Ray Tracing with Procedural Texturing

Jiahui Huang, Ling Gan, Ye Chan Lee
USC CSCI580 3D Graphics and Rendering Final Project

Abstract: Use C++ to implement a ray tracer that supports .obj file and has multiple features, such as anti-aliasing, reflection, soft shadow and procedural texturing.

1. Introduction

As we have spent the entire semester exploring the topic of rendering, we would like to utilize our knowledge to implement our own ray tracer. This project allows us to explore in terms of computational cost, visual fidelity, and structure design of rendering 3D models. It also opened the door and connected us to the real-world 3D rendering with respect to different input files in polygon mesh, material files, and etc.

2. Architecture

The high level structure of our ray tracer is as Figure 1.

3. Implementation

3.1. Input

In this project, we explored the widely used wavefront 3D model file format of .obj and .mtl. In order to parse the .obj file[1], we referenced many light weight obj parsers and adapted them to fit our objective. Our ray tracer is able to render spheres separately from triangle mesh, which requires manual addition to the parsed scene file because in an .obj file, spheres are represented by very smooth, and high poly triangle mesh. Through this design, we are able to boost the speed of our ray tracer in circumstances described above.

3.2. Backward Ray Tracing

The basic idea of backward ray tracing is to fire a ray from the view point (camera) to each pixel and 'trace' this ray, and thus is a pixel-wise implementation. Moreover, we can support two kinds of objects: triangles and spheres. And this implementation refers to the instructions of homework 3 of CSCI420[2].

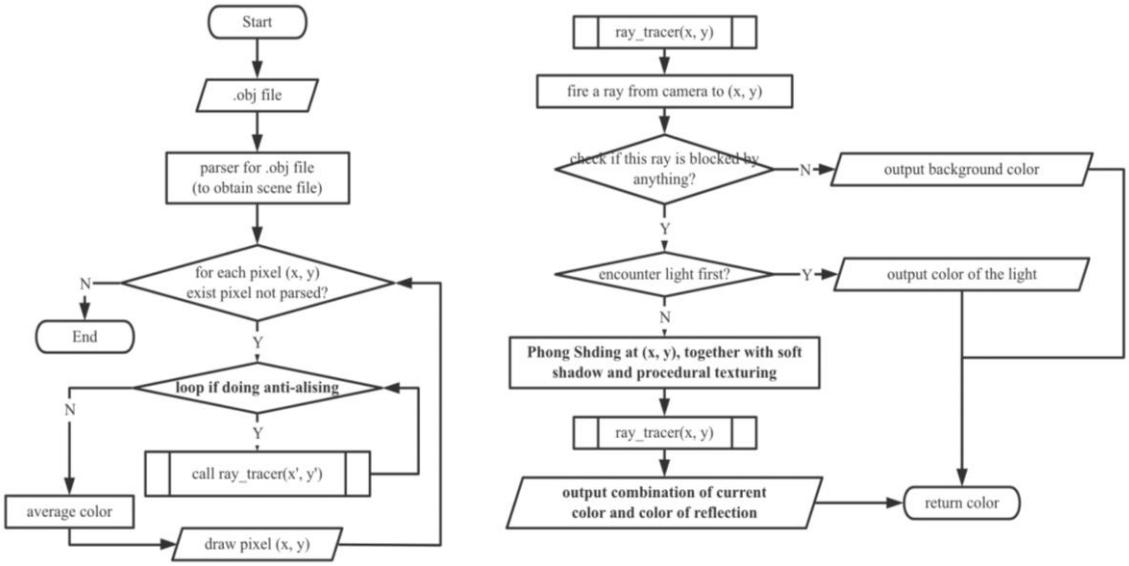


Figure 1 Architecture of Backward Ray Tracing

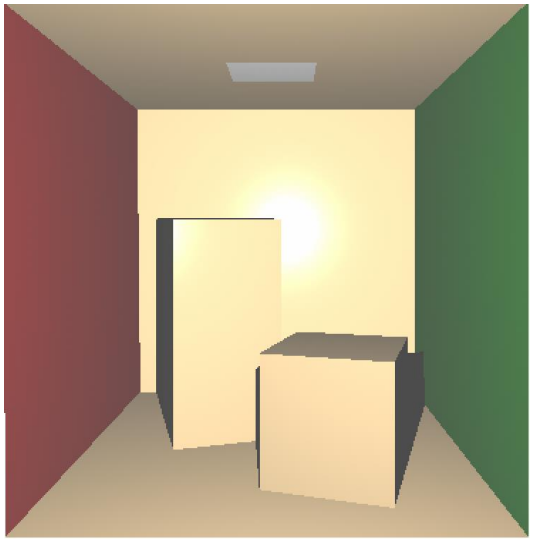


Figure 2 Rendering with basic ray tracer

For each fired ray, we loop all the lights and objects to test if it encounters anything, and there are three possibilities:
(a) If the ray encounters nothing then the color of corresponding pixel is default background color(white).
(b) If the ray first encounters lights, then the color of corresponding pixel is the color of the lights.
(c) If the ray first encounters objects, this case is the most complicated one. After obtaining the intersection point, for each light source we fire a shadow ray. If the shadow ray is blocked by objects, then there is no color contribution from that light. Otherwise, use Phong Shading to obtain the color of that point. In the end, we add up contributions from all lights to obtain the final color at that point.

