Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

# Multi-GPU parallelization of a dynamic heat transfer model on the Moon

Mohammad Alhasni

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

# Multi-GPU parallelization of a dynamic heat transfer model on the Moon

| | |
|---|---|
| Author: | Mohammad Alhasni |
| 1st examiner: | Univ.-Prof. Dr. Ulrich Walter |
| 2nd examiner: | Univ.-Prof. Dr. Hans-Joachim Bungartz |
| 1st Assistant advisor: | Dipl.-Ing. (Univ.) Matthias Killian |
| 2nd Assistant advisor: | M.Sc. (hons) Benjamin Rüth |
| Submission Date: | March 29th, 2018 |

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

March 29th, 2018 Mohammad Alhasni

# Acknowledgments

I would like to thank my supervisors Matthias Killian and Benjamin Rüth for their continued support during my Master's thesis. Their feedback was invaluable to the completion of this work.

*"Everything has a natural explanation. The moon is not a god, but a great rock, and the sun a hot rock."*

*-Anaxagoras (c. 450 BC)*

# Abstract

Thermal models in aerospace engineering are constantly growing in complexity to achieve increasingly accurate results and improve safety and equipment design efficiency. One such model is the Thermal Moon Simulator (TherMoS), which computes the transient thermal profile of man-made objects on the surface of the Moon. It uses ray tracing to numerically calculate the radiative heat fluxes across the lunar surface.

Computing the heat fluxes by ray tracing can be computationally intensive due to the rugged geometry of the Moon, the high variation in temperature over small distances across its surface and the recursive nature of radiation. This means that a rather large number of rays needs to be traced to obtain accurate results. To maintain reasonable computation time, it is vital to make effective use of available computing power and resources. In this work, such an effort is undertaken. A parallel multi-GPU implementation of the TherMoS ray tracer is presented. The implementation performs the ray tracing on a cluster of GPUs using the NVIDIA OptiX library.

For the test scenarios simulated, strong and weak scaling benchmarks were performed on a system of 1 to 8 NVIDIA M2090 GPUs. A near linear scaling was obtained, where the compute time was reduced from 67.4 seconds *per iteration* to 8.7 seconds compared to the reference solution. Even greater improvements can be achieved using more modern hardware due to the modular and portable code design. The improved code would now allow for more detailed scenes to be simulated at higher accuracy while maintaining short simulation times.

Furthermore, the radiative heat transfer model was extended to improve its accuracy. This included a new method of solar ray generation, a more detailed material system where per-face thermo-optical properties can be assigned, and a model for specular reflections. Finally, the interface between the ray tracer and the rest of the TherMoS application was optimized to reduce overhead. The old interface based on simple text files was replaced with a new interface that used a memory-mapped file for communication and synchronization. This reduced the non-compute time by a full order of magnitude.

# Contents

# Nomenclature

## Symbols

| | | |
|---|---|---|
| $\alpha$ | Absorptivity | – |
| $\alpha_s$ | Solar azimuth | $^\circ$ |
| $\varepsilon$ | Emissivity | – |
| $\varepsilon_s$ | Solar elevation | $^\circ$ |
| $\theta$ | Emission zenith | $^\circ$ |
| $\lambda$ | Wavelength | $m$ |
| $\rho$ | Reflectance | – |
| $\tau$ | Transmittance | – |
| $\phi$ | Emission azimuth | $^\circ$ |
| $A_{min}$ | Parallelogram origin | $m$ |
| $A_{span}$ | Parallelogram span | $m$ |
| $\vec{d}$ | Ray direction | $m$ |
| $E$ | Total emissive power | $W \cdot m^{-2}$ |
| $F$ | View factor | – |
| $I$ | Spectral intensity of emission | $W \cdot m^{-2} \cdot \mu m^{-1}$ |
| $N_{IR}$ | Infrared rays per triangle | – |
| $N_s$ | Square root of total solar rays | – |
| $\vec{o}$ | Ray origin | $m$ |
| $P$ | Ray payload | $W$ |
| $\dot{Q}$ | Thermal energy rate | $W$ |

| | | |
|---|---|---|
| $q$ | Heat flux | $W \cdot m^{-2}$ |
| $T$ | Temperature | $K$ |
| $t$ | Time | $s$ |

