CPS5124-Assignment

Zachary Cauchi zcau0001

# Overview

In this report, the code for a ray tracing application is described. This application being capable of using ray tracing to produce images with better accurate light simulation than rasterisation-based methods. Furthermore, the performance of the ray tracer – both from the amount of time required to render an image, and the accuracy of the images when compared to similar reference images – are shown and discussed. The document finishes with a conclusion and summary of the final deliverables, including areas of future expansion for the application to better meet the reference images and go beyond them.

# System design

The program is written in C++, compiled using cmake as a build generator, and executable on Linux environments. The project – including source code and resource files – are available on [Github](https://github.com/Zach128/CPS5124-assignment-1) as of 1st August 2021. The ray tracer is designed to run on multiple CPU cores, using OpenMP for multi-core support and performs load-balancing to ensure each core is used optimally throughout runtime. For input, files are read from a directory called “res/scenes” relative to the program’s current working directory and outputs the rendered results relative to that directory (so a file output path of “../output/example.ppm” will output to “res/output/example.ppm”). JSON scene files are loaded and parsed using the [nlohmann::json](https://github.com/nlohmann/json) library. The program is compiled and executed through the following sequence:

* Open a terminal in a Linux environment in the current directory of the program.
* Run “cmake .”
* Run “make release” to compile with optimisations (“make debug” will not use optimisations but will produce debug symbols).
* Run “./keithray”.
* The program will load and process all the given files in “res/scenes”.

## Loading scene files

The program begins by loading a given JSON file from the “res/scenes” directory. The program expects a JSON file following the schema given by the sample input files labeled “assignment\_01.json” through “assignment\_08.json”. Each renderable component of a scene (namely lights, cameras, materials, and shapes) are loaded prior to constructing the scene. Each component has its own class/struct model with its own `*from\_json*` routine for parsing JSON objects. Additionally, the general naming scheme of the objects is preserved except for the types, which are converted to enums unique to the category of object being created. With all renderable objects loaded, the primitives are loaded and created accordingly, with the final renderer being created. In the case of both and cameras, there can be more than one camera or light source, but there can only be one active camera, and only the lights specified in the “*scene*” object are loaded and rendered. With these finally loaded, the Scene object is constructed and returned for processing.

## The renderers

There are two main renderers available to the program:

* The whitted-renderer follows a Whitted-style ray tracing approach, focusing only on point-based direct lighting.
* The path-renderer which builds upon the whitted-renderer by being capable of global illumination through indirect lighting and support of geometry-based area lights.

In the initial preparation routine, the framebuffers are first initialized according to the resolution of the image. A ray sampler is also initialized which will generated camera rays given {x,y} pixel coordinates.

## Ray sampler

The sampler performs the ray generation in place of the camera to keep the camera structs clean and data oriented. Given the camera, image resolution, and sampling method, it will calculate an orthogonal matrix with which to convert generated rays from camera-space (relative to the camera’s orientation and position) to world-space (aligned with the world’s XYZ axis). The sampler can generate a set of coordinates for a given xy pixel coordinate, with slight random offset for each ray to create an antialiasing effect on the final image. The method by which rays are generated depends on the type of camera used:

* Pinhole cameras result in rays originating from the camera’s position to where the pixel would lie relative to the size of the image, aspect ratio, and field-of-view.
* Lens cameras follow a similar process to pinholes cameras with two exceptions; the orthogonal matrix to make rays want to converge at the focal point, which is defined as being a given distance directly forward of the camera position and orientation (this distance being the “*focal\_distance*”). A camera aperture is simulated by picking random points on this aperture and using those as the new ray origin in place of the camera position. This model is follows the “Thin Lens Approximation”, which causes objects away from the focal point of the camera to appear out-of-focus and blurry while keeping objects near to the focal point sharp and in-focus.

