a separate copy of the said variable for each thread in a multi-threaded code. In a single-threaded version it has no observable effect.
- `scalar.h` includes the math library, as well as provides some helper functions on its own for scalars.

1.4 Basic Vectors (20 + 10 Points)

You are given the first version of the framework. Your first goal is to set up the basic math functionality. To that end you will need to provide the implementation to the following classes:

- `Vector` (in `core/vector.h`) — the math operations on a 3-dimensional vector, as well as operators between vectors and points.
  - `Vector::length()` should return the length of the vector in Euclidean space.
  - `Vector::lensqr()` should return the square of the length of the vector, which is easier to compute.
  - `min` and `max` on a vector should apply the minimum/maximum operation component-wise.
  - `Float4` is not provided at the moment. You can skip the implementation of the constructor depending on it.
- `Point` (in `core/point.h`) — the math operations on a 3-dimensional point.
- `RGBColor` (in `core/color.h`) — the math operations on an RGB color value.
  - Binary operators `+ - *` should work component-wise.
  - `RGBColor::clamp()` — clamps the component values between 0 and 1.
  - You can skip the implementation of `RGBColor::gamma()` and `RGBColor::luminance()` at this time.

In addition, you should implement the `Renderer` class which is the main entry point for the future ray-tracing system. In the current version however, the `Renderer` should ignore its constructor arguments `Camera` and `Integrator` which are left undefined. Instead, for the input empty image the `Renderer::test_render1()` should iterate over all pixels. The function should for each pixel set its color to the value returned from `a1computeColor` which is provided to you with the assignment.

1.5 Ray (1 Points)

Implement the concept of a ray. Ray is defined by the origin and direction. `Ray::getPoint` should return a point on the ray at a specified distance. We are however not testing this function in the current assignment.

1.6 Camera (10 + 10 + 5 Points)

We are introducing a concept of `Camera` into the framework. A camera has one functionality: to create *primary rays* for given uniform coordinate, which is in range $[-1, 1]$ in x and y direction. Your task is to implement `PerspectiveCamera` and `OrthographicCamera` as two possible realizations of this concept.
You should also update your `Renderer` to take the provided `Camera` object into account. It is the job of the `Renderer` to convert image pixel coordinates into the uniform camera coordinates. Also note, that the image y axis is pointing downwards, while the camera is expecting y axis pointing upwards.
Your rendering loop, defined in `Renderer::test_render2`, should generate primary rays, and for each - invoke `a2computeColor`, the testing function provided to you with the assignment.

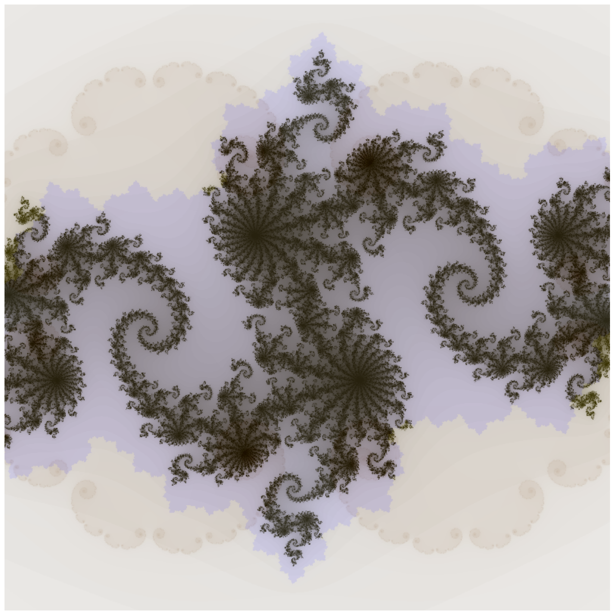

Figure 1: The image produced by `a1computeColor`

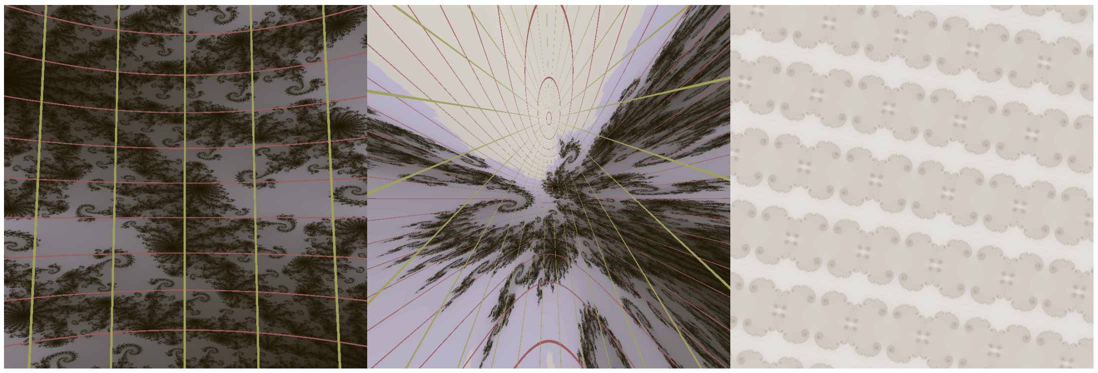

Figure 2: Final images produced by cameras, using perspective camera (first two) and orthographic camera (last image).