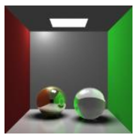


2023/2024, 4th period

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 complete folder and submit it on Blackboard. If your zip-file is many megabytes 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 useful 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.”
– Structure and Interpretation of Computer Programs

Grading:

If you implement the minimum requirements correctly and stick to the above rules, you score a 6.

We deduct points for: a missing readme.txt file, a solution that was not cleaned, a solution that does not compile, a solution that crashes, inconsistent coding style, insufficient comments to explain the code, or missing or incorrectly implemented features.

Implement additional features to earn additional points (up to a 10).

Deliverables:

A ZIP-file containing the contents of your (cleaned) solution folder, including:

(a) All your **source code** (.cs files)

(b) All your **project files** (.sln and .csproj files)

(c) All your **assets files**

(d) The completed **readme.txt** file

The readme.txt file should contain:

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

[1–3 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.

[We will only award grades for features that are mentioned here. We will not search for other features in your code. Tell us how to enable a feature if necessary.]

(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.

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 to click on the final Submit button!
- Re-download your submission from Blackboard, unzip it into a different folder, and check that it runs and looks like the version you intended to submit. This catches most mistakes like missing files or submitting the wrong version. You can correct these mistakes and re-submit until the deadline.
- If you submitted multiple times, we only grade the last version that was submitted before the deadline.

Deadline:

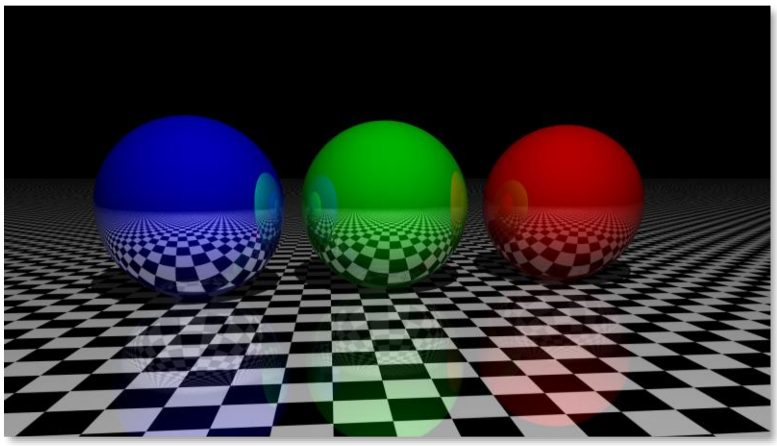
Friday, June 2, 2023, 17:00

This is a hard deadline. If you miss this deadline, your work will not be graded.

Time management: Don’t postpone working on this assignment. It only increases the pressure and stress, and you may run out of time.

Fraud & plagiarism:

- Never look at other students’ code. Don’t discuss implementation in detail. Reference every source in your readme.txt and/or in code comments.
- We use automated content similarity detection tools to compare all submissions of this year and previous years, and online sources. All suspected cases of fraud or plagiarism must be reported to the Board of Examiners.



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').

For this assignment you're **not** expected to write or modify any OpenGL code (any code involving `GL.`*`Something`*). The template provides a `Surface` object called '`screen`' that is the displayed image. You can set the color of individual pixels in that image based on the results of rays that you trace. You can also draw lines, print text, etc. for your debug view. You may write or modify OpenGL code if you like a challenge but you won't earn a higher grade for it (except as part of the GPU bonus feature).

You're **not** expected to re-implement any features that are already provided by the OpenTK library or the template. Such features include basic 2D/3D vector math (dot product, cross product, length, normalization) and loading texture images.

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.

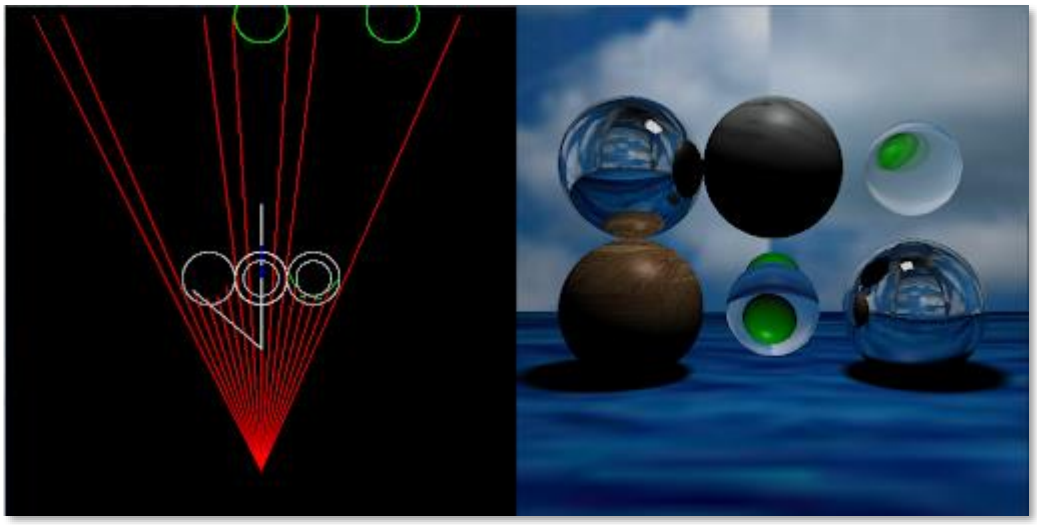
Light, which stores the **location** and **intensity** of a light source. For a Whitted-style ray tracer, this will be a point light. Intensity should be stored using float values for red, green and blue.

