Universidad del País Vasco
Euskal Herriko Unibertsitatea

ZIENTZIA ETA TEKNOLOGIA FAKULTATEA
FACULTAD DE CIENCIA Y TECNOLOGÍA

Gradu Amaierako Lana
Ingeniaritza Elektronikoko Gradua

Implementation and performance analysis of several versions of a Realistic Ray Tracer (Python, CUDA and C++)

Egilea:
Kerman Alonso Ajuria
Zuzendariak:
Iñigo Arredondo Lopez de Guereñu
Jon Sanchez Herrasti

© 2020, Kerman Alonso Ajuria

Leioa, 2020ko irailaren 7a

Contents

1 Introduction **1**
- 1.1 A bit of history . . . 1
- 1.2 Objectives . . . 2

2 Development of the ray tracer **3**
- 2.1 Choosing the development tools . . . 3
  - 2.1.1 Programming language . . . 3
  - 2.1.2 Version control . . . 4
  - 2.1.3 Operating system and hardware . . . 4
- 2.2 Ray casting . . . 5
- 2.3 Scene objects and lights . . . 6
  - 2.3.1 An object-ray intersection example: spheres . . . 7
  - 2.3.2 The general `intersection` algorithm . . . 8
- 2.4 Local illumination . . . 8
  - 2.4.1 Diffuse reflection . . . 9
  - 2.4.2 Shadows . . . 9
  - 2.4.3 Specular reflection . . . 10
  - 2.4.4 Ambient light . . . 12
- 2.5 Color . . . 12
- 2.6 Accelerating Python . . . 14
  - 2.6.1 A small survey of the available tools . . . 14
  - 2.6.2 Numba . . . 15
- 2.7 Global illumination . . . 18
  - 2.7.1 Reflection . . . 19
  - 2.7.2 Transmission . . . 20
  - 2.7.3 Attenuation . . . 22
  - 2.7.4 The `ray_trace` algorithm . . . 22

3 Enhancements **24**
- 3.1 Anti-aliasing . . . 24
  - 3.1.1 Uniform supersampling and jittered supersampling . . . 24
  - 3.1.2 Adaptive supersampling . . . 25
- 3.2 Further development on scene objects . . . 25
  - 3.2.1 Triangle meshes . . . 26
  - 3.2.2 Boolean operations with primitives . . . 27
- 3.3 Steps towards photorealism . . . 29
  - 3.3.1 Depth of field . . . 29

3.3.2 Soft shadows 30
3.3.3 Glossy reflections 31

4 Performance comparison **33**
4.1 Preparation 33
4.1.1 Features and variables 33
4.1.2 The designed scene 34
4.2 Measurements 34
4.2.1 Time 35
4.2.2 CPU and memory utilization 37
4.2.3 GPU utilization 39

5 Conclusions **40**

Bibliography **42**

1 Introduction

With the release of Nvidia's GeForce RTX GPU series, the computer image rendering technique known as ray tracing has caught the attention of the public. Ray tracing simulates the propagation of light rays in 3D scenes, and can generate or render highly realistic images. The main drawback is that it is computationally intensive, which has limited its applicability in the past. However, hardware technology has greatly advanced since the conception of the technique, and Nvidia's RTX technology has made possible to render ray traced images at a fast enough rate to be applied in video games. This is called real time ray tracing and it means that the whole ray tracing rendering process is completed between 30 and 60 times per second.

1.1 A bit of history

Ray tracing computer models were first developed in the 1960s, based on the principles of classical geometrical optics. Much like physics undergraduate students calculating the path a light ray follows after being refracted by a lens, ray tracers apply these principles to trace the path of rays through virtual 3D scenes. Considering its basis, it should not come as a surprise that, in addition to being used to produce realistic images in the field of computer graphics, ray tracing has applications in the fields of optical design and thermal radiation analysis [1, 2]. However, this work lies in the field of computer graphics, which emphasizes on the quality of images and the speed of the rendering process, rather than on the accurate representation of physical processes.

Most of the basic techniques that ray tracers employ were developed in the 1970s. These include direct illumination models that describe how rays interact with the surfaces in the 3D scene to produce color, shading models that render shadows, and global illumination models that simulate reflection and refraction. Despite the considerable realism of the rendered images, they required too much time to be rendered due to the limitations of the hardware at the time. The 1980s saw a rise in ray tracing research that mainly focused on accelerating techniques. More efficient algorithms were developed, as well as hardware acceleration techniques based on parallel computing. However, ray tracing continued to be too slow, and during the 1990s other image rendering techniques were favored.

As hardware acceleration improved, many image rendering software began to include ray tracing techniques to compute some illumination features. In computer generated movies ray tracing slowly but steadily substituted other techniques, becoming the preferred movie rendering technique [3].

The major advancement in hardware acceleration in recent years has been the afford mentioned RTX technology that was released in Nvidia's Turing architecture GPUs in 2018. This has renewed the interest in hardware acceleration research, and has made real time ray tracing accessible to game developers and the general public [4, 5].

1.2 Objectives

Modern ray tracers are very complex algorithms that model a wide range of optical phenomena and apply several acceleration techniques. It would be unrealistic to attempt to implement such an algorithm in this work, so a more basic model has been developed, more akin to the ray tracers of the 1970s.

The basic ray tracer has the capability of rendering simple geometric shapes, and it can reproduce the reflection and transmission of light rays. Its characteristics and limitations are explained throughout chapter 2. Then, some of this limitations are overcome by the enhancing techniques presented in chapter 3.

On the other hand, the basic ray tracer model has been used to explore some of the acceleration tools available for the Python language, the language used to develop the model. The ray tracer has also been compared to a similar model implemented in the C++ language, provided by the DigiPen Institute of Technology. The procedure followed to do the comparison and the results can be found in chapter 4.

2 Development of the ray tracer

Ray tracers are a type of image *renderer* or *synthesizer*; they take a 3D *scene* as an input and produce a 2D image. So, in essence, ray tracers fulfill the same function as a real camera, and can be understood as a model of one. In principle, one could design a ray tracer that models a complex camera with all its complex optics, however, the usual approach is to model a *pinhole camera*.

Section 1.2 of chapter 1 in [1] explains how pinhole cameras work and how ray tracers implement them. A pinhole camera consists on a box with a small hole in the middle of one of its faces and a photographic film that covers the inside side of the opposite face. The pinhole lets through a small number of rays into the box, so that each point in the photographic film receives only the rays coming along the direction of the line joining that point and the hole. By eliminating all the other rays, pinhole cameras get a 2D image of the 3D scene outside the box. Ray tracers also render the scene by projecting a ray that goes through a *eye point* (that functions as the pinhole) into each point of the *screen* (that is analogous to the photographic film). However, the screen in a ray tracer is placed between the eye point and the scene, as oppose to behind the eye point like in the pinhole camera. If the film was placed outside the box, the pinhole camera wouldn't work, but it does work for the ray tracing computer model; in fact, it simplifies the model without any compromise.

There are multiple ways to implement a ray tracers depending on what kind of scenes and optical phenomena one wants to render. That said, there are some features that all ray tracers share, and that are explained across many introductory level ray tracing books. In this case, the the basic ray tracer was mainly developed using [6], but as stated, many other references like [1] or [7] could be used instead.

2.1 Choosing the development tools

Before starting to code the ray tracer, the development tools to be used were explored. This section presents the tools that were selected and the reasoning behind the choices that were made.

2.1.1 Programming language

A quick look into ray tracing textbooks is enough to see that ray tracers are commonly written in C++ (or C) [1, 8, 7, 9]. Its high performance and memory management features make C++ one of the most common language used in image rendering.

Before beginning the project, the student had taken courses that taught the basics of Python, Java and Fortran. Learning C++ was considered at the beginning, as it would allow the student not only to develop a well performing ray tracer, but to create a both a C++ and a Python version to analyze performance differences. Nonetheless, the idea was discarded due to time constraints. Speed of development was an important factor, so Python 3 was chosen, due to familiarity and its inherent beginner friendly features (interpreter, dynamic typing...). In addition, NumPy was used to ease array creation, but the necessary vector and matrix operations were programmed from scratch.

The implementation of a ray tracer is challenging in itself, but to make the project more complete, it was decided to compare the Python version with a C++ one. With this in mind, Jon Sanchez from Digipen Institute of Technology became a co-director, and provided a compiled C++ ray tracer of similar characteristics.

2.1.2 Version control

The tools chosen to keep track of the project versions were Git and GitLab, which also allowed to store the project in a remote repository and keep the team up to date on its development. Learning and applying the basics the version control software was enough to drastically improve the development process, and Git became an invaluable tool to create and manage the various versions of the code.

This was specially true in the process of measuring and comparing the performance of the ray tracers implemented using different coding languages and tools (chapter 4). Each version was contained in a different branch, and switching between versions was reduced to just typing a Git command. While working on a branch locally, GitLab was used to view the code of any branch stored in the remote repository. Thus, making small corrections across the ray tracer implementations became simpler with the usage of these tools.

2.1.3 Operating system and hardware

The capabilities of the student's personal computer were determined to be sufficient to use it as the sole development platform. The relevant specifications of the computer are listed in table 2.1, for use as a reference in the performance measurements of chapter 4.

Table 2.1: *Computer specifications.*

OS	Microsoft Windows 10 Home
RAM	8.0 GB DDR3
External memory	TOSHIBA MQ02ABD100H
CPU	Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, 2592 Mhz
GPU	NVIDIA GeForce GTX 950M

2.2 Ray casting

The fist step to implement the ray tracer was to parameterize the screen and the eye point, that together form what from now on will be referred to as the *camera*. Chapter 2 of [6] presents one way to achieve this by first establishing some conventions, and then defining the necessary parameters.

The conventions are that all vectors are defined in a right-handed coordinate system, and that even though the units of length are arbitrary, the magnitude of the vectors that define the *scene objects* (section 2.3) are generally close to unity.

On the other hand, the camera is defined by three vectors ($\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{e}$) and a point (C). Point C is the center of the screen which has a rectangle shape. Horizontal vector $\boldsymbol{u}$ and vertical vector $\boldsymbol{v}$ are orthogonal vectors that define both the orientation of the screen (the direction of its the horizontal and vertical symmetry axes), and its dimensions ($2|\boldsymbol{u}|$ wide and $2|\boldsymbol{v}|$ tall). The addition of vector $\boldsymbol{e}$ and point C gives the position of the eye point (P_e). It is important to note that $\boldsymbol{e}$ must be orthogonal to both $\boldsymbol{u}$ and $\boldsymbol{v}$ in order to produce images with correct perspective. Figure 2.1 shows the components of a camera.

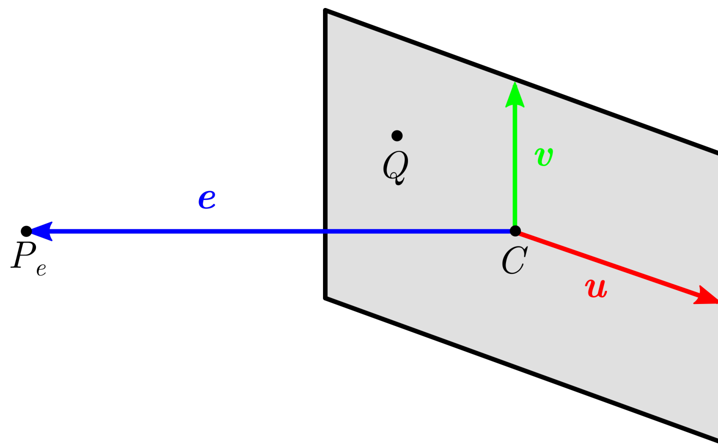


Figure 2.1: *Components of a camera ($\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{e}$, C) and a point on the screen (Q).*

As stated in section 1.1 ray tracing is based on tracing the paths of light rays in a scene. In the model, light rays are defined by some spatial information, and the corresponding color is stored separately (sections 2.4 and 2.5). For now, let's focus on the light rays that go through both the screen and the eye point, which are known as *eye rays*; these are the rays that ultimately contribute to the rendering of the 2D image. The spatial information such rays is expressed as line segments (ρ) that have one end at point P_e, and go through a point Q in the screen into the scene:

$$\rho(t) = \{P_e + t(Q - P_e) \mid 0 \leq t \leq \infty\} \tag{2.1}$$

Point Q can be seen on figure 2.1 and can be written as follows:

$$Q = \{C + \alpha\boldsymbol{u} + \beta\boldsymbol{v} \mid -1 \leq \alpha \leq 1,\ -1 \leq \beta \leq 1\} \tag{2.2}$$

Now that the camera and light rays have been defined, there remain two key ideas to complete the understanding of how a ray tracer renders an image.

The first one illustrates an important difference between the source of light rays in reality versus their source in a ray tracer. In reality, light rays originate in light sources (that will be referred to as *lights*) and travel through the scene before reaching the pinhole camera. One can easily realize that most light rays do not reach the pinhole, and do not therefore contribute to the image formed in the photographic film. Ray tracers take the inverse approach, originating light rays in the eye point and casting them into the scene. The rays travel through the scene until they reach a light. This method greatly reduces the number of created light rays, limiting it to the ones that contribute to the final image. This idea is explained in further detail in sections 2.1 and 2.2 of chapter 1 in [1].

The final idea is that the virtual screen, the analog to the photographic film in the pinhole camera, is directly related to the rendered image in the following way. The screen is subdivided into rectangular areas depending on the size of the image (the number of horizontal and vertical pixels). Each rectangular area corresponds to one of the pixels of the image, and all the light rays that go through this area contribute to the final color of the pixel. This is shown graphically on figure 2.2.