## Main render loop

Inside the main render function, the program iterates over every single pixel in the framebuffer, performing the ray tracing calculations to generate the final colour for that pixel. Given that the renderers follow a distributed ray tracing architecture whereby each pixel is computed from multiple sample rays, the loop is parallelized, and the workload distributed evenly amongst all CPU cores on the host machine. For each pixel, a set of sample rays are generated, which when all are cast by the renderer will return the average overall colour for that pixel. Each renderer follows the below process for casting rays:

### Whitted-renderer

* Check if the ray hits an object. If it does – and the ray isn’t nested too deep, – record the hit position, the normal of the hit surface, the distance, the hit primitive, and the depth of the hit.
* If the hit material is diffuse, calculate the overall diffuse contribution for all the lights in the scene.
* If the material is glossy or specular, calculate a reflection ray from the hit point and cast it, returning the final colour for that ray.
* If the material is a Fresnel material, calculate the ratio of reflectance to refractance. If the ray exhibits some refraction on hitting the surface, cast a refracted ray through the shape. In any case, reflect and cast a ray from the hit position on the surface. Mix the two returned colours together to produce the final Fresnel contribution.

### Path-renderer

* Perform Russian roulette testing, determining if the ray has gone too deep or not by rolling a random number and seeing if it meets the criteria to return immediately or not. Russian roulette is performed upon crossing a depth threshold and not immediately from the first ray onwards.
* After Russian Roulette is passed, check if the ray hits an object, recording the same data on that hit as in the whitted-renderer.
* If the hit material is diffuse, calculate the overall direct contribution for all the lights in the scene.
  + Calculate the overall indirect light contribution in addition to the direct contribution.
* If the material is glossy or specular, calculate a reflection ray from the hit point and cast it, returning the final colour for that ray.
* If the material is a Fresnel material, calculate the transmission and reflection colours for that point and mix them to obtain the final Fresnel contribution.

Both renderers inherit most of the material computation from a shared parent class called “*renderer*”. This defines the behaviour involved in calculating colours for the kind of materials available, including how to reflect, refract, and perform intersection testing with objects in the scene. Lighting sources are provided by the child renderers to allow greater control over how the light is being sampled.

## Lighting

Lighting is supported for point lights and spherical area lights. A point lights are measured from their position and only use a single sample. Point lights have an intensity vector which holds the intensity across the individual red, green, and blue components. Area lights are treated as emissive primitives, having no position of their own but an assigned shape and material instead. When being sampled, 4 random spots are sampled from the sphere using the below process:

* Create an orthogonal matrix which can transform points around the light.
* Pick a random spot on a disc equal in radius to the spherical area light and at the same position as the light.
* Transform the position of the random point using the orthogonal matrix. This will position the point on a hemispherical subsection of the sphere which is facing the initial hit of the ray. This effectively makes the sure all samples are in view of the ray.
* Calculate the mean contributing diffusion from all the sampled light points.

By sampling multiple points of the light source, the overall intensity can be attenuated based on how much of the light is obscured by other objects, creating soft shadows in the process.

## Post-processing

Once the main render function has completed rendering the image by iterating over all pixels, a post-processor can be triggered on the final image. Currently there are 3 supported post-processors:

* Box-blur: Applies a static blurring effect to the image.
* Linear tonemap: Applies a linear tonemapping function to the image.
* Sigmoidal tonemap: Applies a sigmoidal tonemapping function to the image.

The tonemaps begin by pre-scaling each input colour in the framebuffer to a range of {0..1.0}. The framebuffer is then iterated over, applying the selected tonemap function to each pixel. The tonemapped framebuffer is then scaled back up to the RGB range of {0..255.0}.

## Saving the image

The image is saved in PPM format with each RGB component having a max colour value of 255. The save path is obtained from the scene object and is determined relative to the input path of the scene. Two files are saved for each scene: A colour image rendering of the scene, and a greyscale depth map of the scene.