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.

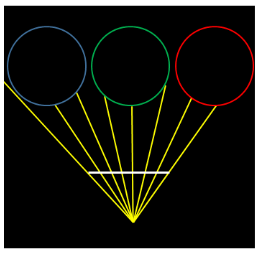
First Steps - Details

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.

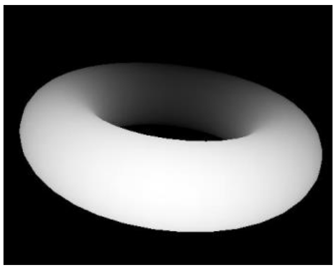
3. Generate primary rays.

Use the loop in your 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.

[*] See the class Raytracer on page 4 for a brief overview of the loop that you should implement in this method.

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 the full Phong shading model including diffuse and glossy materials and ambient light. Your ray tracer must also support pure 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 help verify that reflections work correctly.

Demonstration scene:

- Your application must include a scene that demonstrates all implemented features.

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] A triangle primitive (0.5 pt) with optional normal interpolation (0.5 pt)
- [EASY] Spotlights, and demonstrate these in your scene (0.5 pt)
- [EASY] Stochastic sampling of glossy reflections (1 pt)
- [EASY] Anti-aliasing using a grid or stochastic sampling of rays per pixel (0.5 pt)
- [EASY] Multi-threading (0.5 pt)
  NOTE: multi-threading may break the random number generation for stochastic features; you must fix that to keep those features working
- [MEDIUM] Texturing on all primitives (1 pt) with optional normal maps (1 pt)
- [MEDIUM] Environment mapping to show a cube map or sphere map texture in the background and in mirror reflections and refractions (1 pt)
  NOTE: must be implemented **without adding a cube or sphere object**
- [MEDIUM] Add refraction (1 pt)
  NOTE: each refracted ray also requires a reflected ray to earn the point
- [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 in GPU shaders (3 pts)

Important: many of these features require that you investigate these yourself, i.e. they are not necessarily covered in the lectures or in the book. 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.

Honours Students

If you are a student of the Science Honours Academy (SHA), or considering becoming one, 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 challenge, please indicate clearly in your readme.txt file:

- whether you are (considering becoming) an honours student, or just someone who did the challenge for extra bonus points; and
- which option(s) you implemented.

Example of spectral ray tracing.

Do’s and don’ts

DO	DON’T
Use the Vector3 type from OpenTK.Mathematics or System.Numerics	Don’t implement your own Vector3 class, because it may be buggy or slow, and it won’t give you a higher grade anyway
Use or implement a Color type that stores and calculates with floating point values, where material colors should stay in the range [0, 1] but light colors could be in the range [0, a lot] You could use OpenTK.Mathematics.Color4 but be careful because the constructor will misinterpret four whole numbers as [0, 255], so • new Color4(1.0f, 1, 1, 1) is white • new Color4(1, 1, 1, 1) is almost black	Don’t use a Color type such as System.Drawing.Color that calculates with whole numbers in the range [0, 255], because this would add more factors 255 to every color multiplication. This range should only be used for putting final pixel colors on the screen. Don’t use a tuple or array or three separate variables instead of a proper Color type, because they don’t provide any convenient math operations
Use the vector math operations provided by the Vector3 type	Don’t implement your own dot product, cross product, and other provided vector math operations, because they may be buggy or slow, and it won’t give you a higher grade anyway Don’t write element-wise vector math operations as 3 scalar math operations yourself
Normalize every vector, unless • **you’ve checked that the math works for unnormalized vectors, or** • **you’re certain that it’s already normalized**	Don’t assume that the math works for unnormalized vectors Don’t assume that vector parameters passed to a function have already been normalized
Use single-precision floating point, meaning float and Vector3 types and MathF functions	Don’t use double (e.g., floating point constants without the suffix f), the Vector3d type, or Math (without the suffix F) functions, because they tend to be slower and that much precision isn’t needed in ray tracing
Include the readme.txt file inside your ZIP file	Don’t include the readme.txt as a comment or as a separate file on Blackboard, because it might be lost
Submit your whole project folder, including source code, shaders, assets, and IDE-related files	Don’t omit any files, e.g., the template files or the shaders, because we should be able to run your project as submitted
Include the readme.txt, clean the project, write comments where necessary, use formatting and clear naming to make the code more readable, etc.	Don’t miss out on the easy points that you could earn for making our grading work a bit easier

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.