Middle East Technical University | Department of Computer Engineering

CENG 795

Advanced Ray Tracing

Fall '2024-2025
Assignment 1 - Recursive Ray Tracing
(v.1.0)

Due date: October 22, 2024, Tuesday, 23:59

1 Objectives

Ray tracing is a fundamental rendering algorithm. It is commonly used for animations and architectural simulations, in which the quality of the created images is more important than the time it takes to create them. In this assignment, you are going to implement a recursive ray tracer that simulates the propagation of light in the real world.

Keywords: *recursive ray tracing, light propagation, geometric optics, ray-object intersections, surface shading, conductors and dielectrics*

2 Specifications

1. You should name your executable as "raytracer".
2. Your executable will take an XML scene file as argument (e.g. "scene.xml"). A parser will be given to you, so that you do not have to worry about parsing the file yourself. The format of the file will be explained in Section 3. You should be able to run your executable via command "./raytracer scene.xml".
3. You will save the resulting images in the PNG format. You can use a library of your choice for saving PNG images.
4. The scene file may contain multiple camera configurations. You should render as many images as the number of cameras. The output filenames for each camera is also specified in the XML file.
5. There is no time limit for rendering the input scenes. However, you should report your time measurements in your blog post. Try to write your code as optimized as possible. But if a scene takes too long to render, do not worry yet. Acceleration structures that we will learn later should help you in the second homework.

6. You should use Blinn-Phong shading model for the specular shading computations. Mirrors and dielectrics should obey Fresnel reflection rules. Dielectrics are assumed to be isotropic and should obey Beer's law.

7. You will implement two types of light sources: point and ambient. There may be multiple point light sources and a single ambient light. The values of these lights will be given as (R, G, B) color triplets that are not restricted to $[0, 255]$ range (however, they cannot be negative as negative light does not make sense). Any pixel color value that is calculated by shading computations and is greater than 255 must be clamped to 255 and rounded to the nearest integer before writing it to the output PNG file. This step will be replaced by the application of a tone mapping operator in our later homeworks.

8. Point lights will be defined by their intensity (power per unit solid angle). The irradiance due to such a light source falls off as inversely proportional to the squared distance from the light source. To simulate this effect, you must compute the irradiance at a distance of d from a point light as:

$$E(d) = \frac{I}{d^2},$$

   where I is the original light intensity (a triplet of RGB values given in the XML file) and $E(d)$ is the irradiance at distance d from the light source.

9. **Back-face culling** is a method used to accelerate the ray - scene intersections by not computing intersections with triangles whose normals are pointing away from the camera. Its implementation is simple and done by calculating the dot product of the ray direction with the normal vector of the triangle. If the sign of the result is positive, then that triangle is ignored. Note that shadow rays should not use back-face culling. In this homework, back-face culling implementation is optional. Experiment with enabling and disabling it in your ray tracers and report your time measurements in your blog post.

10. **Degenerate triangles** are those triangles whose at least two vertices coincide. The input files given to you should be free of such cases, but models downloaded from the Internet occasionally have this problem. You can put a check in your ray tracer either to detect or ignore such triangles (using such triangles usually produce NaN values).

3 Scene File

The scene file will be formatted as an XML file (see Section 7). In this file, there may be different numbers of materials, vertices, triangles, spheres, lights, and cameras. Each of these are defined by a unique integer ID. The IDs for each type of element will start from one and increase sequentially. Also notice that, in the XML file:

· Every number represented by X, Y and Z is a floating point number.

· Every number represented by R, G, B, and N is an integer.

Explanations for each XML tag are provided below:

- **BackgroundColor:** Specifies the R, G, B values of the background. If a ray sent through a pixel does not hit any object, the pixel will be set to this color. Only applicable for primary rays sent through pixels.

- **ShadowRayEpsilon:** When a ray hits an object, you are going to send a shadow ray from the intersection point to each point light source to decide whether the hit point is in shadow or not. Due to floating point precision errors, sometimes the shadow ray hits the same object even if it should not. Therefore, you must use this small ShadowRayEpsilon value, which is a floating point number, to move the intersection point a bit further from the hit point in the direction of the hit point's normal vector so that the shadow ray does not intersect with the same object again. Note that ShadowRayEpsilon value can also be used to avoid self-intersections while casting reflection and refraction rays from the intersection point.