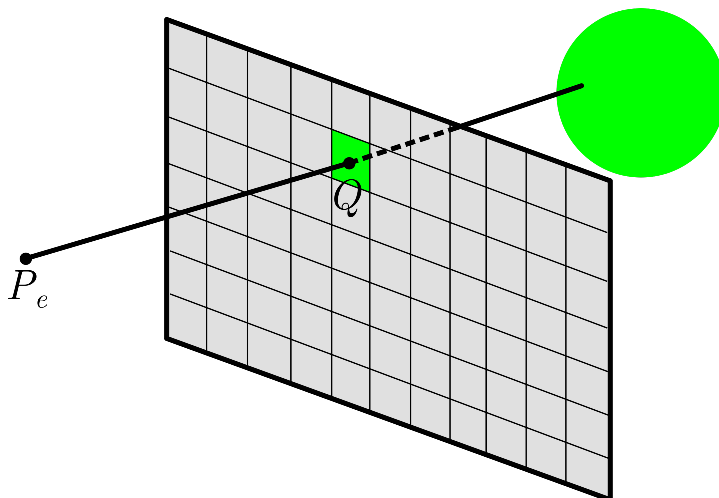


Figure 2.2: *A light ray that originates in the eye point P_e, goes through a screen area that corresponds to a pixel and intersects a sphere in the sceene. The green color of the intersected object is displaied on the corresponding pixel.*

2.3 Scene objects and lights

The 3D scenes that the ray tracer renders are constituted by light sources and objects. Ray tracers often implement several kinds of light sources, that can have different shapes and characteristics [10]. Nonetheless, the implemented ray tracer only includes *point lights*, which are defined by a color and a point in space. Scene objects are 2D or 3D

geometric shapes that have *surface properties* that define how light rays interact with them.

This section will focus on explaining how the intersection of geometric shapes and light rays is computed. The surface properties of objects will be mentioned in sections 2.4 and 2.7, along with the explanations of what each represents and how is used in the model.

An algorithm that solves the object-ray intersection problem first determines if the object and the ray intersect. If they do, it outputs the point of the intersection (P), the normal vector to the surface of the object at that point ($\boldsymbol{n}$) and the distance between the origin of the ray and P (t_P). An object and a ray can intersect at multiple points, but for ray tracing purposes only the point closest to the origin of the ray is needed. The reason is that once a light ray reaches an object surface, its direction and color change; which means that it disappears, and that new reflected or refracted rays are created in its place (section 2.7). $\boldsymbol{n}$ is always chosen to point outside the objects if they are 3D, to make shading easier (subsection 2.4.2). In the case of 2D objects, the normal always points to the region in space where the incident ray comes from.

2.3.1 An object-ray intersection example: spheres

Chapter 3 in [6] describes how to compute object-ray intersections for spheres, boxes (quadrilaterally-faced hexahedron), polygons and ellipsoids; all of which have been implemented in the model. Although each object has each own nuances, studying one of them is sufficient to get a good idea of how P, $\boldsymbol{n}$ and t_P are computed. Since the sphere is the easiest to implement, it is the best candidate to illustrate the procedure.

The first step is to parameterize the spatial information of the object. The sphere is defined by giving the position of its center (C) and its radius (r). Every point P in its surface is defined by this equation:

$$(P - C) \cdot (P - C) = r^2 \tag{2.3}$$

The light ray must also be parameterized, by determining the point of origin of the ray (P_0) and the direction vector ($\boldsymbol{i}$), and using them to define the line that constitutes the spatial information of the ray.

$$\rho(t) = P_0 + t\boldsymbol{i} \tag{2.4}$$

Then, equations (2.3) and (2.4) are used to create a system and get the t value at which the sphere and the ray intersect. This leads to the following second order equation:

$$at^2 + bt + c = 0, \text{ where } \begin{cases} a = |\boldsymbol{i}|^2 \\ b = 2[(P_0 - C) \cdot \boldsymbol{i}] \\ c = |P_0 - C|^2 - r^2 \end{cases} \tag{2.5}$$

Finally, the various kinds of solutions equation (2.5) have to be interpreted to give the correct output of the intersection. The following pseudocode shows the logic behind this analysis:

if $b^2 - 4ac < 0$:
    no intersection
else:
    $t_{\pm} = (-b \pm \sqrt{b^2 - 4ac})/(2a)$
    if $t_+ < 0$:
        no intersection
    elif $t_- < 0$:
        $P = P_0 + t_+\boldsymbol{i}, \quad \boldsymbol{n} = P - C, \quad t_P = t_+$
    else:
        $P = P_0 + t_-\boldsymbol{i}, \quad \boldsymbol{n} = P - C, \quad t_P = t_-$

The basic idea, that holds true for every object-ray intersection, is that the algorithm must find the intersection with the smallest non-negative t value.

2.3.2 The general `intersection` algorithm

Scenes usually contain more than one object, so apart from the functions that compute each kind of object-ray intersection, the model has a general `intersection` function to manage multiple objects. The function is invoked for every light ray, and outputs the P, $\boldsymbol{n}$ and t_P values of the closest object-ray intersection to the origin of the ray. To achieve this, the algorithm simply loops through all the objects of the scene calling the appropriate object-ray intersection function in each case. If the t_P value for the current object-ray intersection is smaller than the previous one, the P, $\boldsymbol{n}$ and t_P values are updated.

2.4 Local illumination

Section 2.2 explained how the camera renders a 2D image from light rays, and section 2.3 showed how scene object and light ray intersections are computed. This section completes the explanation of the basic model by illustrating how the color of the light rays is computed.

Chapter 4 in [6] begins by explaining that a *local illumination model* is a model that computes the interaction between *first order light rays* and scene objects. First order light rays are the ones that, after being casted from the camera, intersect an object,

reflect off of it and reach a light source without interacting with any other object in the process. Therefore, a ray tracer that implements a local illumination model renders the scene objects (or rather object surface points) that are directly visible for the camera and are directly lit by at least one light.

The local illumination model attempts to emulate lights behavior by implementing two kinds of reflection *(diffuse* and *specular*), shadows and ambient light. But, before going into the explanation of these features, it is necessary to introduce the concepts of *radiant flux* (ϕ) and *radiance* (ρ). As stated in [7], radiant flux is the amount of electromagnetic energy that passes through a surface per second. Radiance measures the radiant flux at a specific point coming from a specific direction, and is defined as the radiant flux per unit of projected area (perpendicular to the direction) per unit of solid angle. Its association to a specific direction makes radiance very well suited to quantify the color of light rays.

2.4.1 Diffuse reflection

Diffuse reflection, also known as matte reflection [8], models the reflection phenomena that gives color to most objects in reality: the absorption of photons by surfaces and their subsequent re-emission in a different spectra [1]. The model implements this photon-surface interaction by multiplying the radiance of the incident ray at a specific wavelength λ ($\rho_i(\lambda)$) with a *diffuse reflection coefficient* (k_d), which is a property of the surface.

The re-emitted photon can leave the surface in any direction, and it can be assumed that the probability of all the possible directions is equal. Therefore, with respect to geometry, the amount of reflected light at any given direction will only depend on the relative orientation of the incident light ray ($\boldsymbol{i}$) and the normal vector to the object surface ($\boldsymbol{n}$). The model takes this into account introducing a $|\boldsymbol{i} \cdot \boldsymbol{n}|$ term. Figure 2.4a presents a graphical representation of diffuse reflection, showing that the reflected ray can have any direction.

Taking the photon-surface interaction and the geometry term into account, one can get the formula for the radiance of the diffusely reflected ray ($\rho_d(\lambda)$) [1, 6]:

$$\rho_d(\lambda) = k_d|\boldsymbol{n} \cdot \boldsymbol{i}|\rho_i(\lambda) \tag{2.6}$$

2.4.2 Shadows

Shadows were implemented following the method described in section 4.2 of [6]. The model is based on the idea that a point P in the surface of a scene object is directly lit by a light only if the line segment linking P and the light does not intersect any object in the scene. This line segment is appropriately called a *shadow feeler*, and is implemented using this formula:

$$\rho(t) = \{P + t\boldsymbol{l} \mid 0 \leq t \leq D\} \tag{2.7}$$

where D is the distance from P to the light, and $\boldsymbol{l}$ is a normalized vector that points to the light if placed at point P.

There is a important note to make about the implementation of the shadow feeler in the model. If a light ray hits a surface from outside, there is a possibility of spawning the shadow feeler inside the object, due to finite precision in floating point representation. To avoid this, a normal outward vector with a small magnitude (ϵ) is added to point P before using it to calculate the shadow feeler. This addition ensures that the shadow feeler is always spawned outside the object that contains P. Figure 2.3 shows where shadow rays are spawned and their orientation.

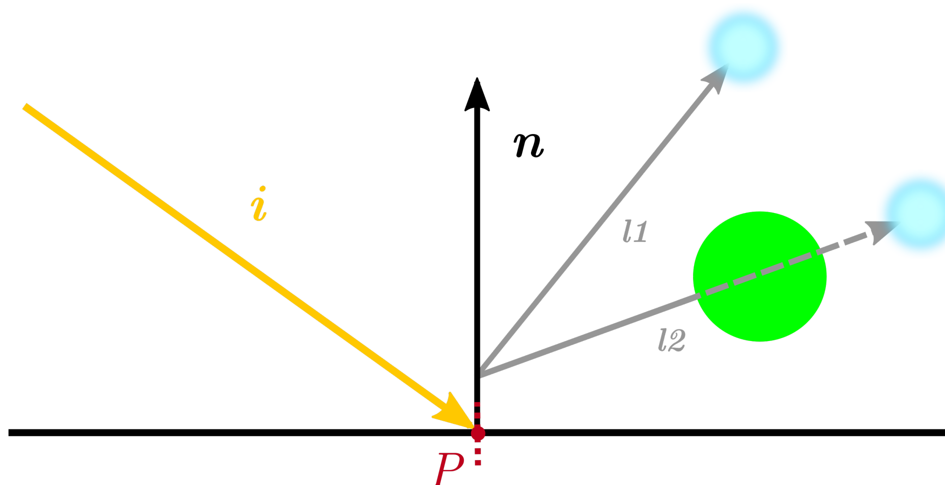


Figure 2.3: *Two shadow rays spawned from a point slightly above P, represented by their direction vectors $\boldsymbol{l_1}$ and $\boldsymbol{l_2}$. The first shadow ray reaches the light, but the second intersects a scene object. The yellow vector represents the direction of the incident ray that has intersected the surface at point P.*

Each time the intersection point P between a light ray casted from the camera and the nearest object is computed, shadow feelers are spawned to find out which lights directly illuminate P. Then, local diffuse and specular reflections are computed taking into account only those lights.

This simple shadow model has a big limitation that will become more apparent at subsection 2.7.2: transparent objects cast a shadow as dark as opaque objects do.

2.4.3 Specular reflection

The implementation of specular reflection in a ray tracer aims to reproduce the reflection phenomena that produces bright highlights in polished surfaces, like the ones seen in cars on a sunny day. This kind of reflection was implemented in the program by using the *Phong reflection model*, described in subsection 4.3.2 of [6].

Specular reflection is related to the reflection law of light waves. According to this law, the incident ray, the reflected ray and the normal vector ($\boldsymbol{n}$) to the surface lie in the

same plane, and the angle of incidence and the angle of reflection are the same (but the incident ray and the reflected ray are at opposite sides of the normal). The expression of the direction vector of the reflected ray ($\boldsymbol{r}$) in terms of the direction of the incident ray ($\boldsymbol{i}$) can be derived from the stated law:

$$\boldsymbol{r} = \boldsymbol{i} - 2(\boldsymbol{i} \cdot \boldsymbol{n})\boldsymbol{n} \tag{2.8}$$

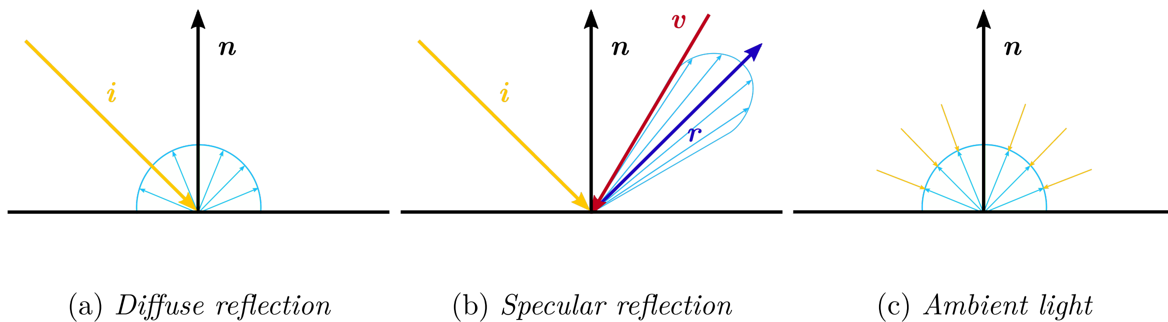
where all vectors are assumed to be normalized. Note that the program follows the convention that $\boldsymbol{n}$ and $\boldsymbol{r}$ point away from the surface, and $\boldsymbol{i}$ points towards the surface.

Perfect specular reflections on a point P are only visible when the viewer is looking in the exact direction of the reflected ray. In the program, the direction vector of the line of sight ($\boldsymbol{v}$) points towards the surface. Therefore, perfect specular reflections are only visible when $\boldsymbol{r}$ and $\boldsymbol{v}$ are anti-parallel to each other. However, specular reflection on real objects is not perfect, due to microscopic surface irregularities, and the reflected ray does not always have the direction vector given by (2.8). The Phong model approximates this by applying a $|\boldsymbol{r} \cdot \boldsymbol{v}|$ term, that is equal to one when the direction vectors are anti-parallel and decreases towards zero as the angle between vectors decreases. This can be seen in figure 2.4b. The specular reflection radiance of the model is given by:

$$\rho_s(\lambda) = k_s |\boldsymbol{r} \cdot \boldsymbol{v}|^m \rho_i(\lambda) \tag{2.9}$$

where k_s is the *specular reflection coefficient* (the fraction of light that is specularly reflected), m is the *specular reflection exponent* (which controls how close the reflections are to perfect reflections) and $\rho_i(\lambda)$ is the radiance at the wavelength λ of the incident ray [6, 7]. Note that if $\boldsymbol{r} \cdot \boldsymbol{v} > 0$ the $\rho_s(\lambda)$ is set to zero, as the viewer would not be able to see the specular reflection in this case.

