Project Report
INF442 - Traitement des données massives

Ian Duleba Christopher Aykroyd

2018
École Polytechnique

1 - Description of the Project

The project consists of the implementation of classical Ray Tracing algorithms to capture images by simulating the illumination of objects on a 3D environment. Given a 3D scene containing a camera, a set of punctual lights and objects (spheres) of different materials our program will render a ppm image file as output.

The source code is available at github.com:

- https://github.com/iduleba/Ray_Tracer/tree/mpi
- https://github.com/iduleba/Ray_Tracer

(the first link refers to the branch which supports MPI, while the second is the version coded in a prior phase of development for use on a single processor)

2 - Algorithmic Approach

The problem was divided into smaller subproblems which we could approach separately. Simple, but crucial methods were the starting point of the algorithms, such as detecting an intersection between a ray and a sphere and solving for the points of intersection; finding the surface normal vector given a point in the surface; calculating reflected rays, given the surface and an incoming ray; retrieving the projection of a vector onto another; etc.

As instructed in the project description, we have implemented a version of the Phong reflection model to render the image of the spheres, as well as reflections and shadows. Finally, we distributed our calculations to multiple processes to speed up computational time.

Due to its simplicity, the output of the program was chosen to be a ppm file, which has very simple, but inefficient encoding[1], but nonetheless, was in tune with the scope of the project.

2.1 - Backward Ray Tracing

As the name suggests, *Ray Tracing*, as opposed to *Rasterization* methods, consists of computing the trajectory of rays of light in a 3D scene which leave each source, are reflected on surfaces and finally hit the camera. In such a way, Ray Tracing methods can be more computationally expensive and are not used as frequently as rasterization methods in real-time computer graphic applications[2].

Our implementation in particular made use of backwards ray tracing, meaning that for a camera and a given direction, we compute the inverse path of rays of light which hitting the former to determine its intensity and color. We can in such a way compute a discrete mesh of directions, one by one, in order to determine individual pixels and form an image.

2.2 - Phong model

Objects are made of different materials, which can be either glossy or specular in nature (mirror-like), or diffuse (in which case incident light rays get scattered as they hit the surface). The Phong Reflection Model is an empirical model for computing the reflected light intensities of points on a surface. It may be useful to point out, however, that despite its visual elegancy, the empirical nature of this model does not guarantee the physical conservation of energy[3].

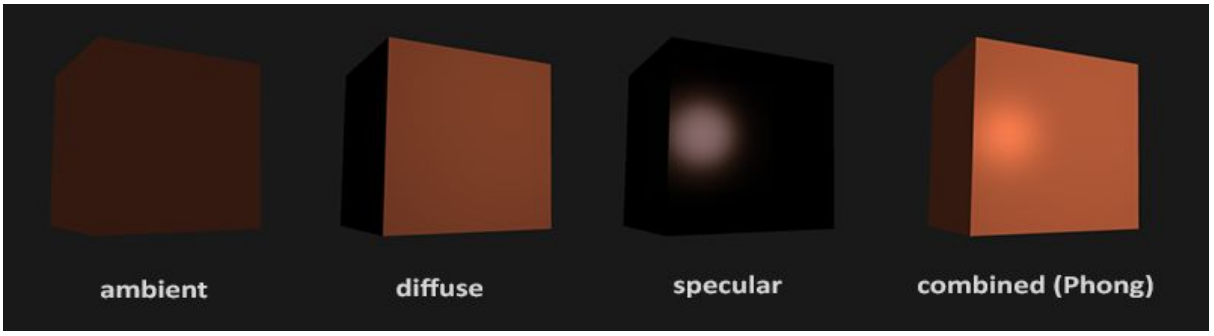

Figure 1 - Wow! Much Phong![4]

The Phong model takes its strength from adding three components to the intensity of the reflected ray of incoming light from a given source, namely:

- an ambient component: light which scattered throughout the scene can be supposed coming equally from all directions;
- a diffuse component: light coming from the source is scattered can in all directions; here, light insiding perpendicularly into a surface will have greatest intensity.
- and specular component: this component is what gives the object a mirror-like appearance, in which the angle of reflection is equal to the incident in relationship to the local normal;

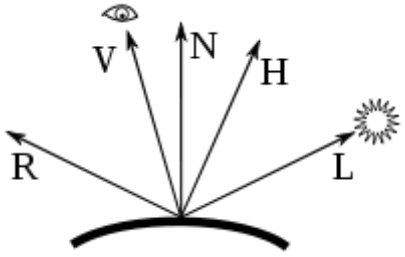

Figure 2 - Vectors for calculating Phong shading[5]

Given a particular surface, made of a material of defined ambient, diffuse and specular constants and a particular color, the total luminosity of a point (for a particular color channel represented in RGB values) can be calculated by adding the luminosity values for each light source in the scene:

$$I_\mathrm{p} = k_\mathrm{a} i_\mathrm{a} + \sum_{m \in \text{ lights}} (k_\mathrm{d} (\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + k_\mathrm{s} (\hat{R}_m \cdot \hat{V})^\alpha i_{m,\mathrm{s}}).$$

The material's color is then compounded with the diffuse and ambient components of the luminosity and finally added to the specular component to obtain the surface point's reflected color.

