Submission deadline for the exercises: 5. November 2018

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 sent by email to lemme@cg.uni-saarland.de 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 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 Orthogonal Vectors (3 + 3 Points)

Given two vectors \( u, v \) of length 1, provide two versions of a formula computing a vector \( t \) that is perpendicular to \( u \) and lying on the \( uv \) plane. Both versions can contain vector addition and subtraction, and...

a) the first version of the formula should consist of only cross products.

b) the second version of the formula should consist of only dot products.

Provide geometric interpretation of these formulas.

1.2 Primary Ray-Generation for a Perspective Camera Model (10 Points)

A Perspective Camera Model can be defined by the following parameters:

- Camera origin: \( \text{pos} \)
- Viewing direction: \( \text{dir} \) (normalized). The image plane is centered around \( \text{dir} \) and perpendicular to it.
- (Full) vertical opening angle of the viewing frustum (in degrees): \( \theta \)
- Up-vector: \( \text{up} \)
- Image resolution: \( \text{resX} \times \text{resY} \). Pixels are squares.

Given the above camera description, derive the direction of a ray (\( \text{ray.d} \)) passing through the center of a given pixel with coordinates \( x, y \). Using your formula, formulate the primary ray for the center of pixel (128, 5), assuming the image resolution (1024, 768)

Note, the \( \text{up} \) vector is not necessarily perpendicular to \( \text{dir} \).
1.3 Camera Movement (9 + 9 Points)

In virtual worlds, as well as first-person-perspective games the camera is tied to an object controlled by the user. How are the values of \( \text{pos} \), \( \text{dir} \) and \( \text{up} \) changing when the user decides to:

- Rotate left (yaw) by an angle \( \alpha \).
- Look up (pitch) by an angle \( \beta \).
- Move forward by a distance \( d \).

There are actually two answers for this question.

- One is typical for air or space movement, where there is no fixed definition of \( \text{down} \).
- The other is typical for an on-ground movement, when the player-controlled object is moving on a flat surface and gravity direction is well defined. In this case, the angle at which you can look up/down is limited, but you can assume that the pitch change \( \beta \) is not exceeding the constraints.

Please provide your answer for both cases.

1.4 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 \text{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:

- \text{core} — containing basic utility functions and classes. These functions are used heavily in the raytracer, but are not specific to ray tracing in general.
- \text{rt} — containing code specific to ray tracing.
- \text{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 \text{CMakeLists.txt} to allow you to use your favorite build system. The respective \text{assignment-xx.cmake} lists all files that were added with an assignment. You may add your additional files here too, but keep the original \text{CMakeLists.txt} unchanged. On Linux \text{libpng-dev} is required for the build and should be automatically detected.

By invoking:

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

For release mode use \text{cmake -DCMAKE_BUILD_TYPE=Release ..} in a different \text{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:
    ```cpp
    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-dimensional 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.

1.5 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.

1.6 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.

1.7 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 testing function provided to you with the assignment.
Figure 2: Final images produced by cameras, using perspective camera (first two) and orthographic camera (last image).