Figure 2 is the rendering result with basic implementation described above.

3.3. Anti-Aliasing

We implement the anti-aliasing feature by super-sampling: use a parameter called SAMPLE_RATE to control sampling, which is an integer no less than 1. For each pixel, we fire number of SAMPLE_RATE^2 rays, and in the end, average every color obtained to compute the color of that pixel.

This is a really simple but effective method, and Figure 4 is the result comparison for different level of anti-aliasing.

3.4. Recursive Reflection

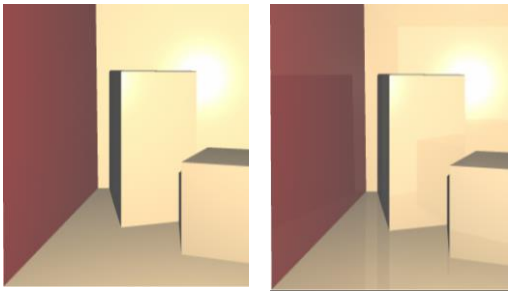
For reflective rays, we have a parameter called reflect_num to control the number of reflections or so called 'bounces'. If the number if 0, there is no reflection. Otherwise, every fired ray that first encounters objects will bounce reflect_num times unless the reflected rays fall in to empty space or first encounter lights, that is, we call our ray tracer recursively. In this way, we treat all surfaces of objects as reflective.

Suppose the vector from reflection point to viewer is $\boldsymbol{l}$, and the normal at that point is $\boldsymbol{n}$. Then use mathematics to compute the reflected ray direction $\boldsymbol{r}$:

$$\boldsymbol{r} = 2(\boldsymbol{l} \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{l}$$

with origin at somewhere a little bit away from the true reflective point, in order to avoid intersection with reflector.

And Figure 3 is the reflection effect comparison:



(a) (b)

Figure 3 Reflective effect:
(a) rendering without reflection
(b) rendering with reflect_num=2

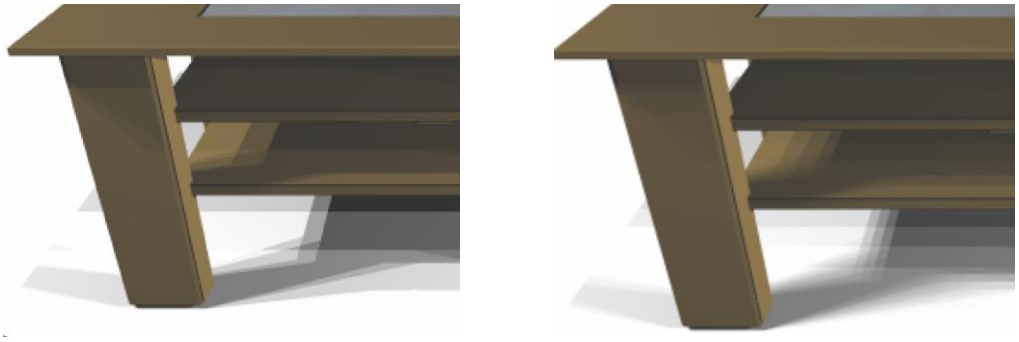
(a) (b) (c)

Figure 4 Anti-Aliasing: (a) rendering without anti-aliasing (SAMPLE_RATE=1) (b) rendering with SAMPLE_RATE=2 (c) rendering with SAMPLE_RATE=4

3.5. Soft Shadow

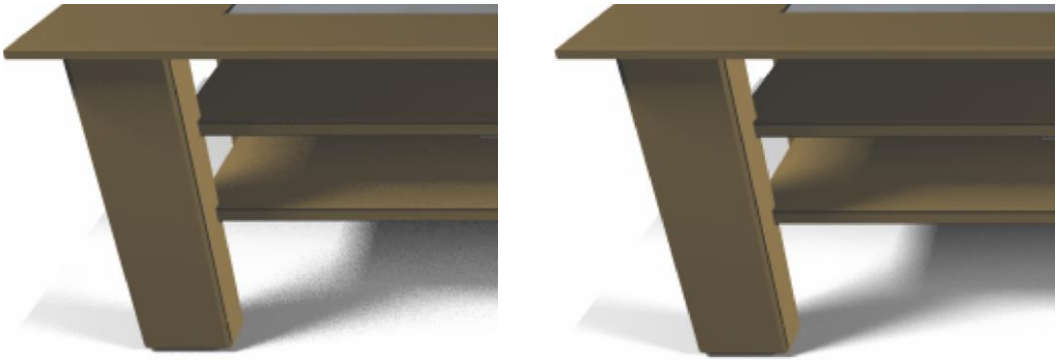
In the real world, light does not exist as a singular point or direction, but rather has a volume or area that illuminates its surroundings. This is what creates the hard shadow (umbra) and the soft shadow (penumbra) depending on the particular size of the light.

