Middle East Technical University    Department of Computer Engineering

# CENG 795

Advanced Ray Tracing

Fall '2022-2023
Assignment 3 - Multisampling and Distribution Ray Tracing
(v.1.0)

Due date: November 27, 2022, Sunday, 23:59

## 1 Objectives

In ray tracing, we learn about the actual light distribution in a given scene by sampling it using a fixed amount of rays. In the previous assignments, we rendered each scene by sending a primary ray through the center of each pixel – i.e. by single sampling. This approach is prone to aliasing artifacts and multisampling is a technique that aims to mitigate the aliasing problem. In this assignment, you will be implementing multisampling and a classical technique called distribution ray tracing that aims to add various effects at little cost over multisampling. While distribution ray tracing can be implemented without multisampling, the results will appear noisy and therefore the two techniques are often combined together.

**Keywords:** *multisampling, aliasing, reconstruction, filtering, distribution ray tracing*

## 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. It is expected that you have implemented an acceleration structure in the previous assignment. Without this, the rendering times with multisampling can be prohibitively long. So make sure your acceleration structure works well enough before attempting this 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 three types of light sources: point, ambient, and area light sources. Note that area lights is a new addition in this homework. More details about this light source are given below. There may be multiple point and area 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. Area lights will be defined by their area, radiance, position, and surface normal direction. See the next section for their detailed description.

10. **Back-face culling** is optional as before. Please see previous assignment texts for more details.

## 3 Scene File

Please see the previous assignments for a detailed description of the scene file. In this section, only the elements introduced in this homework are explained.

- **NumSamples:** A child of the Camera element. It defines how may samples you need to send through each pixel. Jittered sampling (i.e., stratified uniform sampling) is the recommended approach to use. Please refer to the lecture slides for a pseudocode of this technique. In all scene files, this element is defined as a perfect square (e.g., 1, 2, 4, 16, etc.) so that you can divide the pixel into equal number of rows and columns.

- **ApertureSize:** This element is also a child of the Camera element. If it exists, it means you must also pick a sample on the lens to create the depth-of-field effect. We assume that the lens is flat square that covers the aperture of the camera. The "ApertureSize" element determines the edge length of this square lens. If you prefer, you can implement the aperture as a flat disk, but be careful when uniform sampling this disk. In this case, you can take this value as the diameter of the disk. Note that the aperture's center position is the camera origin.

- **FocusDistance:** FocusDistance represents the vertical distance to an imaginary focal plane. The objects that are on or near this plane will appear to be in sharp focus in the final image. Objects that are closer or further will appear blurry creating a depth of field effect. See distribution_ray_tracing_explained.pdf for a visualization of this element.

- **AreaLight:** Area lights are imaginary squares that provide illumination for the scene. They are typically used for effects such as soft shadows. An area light is defined by its normal, size (edge length), radiance, and position:

  **Position:** The center point of the area light

  **Normal:** The surface normal of the light

  **Size:** The edge length of the area light. Its area will be equal to the square of this value

  **Radiance:** Radiance of the light

  The orientation of an area light needs to be taken into account when computing the light that arrives from it. Although we do not attenuate the *radiance* of this light based on the distance (remember that radiance is preserved in vacuum), the received energy will still be affected as the area light will cover a smaller solid angle based on the distance from it. This relationship is given by:

  $$dw_i = A\frac{\mathbf{n_l} \cdot \mathbf{w_i}}{d^2}, \tag{1}$$

  where $A$ is the surface area of the area light, $\mathbf{n_l}$ is its surface normal, $\mathbf{w_i}$ is the emitted light direction, and $d$ is the distance between the sampled point on the light and the intersection point for which we are computing the shading. Note that the latter two terms can be different for each sample. So while $L_i$ in the rendering equation does not change, the product of $L_i dw_i$ changes as a function of distance and orientation. For a more thorough explanation of this phenomenon, please refer to areaLight.pdf in the resources folder.

- **MotionBlur:** Motion blur simulates the blurry appearance of fast moving objects. It is defined for objects using the "MotionBlur" element (e.g. dragon_dynamic.xml). For simplicity motion blur will be assumed to be only for translational movements. MotionBlur element defines the net translation of the object during which the image capture was taking place. We implement this by adding a "time" element to our rays. You can assume that the object is in its original position (computed after applying all of the transformations except the motion blur) at time is equal to 0 and it is in its final position (transformed position plus the motion blur vector) at time is equal to 1. As each sample will have a random time value between 0 and 1, the object will appear at slightly different positions for each sample creating the illusion of a fast moving object. Please see distribution_ray_tracing_explained.pdf for more details.

- **Roughness:** A child of the Material element. It defines the roughness of the mirrors and conductors. This can technically be defined for dielectrics as well but supporting it for dielectrics is optional. Roughness is a single scalar value that determines how much the reflected ray may deviate from the ideal reflection direction. The larger this value, the more rough a metallic object will appear and vice versa. In general, we call objects with a smaller roughness to be more glossy and polished. Again please see distribution_ray_tracing_explained.pdf for more details.

## 4 Hints & Tips

In addition to those for the previous homeworks, the following tips may be useful for this homework.

1. Although the visual results of this homework are quite appealing, the implementation of the concepts is not difficult. So do not be intimidated by the results.

2. For each ray (i.e. for each sample), you must generate 5 uniform random numbers between 0 and 1. Two are for the position within a pixel, two for position within the aperture, and one for time.

3. You can use the Mersenne Twister algorithm for random number generation, which fortunately is implemented in C++ standard template library. The usage of this library to generate a uniform random number in range $[0, 1)$ is illustrated below:

```
std::mt19937 gRandomGenerator;
std::uniform_real_distribution<> gNURandomDistribution(0, 1);
float psi = gNURandomDistribution(gRandomGenerator);
```

## 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!

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

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.

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.