Ray-tracer

Roger Leite Lucena – roger.lucena@polytechnique.edu
Alexandre Ribeiro João Macedo – alexandre.macedo@polytechnique.edu

1 Introduction

The main goal of this project is to implement a simple Ray-Tracer algorithm in C++ that can calculate shadows and reflections and that is parallelized with MPI. In addition to the standard library and MPI, we also used Boost MPI and Boost Serialization, in order to simplify the parallelization with MPI. The group also used OpenCV to generate the visualization and save the generated image in the JPG format. The figure 1 shows some of the elements used in the algorithm and the general idea of casting rays to see what color is seen by each point.

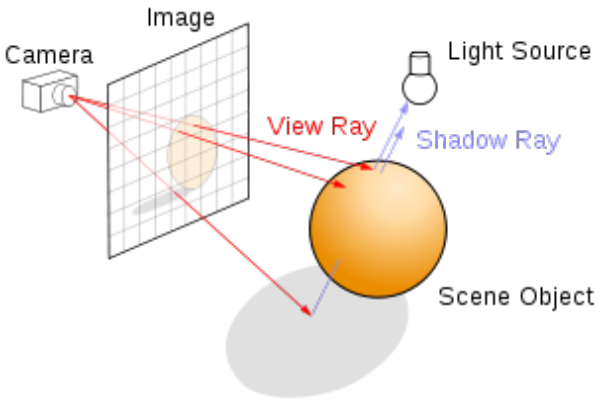


Figure 1: Ray tracer model. Source: https://en.wikipedia.org/wiki/File:Ray_trace_diagram.svg

2 Implementation

2.1 Classes and RtTools

We used 8 classes: vector, ray, spheres, scene, light, camera, color, image. The vector is a simple 3D vector class with the most common operations. The ray is to to represent a light ray, with the origin and the direction. The sphere is to represent the spherical objects and have the center of the sphere, the radius, the color, and the reflection coefficient. The scene is just to encapsulate a set of spheres. The light represents the light source and has its origin and light color. The camera has three vectors, the eye position, the target position, and the orientation of the image (called up), it also has the dimensions of the "panel" where the image shall be created. The color is just to encapsulate the RGB information. The image is where we define the number of horizontal and vertical pixels.

RtTools is the namespace where we defined the functions that actually calculate the image and that generate the visualization. The functions such as calculate the color of one point using the Phong reflection model, explained below, find the first point in the scene crossed by a ray, calculate the color of every pixel in the image, with and without MPI, and the wrappers for the OpenCV functions to visualize the result and save it.

2.2 Phong reflection model

To generate the color of each point we used the model of calculation of Bui Tuong Phong, published as his PhD dissertation at the University of Utah in 1975 and described in [1]. As ilustrated in the figure 3 , the model is a basic empirical one that simulates the local illumination of points on a surface, superposing ambient, diffuse and specular contributions, as shown in the figure 2 bellow:

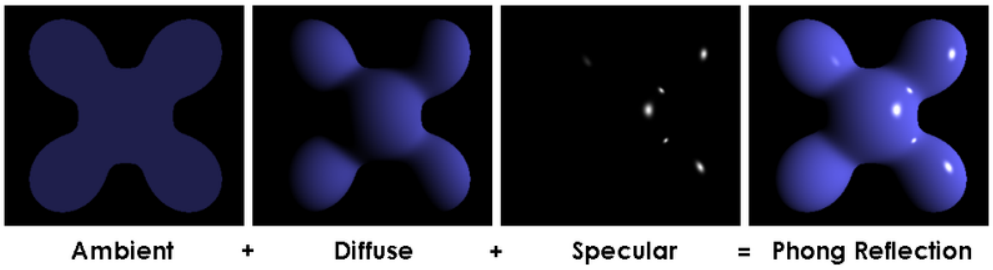


Figure 2: Phong - contributions. Source: `https://en.wikipedia.org/wiki/File:Phong_components_version_4.png`

The model takes also into account:

- The vector pointing towards the light source (L);
- The vector point towards the eye of the viewer (V);
- The Normal of the surface in that point (N);
- The symmetric R of L regarding N (the direction of the reflected ray from the light source).

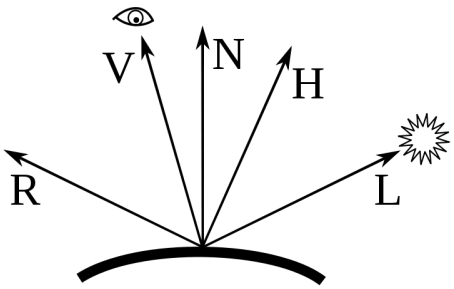


Figure 3: Phong reflection model. Source: `https://en.wikipedia.org/wiki/File:Blinn_Vectors.svg`

The angles between these vectors are included in the calculus of the color when taking their inner products in the following formula.

I_p represents the final intensity of the point's color, i_a, i_d and i_s the contributions of the ambient, the diffuse and the specular intensities of the ray, in this order, k_a, k_d and k_s the constants related to each one of them and α the shininess of the material.

$$I_p = k_a \cdot i_a + \left(k_d \cdot (\hat{L} \cdot \hat{N}) \cdot i_d + k_s \cdot (\hat{R} \cdot \hat{V})^{\alpha} \cdot i_s\right)$$

It is important to remember that the model works also for multiple sources of light, but in the case of this project, just for simplicity, we are dealing only if one source.