## Constants

| | | |
|---|---|---|
| $\sigma$ | Stefan–Boltzmann constant | $5.67 \times 10^{-8}\, W \cdot m^{-2} \cdot K^{-4}$ |
| $c_0$ | Speed of light in vacuum | $299,792,458\, m \cdot s^{-1}$ |
| $h$ | Planck constant | $6.626 \times 10^{-34}\, J \cdot s$ |
| $k$ | Boltzmann constant | $1.381 \times 10^{-23}\, J \cdot K^{-1}$ |
| $S$ | Solar constant at 1 AU | $1367\, kW \cdot m^{-2}$ |

## Indices

| | |
|---|---|
| $bb$ | Black body |
| $e$ | Emitter |
| $g$ | Generation |
| $in$ | Incoming |
| $out$ | Outgoing |
| $IR$ | Infrared |
| $s$ | Solar |
| $T_k$ | $\mathrm{K}^{\mathrm{th}}$ triangle |

# Part I.

# Introduction and Background Theory

# 1. Introduction

Humans have been fascinated by celestial bodies since the dawn of civilization. They saw shapes in the stars, omens in the comets and deities in the planets. The Moon was no exception. But outside such mythological regard, astronomers knew of the Moon's importance for millenia. They understood its effects on the tides, they hypothesized about its physical rocky nature and attempted to calculate its size and orbit. Such was the interest in the Moon that a journey to it was envisioned in the earliest known work in science fiction [12]. The tale, which was narrated in The True History by Lucian of Samosata in 175 AD, spoke of a boat caught up in a whirlwind that carried it all the way to the Moon, prompting its occupants to join the Moon people in a war against the Sun [22].

Such a journey by a human to the Moon was first made reality in 1969 with the manned Apollo 11 mission. Unlike Lucian's tale, humans relied on their own skills and abilities to reach the Moon. It was the culmination of centuries of scientific and engineering advances. Not only were these advances necessary to overcome the many hurdles faced during the journey, such as escaping the Earth's gravitational pull and guiding the machinery accurately towards the Moon, but also to protect this machinery and its inhabitants once they arrive at their destination. They must be shielded from the harsh environment of the lunar surface.

One of the biggest difficulties in surviving such an environment arises from the extreme temperatures that the lunar landers, rovers and astronauts are exposed to. On the lunar surface, the temperature varies from extremely cold to extremely hot, with estimated surface temperatures ranging from $25\,\mathrm{K}$ to $400\,\mathrm{K}$ [13], with temperatures varying up to hundreds of Kelvin over short distances. For this reason, a lot of effort was put into analyzing the temperature profiles of the lunar surface. The objective was to provide a better understanding of the environment, an understanding which would in turn be used in designing suitable equipments, instruments, landers, rovers and spacesuits. Such an effort was crucial in guaranteeing the safety of the astronauts, survivability of any vital equipment and thus the success of the mission.

The analysis and modelling techniques used historically varied in their methodology, scope and accuracy. Such models date all the way back to 1930 where a simplified static model was developed using infrared telescope measurements [27]. With the advent of computing, models developed in the 1940s onward began using numerical simulations to compute changes in the surface temperature profiles. As the available computational power continued to increase, the use of more advanced modeling techniques such as finite elements became widespread in the 1990s. Over time, the models continued to provide an increased level of detailed results down to the crater and even pebble level. A more detailed review of the developed lunar thermal models can be found in [13].