# Results

The results were obtained on a computer with 8 logical cores and 24GB of memory. Results are obtained over 3 separate passes. The program is compiled with optimisations (-O3) enabled and no debug symbols. All measures are in seconds and generate the images found in the comparisons. Additional details about the results can be found in the spreadsheet included with the source code in the zip file.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Time to render (s) | | | | | | | |
| Run | Scene 1 | Scene 2 | Scene 3 | Scene 4 | Scene 5 | Scene 6 | Scene 7 | Scene 8 |
| 1 | 0.715 | 0.300 | 113.136 | 297.518 | 776.712 | 1,036.330 | 1.669 | 1,885.740 |
| 2 | 0.732 | 0.293 | 120.083 | 274.024 | 851.205 | 1,006.670 | 2.338 | 1,823.260 |
| 3 | 0.651 | 0.283 | 110.185 | 254.004 | 791.628 | 918.542 | 1.709 | 1,642.980 |
| Avg | 0.698 | 0.292 | 114.320 | 274.038 | 805.260 | 984.575 | 1.861 | 1,777.868 |

Table 1: Render result times for the 8 sample scenes across 3 separate runs.

# Image comparisons

Below is a comparison of the generated images from scene files assignment\_01.json through to assignment\_02.json. On the left are the generated files and on the right are reference images of the scenes used throughout development. As is the case with the reference images, the same scene settings were used with the final output being tonemapped using a sigmoidal-tonemapper.

|  |  |
| --- | --- |
| Generated images | Reference images |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Table 2: Comparison of generated images (left) to reference images(right)

An example scene can be found below. This is based on the assignment\_03 scene, but with the addition of another area light, purple-hued Fresnel sphere, and use of a lens camera. In addition to this, an example scene called “furnace” is available in the scene files. This is a test scene in which a diffuse sphere is surrounded by an area light, created as a test for the area lighting implementation.

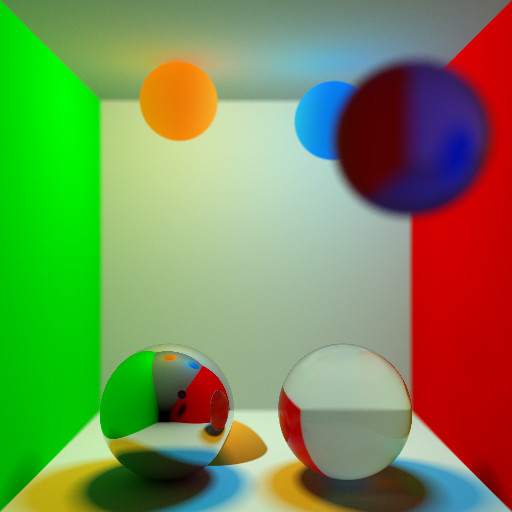


Figure 1: Image of the many\_spheres scene, rendered with 1024 samples per-pixel.

# Discussion

The renderer manages to position and render all geometry as dictated by the scenes, omitting those lights, cameras and primitives which were excluded from the scene object in the file (such as the lens camera in assignment\_03 or the point light in assignment\_08). Additionally, soft shadows are rendered correctly, and in some cases with fewer artifacts than the reference images. Transmissive surfaces and specular/glossy surfaces both appear to accurately reflect/refract rays as in the references. However, glossy/specular surfaces do not scatter the rays, leaving the surfaces with sharp reflections. The rendering function also does not render caustics with the same intensity as the reference images likely due to convergence speed. Area lights emit light accurately and are sampled for global illumination properly. Their drawback is that they lose their bright radiance in the final images. Better tonemapping which preserves bright highlights such as the lights could be developed to solve the issue.

The biggest discrepancy between the generated images and the reference images is the lighting: the renderer does preserve light well enough. The images generated are very brightly lit across both the Whitted-style and path renderers. The problem could likely be due to the lighting calculation of diffuse surfaces. In addition to this, better distance-based light falloff could be implemented to create a tamer area of effect by the light.

# Conclusion

This document described ray tracing application. It described its design and processes, its rendered output, the times taken for the results, and discussed its strengths and limitations. The rendered images were presented for comparison to their references, showing the strengths and weaknesses of the renderers. Future improvements have also been discussed, namely revised lighting and diffuse calculations, improved caustics rendering, better tonemapping, and better rendering of area lights.