2.3 Calculating the image

In order to fill the image, we need to to map the points in "space" to the point in the image. To do that we use the following formulas:

$$\mathbf{increment_{vertical}} = -\mathbf{camera_{up}} * \frac{\text{camera}_{\text{height}}}{\text{image}_{\text{height}}}$$

$$\mathbf{increment_{horizontal}} = -\frac{\mathbf{camera_{up}} \times (\mathbf{camera_{target}} - \mathbf{camera_{eye}})}{|\mathbf{camera_{up}} \times (\mathbf{camera_{target}} - \mathbf{camera_{eye}})|} * \frac{\text{camera}_{\text{width}}}{\text{image}_{\text{width}}}$$

$$\mathbf{initial} = \mathbf{camera_{target}} - [\mathbf{increment_{vertical}} * 0.5 * (\text{image}_{\text{height}} - 0.5)] - \mathbf{increment_{horizontal}} * 0.5 * (\text{image}_{\text{width}} - 0.5)]$$

$$\mathbf{current} = \mathbf{initial} + j * \mathbf{increment_{horizontal}} + i * \mathbf{increment_{vertical}}$$

The color will then be calculated by the intersection of the ray whose origin is at **current** and the direction $\mathbf{camera_{target}} - \mathbf{camera_{eye}}$. The intersection point between the ray and the scene is simply the closet point to the ray origin between the intersection of all spheres and the ray. Once we have the point, we use the Phong model to reflect its color. If we also want shadows, we calculate the intersection point between a ray coming from the light source and whose direction is the intersection point found before. If these points are different, we can conclude that we have a shadow point and we make the color darker by a percentage. In the case we are calculating the reflection, we calculate the point color recursively, and the final color will be a percentage (based on the sphere's reflection coefficient) of the color calculated the way explained above and a percentage based on the reflect ray in the sphere (that we calculate using a simple linear algebra formula). If the reflected ray does not intersect the scene, it will be reflecting the "environment" that we gave the color white. Finally the color of the pixel image$[i][j]$, will be calculated for each i and j.

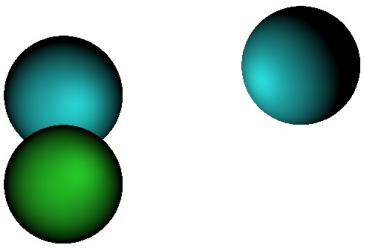
2.4 MPI

For the parallelization, we initialize all objects in all process and, based on the process rank, we assign a set of pixels for it to calculate based on the methods explained above. The only message passed between the process are color for the set of pixels. Since we are using a class to represent colors, we need to create a serialize method in this class and use Boost MPI to transfer this information.

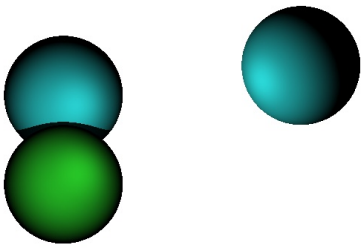
In general lines, each process, based in its rank, is assigned to calculate the colors of a number of pixels of the image and so send them back to the process root (rank 0), that is responsible to put all the pixels together, build and print the final image. Using MPI boost it was possible to the vectors of objects "Color", previously defined in a class "Color", as explained above.

3 Results

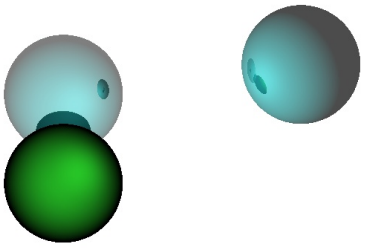
Figure 4 shows the different results obtained by running the algorithm with different parameters. It shows a satisfactory result for every parameter combination. One limitation of our implementation is the absence of a depth effect due to the direction chosen for the rays. Initially, we implemented the direction coming from the eye as shown in figure 1 but it created a distortion in the objects that were not centralized with the eye. Therefore we decided to make the ray direction perpendicular to the "panel" in the camera model as shown below in figure 5 .



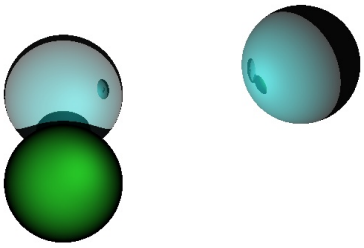
(a) Only objects.



(b) Objects with shadows only.



(c) Objects with reflection only.



(d) Objects with reflection and shadows.

Figure 4: Results

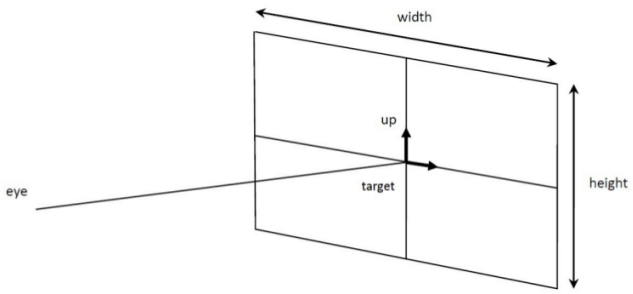


Figure 5: Camera Model. Source: Project guide.

Another aspect worth mentioning is that the MPI implementation is not necessarily faster than the serialized version. This is due to the overhead in the communication between the processes. Therefore is worth using MPI only when the scene is complex and it would take to much time for a single process to render it.

4 Conclusion

With simple algorithms and a not exhaustive implementation it was possible to obtain good and very interesting results in C++. The images were generated with all the effects, such as shadows and reflections, and we even managed to make some simple simulations, like small videos, changing the position of the point of light.

The MPI tool is also very useful and makes the distribution of the calculations direct and intuitive, in a point that this organized and robust framework made the implementation of the last part of the project (parallelization) pretty fast. The success obtained using this tool will be for sure remembered by the students for future projects involving parallelization, and the familiarity with the C++ language and other tools for developing projects, like the Visual Studio Code and GitHub, beside the experience of coding in a group, will also be remembered.

References

[1] Wikipedia - Phong reflection model. Available at
`https://en.wikipedia.org/wiki/Phong_reflection_model`