It is important to note that the Phong model is an empirical model and that it is not physically accurate. For instance, it does not conserve energy (section 15.1 in [7]) and does not take into account that the color of the reflected ray can be slightly affected by the surface (subsection 4.1 of chapter 4 in [1]).



(a) *Diffuse reflection* (b) *Specular reflection* (c) *Ambient light*

Figure 2.4: *Diagrams of the local ilumination components. The yellow vectors correspond to incident rays, and the blue ones to reflected rays.*

2.4.4 Ambient light

Ambient light is used as an approximation of *indirect diffuse illumination*; which happens when an object A is illuminated by diffusely reflected light rays coming from a object B [6, 7]. In figure 2.4c, ambient light is represented by yellow incident rays coming from all directions. Ambient light is a light present at every point in the scene, and its radiance component ($\rho_a(\lambda) = k_d\rho_i(\lambda)$) added to the diffuse and specular reflection radiance components every time local illumination is computed. The results obtained by implementing a local illumination model are shown in figure 2.5.

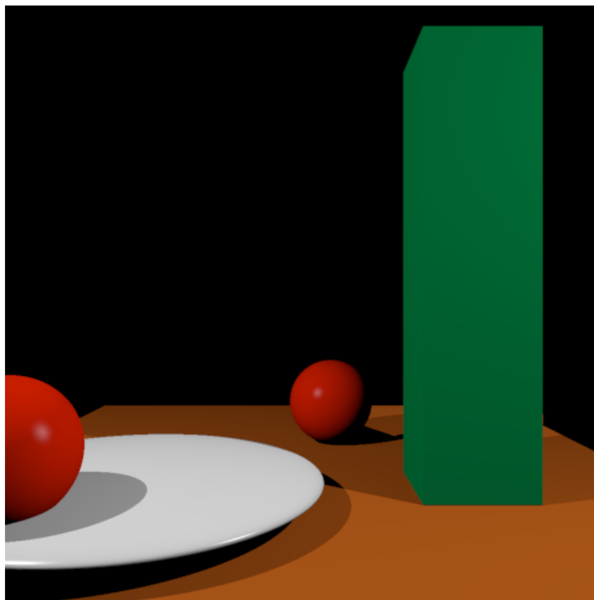


Figure 2.5: *Render of a scene using the local illumination model. A brown polygon object, a green box, a white ellipsoid and two red spheres can bee seen. There are two sources of light, so shadows are either partial or complete. The color of the objects comes mostly from the diffuse component, but specular highlights can be seen on the spheres and the ellipsoid.*

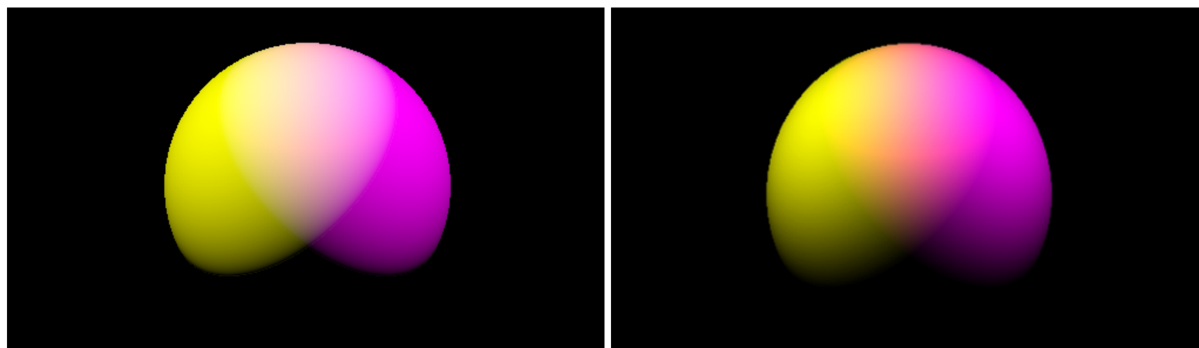
2.5 Color

Up until this point the radiance of the light rays has been given in terms of wave length λ. There are ray tracers that use spectral distributions to implement the radiance of each light ray, specially in the fields of optical analysis and thermal radiation transfer analysis. However, the ray tracer implemented in this work is not aimed to be physically accurate, but to render a realistic seeming scene. On that note, radiance has been implemented using the *linear RGB* color space.

RGB stands for Red Green Blue, which are the three components of the vectors used to represent colors in this color space (values of red, green and blue components range from 0.0 to 1.0 in linear RGB). The "linear" part of the name specifies that linear RGB is a linear color space. In linear color spaces one can apply math (addition and multiplication) to colors and expect the same result as one would get in reality. For

example, if one adds two light rays of the same color, the result is a light ray with a color twice as bright. Therefore, linear RGB is well suited to implement the radiance of a light ray, because it is compatible with all the mathematical operations performed in the rendering process.

Nevertheless, it turns out that the human eye does not perceive light linearly. For instance, in a monochromatic linear color gradient, it is far better sensing differences of brightness between darker shades than between brighter ones. Because of this, most digital images are encoded in non-linear color spaces (like sRGB), that devote more information bits to darker shades. At the same time, display monitors assume that images are encoded in non-linear color spaces, and apply corrections to the images to return to a linear color space before displaying them. If the ray tracer outputs an image encoded in linear RGB, the image displayed by the monitor would look wrong. On the other hand, as stated in [11], computing color addition or multiplication on non-linear color spaces leads to incorrect results, so they should not be used in the rendering process. Figure 2.6 is an illustrative example of the difference between rendering a scene using linear RGB and rendering the same scene using sRGB.



(a) *Linear RGB render* (b) *sRGB render*

Figure 2.6: *Rendring of a withe sphere lit by two lights of yellow ((R,G,B) = (1,1,0)) and magenta ((R,G,B) = (1,0,1)) colors respectively. Figure 2.9a shows the spected white ((R,G,B) = (1,1,0)) color in the region lit by both lights. Figure 2.9b shows a diferent, and therefore erroneous, color in the same region.*

The above explanations lead to the conclusion that the program needs to be able to transform colors between the linear RGB and sRGB color spaces. This is done by using the following formulas [12]:

sRGB to linear RGB:

if $R,\ G,\ B \leq 0.04045$

$$X_L = X/12.92 \tag{2.10}$$

if $R,\ G,\ B \geq 0.04045$

$$X_L = ((X + 0.055)/1.055)^{2.4} \tag{2.11}$$

Linear RGB to sRGB:

if $R_L,\ G_L,\ B_L \leq 0.0031308$

$$X = 12.92X_L \tag{2.12}$$

if $R_L,\ G_L,\ B_L \geq 0.0031308$

$$X = 1.055X_L^{(1/2.4)} - 0.055 \tag{2.13}$$

where X_L stands for one of the three primary color components ($R_L,\ G_L$ or B_L) of a linear RGB color, and X for one of the color components of a sRGB color ($R,\ G$ or B).

The implemented ray tracer assumes that the input colors are given in sRGB. So first, it transforms all colors into linear RGB, and then performs the rendering process in that color space. Finally, it transforms the colors of the rendered image to the sRGB color space, so that the resulting image is shown correctly by the monitor.

2.6 Accelerating Python

The ray casting camera, the scene objects and lights, and the local illumination model are the backbone of the ray tracer. Once a functioning ray tracer had been implemented, it was decided to find tools to accelerate its execution.

Ray tracers are very costly algorithms that require a lot of calculations to render each pixel. However, the rendering process of one pixel does not depend on the rendering of another, they are completely independent. This makes the ray tracing algorithm "embarrassingly parallel". Therefore, the simplest and most effective method of accelerating a ray tracer is to render pixels in parallel.

As stated before, the C++ language is well suited to efficiently render each pixel. In addition, its native multithreading capabilities allow the programmer to access all the CPU cores of the computer to run parallel processes. In contrast, Python performs serial operations slower than C++, and it has a built in *global interpreter lock* (GIL) that limits a program's execution to a single core [13].

2.6.1 A small survey of the available tools

There are several tools that can be used to accelerate the serial execution of Python code and to enable parallel programming [14]. Some tools allow to call functions written in other languages; examples include Cython for C/C++ [15], F2Py for Fortran [16], and Jython for Java [17]. Cython and Jython also allow to compile Python code into C code (that can be compiled by a C/C++ compiler into runtime code) and into code that runs in the Java Virtual Machine respectively. Both these compilation processes create code that runs faster than the native Python byte code created by CPython (the standard built-in Python compiler). IronPython is another alternative to CPython [18]; it creates

byte code that runs in the Common Language Runtime virtual machine that is part of the Microsoft .NET Framework. Cython, Jython and IronPython also allow to release the GIL and execute code in multiple CPU cores simultaneously.

Despite the advantages offered by the afford mentioned tools, all of them require to write code using specialized syntax or programming languages other than Python. Fortunately there are at least two tools that can speed up the execution of code written in the core Python language: PyPy and Numba [19, 20]. These are both *just in time* (JIT) compilers, that compile the code during its execution to optimize it and shorten the runtime. They both support the basic NumPy features used in the ray tracer program, and could offer a substantial performance improvement in the serial execution of the code. Nonetheless, Numba has a big advantage when compared to PyPy: it allows parallel programming in the CPU and the GPU. PyPy has a version that implements Software Transactional Memory to implement parallel programming in the CPU, but it does not support Python 3 [21]. For this reason, Numba was chosen to accelerate the ray tracer program.

It has to be mentioned that Python does offer the option to do parallel programming by using the `threading` and `multiprocessing` modules [22, 23]. The `threading` module does not allow to release the GIL; instead it is used to create several threads and switch from one to another whenever one of them is waiting for some external process to finish (creating a program that seemingly runs on parallel). This can greatly reduce execution time in I/O bound applications, which expend most of their time waiting for data from a remote source. However, the ray tracer program is a CPU bound program, as it expends the majority of the time performing arithmetic operations in the CPU. This means that using the `threading` module would not improve performance. The `multiprocessing` module on the other hand, can be used to create several processes. Each process is executed in a different CPU core and has its own GIL, so different sections of the program truly run in parallel. In principle the `multiprocessing` module could have been used to improve the performance of the ray tracer, but since Numba already offers parallel programming in the CPU its use was not necessary. Nevertheless, the module was used to create a process that runs in parallel to the ray tracer program and measures its performance (chapter 4).

2.6.2 Numba

Numba performs JIT compilation by using the LLVM compiler library, producing machine code at runtime. As stated in [24], it works best when optimizing code that relies heavily on loops and NumPy arrays. This makes it very well suited to accelerate the ray tracer algorithm.

Numba is based on the use of decorators that specify how to compile Python functions. The decorators are simply written above a function, and Numba takes care of the JIT compilation and optimization. The basic decorator is `@numba.jit()`, which has some arguments to select compilation options. The relevant ones for this work are `nopython` and `parallel`. There is also an option to specify the types of the arguments of the function, but Numba can perform type inference so this feature was not used.

The `nopython` argument is related to the two Numba compilation modes: the nopython mode and the object mode. The first mode generates code that does not access the Python C API, and is much faster than the code generated by the object mode. Numba always tries to use the first mode, but it switches to object mode when nopython cannot be applied. Setting a `True` value to `nopython` avoids this by issuing an error message if nopython mode fails. Fortunately, all the functions of the ray tracer are compatible with the nopython mode, and the `nopython=True` option ensures that only this mode is used.

On the other hand, `parallel` marks if a function makes use of automatic parallelization in the CPU. When set to `True`, some array operations are automatically parallelized. It also offers the possibility of explicitly parallelizing loops when there are no cross iteration dependencies. "Embarrassingly parallel" algorithms can be easily parallelized by using this second capability. In the ray tracer, the only necessary change is to write `prange` instead of `range` in the loop that iterates over the pixels of the image.

As previously mentioned, Numba also supports GPU programming. More specifically, it is compatible with Nvidia GPU devices that use the CUDA interface [25]. GPUs have hundreds of cores, and are purposefully built to process parallel programming, which in CUDA is based around CUDA kernels and device functions. Numba offers the capability of turning Python functions into kernels and device functions by using decorators `@cuda.jit()` and `@cuda.jit(device=True)` respectively.

CUDA kernels are functions that are called from the CPU but execute on the GPU. When a kernel is called, its code is executed in multiple threads at the same time. This threads are grouped into blocks, that all together form a grid. The programmer can select the total number of threads and the number of threads per block, which are specified each time the function is called. On the other hand, device functions are functions that are executed on the GPU and can only be called from the GPU (by kernels or other device functions).

In the Numba CUDA implementation of the ray tracer, a kernel function performs the rendering process of a subsection of the image, calling various device functions in the process. The size of the image section depends on the number of total threads used: the bigger the number of threads, the smaller the section of the image each of them has to render. It is apparent then, that the speed of the rendering process will depend on the total number of threads that is selected. The number of concurrently executing threads is limited by the hardware, and is far smaller than the amount of pixels the ray tracer renders. However, the total number of threads called by the programmer can be as big as the number of pixel. This leads to the following question: is it better to call as many threads as there are pixels, even though they will not all be executed at the same time? Or is it better to create the maximum number of threads that can be executed at the same time, and let each of them handle multiple pixels?

As it turns out, the number of threads per block also influences the speed of rendering, so getting the answer to the question required some testing. First, the execution time for different numbers of total threads were measured, with fixed values of threads per block. The results are shown in figure 2.7, that lead to the conclusion that in general the "as many threads as there are pixels" approach works better. Knowing this, the next

step was to measure the execution time for a variety of block sizes keeping the number of total threads equal to the number of pixels. Figure 2.8 shows the results of this second test, which established that a number 32 threads per block was the optimum, at least for the tested scene.

