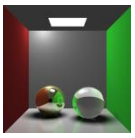


2021/2022, 4th period

INFOGR: Graphics

Practical 1: Ray Tracing

Author: Peter Vangorp, based on a previous version by Jacco Bikker

The assignment:

The purpose of this assignment is to create a Whitted-style ray tracer. The renderer should be able to render a scene consisting of diffuse and mirror spheres and planes, illuminated by point lights. For a full list of required functionality, see section "Minimum Requirements".

The following rules for submission apply:

- Your code has to compile and run on other machines than just your own. If this requirement isn't met, we may not be able to grade your work, in which case your grade will default to 0. Common reasons for this to fail are hardcoded paths to files on your machine.
- Please **clean** your solution before submitting (i.e. remove all the compiled files and intermediate output). This can easily be achieved by running `clean.bat` (included with the template). After this you can zip the solution directories and submit them on Blackboard. If your zip-file is multiple mega-bytes in size something went wrong (not cleaned properly).
- The debug feature mentioned in this document is mandatory. Since you have to program it anyway, consider doing so early on in the project, so it actually is useful for debugging. 😉 If you ask a question about a bug in your ray tracer, we will typically suggest to implement this feature first.
- We want to see a consistent and readable coding style: formatting; descriptive names for variables, methods, and classes; and comments. Most code editors have tools to help with formatting and indentation, and with renaming things ("refactoring") if necessary.

  "Programs are meant to be read by humans
  and only incidentally for computers to execute." – Donald E. Knuth

Grading:

If you implement the minimum requirements, and stick to the above rules, you score a 6. Implement additional features to obtain additional points (up to a 10).

Additional grading details:

- From the base grade of 6, we deduct points for a missing readme, a solution that was not cleaned, a solution that does not compile, or a solution that crashes (up to 1 point for each problem).
- Up to 1 point will be deducted for an inconsistent coding style.
- Up to 1 point will be deducted for code that is not properly commented.

Deliverables:

A ZIP-file containing:

1. **The contents of your (cleaned) solution directory**
2. **The readme.txt file**

The contents of the solution directory should contain:

(a) Your **solution file** (.sln)

(b) All your **source code**

(c) All your **project** and **content files**

The readme.txt file should contain:

(a) The names and student IDs of your team members.

[1 or 2 students, not more!]

(b) A statement about what minimum requirements and bonus assignments you have implemented (if any) and information that is needed to grade them, including detailed information on your implementation.

[We will not search for special features in your code. If we can't find and understand them easily, they may not be graded, so make sure your description and/or comments are clear.]

(c) A list of materials you used to implement the ray tracer. If you borrowed code or ideas from websites or books, make sure you provide a <u>full and accurate overview</u> of this.
Considering the large number of ray tracers available on the internet, we will carefully check for original work.

Please put the solution directories and the readme.txt file directly in the **root** of the zip file.

Mode of submission:

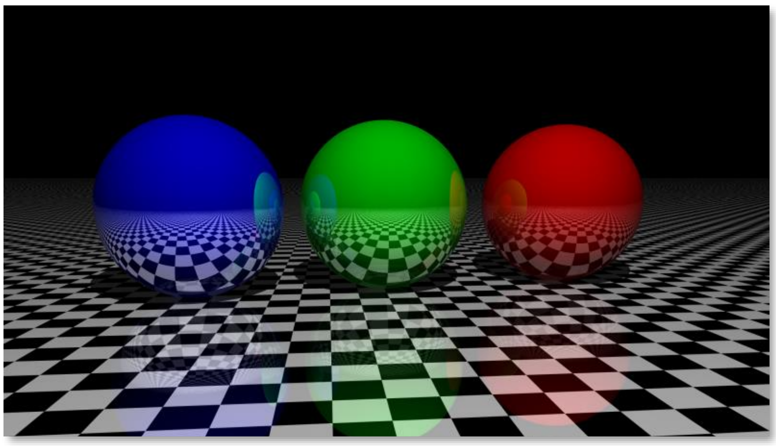
- Upload your zip file before the deadline via Blackboard. The Blackboard software allows you to upload without submitting: please do not forget to hit 'submit' once you are sure we should see the final result. Please do not forget the final submit!
- Note that we only grade the last submitted version of your assignment.

Deadline:

Tuesday, May 31, 2022, 17:00

If you miss this deadline, you may hand in your work up to 7 hours late, i.e., until midnight, but this decreases your grade by 0.5 points; or up to 24 hours late, i.e., until Wednesday June 1, 17:00, but this decreases your grade by 1.0 point.

If you miss these deadlines as well, your work will not be graded.



High-level Outline

For this assignment you will implement a Whitted-style ray tracer. This is a recursive rendering algorithm for determining light transport between one or more light sources and a camera, via scene surfaces, by tracing rays backwards into the scene, starting at the camera.