2.3 - Shadows

The next step in our backwards ray tracing algorithm was to extend the *Phong Model* in order to detect shadows. In *Phong*, whenever calculating the luminosity of a single point, we would always trace the rays hitting such a point, coming from each light source. By adding a collision detection of such rays with objects along its path, we could determine whether light rays coming from such a source would be intercepted before hitting our point, casting therefore a shadow at our object by not contributing to the diffuse or specular intensities of its luminosity.

2.4 - Indirect Illumination: Specular Reflections

Reflection can be either specular or diffuse, as explained above. While the Phong model is extremely useful to calculate reflection of rays originating directly from scene light sources, it does not consider the light bounced by nearby objects. Such a process of lighting is called indirect illumination.

Whereas diffuse indirect illumination is generally calculated by a Monte-Carlo integration method which computes numerous rays reaching the surface from random directions and recursively searching whether they hit another surface, this will lead to exponentially expensive calculations. Our chosen path was to determine the specular reflections such as those displayed by mirror-like surfaces, by calculating rays of perfectly reflected surfaces (as in the law of reflection) and recursively tracing its origin as in the pseudo-algorithm below:

```
Color Reflection(Sphere sphere, Vector point, Scene scene, Vector reflected_ray, float r){

  Color cs = PhongReflection(sphere, point, scene);

  // End condition: the composite reflectivity is small the bounced light will be irrelevant
  if r < Epsilon
    return (cs);

  // Calculate normal vector at the surface point, and the incoming ray
  Vector normal = sphere.Normal(point);
  // the projection onto the vector normal to the sphere
  Vector proj = normal * DotProduct(normal, reflected_ray);
  // direction vectors representing the reflected ray
  Vector incoming_ray = (proj * 2 - reflected_ray;).Normalized();

  // Calculation of the closest sphere intersected by the ray
  foreach (object in scene)
    if reflected_ray.Intersects(object)
      let Obj be the closest such object,
      let p be the point of such intersection;

  if (Obj != null)
    Color cr = Reflection(Obj, p, scene, -incoming_ray, r * k);
    return r * cr + (1-r) * cs;

  else
    return cs;

}
```

Figure 3 - Wow! Much pseudocode!

2.5 - MPI

The adopted strategy to parallelize the code was to break the backwards ray tracing algorithm into chunks: each processor handles a block of the image. Insofar as the pixels of each block are calculated independently from the other blocks, the problem is parallelizable in this manner.

Some of the required changes and main ideas to this effect were: every process reads the input file but it will only render a portion of the pixels of the final image, sending the result of its computation to the root process (first process) which, in turn, will combine all pixels computed into a single image and then export it to the ppm file.

3 - Overview of the classes and encoding

The basic classes of the problem were created, namely, Vector, Color, Ray, Light, Image, Spheres, Scene and Camera. We have only used std::vector as data structure in our program.

- **Vectors**: This is the chosen representation of a 3D vector: three float numbers specifying its components. Usual vectorial operations were defined: addition, cross and dot product, multiplication by a float.

- **Ray:** Defines a ray of light. It has a vector representing its position and another indicating its direction.

- **Color:** A color is defined by its RGB values. It was used to represent pixels and light colors.

- **Spheres**: A sphere contains: a color; a float defining its radius; the position of its center; five floats whose values define how each component of light it will interact with its surface.

- **Scene:** Represents the setup of the scenario. It contains all spheres and light sources in std::vectors.

- **Light:** Source of light. A vector characterizes its position and three colors identify its components.

- **Camera:** Contains a vector representing the "eye", a vector representing the target (looked by the camera), a "up vector" representing the orientation of the camera, and the camera resolution. It also implements the algorithms described in the previous section.

- **Image:** Picture taken by the camera. It is a grid of colors (pixels).