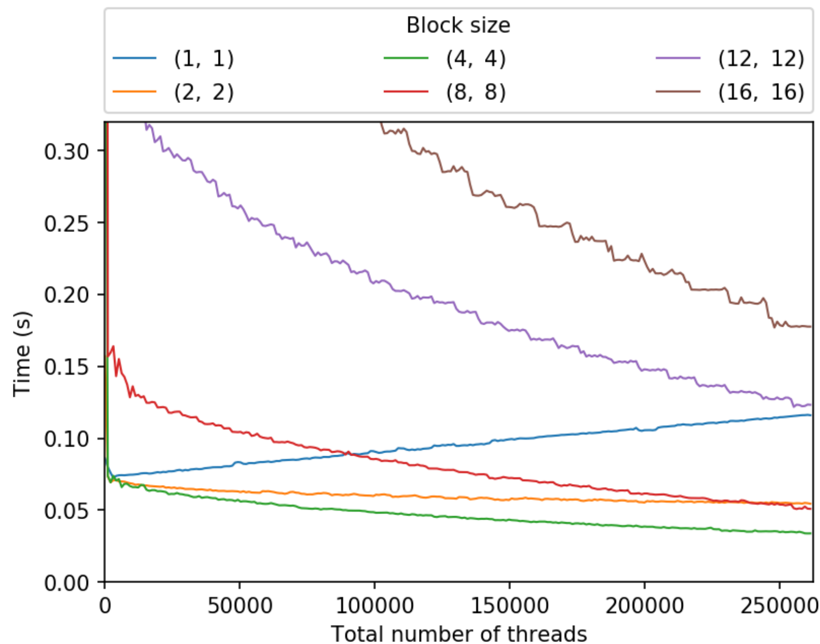


Figure 2.7: *Execution time of the Numba CUDA ray tracer with respect to the total number of threads. Each line corresponds to a two dimensional block size, indicated by the number of threads the block has in its horizontal and vertical directions.*

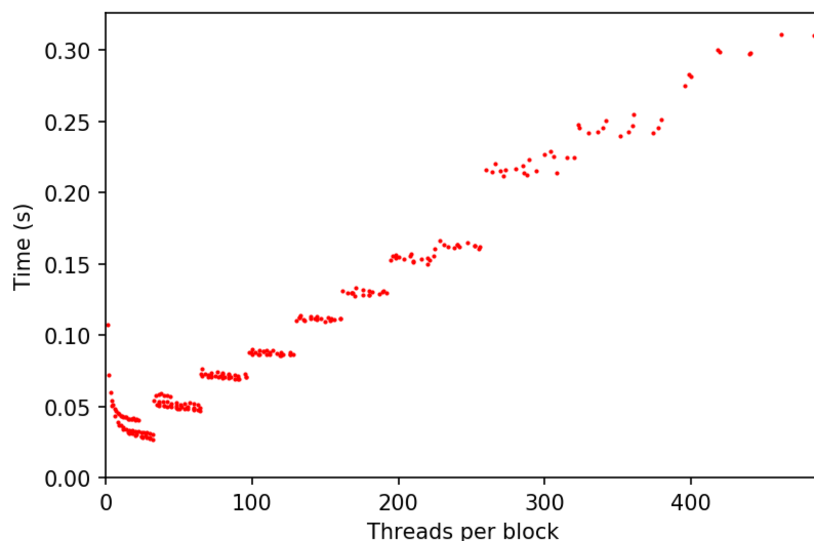
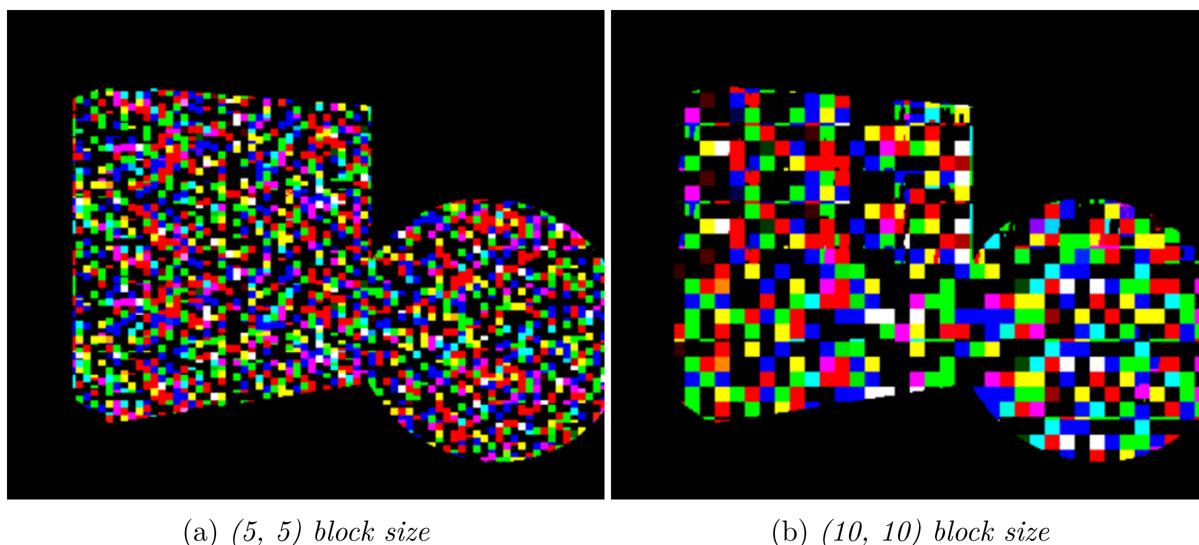


Figure 2.8: *Execution time of the Numba CUDA ray tracer for various kernel block dimensions. The total number of threads is fixed and equal to the total number of rendered pixels. Each dot corresponds to a different two dimensional block size.*

When discussing the performance of CUDA functions memory use is very important. Numba gives access to three main types of memory: global device memory, local memory and on-chip shared memory. Global device memory is the memory that is used when an array is transferred from the CPU to the GPU, and can be accessed by all threads. Local memory is private to each thread, and shared memory can be accessed by all the threads in a block. Local and global memory access speed is similar, and much slower than shared memory access. However, shared memory is limited and requires much more careful management.

At the beginning of the project an attempt was made to use shared memory to store the read-only data of the camera, the objects and the lights. The results can be seen in figure 2.9, in which the surface colors of the rendered objects change following a checkered pattern. The size of the squares that form the pattern changes when the number of threads per block is altered, but the underlying reason for that was not found. Due to this complications, shared memory was not used in subsequent versions of the CUDA ray tracer.



(a) *(5, 5) block size* (b) *(10, 10) block size*

Figure 2.9: *Incorrect renderings of a scene caused by poor use of CUDA shared memory. Each image is the result of a kernel call with a different block size.*

After including Numba acceleration, there are 4 versions of the Python ray tracer: the core language version, the mono-threaded Numba CPU version, the multi-threaded Numba CPU version and the Numba CUDA GPU version. The performance comparison between all of them and the C++ ray tracer can be found in chapter 4.

2.7 Global illumination

A *global illumination model* is defined as a ray tracing model that computes the color contributions of light rays of second order or higher. These rays are reflected more than once before reaching a light source, as shown in figure 2.10.

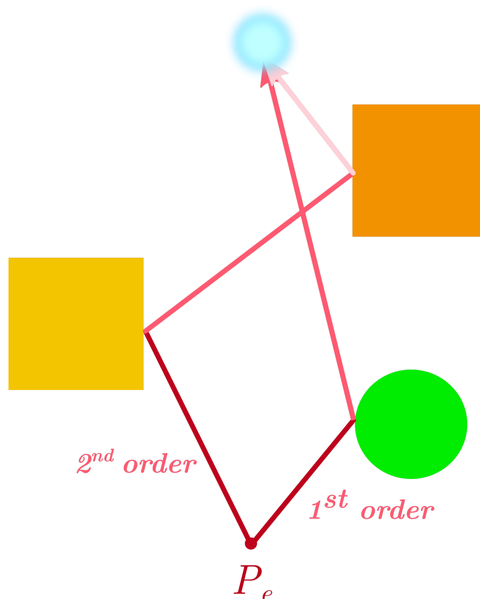


Figure 2.10: *The paths followed by two rays from the eye point P_e to a light. The* 1st *order ray reflects once, and the* 2nd *order ray reflects twice.*

2.7.1 Reflection

When a light ray reflects off a surface its direction and color change. In the model, these changes are implemented by first calculating the new spacial and color values, and then creating a new *reflected ray.*

As in the case of the shadow feeler, the point of origin of the reflected ray is placed at a distance ϵ from the surface, to prevent the ray from intersecting the surface it reflected off of. The direction of the reflected ray is computed by applying equation (2.8). This makes all the reflections perfect, so the model only implements surfaces that are perfectly smooth. Section 24.4 of [7] points out that assuming the reflections on the global illumination model are perfect is inconsistent with the implementation of glossy specular reflections (Phong model) in the local model. Since the glossiness of a reflection is determined by the object surface, the amount of glossiness applied to global reflections should be the same to the one applied to specular reflections on the local illumination model.

When a light ray reflects off a surface, a fraction of its energy is lost. This can be modeled by simply multiplying the color vector of the incident ray by the specular reflection coefficient (k_s). On top of that, if the surface isn't fully reflective, a fraction of the energy of the incident ray will end up in the *transmitted ray.* Since the implemented model includes transmissive objects, it is necessary to compute what fraction of the incident ray's energy the reflected ray has. This fraction is called the *reflection coefficient* (R), and can be obtained from the Fresnel equations. Since the model assumes that all

light is unpolarized, R is computed by using the following equation:

$$R = \frac{1}{2}\left\{R_{\perp} + R_{\parallel}\right\} \tag{2.14}$$

where $R_{\perp}$ is the reflection coefficient for light polarized in a perpendicular direction to the surface, and $R_{\parallel}$ is the reflection coefficient for light polarized in a parallel direction to the surface. Their values are given by:

$$R_{\perp} = \left(\frac{\frac{n_i}{n_t}\cos\theta_i - \frac{\mu_i}{\mu_t}\sqrt{1-\left(\frac{n_i}{n_t}\right)^2(1-\cos^2\theta_i)}}{\frac{n_i}{n_t}\cos\theta_i + \frac{\mu_i}{\mu_t}\sqrt{1-\left(\frac{n_i}{n_t}\right)^2(1-\cos^2\theta_i)}}\right)^2 \tag{2.15}$$

$$R_{\parallel} = \left(\frac{\frac{\mu_i}{\mu_t}\cos\theta_i - \frac{n_i}{n_t}\sqrt{1-\left(\frac{n_i}{n_t}\right)^2(1-\cos^2\theta_i)}}{\frac{\mu_i}{\mu_t}\cos\theta_i + \frac{n_i}{n_t}\sqrt{1-\left(\frac{n_i}{n_t}\right)^2(1-\cos^2\theta_i)}}\right)^2 \tag{2.16}$$

where subscript i refers to incident, t to transmitted, θ_i is the angle between the surface normal $\boldsymbol{n}$ and $-\boldsymbol{i}$ (following the convention given in subsection 2.4.3), μ is the relative magnetic permeability (assumed to be equal to 1 in the equations implemented in the model) and n is the index of refraction.

To be able to apply these equations in the model, it is necessary to know in which medium the incident ray travels and which medium the transmitted ray will travel in. Following the convention of subsection 5.2.3 of [6], all objects are assumed to be surrounded by air. The model keeps track of the value of n_i; if it's equal to 1, the incident ray is traveling through air and n_t is computed from the surface properties of the intersected object, if it's not, the incident ray is traveling inside the object and $n_t = 1$.

Once R is computed, the color of the reflected ray is computed multiplying the color of the incident ray by $k_s R$.

2.7.2 Transmission

Up until this point, light rays haven't been allowed to enter and travel thorough any object, so all the rendered scenes have contained opaque objects. With the implementation of transmission, the ray tracer becomes able to render translucent objects. The implementation of transmission is very similar to the implementation of reflection: each time a light ray intersects an object surface a transmitted ray is created.

The color of the transmitted ray is obtained multiplying k_s and the *transmission coefficient* ($T = 1 - R$) to the color of the incident ray.

On the other hand, the computation of the direction of the ray is based on *Snell's Law of Refraction*. This equation can be rearranged to get an expression for the direction vector ($\boldsymbol{t}$) [6]:

$$\boldsymbol{t} = \left[(\boldsymbol{t} \cdot \boldsymbol{n}) - \frac{n_i}{n_t} (\boldsymbol{i} \cdot \boldsymbol{n})) \right] \boldsymbol{n} + \frac{n_i}{n_t} \boldsymbol{i} \tag{2.17}$$

In the model, $\boldsymbol{n}$ can be on either side of the surface, so $(\boldsymbol{i} \cdot \boldsymbol{n})$ and $(\boldsymbol{t} \cdot \boldsymbol{n})$ can be negative. This is not a problem, because the $\boldsymbol{n}$ outside the brackets compensates the sign. However, the signs of $(\boldsymbol{i} \cdot \boldsymbol{n})$ and $(\boldsymbol{t} \cdot \boldsymbol{n})$ must be the same. $(\boldsymbol{t} \cdot \boldsymbol{n})$ is calculated using the following equation:

$$(\boldsymbol{t} \cdot \boldsymbol{n}) = \begin{cases} -\cos\theta_t, & \text{if } (\boldsymbol{i} \cdot \boldsymbol{n}) \leq 0 \\ \cos\theta_t, & \text{if } (\boldsymbol{i} \cdot \boldsymbol{n}) > 0 \end{cases} \tag{2.18}$$

where $\cos\theta_t$ is given by:

$$\cos\theta_t = \sqrt{1 - \left(\frac{n_i}{n_t}\right)^2 (1 - (\boldsymbol{i} \cdot \boldsymbol{n})^2)} \tag{2.19}$$

Finally, as in the case of the reflected ray, the origin point of the transmitted ray has to be placed a small ϵ distance away from the surface to avoid false collisions. However, the origin of the transmitted ray and the origin of the reflected ray have to be placed at opposing sides of the surface. So, if the incident ray is outside an object, the origin of the reflected ray is placed outside and the origin of the transmitted ray inside.