A Whitted-style ray tracer requires a number of basic ingredients:

- a camera, representing the position and direction of the observer in the virtual world;
- a screen plane floating in front of the camera, which will be used to fire rays at;
- a number of primitives that will be intersected by the rays;
- a number of light sources that provide the energy that will be transported back to the camera;
- a renderer that serves as the access point from the main application. The renderer ‘owns’ the camera and scene, generates the rays, intersects them with the scene, determines the nearest intersection, and plots pixels based on calculated light transport.

Optionally, you can define materials for the primitives. A material stores information about color or texture, reflectivity, and transmission (‘refractivity’).

The remainder of this document describes the implementation of such a ray tracer. You are free to ignore these steps and go straight to the required feature list. Note however that the **debug view** is a **mandatory** feature.

Architecture

To start this project, create classes for the fundamental elements of the ray tracer:

Camera, with data members **position**, **look-at direction**, and **up direction**. The camera also stores the **screen plane**, specified by its four corners, which are updated whenever camera position and/or direction is modified. Hardcoded coordinates and directions allow for an easy start. Use e.g. (0,0,0) as the camera origin, (0,0,1) as the look-at direction, and (0,1,0) as the up direction; this way the screen corners can also be hardcoded for the time being. Once the basic setup works, you must make this more flexible.

Primitive, which encapsulates the ray/primitive intersection functionality. Two classes can be derived from the base primitive: Sphere and Plane. A sphere is defined by a **position** and a **radius**; a plane is defined by a **normal** and a **distance** to the origin. Initially (until you implement materials) it may also be useful to add a color to the primitive class.

Important: colors should be stored as floating point RGB vectors. We will convert the final transported light quantities to integer color as a final step; keeping everything in floats is accurate, more natural, and easier.

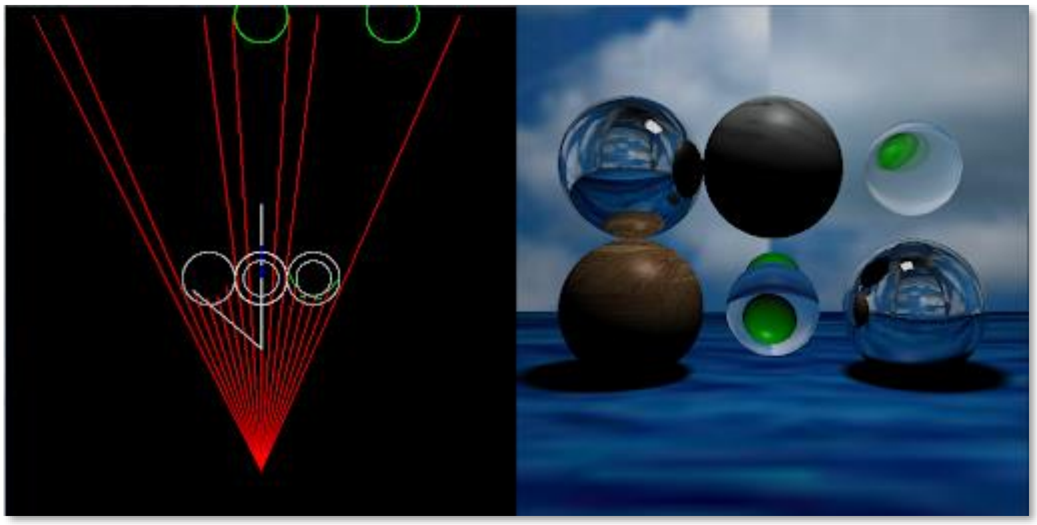
Light, which stores the **location** and **intensity** of a light source. For a Whitted-style ray tracer, this will be a point light. Intensity should be stored using float values for red, green and blue.

Scene, which stores a **list of primitives** and **light sources**. It implements a scene-level Intersect method, which loops over the primitives and returns the closest intersection.

Intersection, which stores the result of an intersection. Apart from the intersection **distance**, you will at least want to store the nearest **primitive**, but perhaps also the **normal** at the intersection point.

Raytracer, which owns the **scene**, **camera** and the display **surface**. The Raytracer implements a method Render, which uses the camera to loop over the pixels of the screen plane and to generate a ray for each pixel, which is then used to find the nearest intersection. The result is then visualized by plotting a pixel. For the middle row of pixels (typically line 256 for a 512x512 window), it generates debug output by visualizing every Nth ray (where N is e.g. 10).

Application, which calls the Render method of the Raytracer. The application is responsible for handling keyboard and/or mouse input.



Example of valid debug output. Ray tracer by Rens van Mierlo.

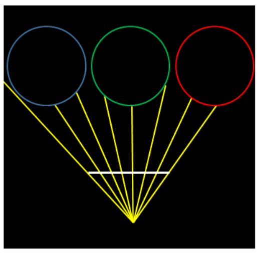
First Steps - Details

With the basic structure of the application in place, it is time to implement the functionality. It helps to get to something that produces sensible output as quickly as possible.

1. Prepare the scene.

A good scene to start with is a floor plane with three spheres on it. Keep everything within a 10x10x10 cube and position the spheres so that the default camera can easily 'see' them. Make sure the sphere centers are at y=0, which is convenient for the debug view.