Throughout the lifetime of the spaceflight programs in the 20th century, numerous problems arose in the different lunar missions as a direct result of the thermal conditions on the lunar surface. For example, the Modular Equipment Transporter (MET) during Apollo 14 faced a problem where one of its wheels was exposed to a temperature lower than the design threshold simply due to being shadowed by the rest of the MET [5]. Also, the Seismic Experiment Package experienced temperatures 28 °C higher than the design threshold due to its radiators being covered by dust and debris. In Apollo 16, the plastic material used in the Cosmic Ray Detector was overheated due to being coated with dust [10]. The Surveyor landers routinely experienced temperatures 25 °C higher than anticipated due to lack of knowledge of lunar surface properties [16]. In Apollo 15, 16 and 17 the batteries of the Lunar Roving Vehicles repeatedly experienced overheating due to insufficient dust brushing on the radiators, despite re-evaluating temperature level predictions after each mission [9].

These are just a few examples that demonstrate the necessity for continued development of more detailed thermal models. This would allow for more accurate temperature predictions, reducing the inherent uncertainty associated with space flight applications.

Traditionally, space vehicles are desined against static worst case thermal conditions, as opposed to a transient approach. Both low and high temperature extremes are taken into account during the design process to ensure that all equipment operate inside that range, within a safety margin. The design scenario would thus assume the vehicles to be operating in shaded area due to craters or boulders. However, such a worst-case scenario would realistically only occur for a short period of time during a mission.

In the case of static lunar landers, this criterion can be acceptable. This design criterion also produces acceptable results when planning missions to planets or moons with relatively forgiving environments. However, in the case of moving equipment such as a space exploration vehicle, coupled with the harsh environment of the Moon, better design criteria must be used. With static worst-case models, the resulting designs would needlessly produce large and heavy vehicles. It is better to optimize the movement of the vehicles so that they can avoid such situations, reducing the temperature gradient between the vehicles and the environment. This can be achieved by replacing the static calculations with a transient design approach where time-dependent local heat fluxes are calculated dynamically. As a result, a more efficient and compact design can be reached, with less uncertainty of the heat transfer experienced by the vehicle, or any other type of moving object.

One example of such a dynamic thermal simulator was proposed by Philipp Hager [13]. In that work, a tool called a Thermal Moon Simulator (TherMoS) was developed to calculate the transient thermal profile of man-made objects on the surface of the Moon. It achieved this by calculating time-dependent local infrared and solar heat fluxes, which were dependent on the position of the Sun relative to the Moon and on local surface geometry, temperatures and material properties. The underlying incentive was to produce high temporal resolution of the core temperature of moving sample bodies.

TherMoS was divided into two main components as shown in Figure 1.1. One component was responsible for computing the radiative heat transfer via the ray tracing technique. The radiation was split into solar and infrared radiation. Solar radiation was emitted by the Sun, whose position was described using elevation and azimuth angles. Infrared radiation was emitted by the lunar surface and the sample body as a function of their temperature. The second component, the solver, would then use the computed radiative heat exchange to solve the heat transfer equation numerically, producing the temperature profile across the sample body and the lunar surface. This followed an iterative approach where the temperatures were continually computed at fixed intervals as the sample moved on a path along the surface.