This project makes use of the general algorithm described by Jamis Buck to achieve soft shadow in ray tracers[3]. The algorithm describes calculating the intensity of each pixel determined by occlusion from light sources, calculating color from the intensity, and reducing banding via jittering. In order to incorporate this algorithm in our ray tracer, we had to come up with a method that modified our point lights to area lights. The simple solution was to add a virtual area to the point light by defining AREA_LIGHT_WIDTH and AREA_LIGHT_LENGTH, which are used to create a plane on the XZ-axis with the point light as the center. Then, we sent out shadow rays (number defined by LIGHT_CELL_SIZE^2) to each cell in the area light to calculate the average intensity. The intensity factor ranged from $[0.0, 1.0]$, where 1.0 meant no occlusion to any cells, and 0.0 meant occlusion to every cell defined. We then multiply the intensity with the diffuse and specular portions to produce shadows. With higher sample rate, the "softer" the shadow becomes (see Figure 5); however, without random sampling, there is an issue with banding, which produces nonrealistic results. To address this, we need to use jittering, or also known as random sampling. By taking a random sample for each cell, we can reduce banding (see Figure 6), but need higher sampling rates to produce better results.



(a) (b)

Figure 5 Soft Shadow: (a) light_cell_size = 2 (b) light_cell_size = 4



(a) (b)

Figure 6 Soft Shadow with jittering: (a) light_cell_size = 2 (b) light_cell_size = 4

3.6. Procedural Texturing

In this ray tracer, we also implemented procedural texturing[4] to add a little flavor to our scene. The result is showcased in Figure 7 below.

Figure 7 Marble Textured Vase with different color and turbulence

Currently we implemented the texture of marble with the support of the Perlin Noise algorithm[5]. To achieve a more natural marble effect, we incorporated turbulence as a sum of weighted noise (x, y, z). With different weight distribution and levels of the summation, we are able to get different effects as demonstrated above. The formula we referenced is shown below:

$$\begin{aligned} noiseCoef = \ & \sin((x + y \\ & + turbulence \\ & * noiseAmplitude) \\ & * marbleFrequecy) \end{aligned}$$

$$\begin{aligned} Marble = \ & color1 * noiseCoef \\ & + color2 * (1.0 \\ & - noiseCoef) \end{aligned}$$

The bigger the $marbleFrequecy$, the smaller the pattern of such a texture. Figure 8 puts the rendered result of different amplitude and frequency side to side to give a more apparent comparison.

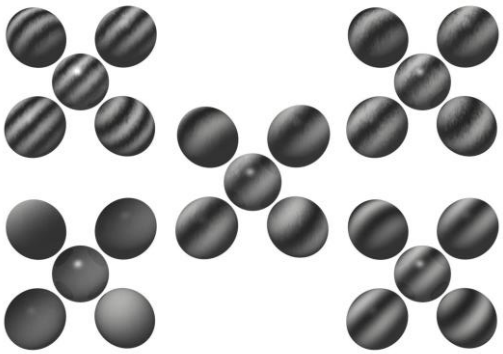


Figure 8 Comparison marble effect: (a) left top f=0.8 (b) left down f =0.1 (c) center a=0.4, f=0.4 (d) right top a=0.4 (e)right down a=0.1

4. Results & Conclusion

In conclusion, we are able to create a ray tracer that implements reflectivity, anti-aliasing, soft shadowing, and procedural texturing. Figure 9 shows the final image. For future work, we plan on adding more procedural texture materials like grass or clouds. Also, we could add transparency and refraction to enhance the result.

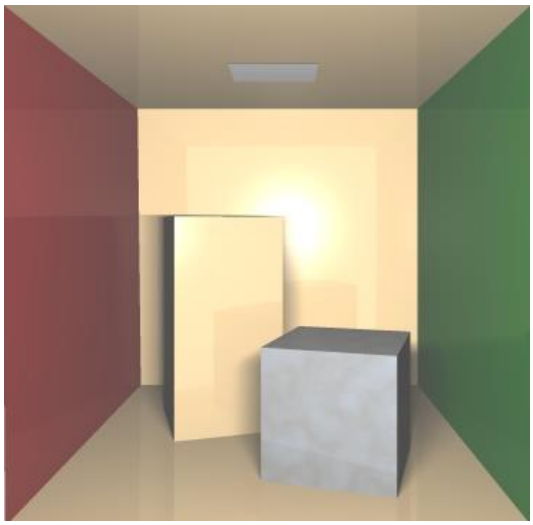


Figure 9 Final Result on Cornell Box

5. References

[1] Github Bly7OBJ-Loader: https://github.com/Bly7/OBJ-Loader

[2] CSCI420 Homework 3 Instruction: https://bytes.usc.edu/cs580/s22_CG-012-Ren/lectures/Lect_RT/CS420_RT_HW.pdf

[3] Buck, Jamis *The Ray Tracer Challenge: A Test-Driven Guide to Your First 3D Renderer*: http://raytracerchallenge.com/bonus/area-light.html

[4] Procedural Texture Generation: http://people.wku.edu/qi.li/teaching/446/cg13_texturing.pdf

[5] Perlin Noise: https://cs.nyu.edu/~perlin/noise/