- **MaxRecursionDepth:** Specifies how many bounces the ray makes off of conductor and dielectric objects. Primary rays are assumed to start with zero bounce count.

- **Camera:**

  ·**Position** parameters define the coordinates of the camera.

  ·**Gaze** parameters define the direction that the camera is looking at. You must assume that the Gaze vector of the camera is always perpendicular to the image plane.

  ·**Up** parameters define the up vector of the camera.

  ·**NearPlane** attribute defines the coordinates of the image plane with Left, Right, Bottom, Top floating point parameters, respectively.

  ·**NearDistance** defines the distance of the image plane to the camera.

  ·**ImageResolution** defines the resolution of the image with Width and Height integer parameters, respectively.

  ·**ImageName** defines the name of the output file.

  Cameras defined in this homework will be right-handed by default. The mapping of Up and Gaze vectors to the camera terminology used in the course slides is given as:

$$\begin{aligned} \text{Up} &= v \\ \text{Gaze} &= -w \\ u &= v \times w \end{aligned}$$

- **AmbientLight:** is defined by just an X, Y, Z radiance triplet. This is the amount of light received by each object even when the object is in shadow. Color channel order of this triplet is RGB.

- **PointLight:** is defined by a position and an intensity, which are all floating point numbers. The color channel order of intensity is RGB.

- **Material:** A material can be defined with ambient, diffuse, and specular properties for each color channel. The values are floats between 0.0 and 1.0, and color channel order is RGB. PhongExponent defines the specularity exponent in Blinn-Phong shading. For reflection and refraction, the material type attribute must be checked. The values can be "mirror", "conductor", and "dielectric". For mirrors, do not apply Fresnel reflection. It is assumed that such

surfaces have constant reflectivity (as indicated by the MirrorReflectance element). For both conductors and dielectrics, you should apply Fresnel reflection. Fresnel calculations should be done using the values of the RefractionIndex and AbsorptionIndex elements. For conductors you should still use MirrorReflectance to further modulate the reflectivity of different color channels. We do this to induce a sense of color for metal materials. You can see some examples of gold materials in the input files. Finally for dielectrics you must use the "AbsorptionCoefficient" element to attenuate different colors using different amounts. This way we can also induce color for dielectric (e.g. glass) materials. You need to use this coefficient for the "c" parameter in the Beer's Law formula:

$$L(x) = L(0)\exp(-cx),$$

where $L(x)$ is the luminance after a refracted ray travels a distance of x inside the material, $L(0)$ is the luminance at the interface of the material, and c is equal to the "AbsorptionCoefficient". Note that c is a triplet to simulate the fact that a material may absorb different colors by different amounts.

- **VertexData:** Each line contains a vertex whose x, y, and z coordinates are given as floating point values, respectively. The first vertex's ID is 1.
- **Mesh:** Each mesh is composed of several faces. A face is actually a triangle which contains three vertices. When defining a mesh, each line in Faces attribute defines a triangle. Therefore, each line is composed of three integer vertex IDs given in counter-clockwise order (see Triangle explanation below). Material attribute represents the material ID of the mesh. Some meshes use Stanford Polygon File Format (PLY). See `http://paulbourke.net/dataformats/ply/` for the definition of this format and parser codes.
- **Triangle:** A triangle is represented by Material and Indices attributes. Material attribute represents the material ID. Indices are the integer vertex IDs of the vertices that construct the triangle. Vertices are given in counter-clockwise order, which is important when you want to calculate the normals of the triangles. Counter-clockwise order means that if you close your right-hand following the order of the vertices, your thumb points in the direction of the surface normal.
- **Sphere:** A sphere is represented by Material, Center, and Radius attributes. Material attribute represents the material ID. Center represents the vertex ID of the point which is the center of the sphere. Radius attribute is the radius of the sphere.

4 Hints & Tips

1. Start early. It takes time to get a ray tracer up and running – especially if this is your first ray tracer!
2. You may use the -O3 option while compiling your code for optimization. This itself will provide a huge performance improvement.
3. Try to pre-compute anything that would be used multiple times and save these results. For example, you can pre-compute the normals of the triangles and save it in your triangle data structure when you read the input file.

4. If you see generally correct but noisy results (black dots), it is most likely that there is a floating point precision error (you may be checking for exact equality of two FP numbers instead of checking if they are within a small epsilon).

5. For debugging purposes, consider using low resolution images. Also it may be necessary to debug your code by tracing what happens for a single pixel (always simplify the problem when debugging).