As stated in subsection 5.2.4 of [6], the biggest limitation of this implementation of transmission is that the shadows of the translucent objects are as dark as the ones the opaque objects cast. The implemented shadow feeler is directly aimed at the point light sources, and is not allowed to go through any scene object. Since the shadow feeler and the spatial part of a light ray are mathematically the same, the transmission model could be applied to compute transmitted shadow feelers. The problem is, that these transmitted rays would no longer be aimed at the light source, and thus couldn't be used as shadow feelers.

More complex ray tracers implement photon maps to render correct shadows of transmissive objects and simulate caustics (like the bright spot that can be produced by concentrating sunlight with a magnifying glass). Photon mapping is performed before rendering. First, many photons (analogous to the implemented light rays) are emitted from each light source. Then, the photon-object intersections are computed, and the photons are reflected, refracted or absorbed based according to probability functions. Finally, the scene points in which photons have been absorbed are stored to create a photon map. After the standard rendering process is finished, the photon map is superimposed to create the final image [26].

2.7.3 Attenuation

Light attenuation refers to the absorption of light as it passes through a material [6]. The model implements the *Beer-Lambert Law* that can be written in the following way:

$$\rho_\lambda(d) = \rho_{\lambda,0} A_\lambda^d \tag{2.20}$$

where $\rho_{\lambda,0}$ is the radiance of the ray at its origin point, d is the distance traveled inside the material and A_λ is the *attenuation coefficient.*

Attenuation is applied each time a light ray intersects an object; multiplying the color vector of the ray by the attenuation factor A_λ^d.

2.7.4 The `ray_trace` algorithm

The `ray_trace` function is the main function of the part of the program that renders the image. At its core, the function takes a light ray as an input, calls the `intersection` function to get the P, $\boldsymbol{n}$ and t_P values of the closest object, and uses this information to apply the local illumination model and get the color of the ray.

To render a scene applying local illumination only, it would be sufficient to pass each ray that originates in the eye point to the `ray_trace` function, and use the output ray colors to set the image pixel sRGB values. However, global illumination requires the implementation of higher order rays, which will also need to be passed to `ray_trace` in order to get their color. For this reason, `ray_trace` has been implemented as a recursive function in the case of the CPU implementations [6].

On each iteration, the function spawns a reflected and a refracted ray, and then calls itself with each of the new rays as the new ray argument. Once the color of the reflected and the refracted rays have been computed, they are added to the color given by the local illumination model. Finally, t_P is used along with the surface properties of the closest object to apply attenuation and get the final color.

The recursion is set to terminate if a given ray doesn't intersect any object, and there is also a maximum depth of recursion that limits the maximum order of the rays.

Recursion is the easiest solution to implement `ray_trace`, but unfortunately, Numba CUDA doesn't support recursion. Therefore, in the GPU implementation, recursion was substituted by a equivalent iterative algorithm, based on the indications and procedures given in [27].

The iterative version first computes the maximum possible number of rays (N) for a given value of depth of recursion (d) by using the following equation:

$$N = 2^{d+1} - 1 \tag{2.21}$$

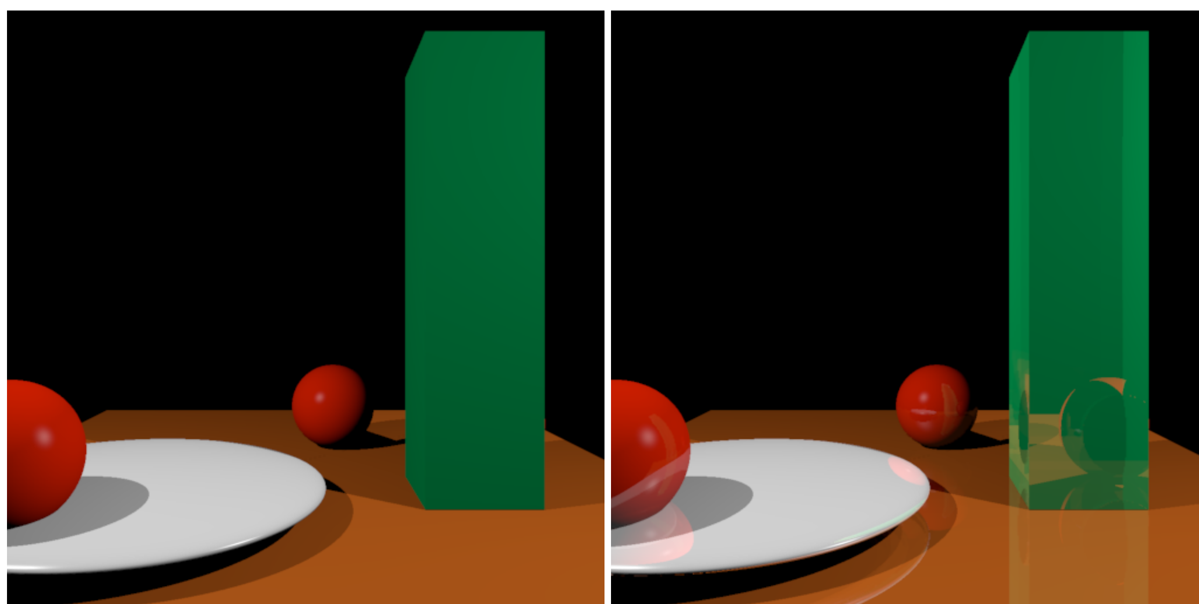
$d = 0$ means that only first order rays are considered (local illumination only), $d = 1$ that up to second order rays are considered and so on. Equation (2.21) arises from the

fact that that each lower order ray spawns two higher order rays: a reflected ray and a refracted one.

The characteristics of each ray depend on the ray from which it originated (*parent ray*), up to the first order ray. For this reason the information of all the rays is stored in a 2D array (`ray_data`) with N rows that contains information about a ray in each row. Specifically, each row stores the defining parameters of a ray, the index of refraction of the medium it is traveling through and the *cumulative factors* that account for the effect of the reflection, transmission and attenuation coefficients of the parent rays.

The algorithm loops through the rows in `ray_data`, applying local illumination for each ray and multiplying it by the cumulative factors before adding the resulting color to the color of the corresponding pixel. Much like in the recursive version, each iteration spawns a reflected and refracted ray, that in this case are stored on higher index rows of `ray_data`.

Either by using recursion or iteration, the ray tracers use higher order rays to implement global illumination and enhance the local illumination model by adding reflection and transmission. The results of this can be seen on figure 2.11.



(a) *Local illumination only* (b) *Global illumination included*

Figure 2.11: *Renders of the same scene using local illumination only, and local illumination with global illumination. The render that includes global illumination shows the reflections of objects on the surfaces of other objects, and also reveals that there is a hidden red sphere behind the green box.*

3 Enhancements

Once the basic ray tracer was completed, the model was further developed by implementing some enhancement techniques. These can be organized in three categories: anti-aliasing techniques, more advanced scene object creation and enhancements that improve photorealism.

3.1 Anti-aliasing

In the context of image rendering, aliasing refers to the stair-like borders (informally known as "jaggies") that appear where smooth borders are expected. Since computer screens are made up of pixels and pixels are not infinitesimally small, aliasing cannot be completely avoided. However, there are several anti-aliasing techniques that can reduce aliasing and effectively hide it from the human eye. This is achieved by taking multiple samples per pixel, computing the average color of those samples and assigning the result to the pixel. The process is known as *supersampling,* and the results it can produce are shown in figure 3.1. There are various supersampling techniques, among which three have been implemented in the model: uniform supersampling, jittered supersampling and adaptive supersampling.

3.1.1 Uniform supersampling and jittered supersampling

These techniques are considered brute force supersampling techniques, because they apply the same level of supersampling to every pixel of the image. Anti-aliasing is only needed when there is a big color difference between adjacent pixels, so brute force supersampling techniques are inefficient. Nevertheless, they are the simplest to implement, and produce good results.

In uniform supersampling, each pixel (region of the screen) is subdivided into smaller equally sized subareas. The center of the subareas is used, in conjunction with the eye point, to aim the light rays. Once the colors of all the rays that go through a pixel are computed, the color of the pixel is computed using the following formula:

$$\rho_{pixel}(\lambda) = \frac{1}{N}\sum_{i=1}^{N}\rho_i(\lambda) \tag{3.1}$$

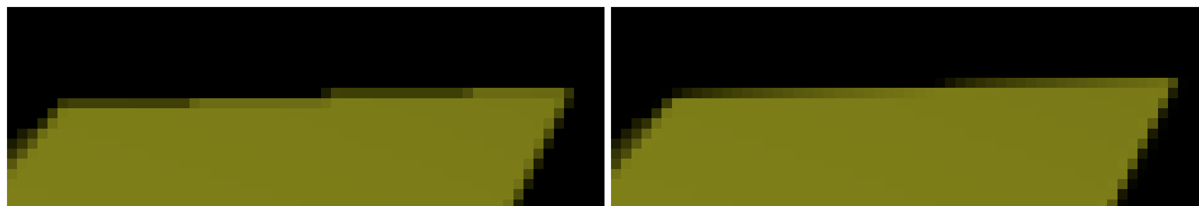
where $\rho_{pixel}(\lambda)$ is the radiance that corresponds to the pixel, $\rho_i(\lambda)$ are the radiances of the pixel subareas and N is the number of subareas per pixel.

Jittered supersampling combines uniform supersampling and the *Monte-Carlo integration technique*. The pixels are subdivided in the same way, but instead of using the center of the subareas to aim, a random point is chosen. This choice makes an image rendered by applying jittered supersampling different from one rendered by using uniform supersampling. Jittered supersampling is better eliminating the undesired lines that can appear between the triangles that constitute a polygon. However, it can worsen the render of vertical and horizontal borders when the number of samples is small.

As a final note, it must be mentioned that with a high enough number of samples per pixel uniform and jittered techniques give the same result.

3.1.2 Adaptive supersampling

Adaptive supersampling increases the efficiency of supersampling by applying it only where necessary. Instead of taking a sample of the center of the pixel, the algorithm uses the corners of the pixel to aim the rays. If the colors of the casted rays are similar to each other the color of the pixel is computed by taking the average. However, if the colors differ more than a specified amount the pixel is subdivided into 4 equal sized subareas. The color of each subarea is computed the same way as the color of a pixel. So each subarea might be divided into smaller and smaller subareas until the colors of the corners of the smallest subarea are similar enough or a specified maximum recursion depth is reached.



(a) *Without supersampling* (b) *With supersampling*

Figure 3.1: *Amplified sections of renders obtained without supersampling and with supersampling. The horizontal border of the yellow object presents smother color transitions between adjacent pixels in the render that includes supersampling.*

3.2 Further development on scene objects

Section 2.3 explained how simple shapes, also known as primitives, were implemented. Spheres, boxes, polygons and ellipsoids are enough to create some interesting looking scenes, but the kinds of scenes that can be created are quite limited. With that in mind, two additional features were included as an enhancement to the basic ray tracer described in [6]: triangle meshes and Boolean operations with primitives.

3.2.1 Triangle meshes

Triangle meshes are 3D shapes constituted by interlocking triangles with common edges and vertices. With enough triangles, meshes can be used to represent any shape and are widely used in commercial modeling, rendering and animation software [7].

Triangle meshes are defined in a similar way to other objects with the same surface properties and some spatial data. The spatial data contains the spatial data of each constituent triangle, but unlike in the case of primitives some this information is stored in a separate file. There are several file formats that store triangle mesh data. The model uses Wavefronts OBJ file format (.obj) to store the mesh definition [28]. The mesh of the OBJ file contains triangles defined relative in a certain frame of reference that might not coincide with the scene in which the mesh must be placed. For that reason, there are three additional fields that constitute the spatial data of a triangle mesh: a origin position (to place the mesh in the scene), Euler angles (to orient the mesh) and a scaling factor (to determine the size of the mesh in the scene).

Among the information that OBJ files can store, the model only uses the vertex (v) and face (f) fields. Vertex fields store the spatial positions of all the vertices of the constituent triangles, without repetition. The face fields reference the vertices to define the constituent triangles (or polygons). When parsing a OBJ file (using the `mesh_parser` function), the model extracts the vertices of each and uses them to create triangle primitives (a kind of polygon previously implemented). It then applies a rotation matrix based on the Euler angles to each vertex [29]. Followed by the scaling factor, and a translation using the origin position. Finally, it adds the surface properties information to each newly created triangle primitive.

Section 2.3 explained that the normal vectors of 3D objects are always defined as pointing outward from the object, but the normal vectors of 2D objects are defined as pointing to the side the incident ray comes from. In principle, triangle meshes are 3D objects, so the normal vectors should point outward. However, the model parses meshes by creating a bunch of 2D triangle primitives. Because of this, when a ray traveling inside a mesh hits a constituent triangle, the shadow feeler is spawned inside the object and never reaches a light. If the shadow feeler was instead spawned outside it might have reached a light, contributing color to the pixel that is being rendered. This means that, when a light ray exits a mesh, the local illumination contribution that should be added is not. It is important to note that this doesn't affect the orientations of the reflected and refracted rays, which are spawned in the correct sides of the triangles and can contribute to the color of the pixel.

There are methods to correctly define the normal vectors, like the ones explained in section 12.5 of [8] and subsection 23.4.2 of [7]. However, the model gives correct results for non-refractive meshes, and correct enough for refractive meshes (it only makes translucent objects a bit more see-through). An example of a rendered mesh is shown in figure 3.2.