2. Prepare the debug output.



Draw the scene to the debug output. Use a dot (or 2x2 pixels) to visualize the position of the camera. Use a line to visualize the screen plane. Draw ~100 line segments to approximate the spheres. Use the coordinate system translation from the tutorial to get a view where camera and spheres fit in the 512x512 debug window. Skip the ground plane in the visualization: you can't really draw an infinite plane with a few lines, after all.

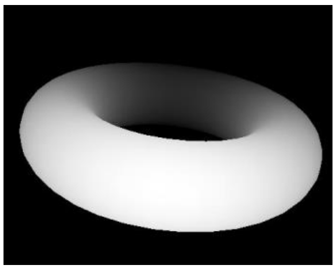
3. Generate primary rays.

Use the loop in the Raytracer.Render method to produce the primary rays. Note that you can use a single ray here; it can be reused once the pixel color has been found. In the debug window, draw a red line for the normalized ray direction. Verify that the generated rays form an arc with radius 1. Verify that no rays miss the screen plane.

4. Intersect the scene.

Use the primary rays to intersect the scene. Now the full rays can be displayed in the debug output. If you visualize rays for y=0 (i.e., line 256 of the 3D view), these rays should exactly end at the sphere boundaries.

5. Visualize the intersections.



Once you have an intersection for a primary ray, you can use its data to make up a pixel color. Plot a black pixel if the ray didn't hit anything. Otherwise, take the intersection distance, and scale it to a suitable range (i.e., the scaled distances should be in the range 0..1). Now you can use this value to plot a greyscale value. This should yield a 'depth map' of your scene.

From here on you can slowly implement the remaining features, and you will always be able to add bits of visualization to see what's happening. E.g., normals can be visualized as little lines pointing away from intersection points, and shadow rays can be colored based on whether they hit an obstruction or not. Use the debug view extensively to verify your code.

Minimum Requirements

To pass this assignment, implement the following features:

Camera:

- Your camera must support arbitrary field of view. It must be possible to set FOV from the application, using an angle in degrees.
- The camera must support arbitrary positions and orientations. It must be possible to aim the camera based on an arbitrary 3D position and target.

Primitives:

- Your ray tracer must at least support planes and spheres. The scene definition must be flexible, i.e. your ray tracer must be able to handle arbitrary numbers of planes and spheres.

Lights:

- Your ray tracer must be able to correctly handle an arbitrary number of point lights and their shadows.

Materials:

- Your ray tracer must support at least diffuse materials and colored specular materials (mirrors). The mirrors must be recursive, with an adjustable cap on recursion depth. There must at least be texturing for a single plane (e.g., the floor plane) to make correct reflections visible (and verifiable).

Application:

- The application must support keyboard and/or mouse handling to control the camera.

Debug output:

- The debug output must be implemented. It must at least show the sphere primitives, primary rays, shadow rays and secondary rays.

Note that there is no performance requirement for this application.

Bonus Assignments

Correctly implementing the minimum requirements earns you a 6 (assuming practical details are all in order). An additional four points can be earned by implementing bonus assignments. An incomplete list of options, with an indication of the difficulty level:

- [EASY] Add triangle support (0.5 pt) with optional normal interpolation (0.5 pt)
- [EASY] Add spotlights, and demonstrate these in your scene (0.5 pt)
- [EASY] Add stochastic sampling of glossy reflections (1 pt)
- [EASY] Add anti-aliasing (0.5 pt)
- [EASY] Implement multi-threading (0.5 pt)
  NOTE: each thread may need its own random number generator for stochastic sampling / anti-aliasing
- [MEDIUM] Add textures to all primitives (1 pt) with optional normal maps (1 pt)
- [MEDIUM] Add a sphere or cube map texture for an environment map (1 pt)
- [MEDIUM] Add refraction (1 pt)
- [MEDIUM] Add stochastic sampling of area lights (1 pt)
- [HARD] Add an acceleration structure and render 5000+ primitives (2 pts)
- [EXTREME] Implement the ray tracer on a GPU using GPGPU, RTX, or DXR (3 pts)

Important: many of these features require that you investigate these yourself, i.e. they are not necessarily covered in the lectures. You may of course discuss these on Teams to get some help. Also note that performance features such as multi-threading or an acceleration structure may be necessary to run some of the more advanced visual features in a reasonable time.

Honors Students

If you are an honors student, we have a special challenge for you. There are in fact two options:

1. Create a path tracer with area lights. You may use the work by James Kajiya ("The Rendering Equation") as a starting point. A path tracer supports indirect lighting by computing secondary rays from any material, not just from mirror reflective/refractive materials.

2. Create a spectral renderer with wavelength-dependent refraction. Instead of RGB values for light transport, a spectral renderer uses a single wavelength. This matters for materials like glass or crystal, where the index of refraction is wavelength-dependent.

If you are doing this assignment as an honors student, please indicate this clearly in your readme.txt file.

Example of spectral ray tracing.

And Finally…

Don't forget to have fun; make something beautiful! Writing a ray tracer can be one of the most fulfilling assignments you'll ever get.