4 - Usage

Firstly, one needs to compile the source code. That task was simplified with the aid of a Makefile: inside the projects folder, type

make

To run the binary, specific inputs are expected:

->for the mpi version

mpirun -np 3 rt **<camera_file> <spheres_and_sources_file> ./output_img.ppm**

->for the single core version

./*rt* **<camera_file> <spheres_and_sources_file> ./output_img.ppm**

where <camera_file> and <spheres_and_sources_file> must have a specific format that is assumed by the program to be correct:

A) <camera_file> format:

width(int) height(int) -> resolution of the camera
x(float) y(float) z(float) -> position of the camera
x(float) y(float) z(float) -> position of the target
x(float) y(float) z(float) -> upward direction

B) <spheres_and_sources_file> format:

Red(0-255) Green(0-255) Blue(0-255) -> Background color
number_of_spheres(int) number_ of_sources(int)

For each sphere, four lines will be read:
radius(float)
ka(float) kd(float) ks(float) sh(float) refl(float) -> reflection parameters
Red(0-255) Green(0-255) Blue(0-255) -> color of the sphere
x(float) y(float) z(float) -> coordinates of the center

Then, for each source, four lines will be read:
Red(0-255) Green(0-255) Blue(0-255) -> ambient component of the light
Red(0-255) Green(0-255) Blue(0-255) -> diffuse component of the light
Red(0-255) Green(0-255) Blue(0-255) -> specular component of the light
x(float) y(float) z(float) -> position of the source

5 - Results

The source code in our git has two input files that were used to generate the images (800x600 pixels) and data shown below. The second image has more spheres and consequently, takes more time to render.

5.1 - Renderings

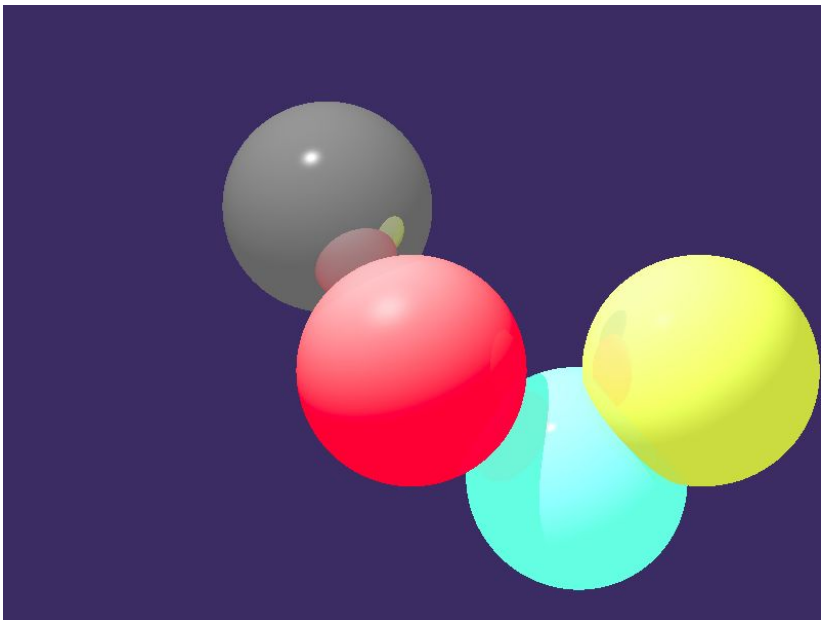

Figure 4 - Wow! Much spheres!

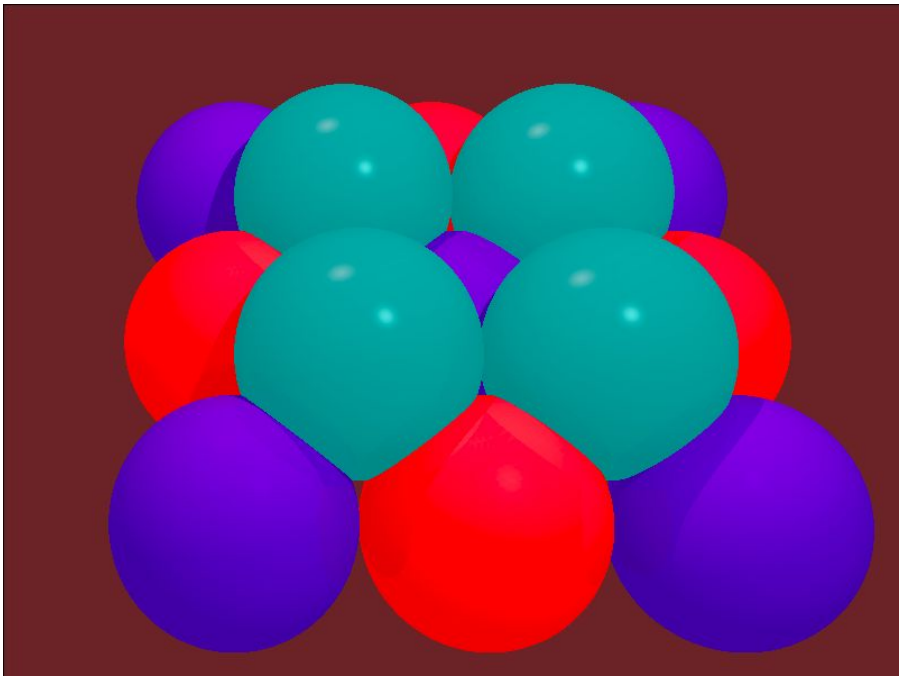

Figure 5 - Wow! Much spheres!