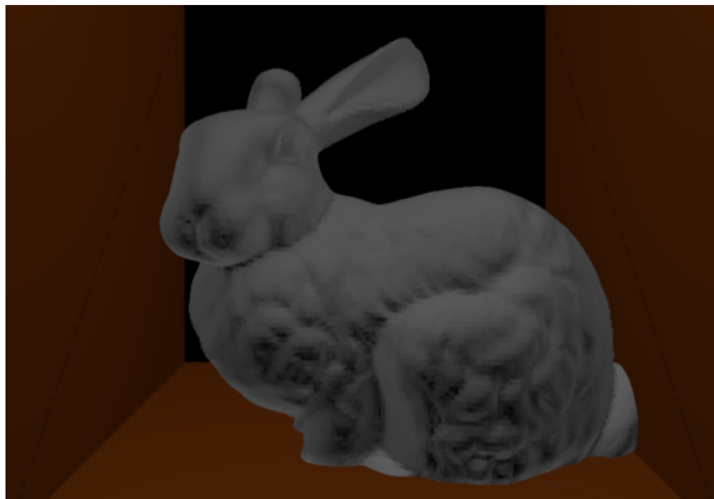


Figure 3.2: *Render of the Stanford bunny mesh object.*

3.2.2 Boolean operations with primitives

Boolean operations with primitives or constructive solid geometry consists on building complex objects by combining simpler primitive objects. The most common operations are the OR operation (also called union operation) and the AND operation (also called intersection operation). The first combines primitive objects to create a single object that includes points inside at least one of the constituent primitives. The second creates an object that only includes points that are inside all constituent primitives. The implementation of Boolean operations with primitive objects is discussed in chapter 14 of [8] and chapter 3 of [1]. The information found there was used to get a general idea of how this feature should work, as the specifics of the implementations there described were not necessarily compatible with the ray tracer developed so far.

As a consequence, the developed version has some limitations when compared to the ones described in [8] and [1]. The main one is that objects can only be constructed by combining multiple primitives with OR operations or by combining multiple AND operations, but not by combining OR and AND operations. In addition, all the combined objects need to have the same surface properties, to maintain consistency with the implemented attenuation model. Finally, only spheres, boxes and ellipsoids can be used as constituent primitives, so polygons are excluded. The reason is that the implementation relies on determining inside which of the constituent primitives a ray is, and a ray cannot be inside a polygon. An example of an OR and an AND construct created by combining the same primitives is shown in figure 3.3.

The implementation of this feature required the addition of a *medium* parameter to both objects and rays. In the case of objects the parameter determines if the object is an isolated object or if it forms part of an OR or AND construct. In the case of rays, it determines if a ray is outside OR/AND constructs, inside an OR construct or inside an AND construct. If a ray is inside an OR construct, the medium parameter also determines inside which of the constituent primitives it is. Medium information is

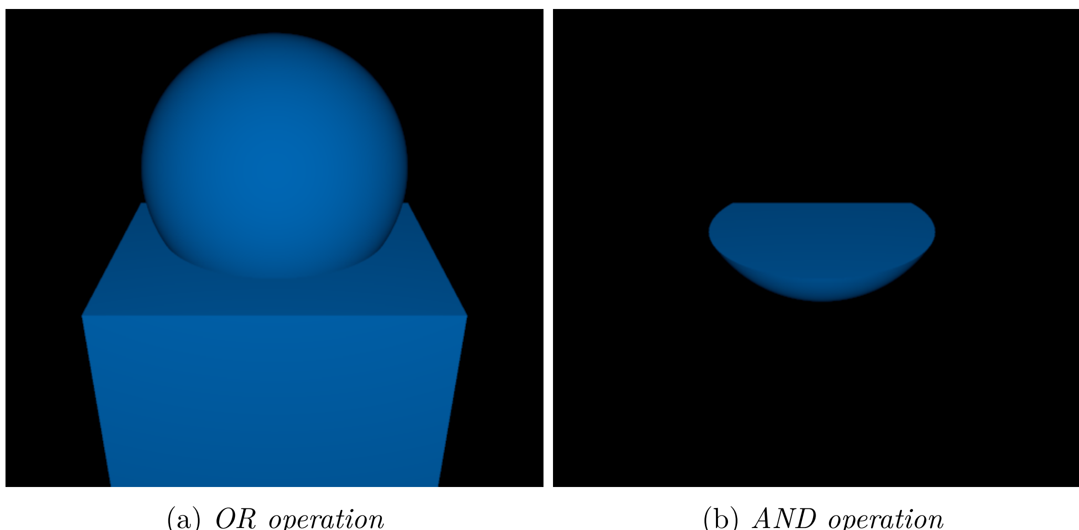
used by two new intersection algorithms that correspond to constructed objects: `OR_intersection` and `AND_intersection`.

`OR_intersection`

This algorithm takes in a list of constituent primitives and an incident ray and returns the closest intersection of the OR construct and the ray. If the ray incident ray is outside the OR construct, the algorithm simply returns the intersection with the closest constituent primitive. However, if the ray is inside one of the primitives the algorithm needs to compute where the ray would get outside the OR construct. The problem is that it has to do this by calculating ray primitive intersections. It does this by entering a loop that computes the closest ray primitive intersection, and then spawns the ray to be used in the next iteration on the other side of the intersected surface. At the same time, it keeps tack of which of the primitives have been entered or exited, and the loop is iterated over until the newly spawned ray is outside all the primitives.

`AND_intersection`

As the previous algorithm, this one takes in a list of constituent primitives and an incident ray and returns the closest intersection of the respective construct and the ray. In this case the simple part is computing the intersection when the incident ray is inside the AND construct. The algorithm simply returns the intersection between the ray and the closest primitive. When the incident ray is outside, the algorithm employs a similar loop to the one described for the `OR_intersection` algorithm. However, in this case the loop is exited when the newly spawned ray is inside all the primitives, or when it is outside a primitive that had been entered in a previous iteration.



(a) *OR operation* (b) *AND operation*

Figure 3.3: *Renders created by combining a box and a sphere by using the OR operation and the AND operation.*

3.3 Steps towards photorealism

The aim of this section is to present three techniques that were implemented to make the rendered images look more photorealistic: depth of field, soft shadows and glossy reflections. In this context, the term “photorealistic” refers to how realistic a rendered image looks, and does not necessarily imply that the rendering process is more physically accurate.

3.3.1 Depth of field

In photography, depth of field refers to the range of distance where the image in the film is in focus. In perfect pinhole cameras where all the rays go through a infinitesimally small pinhole, the depth of field is infinite and the entire image is in focus. However, in real cameras the rays go through a lens with a finite radius, and only a plane parallel to the lens (located at *focal distance* d_o) is in perfect focus: the *focal plane*. With the inclusion of depth of field, the ray tracer becomes able to reproduce this “shortcoming” of real cameras, rendering images that more closely resemble real photographs.

As explained in chapter 10 of [7], the simplest way to simulate depth of field in a ray tracer is to implement a camera with finite-radius (r) lens, assuming the *thin-lens* approximation. Under this assumption, all the objects at distance d_o from the lens produce a perfectly focused image at distance d_i (*image distance*) from the lens. Therefore to render a perfectly focused image of the objects at distance d_o the screen of the ray tracer should be positioned at distance d_i. The scene objects at other distances (d'_o) are out of focus, and the amount of blurriness increases the more d'_o differs with respect to d_o. The blurriness also increases when the radius of the lens is increased, because this allows a higher amount of rays with different directions to reach the same point in the screen.

In the implemented model, distance d_i can be derived by calculating the norm of vector $\boldsymbol{e}$ define in section 2.2. Then, the needed additional parameters to define a thin-lens camera are the focal distance and the lens radius. This parameters are used to modify the 3.6 and 2.2 equations that define the eye rays.

The lens radius is used to get a random point P_l inside the lens plain; this substitutes the eye point P_e. The focal distance is used, along with the screen point Q, to calculate the point where the eye ray intersects the focal plane (Q_o). To understand how this is done, it is convenient to define vectors $\boldsymbol{p}_i$ and $\boldsymbol{p}_o$:

$$\boldsymbol{p}_i = Q - C \quad \boldsymbol{p}_o = Q_o - C \tag{3.2}$$

The thin-lens theory says that the components of $\boldsymbol{p}_i$ and $\boldsymbol{p}_o$ parallel to the lens plain satisfy this relation:

$$\frac{p_{ou}}{p_{iu}} = \frac{p_{ov}}{p_{iv}} = \frac{d_o}{d_i} \tag{3.3}$$

where subscripts u and v refer to orthogonal vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ inside the lens plain.

The $\boldsymbol{e}$ component of $\boldsymbol{p}_o$ is a vector that goes from point C to the center of the focal plane, and its magnitude is defined as:

$$p_{oe} = -(d_o - d_i) \tag{3.4}$$

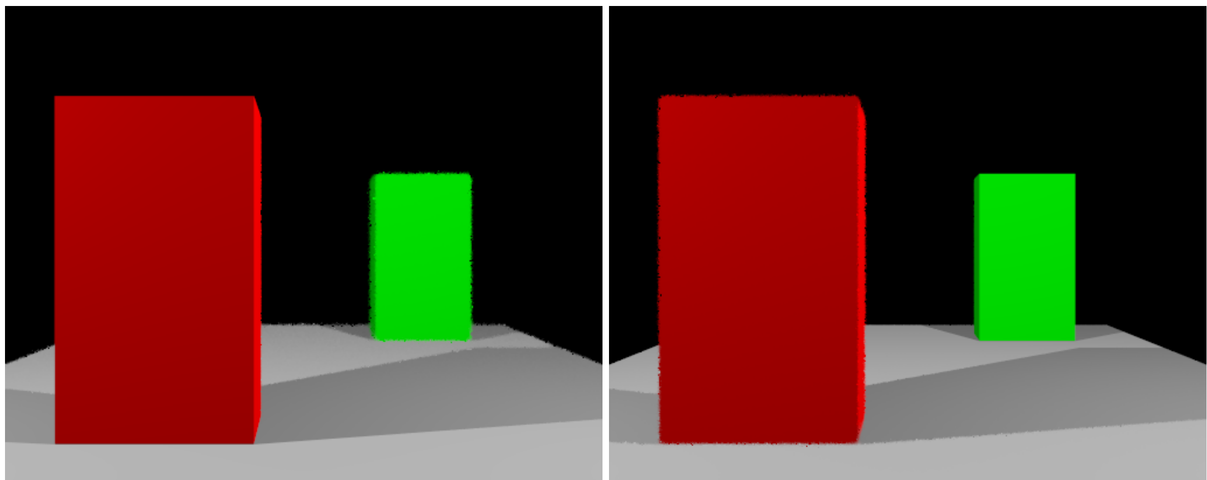
With this, Q_o can be written in terms of the known parameters:

$$Q_o = C + \boldsymbol{p}_o = C + (\alpha \boldsymbol{u} + \beta \boldsymbol{v}) \frac{d_o}{d_i} - (d_o - d_i) \frac{\boldsymbol{e}}{|\boldsymbol{e}|} \tag{3.5}$$

Finally, the equation for the eye ray in a thin-lens camera is given by:

$$\rho(t) = \{P_l + t(Q_o - P_l) \mid 0 \leq t \leq \infty\} \tag{3.6}$$

The results of implementing the thin-lens camera to the model can be seen in figure 3.4. When rendering images that include depth of field, the out of focus images can present considerable noise, due to the randomness introduced when calculating P_l. This noise can be reduced by applying supersampling, which uses a different P_l for each sample. Figure 3.4 includes supersampling, but some noise is still present.



(a) *Red box in focus* (b) *Green box in focus*

Figure 3.4: *Renders of a scene with two settings of depth of field. The renders present noise in the borders of the objects that are out of focus, instead of a uniform blur.*

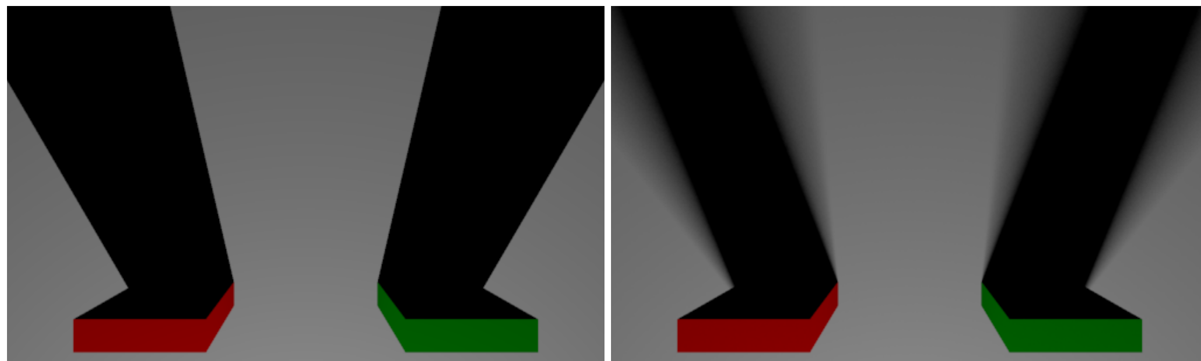
3.3.2 Soft shadows

The shadow model presented in subsection 2.4.2, produces shadows that have a clearly defined sharp edge. This is kinds of shadows are consistent with the point like nature of the lights that the model uses, and could be reproduced in reality by using one-dimensional sources of light. The problem is that most real light sources have a finite area, and create shadows that have a soft edge often referred to as the penumbra. The

regions on the penumbra receive light only from a part of the light, and are therefore partially in shadow. Including soft shadows in the model greatly enhances photorealism, bringing the rendered shadows closer to those seen in reality.

Implementing soft shadows in ray tracer normally involves creating area lights or light emitting objects as explained in chapter 18 of [7] and chapter 6 of [10]. However, the method described in exercise 18.1 of [7] is much simpler, and produces good soft shadows. It consists on giving a radius parameter r to each point light, and computing the local illumination at point P by casting multiple shadow feelers. Each shadow feeler is aimed at a random point r distance away from the point light, so some of them may return that point P is in shadow while others determine that it is not. After each shadow feeler is casted, the model applies local illumination to get the corresponding color for P. Finally, all the computed colors are averaged to get a final value of the color of P.

If the number of casted shadow feelers is small, the resulting shadow will present noise with big variations in the lightness of adjacent pixels. Nevertheless, with as the number of shadow rays is increased, the lightness of each pixel converges to a certain value, and the noise disappears. A render of soft shadows with reasonably low noise is shown in figure 3.5b and can be compared to figure 3.5a that presents hard shadows.