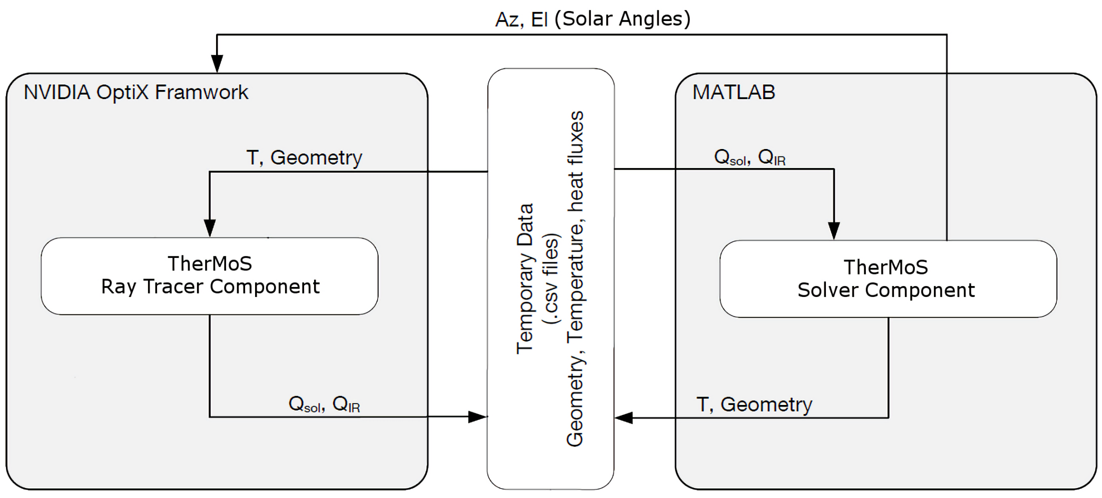


Figure 1.1.: Overview of the interaction between the TherMoS components. Adapted from [13].

This work builds upon the TherMoS model. Specifically, the ray tracing component, where the majority of the overall simulation time is spent. It is a prime candidate for optimization, allowing the performance of TherMoS to be greatly improved.

## 1.1. Motivation

Proposed thermal models continue to increase in complexity in an effort to produce more accurate results. Such results should also be obtainable at reasonable simulation times to allow for efficient development. Fortunately, available computing power and resources also continue to grow, providing the means necessary to meet these increasing computational demands.

One way to utilize such resources is through parallel computing. The performance of established algorithms could be significantly improved by taking advantage of the parallelization capabilities of modern computers. This would allow us to use more complex models and simulate more detailed scenery while maintaining an acceptable computation time. Consequently, more accurate temperature predictions can be produced, reducing the risk of mission failures. It would also allow for more cost-effective and efficient equipment design.

There are many reasons why such lunar thermal models are so computationally intensive. Below is a summary of some of the main difficulties faced when developing algorithms to solve for thermal profiles on the Moon:

- **The rugged geography of the Moon:** The numerous craters scattered across the surface of the Moon coupled with its high reaching mountain ranges causes a lot of occlusion of solar radiation, the main source of heat for the lunar surface. Therefore, a large enough section of the surface must be simulated in order to capture as many of these obstacles as possible. Otherwise radiation that would have ordinarily been blocked by such features would make its way to inaccessible regions, causing them to be over-illuminated.

- **The high variation in temperature across the lunar surface:** Due to the fact that the Moon has almost no atmosphere to trap the Sun's radiation, convective heat transfer is negligible. This means that direct radiation is the main method of heat transfer and so the temperature varies significantly between shaded and unshaded areas during the day. A fine mesh is therefore required to capture this and provide accurate results.

- **The relatively small size of the equipment and astronauts:** A consequence of simulating a large patch of lunar surface is the significant difference in scale between it and any man-made object landed on the Moon. This means that the model will need to make sure that a sufficient level of detail is simulated. Otherwise the temperature profile of any object landed would be physically inaccurate either due to underestimating or overestimating the amount of heat transfer.

- **The recursive nature of radiation modelling:** Incoming radiation does not just stop once it hits a point on the surface. Part of it is reflected back into the scene, which could in turn hit another point on the surface, producing yet more reflected radiation. Reflections are an important aspect that must be captured to produce more accurate results, despite its effect on performance.

## 1.2. Objective

TherMoS spends a significant amount of time performing the ray tracing to determine the solar and infrared heat fluxes. Depending on the complexity of the simulated patch of lunar surface, the ray tracer took between 7 and 40 seconds to run per time step. Scenes with boulders and large craters required more time to trace due to an increase in the number of ray intersections and consequently the number of reflections. In comparison, the other TherMoS component (solver) took less than 0.1 seconds. Therefore, it is quite apparent that optimization efforts should be focused on the ray tracer component.