5 Bonus

I will be more than happy to give bonus points to students who make important contributions such as new scenes, importers/exporters between our XML format and other standard file formats. Note that a Blender exporter[1], which exports Blender data to our XML format, was written by one of our previous students. You can use this for designing a scene in Blender and exporting it to our file format.

6 Regulations

1. **Programming Language:** C/C++ is the recommended language. However, other languages can be used if so desired. In the past, some some students used Rust or even Haskell for implementing their ray tracers.

2. **Changing the Sample Codes:** You are free to modify any sample code provided with this homework.

3. **Additional Libraries:** If you are planning to use any library other than *(i)* the standard library of the language, *(ii)* pthread, *(iii)* the XML parser, and the PNG libraries please first ask about it on ODTUClass and get a confirmation. Common sense rules apply: if a library implements a ray tracing concept that you should be implementing yourself, do not use it!

4. **Submission:** Submission will be done via ODTUClass. To submit, Create a "**tar.gz**" file named "raytracer.tar.gz" that contains all your source code files and a Makefile. The executable should be named as "raytracer" and should be able to be run using the following commands (scene.xml will be provided by us during grading):

   tar -xf raytracer.tar.gz
   make
   ./raytracer scene.xml

   Any error in these steps will cause point penalty during grading.

5. **Late Submission:** You can submit your codes up to 3 days late. Each late day will cause a 10 point penalty.

[1] `https://saksagan.ceng.metu.edu.tr/courses/ceng477/student/ceng477exporter.py`

6. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating will be punished according to the university regulations and will get 0 from the homework. You can discuss algorithmic choices, but sharing code between groups or using third party code is strictly forbidden. By the nature of this class, many past students make their ray tracers publicly available. You must refrain from using them at all costs.

7. **Forum:** Check the ODTUClass forum regularly for updates/discussions.

8. **Evaluation:** The basis of evaluation is your blog posts. Please try to create interesting and informative blog posts about your ray tracing adventures. You can check out various past blogs for inspiration. However, also expect your codes to be compiled and tested on some examples for verification purposes. So the images that you share in your blog post must directly correspond to your ray tracer outputs.

7 Sample Scene File

```
<Scene>
        <BackgroundColor>R G B</BackgroundColor>
        <ShadowRayEpsilon>X</ShadowRayEpsilon>
        <MaxRecursionDepth>N</MaxRecursionDepth>
        <Cameras>
                <Camera id="Cid">
                        <Position>X Y Z</Position>
                        <Gaze>X Y Z</Gaze>
                        <Up> X Y Z </Up>
                        <NearPlane>Left Right Bottom Top</NearPlane>
                        <NearDistance>X</NearDistance>
                        <ImageResolution>Width Height</ImageResolution>
                        <ImageName>ImageName.ppm</ImageName>
                </Camera>
        </Cameras>
        <Lights>
                <AmbientLight>X Y Z</AmbientLight>
                <PointLight id="Lid">
                        <Position>X Y Z</Position>
                        <Intensity>X Y Z</Intensity>
                </PointLight>
        </Lights>
        <Materials>
                <Material id="Mid" [type="mirror"]>
                        <AmbientReflectance>X Y Z</AmbientReflectance>
                        <DiffuseReflectance>X Y Z</DiffuseReflectance>
                        <SpecularReflectance>X Y Z</SpecularReflectance>
                        <MirrorReflectance>X Y Z</MirrorReflectance>
                        <PhongExponent>X</PhongExponent>
                </Material>
        </Materials>
        <VertexData>
                V_{1X} V_{1Y} V_{1Z}
                V_{2X} V_{2Y} V_{2Z}
                .....
        </VertexData>
        <Objects>
                <Mesh id="Meid">
                        <Material>N</Material>
                        <Faces>
                                F_{1_V1} F_{1_V2} F_{1_V3}
                                F_{2_V1} F_{2_V2} F_{2_V3}
                                .....
                        </Faces>
                </Mesh>
                <Triangle id="Tid">
                        <Material>N</Material>
                        <Indices>
                                V_1 V_2 V_3
                        </Indices>
                </Triangle>
                <Sphere id="Sid">
                        <Material>N</Material>
                        <Center>N</Center>
                        <Radius>X</Radius>
                </Sphere>
        </Objects>
</Scene>
```