(a) *Hard shadows* (b) *Soft shadows*

Figure 3.5: *Renders of the same scene using the basic shading model and using the more advanced soft shadow model.*

3.3.3 Glossy reflections

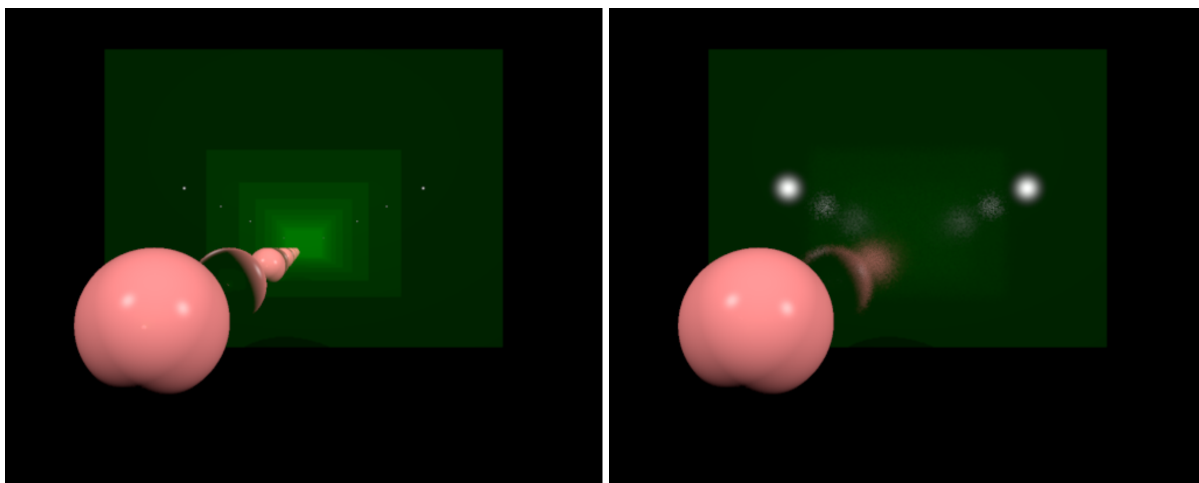
Glossy reflections are the imperfect or blurry reflections that occur when the reflective surface is not completely smooth. In subsection 2.7.1, it was mentioned that the perfect specular reflection model used in global illumination is not consistent with the Phong model applied in local illumination. The problem is that the reflections of point lights are blurry while the reflections of scene objects are perfect. The solution to this lies in substituting perfect reflections by Phong reflections in the global illumination model.

The procedure to achieve glossy reflections in global illumination is described in detail on chapter 25 of [7]. It is based on the Phong model, that states that glossy reflection rays (with direction $\boldsymbol{r}_g$) follow a distribution proportional to $(\cos\theta)^m$, where m is the

specular reflection exponent and θ is the angle with respect to the direction of the perfectly reflected ray ($\boldsymbol{r}$). This was applied to compute radiance of specular reflections in equation 2.9, where the direction of the glossy reflection (denoted by the symbol $\boldsymbol{v}$ in that equation) and $\boldsymbol{r}$ were known.

The difficulty in this case is that $\boldsymbol{r}_g$ is unknown, so instead of using its value to compute the radiance, the distribution $(\cos\theta)^m$ has to be used to get $\boldsymbol{r}_g$. This is done by first writing $\boldsymbol{r}_g$ in terms of azimuth and polar angles $(\phi,\ \theta)$ on a hemisphere with a symmetry axis parallel to $\boldsymbol{r}$. Then, random values of $(\phi,\ \theta)$ that follow a $(\cos\theta)^m$ distribution are computed following the indications of sections 7.1 and 7.2 of [7]. Thus, a $\boldsymbol{r}_g$ direction that follows the Phong model distribution is obtained.

As in the case of soft shadows, spawning a single ray to compute the radiance produces noisy results. For this reason the model offers the possibility to select how many reflected glossy rays are spawned on each recursion iteration of `ray_trace`. When multiple rays are created, the resulting radiance is computed by averaging the values of all the rays, taking into account that each of them has to be multiplied by the corresponding $(\cos\theta)^m$ factor. However, reducing the noise in this case is far more expensive than in the depth of field or soft shadow cases, because increasing the number of reflected rays on each recursion iteration produces an exponential increase in the total number of casted rays. That is why the results of glossy reflections shown in figure 3.6b present considerable noise.



(a) *Perfect mirrors* (b) *Glossy mirrors*

Figure 3.6: *Renders of two parallel mirrors with a pink sphere in between. When the mirrors are perfectly reflective, all the reflections of the sphere are perfectly sharp. When the mirrors are glossy the sphere reflections are blurry, and get blurrier with each reflection iteration.*

4 Performance comparison

With various version of a functioning ray tracer model at hand, it became possible to setup a performance comparison. The main objectives of this comparison were to quantify the acceleration that Numba provides to the core Python implementation, and to measure how the accelerated Python versions fare when measured against the C++ ray tracer.

4.1 Preparation

The previous chapters have shown that ray tracers can include various kinds of features, and that they depend on a large amount of variables. Therefore, before comparing the various ray tracer versions, the student had to decide what features to include in the comparison, as well as what were the adequate values for the variables. A scene was also designed for the specific purpose of making the performance comparison.

4.1.1 Features and variables

To make a fare comparison, all the ray tracer versions need to include the same features. Since the Numba CUDA version does not support Boolean operations with primitives and the enhancement techniques that improve photorealism, all these features were disabled in the rest of the versions. Attenuation was also disabled in all the versions, because the C++ ray tracer does not include it. Finally, anti-aliasing and shadows were excluded, mainly for simplicity sake, but also to minimize possible algorithmic differences between the C++ ray tracer and the Python versions.

Under these conditions, the performance of the ray tracer depends on four main variables. First, there is the total number of pixels (p) that depends on the size of the rendered image. Then, there is the maximum order of the light rays (d), that refers to the maximum number of times a light ray can reflect or refract in the process of rendering the color of a pixel. Finally, there are the number of scene objects (o) and the number of scene lights (l). This variables indicate the number of iterations on the relevant loops of the ray tracing algorithm. When shadows are disabled, the loop hierarchy can be represented in pseudo code as:

loop over pixels (range $= p$)
  loop over recursive rays (range $= d$)
    loop over objects (range $= o$)
    loop over lights (range $= l$)

Therefore, the time complexity (T) of the simplified ray tracers can be written as:

$$T(p, d, o, l) = O(p \cdot d \cdot (o + l)) \tag{4.1}$$

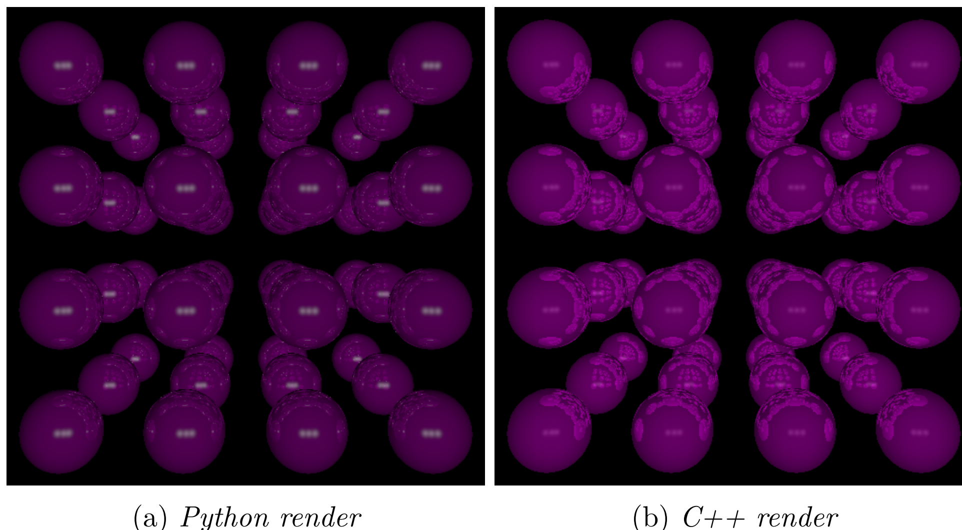
4.1.2 The designed scene

There were two important factors that guided the design of the comparison scene. First, the scene had to be simple, and include as few as possible features to avoid the effect of possible algorithmic differences between the C++ and Python versions. With that in mind, the only kind of objects included in the scene were fully reflective spheres that didn't allow transmission. This choice prevented the execution of the intersection algorithms of boxes, ellipsoids and polygons, as well as avoiding the creation of transmitted rays. The second guiding factor was that the execution time of the fastest version of the ray tracer had to be measurable while not making the execution time of the slowest one too long. This balance was found by placing 48 reflective spheres on a regular grid ($o = 48$) lit by three lights $l = 3$. Besides that, the size of the image was set to 512×512 pixels ($p = 262144$) and the maximum order of light rays to 10 ($d = 10$).

Despite the attempts to minimize the effects of differences between the C++ and Python versions, they were not completely avoided. Figure 4.1 shows the rendering of the same comparison scene by the two versions, and the results are not equal. Some testing indicated that the differences are probably caused by discrepancies in how math is applied to color and the specifics of the implementation of specular reflections. However, this was not confirmed by Jon Sanchez, so the real causes could be other.

4.2 Measurements

After establishing the conditions under which the performance comparison was to be made, it was decided begin the measurements. This section addresses the parameters that were measured, the methods used and the results gotten.



(a) *Python render* (b) *C++ render*

Figure 4.1: *Render of the comparison scene by the Python and C++ ray tracers.*

4.2.1 Time

The standard procedure to evaluate the acceleration provided by a programming tool is to measure the execution or run time of the different versions of a program. Normally, acceleration techniques can only be applied to certain portions or functions of the program, and it is not rare to only measure the run time of this subsets.

The ray tracer algorithm can be divided in three parts: scene loading, rendering and image displaying. In the first, the scene file is read and its information parsed into data structures that the program can use. The rendering process is where the actual ray tracing is performed, by casting the rays and getting the color of each pixel. Finally, the image displaying consists on taking the colors and locations of each pixel and forming the final image. Numba acceleration was only applied to the rendering portion of the program, so only the run time of the rendering portion of the program was measured.

In the Python versions, run time measurements were made using Python's `time` module's `perf_counter()` function. This function provides the best time precision among the timers of the `time` module, with a precision of $150ns$. Its main drawback is that it includes time elapsed during sleep, so if the measured Python process is temporarily set to sleep to execute some other process, the time spent in the second process is included in the measurement. This is generally not a concern when the Python program only uses one CPU core, as the Windows OS uses other cores for other processes. However, when Python uses multiprocessing, external processes might interfere in the run time measurements. In the C++ version, rendering time measurements were made using `std::chrono::steady_clock`, which also includes time elapsed during sleep.

Measuring run time in the C++ and the core Python versions is straight forward; one just needs to take the time at which the rendering process starts the time at which it ends and compute the difference. However, section *How to measure the performance of Numba?* of [24] explains that when measuring the performance of a function accelerated

by Numba one has to be careful with the time JIT compilation takes. The recommended approach is to not include JIT compilation time in the run time. This is done by first calling the function without making any measurements to make Numba JIT compile it, and then calling it a second time to measure its run time after it has been compiled. The measurements of the Numba versions were done following this procedure, with the reasoning that, when measuring the run time of the C++ executable, the compilation time it took to create it is not taken into account either.

This way of measuring the run time of Numba does not give an accurate idea of the real run time improvement that Numba provides. Depending on the application, Numba JIT compilation can take a long time, so the run time of Python programs is not always improved by applying Numba. This is not the case for the implemented ray tracer, but JIT compilation does take a considerable time: about $30s$ in the regular Numba version, $35s$ in the version that includes CPU parallelization, and $15s$ in the Numba CUDA version. Numba does provide a facility for Ahead of Time (AOT) compilation, but it is not compatible with Numba CUDA, and adds further limitations to the acceleration on code that runs in the CPU [24]. For this reason, and due to time constraints, Numba's AOT compilation was not explored in this work.

After setting up the measuring procedures described above, it was decided to make 10 measurements to compute the average of the rendering times of all the ray tracer versions in two cases: when the complete image is rendered, and when only one pixel is rendered. The first case would be indicative of the real performance of each version, and the second case would prevent parallelization, which according to expectations should eliminate the advantages the multi-threaded and GPU versions have.

The results of the measurements can be seen in table 4.1. The complete image rendering times show that Numba greatly accelerates the core Python code, by decreasing the rendering time by two orders of magnitude. Applying multi-threading to Numba also improves performance, but it only reduces the rendering time by a factor of 1.6, which is rather small when compared to the maximal theoretical factor of 8. This is probably caused by the overhead related to spawning parallel processes. C++ multi-threading seems to work better, improving performance by a factor of 3.5 when compared to its mono-thread counterpart. Comparing the mono-thread versions of Numba and C++ indicates that the latter is an order of magnitude faster, so the significant acceleration provided by Numba still is not enough to mach the performance of the C++ language. Lastly, there is the rendering time of the Numba CUDA version, which is surprisingly long considering how well ray tracing lends itself to parallelization. The GPU implementation does outperform the C++ mono-thread version, but not by much, and the multi-thread C++ is faster. There might be many reasons behind this, but the main one is probably the poor use of GPU capabilities. Although the kernel dimensions were modified to optimize the rendering time of the comparison scene, features that can greatly improve the performance of a kernel were not used. The main example of this is shared memory, which when applied correctly reduces the overall time spent in accessing memory.

The "one pixel" rendering times do show the expected elimination of the advantages parallelization offers. In fact, comparing the CPU Numba implementations to the GPU one indicates that a single GPU thread is slower than a CPU thread. Therefore, one

can conclude that the difference between Numba and Numba CUDA complete image rendering times comes from the the amount of parallelization that characterizes the GPU. As a final note, it has to be mentioned that despite rendering the color of one pixel, the rendering process in the python version includes the creation of an array with dimensions of the full image; which explains why multiplying the rendering time of one pixel in the mono-threaded Numba version by the total number of pixels does not give the rendering time of the complete image.