In the original implementation of TherMoS, the heat transfer by radiation was computed on the GPU using ray tracing. However, it only considered the use of one GPU to perform the computation. And while tracing rays on the GPU provides a rather significant performance speed-up when compared to tracing them on the CPU, using more than one GPU offers an opportunity for further speed-up. It adds another level of parallelism on top of the GPU's inherent parallelism capabilities.

The objective of this work is therefore to investigate the benefits of extending TherMoS to a multi-GPU implementation. This would answer the question of whether such an effort is worthwhile and would allow for more complex models and higher resolution scenery to be simulated, generating more accurate temperature profiles and therefore leading to better designs.

## 1.3. Thesis Outline

Beyond this introduction, the thesis is divided into the following structure:

- Chapter 2 contains background information relevant to this work including the physics of heat transfer and basics of ray tracing.
- Chapter 3 introduces the overall model and describes the single GPU implementation of the ray tracer.

- Chapter 4 extends the single GPU ray tracer to a multi-GPU implementation.
- Chapter 5 provides a description of an optimized interface between the TherMoS ray tracer and solver that handles data exchange and synchronization.
- Chapter 6 presents the results of this work.
- Chapter 7 summarizes the work presented and provides an insight into possible future improvements.

# 2. Background

This chapter contains details of the subjects relevant to this work. The importance of each section depends on the background of the reader. Where necessary, external references are provided to the reader should they wish to seek a more in depth explanation. The chapter is structured as follows:

- Section 2.1 details the physics of heat transfer with focus on radiative heat transfer, which is the subject of this work.
- Section 2.2 describes how radiative heat transfer is translated into a computational model. It provides an explanation of the ray tracing technique, its different forms and the concept of acceleration structures.
- Section 2.3 discusses GPU computing and its potential in comparison to traditional computing on the CPU. It also introduces NVIDIA's CUDA platform.
- Section 2.4 introduces NVIDIA's OptiX framework, which is used to perform ray tracing on the GPU in this work.
- Section 2.5 provides an evaluation of the main methods of inter-process communication, which is necessary for building an efficient interface between the TherMoS solver and ray tracer components.

## 2.1. Fundamentals of Heat Transfer

Heat is a form of energy transferred between bodies at different temperatures. Heat transfer is the science that deals with determining the rate of these energy transfers, or *heat fluxes* (i.e. the rate of heat transfer per unit area normal to the direction of transfer). This depends on the rate of change of temperature along that direction, or the *temperature gradient*, such that heat flows from areas of high temperature to areas of low temperature. The greater temperature gradient is, the greater the heat flux.

This section offers a brief explanation of the physics of heat transfer and is intended for readers with little or no background in the subject. For a more in-depth explanation of the science of heat transfer as a whole, the reader is referred to the following books in literature, from which the material in this section was derived: Fundamentals of Heat and Mass Transfer by Frank P. Incropera [14] and Heat Transfer: A Practical Approach by Yungus A. Cengel [4].

### 2.1.1. Heat Balance Equation

Heat transfer is governed by the first law of thermodynamics, which states that energy can neither be created nor destroyed, but can only be transformed from one form to another. This is translated into an equation of energy balance, which describes the change of energy in a system by accounting for all energy entering and leaving the system, in all its forms. When dealing with heat transfer, it is convenient to formulate a heat balance equation instead, with the conversion of other energies (such as chemical, nuclear or electrical) to thermal energy being lumped into a single term. The heat balance equation is:

$$\frac{dQ}{dt} = \dot{Q}_{in} - \dot{Q}_{out} + \dot{Q}_g \tag{2.1}$$

where $\dot{Q}_{in}$ and $\dot{Q}_{out}$ are the rates of thermal energy $(W)$ entering and leaving the system respectively, and $\dot{Q}_g$ is the rate of heat generation such as via chemical reactions, radioactive decay or heat dissipation from electronic devices.

