Computer Graphics Practical 2

Michaelmas Term 2020

Overview

The aim of this practical is give you an understanding of the entire ray-casting rendering process: taking definitions of simple three-dimensional objects and producing an output image. Modern rendering systems are very complex and are implemented mostly behind-the-scenes, making it difficult to understand and appreciate the processes they perform. Simple CPU-based renderers are more understandable, but will quickly run into performance problems with complex scenes or advanced techniques.

The objective of the practical is to develop your own renderer, written in Python. Your renderer should take, as input, the specifications of some simple geometric objects and light sources and provide, as output, a rendered image displaying those objects.

Sphere((-0.5,0.8,4), 0.2),
Sphere((0.5,0.5,3.5), 0.5),
Triangle((-10,1,-10), (10,1,-10), (10,1,10)),
Triangle((-10,1,-10), (-10,1,10), (10,1,10)),
Triangle((-10,1,10), (10,1,10), (10,-10,10)),
Triangle((-10,1,10), (-10,-10,10), (10,-10,10)),
Light((0,-1,5), (1,0,0)),
Light((-1,-1,5), (0,1,0)),
Light((0.5,-1,3.5), (0,0,1))

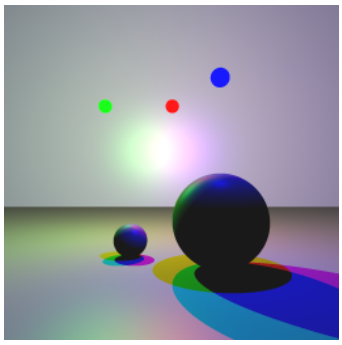

Figure 1: An example input and output

To make this task manageable, you will be provided with a skeleton. Additionally, your renderer has no (strict) time constraints, and does not need to be implemented on the GPU. You are being asked to use Python for this practical to help you focus on the fundamental graphics techniques, instead of dealing with the practicalities of using a specific rendering library or framework.

Your renderer should not depend on external libraries, but instead implement the mathematics you will learn during the course.

Tasks

Your renderer should, at least, perform the following:

- Render arbitrary spheres accurately
- Render arbitrary triangles accurately
- Perform some form of lighting based on given light sources.

Successfully implementing these into your renderer is sufficient to pass this practical. Your renderer could, optionally, support some of the following:

- More complex lighting models, such as shadowing
- More complex geometric shapes
- Antialiasing (such as supersampling)
- Loaded objects, specified in a standard format such as `.obj`
- Texturing
- Orthographic views
- Other improvements that are agreed with the demonstrator

Assessment

This practical will be assessed as follows:

- For the S grade, your renderer should meet the requirements specified above, rendering spheres and triangles with some form of lighting.
- For the S+ grade, your renderer should meet the requirements above, together with at least one of the optional features.

Your renderer will not be assessed based on its performance, although you should consider the performance implications of your implementation choices.

Notes

While your renderer should successfully render the scene, there is no requirement for it to be quick. In fact, it will likely take a long time to render a high resolution image – you should think about why this is the case, and what would be necessary to speed it up.

You should not rely on external libraries to implement any of the 3D graphics for your renderer.

The skeleton is provided to help you get started, although you do not need to use it. If you would prefer to use a different language, agree it with the demonstrators before you begin. Note that, in any case, you should not rely on any external libraries for any of the rendering process.

The skeleton renderer uses a ray-casting method (rather than a rasterisation method), meaning it casts a ray from the camera for each pixel, and finds the first intersection with scene geometry. The colour of the pixel is determined by the lighting calculation at the intersection point.

The skeleton code contains a class called `Sphere`, which you should implement two methods of, and `PhongLight`, which you should implement one method of. You should also add a new subclass of `Renderable` to represent a triangle, and implement it.

When debugging your code, you may want to reduce the size of your output image to get a faster result.

To save some time, the `vec3` class (in `helpers.py`) is provided and implements typical vector operations, such as dot and cross products.