Table 4.1: *Rendering times of ray tracing implementation versions for one pixel and for the complete picture. Variables: scene = comparison.txt,* $p = 512 \times 512$*,* $d = 10$*. The core Python averge for the complete image was computed by only using 5 measurements, due to the long amount of time each takes. All table values are in seconds.*

	one	all
Python	0.0365 ± 0.0005	14800 ± 100
Numba	0.0051 ± 0.0002	149 ± 1
C++	0.000149 ± 0.000006	11.22 ± 0.03
Numba & mult.	0.0041 ± 0.0001	95.6 ± 0.4
C++ & mult.	0.000140 ± 0.000008	3.22 ± 0.02
CUDA	0.0075 ± 0.0003	10.112 ± 0.002

4.2.2 CPU and memory utilization

CPU utilization and memory usage are the next most common measurements used to evaluate the performance of programs. CPU utilization indicates the usage of processing resources by a program during its execution, and it can be expressed by percentages with respect to the total capacity of the CPU. The memory usage of a program is usually measured by taking the peak memory usage value of any process spawned by the program during its execution. There are two kinds of memories that can be measured: Resident Set Size (RSS) refers to the non-swapped physical memory a process has used, and Virtual Memory Size (VMS) indicates the total amount of virtual memory used by the process [30].

Measuring a process's CPU and memory utilization necessarily takes CPU processing and memory resources, so the measurements of one process have to be taken using another process. This was achieved by creating a Python program that uses the `multiprocessing` library to create a process to execute the ray tracers and another process that measures the performance of the first. The second process is appropriately called *measurer*, and makes use of `psutil` library functions to get information about the ray tracer process periodically during run time.

CPU utilization is measured by calling the `cpu_percent()` function, and dividing the result by `psutil.cpu_count()` to get the appropriate percentage. Since the computer used to make the measurements has 8 logical CPU cores, the maximum CPU utilization of a program that runs in a single logical core is 12.5%. Ray tracers make heavy usage of the CPU, so the CPU utilization should be close to 12.5% in the single thread versions and close to 100% in the multi-thread versions.

The measurements shown in figure 4.2 are generally in good agreement with the expectations, however there are a couple things to note. Firstly, the red line of the multi-threaded Numba version clearly jumps from about 12.5% to 100% some time after the beginning of the execution. The time spent before the jump is mostly JIT compilation time, and once compilation is done, the multi-threaded rendering process begins pushing the CPU utilization close to 100%. In fact measurements of single-threaded Numba and Numba CUDA also include measurements taken while JIT compilation occurred. The second notable thing is that the brown line of the Numba CUDA version clearly dips at some points in time, probably due to the GPU taking over to handle the rendering process. However, while the GPU is processing the rendering, the CPU utilization generally remains close to 12.5%. It is unknown what the CPU is processing during this time, or if the measured behavior is common or not, and may require consulting an expert on GPUs.

On the other hand, both RSS and VMS are measured using the `memory_info()` function. In this case, the measurements should show that the Python versions use more memory than the C++ versions, due to the difference approach to memory management each language has. Python handles memory management automatically using a garbage collector, while in C++ it is handled manually which pushes developers to make a better use of it. Table 4.2 shows the results of the measurements, which indicate that C++ performs better according to these measurements. There is also a difference between the core Python version and the Numba CPU versions, with the latter requiring more memory due to the usage of additional libraries. Finally, the reason for Numba CUDA not appearing in the table is that this version uses GPU memory in addition to CPU memory, so it cannot be compared to the CPU versions in a meaningful way.

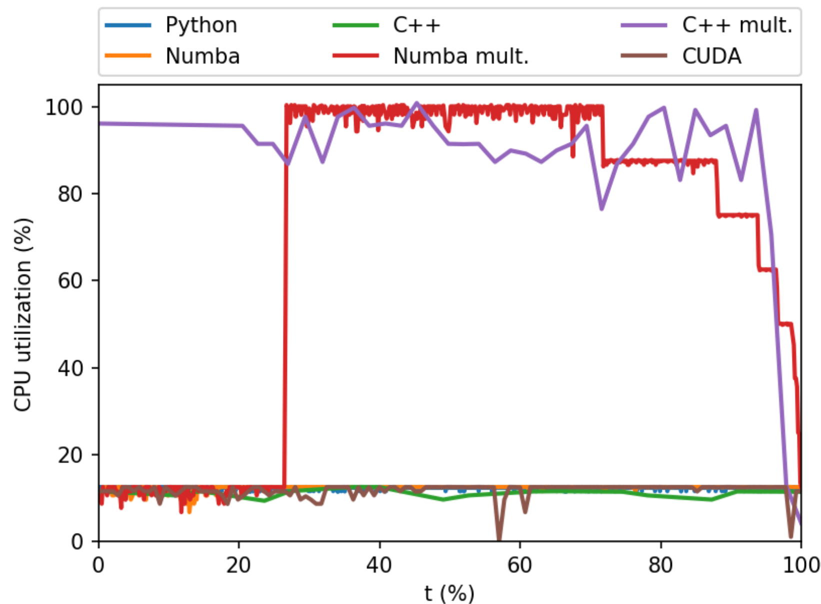


Figure 4.2: *CPU utilization of the ray tracer versions during execution. Variables: scene = comparison.txt, $p = 512 \times 512$, $d = 10$.*

Table 4.2: *Peak virtual and physical memory used by each ray tracer version. Variables: scene = comparison.txt,* $p = 512 \times 512$, $d = 10$.

	RSS (MB)	VMS (MB)
Python	73 ± 2	54.3 ± 0.7
Numba	202.1 ± 0.5	178.0 ± 0.3
C++	8.3 ± 0.1	4.8 ± 0.2
Numba & mult.	200.2 ± 0.6	182.2 ± 0.4
C++ & mult.	8.61 ± 0.05	4.93 ± 0.06

4.2.3 GPU utilization

GPU utilization measures the usage of processing resources by a program during its execution in the GPU. It was measured mainly to make sure that the Numba CUDA version did indeed use the GPU, after seeing that its rendering time was longer that expected. This was done by calling calling `nvidia_smi` module's `nvmlDeviceGetUtilizationRates()` function inside the *measurer* process. The results shown in figure 4.3 indicate that after JIT compilation, GPU utilization suddenly increases to 100% and maintains that value until the rendering process is completed.

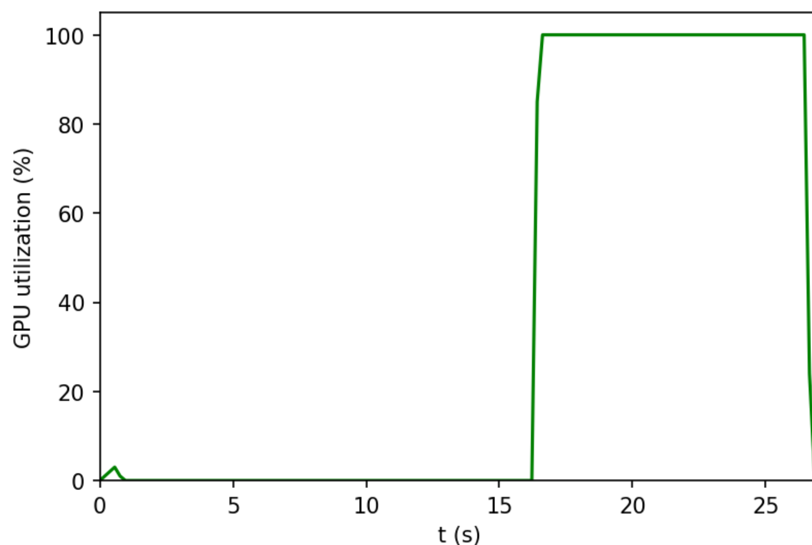


Figure 4.3: *GPU utilization of the CUDA ray tracer during execution. Variables: scene = comparison.txt,* $p = 512 \times 512$, $d = 10$.

5 Conclusions

In this work a basic ray tracer has been developed from scratch using the Python programming language. This has required the application of basic concepts of fields like linear algebra, optics, color theory, programming and computer hardware. Computer graphics combines all this fields and some more in an attempt to recreate the intricate way light interacts with the physical world and the human eye. Ray tracing has proven to be a rather successful attempt at that.

Even the basic ray tracer implemented in chapter 2.2 is remarkably realistic in many aspects. It is easy to understate the precision with which this simple algorithm computes the perspective of the scene, the projection of the shadows and the effects of reflection and refraction. The enhancement techniques that were developed were also simple, but provided great improvements to scene creation and photorealistic rendering. For this reason, this work could serve as material for an introductory course in computer graphics, along with the books on which it is based. It may also be interesting for people who have heard of Nvidia's RTX technology and want to know how this "ray tracing thing" that everybody is talking about works.

As stated many times through the work, the implemented ray tracer has many limitations, so another possibility is to use it as the base of a more complex ray tracer that overcomes these limitations. Improving the Boolean operations with primitives by adding the complement operator and the possibility to mix different operators for example, would greatly expand the range of scenes that can be created. Other features that are relatively simple to include are volume rendering that can be used to render clouds or smoke, and texture mapping that maps 2D images to object surfaces [10]. With respect to illumination, the biggest limitation is that transparent objects cast completely dark shadows, but implementing photon maps to solve this would require much more work. In the end, there is always another feature that can be added to a ray tracer. They are in a certain sense infinitely extendable, as the titles of Peter Shirley's books on the subject suggest: *Ray Tracing in One Week*end, *Ray Tracing: The Next Week* and *Ray Tracing: The Rest of Your Life* [9, 10, 31].

However, the most relevant part of the work may be the application of Numba's acceleration tools and the measured results. Although the measurements in chapter 4 were performed for a ray tracer under very specific conditions, the results show promise in the use of Numba for the acceleration of Python in scientific applications. The performance of C++ is still very superior to the performance of Python, but Numba is in continuous advancement, and the performance gap might become smaller in the future.

To conclude, it has to be mentioned that the most attractive feature of Numba is CUDA GPU programming. The results of the GPU performance were not as good as expected,

but this had more to do the poor use of the tool than with the tool itself. The program could probably been better optimized, and it would probably have benefited from the proper use of features like shared memory. Regardless, being able to program GPUs using Python is by itself remarkable, as it offers the power of a new hardware device to Python developers.

Bibliography

[1] Andrew S. Glassner, *An introduction to ray tracing.* Academic Press, 1989.

[2] Edward R. Freniere and John Tourtellott, "A Brief History of Generalized Ray Tracing." SPIE, 1997.

[3] Per H. Christensen and Wojciech Jarosz, "The Path to Path-Traced Movies," *Foundations and Trends in Computer Graphics and Vision*, vol. 10, pp. 103–175, 2016.

[4] V. V. sanzarov, A. I. Gorbonosov, V. A. Frolov, and A. G. Voloboy, "Examination of the Nvidia RTX." CEUR workshop proceedings, 2019.

[5] "Nvidia Geforce RTX." [Online]. Available: https://www.nvidia.com/es-es/geforce/20-series/rtx/

[6] Jason Hanson, *CS 500: Ray Tracing, Course Notes.* DigiPen Institute of Technology, 2011.

[7] Kevin Suffern, *Ray Tracing from the Ground Up.* AK Peters Ltd., 2007.

[8] Don Cross, "Fundamentals of ray tracing," 2013. [Online]. Available: http://cosinekitty.com/raytrace/

[9] Peter Shirley, "Ray tracing in one weekend," 2018. [Online]. Available: https://raytracing.github.io/

[10] ——, "Ray tracing: The next week," 2018. [Online]. Available: https://raytracing.github.io/

[11] Larry Gritz and Eugene d'Eon, *GPU Gems 3*, 2007, ch. 24, pp. 337–343. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/foreword

[12] I. E. Commission, "Iec 61966-2-1:1999/amd1:2003," IEC Webstore, Tech. Rep., 2003.

[13] "Python global interpreter lock." [Online]. Available: https://docs.python.org/3/glossary.html#term-global-interpreter-lock

[14] "Accelerating python for scientific research." [Online]. Available: https://developer.ibm.com/articles/ba-accelerate-python/

[15] "Cython c-extensions for python." [Online]. Available: https://cython.org/#documentation

[16] "F2py users guide and reference manual." [Online]. Available: https://numpy.org/doc/stable/f2py/

[17] "Jython." [Online]. Available: https://www.jython.org/index

[18] "Ironpython: the python programming language for the .net framework." [Online]. Available: https://ironpython.net/

[19] "Pypy." [Online]. Available: https://www.pypy.org/index.html

[20] "Numba." [Online]. Available: https://numba.pydata.org/

[21] "Pypy. software transactional memory." [Online]. Available: https://doc.pypy.org/en/latest/stm.html#python-3-cpython-and-others

[22] "Python `threading` module documentation." [Online]. Available: https://docs.python.org/3/library/threading.html#module-threading

[23] "Python `multiprocessing` module documentation." [Online]. Available: https://docs.python.org/3/library/multiprocessing.html#the-process-class

[24] "Numba user manual." [Online]. Available: https://numba.pydata.org/numba-doc/latest/user/index.html

[25] "Numba for cuda gpus." [Online]. Available: https://numba.pydata.org/numba-doc/latest/cuda/index.html

[26] Henrik Wann Jensen, *Realistic Image Synthesis Using Photon Mapping.* AK Peters, 2001, ch. 5, pp. 55–65.

[27] Yanhong A. Liu and Scott D. Stoller, "From recursion to iteration: what are the optimizations?" *ACM Press*, 2000.

[28] James D. Murray and William vanRyper, *Enciclopedia of Graphics File Formats.* O'Reilly & Associates, Inc., 1996.

[29] D. Rose, "Rotations in three-dimensions: Euler angles and rotation matrices," 2015. [Online]. Available: http://danceswithcode.net/engineeringnotes/rotations_in_3d/rotations_in_3d_part1.html

[30] "psutil documentation." [Online]. Available: https://psutil.readthedocs.io/en/latest/

[31] Peter Shirley, "Ray tracing: The rest of your life," 2018. [Online]. Available: https://raytracing.github.io/