Middle East Technical University | Department of Computer Engineering

CENG 795

Advanced Ray Tracing

Fall '2025-2026
Assignment 1 - Recursive Ray Tracing
(v.1.0)

Due date: October 29, 2025, Wednesday, 23:59
Blog due date: November 01, 2025, Saturday, 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 a JSON scene file as argument (e.g. "scene.json"). The format of the file should be self-explanatory but some information is provided in Section 3. You should be able to run your executable via command "./raytracer scene.json".

3. You will save the resulting images in the PNG format. You can use a library of your choice for saving PNG images. A popular choice is the stb-image library[1].

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 specified in the scene 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. Conductors and dielectrics should obey Fresnel reflection rules. Dielectrics are assumed to be isotropic and should obey Beer's law for light attenuation. Simple mirrors do not follow the Fresnel 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 <u>not</u> 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 scene 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).

[1] `https://github.com/nothings/stb`

3 Scene File

The scene file will be formatted as a JSON 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 ID. The IDs for each type of element will start from one and increase sequentially. Explanations for the elements in this file are given 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, dielectric, and mirror 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:

$$w = -\frac{\text{Gaze}}{||\text{Gaze}||},$$
$$v = \frac{\text{Up}}{||\text{Up}||},$$
$$u = v \times w.$$

In some scene files, you may find that the given gaze and up vectors are not perpendicular to each other. In such a case, you need to correct the up vector to make it perpendicular to the gaze vector. This processed can be achieved by the following sequence of operations:

$$w = -\frac{\text{Gaze}}{||\text{Gaze}||},$$
$$v' = \frac{\text{Up}}{||\text{Up}||},$$
$$u = v' \times w,$$
$$v = w \times u.$$

Finally, some cameras are defined with the type attribute "lookAt". These cameras assume a symmetric image plane centered along the gaze direction. These cameras take a "GazePoint" to specify the point that the camera is looking at. You can find the gaze direction by subtracting the camera position from this gaze point. The "FovY" parameter specifies the field of view in degrees that the image plane covers in its vertical direction. The aspect ratio is implicitly defined by the resolution of the image plane. For example, if the image width is twice the image height in pixels, the aspect ratio would be two. You can refer to the course slides for computing the near plane parameters for these camera types.

- **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 for refracted rays. You need to use this coefficient for the c parameter in the Beer's Law formula:

$$L(x) = L(0)e^{-cx},$$

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. As dielectrics also reflect light, you can still use the "MirrorReflectance" coefficient to alter the colors of the reflected rays. This way we can control the final color for dielectric (e.g. glass) materials.

- **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 vertex indices. These indices define the triangle 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.

  Meshes can be *flatly* or *smoothly* shaded. In a flat shaded mesh, each triangle has a single surface normal solely derived from the edge vectors of the triangle. Such meshes have a faceted appearance especially if the triangle count is low. Smoothly shaded meshes require computation of per-vertex normals (as opposed to per-face) normals. A per-vertex normal can be computed as the weighted average of triangle normals that share that vertex, where weights are given by the areas of the triangles. That is, a larger triangle contributes more to a per-vertex normal than a smaller triangle for the vertices that they are sharing. The information about which vertices are shared by which triangles can be found from the index information. For example, if a triangle is using indices 5, 6, 7 and another one using 6, 4, 8, we can infer that vertex 6 is shared.

- **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.

- **Plane:** A plane is represented by Material, Point, and Normal attributes. Material attribute represents the material ID. Point represents the vertex ID of a point on the plane. Normal attribute defines the surface normal of the plane.

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 JSON format and other standard file formats. Note that a Blender exporter[2], which exports Blender data to our previously used XML format is available. After producing the XML file, you can run the bash script here[3] to produce the final JSON file. You can use these scripts 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. **Additional Libraries:** If you are planning to use any library other than *(i)* the standard library of the language, *(ii)* pthread, *(iii)* the JSON parser[4], *(iv)* the image IO libraries[5], and *(v)* PLY parsing libraries[6] 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!

3. **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.json will be provided by us during grading):

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

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

4. **Late Submission:** You can submit your codes up to 3 days late. Each late day will cause a 10 point penalty. The blog deadline is 3 days later than the regular deadline and no further late submission is allowed for the blog posts.

[2] `https://user.ceng.metu.edu.tr/~akyuz/courses/795/ceng795exporter.py`
[3] `https://user.ceng.metu.edu.tr/~akyuz/courses/795/xml2json.sh`
[4] `https://github.com/nlohmann/json`
[5] `https://github.com/nothings/stb`
[6] `http://paulbourke.net/dataformats/ply/`

5. **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.

   Using Large Language Models (LLMs) and other AI tools: We note that AI can be a useful learning resource but if it is used for generating code that you are supposed to write, it can have a detrimental effect on your learning. Therefore, using code from other resources, whether it is generated by humans or AI, is considered cheating.

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

7. **Evaluation:** The primary basis of evaluation is your blog posts. Therefore try to write interesting and informative blog posts about your ray tracing journey. You can check out various past blogs for inspiration. However, also expect your codes to be compiled and tested on several input scenes for verification purposes. Therefore images that you share in your blog posts must directly correspond to your own ray tracer outputs.

7 Sample Scene File

```
{
  "Scene": {
    "BackgroundColor": "0 0 0",
    "ShadowRayEpsilon": "1e-3",
    "IntersectionTestEpsilon": "1e-6",
    "Cameras": {
      "Camera": {
        "_id": "1",
        "Position": "0 0 0",
        "Gaze": "0 0 -1",
        "Up": "0 1 0",
        "NearPlane": "-1 1 -1 1",
        "NearDistance": "1",
        "ImageResolution": "800 800",
        "ImageName": "simple.png"
      }
    },
    "Lights": {
      "AmbientLight": "25 25 25",
      "PointLight": {
        "_id": "1",
        "Position": "0 0 0",
        "Intensity": "1000 1000 1000"
      }
    },
    "Materials": {
      "Material": {
        "_id": "1",
        "AmbientReflectance": "1 1 1",
        "DiffuseReflectance": "1 1 1",
        "SpecularReflectance": "1 1 1",
        "PhongExponent": "1"
      }
    },
    "VertexData": {
      "_data": "-0.5 0.5 -2 -0.5 -0.5 -2 0.5 -0.5 -2 0.5 0.5 -2 0.75 0.75 -2 1 0.75 -2 0.875 1 -2 -0.875 1 -2 0 -0.5 0",
      "_type": "xyz"
    },
    "Objects": {
      "Mesh": {
        "_id": "1",
        "Material": "1",
        "Faces": {
          "_data": "3 1 2 1 3 4",
          "_type": "triangle"
        }
      },
      "Triangle": {
        "_id": "1",
        "Material": "1",
        "Indices": "5 6 7"
      },
      "Sphere": {
        "_id": "1",
        "Material": "1",
        "Center": "8",
        "Radius": "0.3"
      },
      "Plane": {
        "_id": "1",
        "Material": "1",
        "Point": "9",
        "Normal": "0 1 0"
      }
    }
  }
}
```