5.2 - Scalability

MPI on a single machine:

Running our code on a laptop (Intel Core i5 6th Generation with four cores), we observe an improvement of performance with multiple cores, but it flattens out with 2:

# of cores	average time (s) - Figure 4	average time (s) - Figure 5
1	2,073	9.123
2	1,482	5.095

3	1,391	4.275
4	1,577	4.762

Note: there is virtually no difference between the single core version of the code and the MPI one running on a single core.

Salloc (salle d'info):

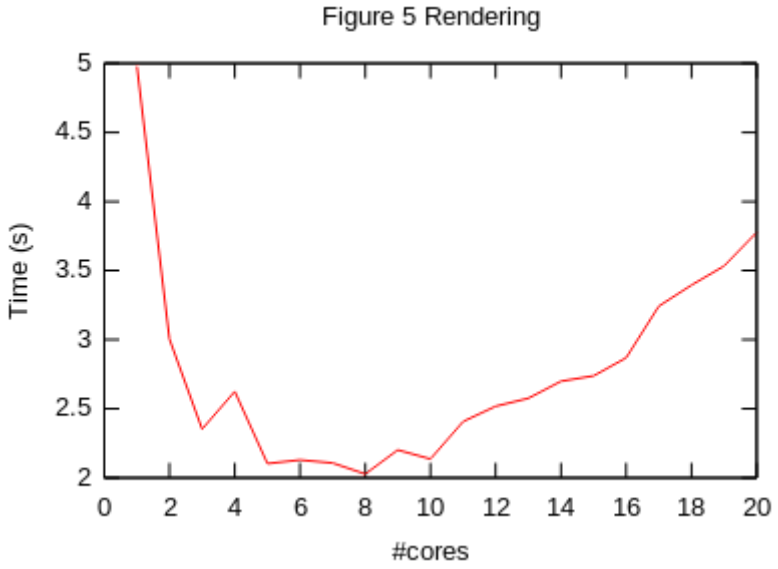

Figure 6 - salloc salle d`informatique rendering

In both cases the curve flattens out and then it will actually worsen with the increase of the number of cores. The gain from the parallel computation is lost in the time that it takes to retrieve/send data across processes. Other strategies, such as communicating the results to more than one process (root) yields similar results.

6 - Issues and Further Development

The development of this project was done in Atom[6], a free and open-source text and source code editor with embedded Git Control, developed by GitHub. We could not recommend it enough.

As expected, MPI is not very friendly toward debugging, a difficulty that required a few hours to cope with. Another nuisance during the adaptation to a parallelized version was that MPI only deals with its basic types, so, quite a few things had to be rewritten.

Some improvements can be further made both to optimize calculation speed as well as to render images more visually pleasing or add extra functionality:

- Add decay to light intensity: the further away light sources are from an object, the weaker they become
- Textures materials: using images skinned to our scene objects would permit the use of different colors for each point in the surface, allowing the rendering of more visually pleasing types of materials.

- Triangle objects: Adapting the ray tracing algorithms to triangular planar surfaces could eventually permit the construction of meshes, and thus enable the rendering of complex volumes in our scenes.
- Acceleration structures: certain data structures, such as the Bounding Volume Hierarchy[7], could greatly boost the velocity of ray-object intersection calculation, which is currently linear in the number of objects in the scene. The adding of meshes explained above would greatly slow computation time and therefore require different optimizations to run smoothly.

7 - References

[1] netpbm.sourceforge.net/doc/ppm.html

[2] https://blogs.nvidia.com/blog/2018/03/19/whats-difference-between-ray-tracing-rasterization/

[3] http://www.rorydriscoll.com/2009/01/25/energy-conservation-in-games/

[4] https://learnopengl.com/Lighting/Basic-Lighting

[5] https://en.wikipedia.org/wiki/Phong_reflection_model

[6] https://atom.io/

[7] http://www.scratchapixel.com/lessons/advanced-rendering/introduction-acceleration-structure/bounding-volume-hierarchy-BVH-part2