In any system, heat is transferred in three different ways: conduction, convection and radiation. Conduction is the transfer of heat due to interactions of particles that are in direct contact where energy is transferred from the more energetic particles to the lesser energetic ones. Convection describes the transfer of heat in solid-fluid or fluid-fluid systems due to both random molecular motion and the bulk motion of the fluid. Radiation is the emission of energy by matter at non-zero temperatures due to changes in the electron configurations of its particles.

In the context of this work, which focuses on the ray tracer component of TherMoS, radiation is the only relevant heat transfer method. Conduction is still taken into account in the overall TherMoS simulation by the solver component to calculate the heat transfer along the interior of the Moon. Convection is not considered at all in either the ray tracer or the solver due to the negligible atmosphere of the Moon. Therefore, only heat transfer by radiation will be discussed in detail in the next section.

### 2.1.2. Heat Transfer by Radiation

Electromagnetic radiation occurs due to changes in the electronic configurations of atoms and molecules. This produces oscillating electric and magnetic fields that travel in discrete quanta called photons. Radiation does not require a medium to travel through and can freely propagate in a vacuum at the speed of light. When travelling through a medium, the radiation is attenuated due to absorption or scattering of photons as they interact with other particles in the medium.

This radiation can occur over an infinite continuous range of wavelengths making up the electromagnetic spectrum. Within that spectrum, wavelengths ranging from $0.1\,\mu\mathrm{m}$ to $100\,\mu\mathrm{m}$ are of particular interest since it is waves within that range that are associated with thermal radiation [14]. Thermal radiation is the part of the electromagnetic spectrum that is caused by and affects the temperature of matter. It is made up of the visible light and infrared regions of the spectrum, as well as a part of the ultraviolet region.

Thermal radiation emitted from any surface varies with the wavelength and the direction of emission. These effects are referred to as the spectral distribution and directional distribution of the emitted radiation, respectively. The spectral distribution describes the radiation magnitude as a function of the wavelength $\lambda$. This in turn depends on the surface temperature, which affects both the individual magnitudes and the distribution itself. The directional distribution describes the variation of the radiation intensity as a function of the direction of emission. Both of these distributions must be taken into account when evaluating the heat transfer rate of radiation. Directions are described by a zenith angle $\theta$ and an azimuth angle $\phi$ over a hemisphere as shown in Figure 2.1.

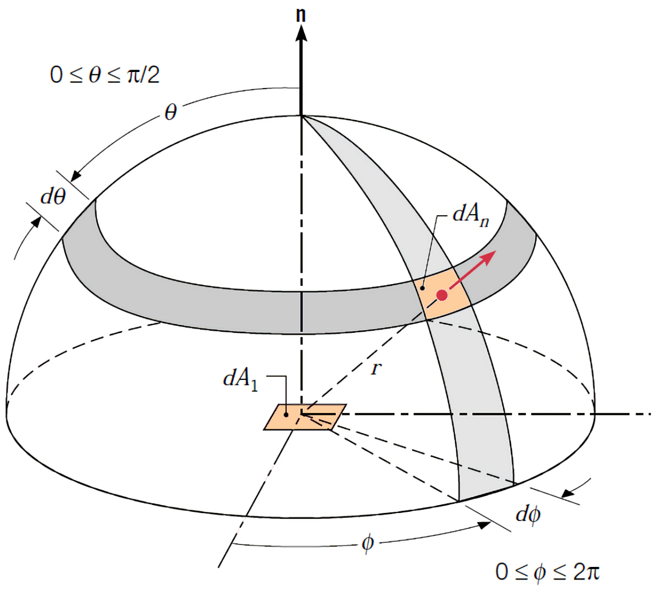


Figure 2.1.: Emission through a hypothetical hemisphere above differential element $dA$. Adapted from [14].

By integrating the spectral and directional distributions of the spectral intensity of emission, denoted by $I_{\lambda,e}(\lambda,\theta,\phi)$, over the space of directions and wavelengths, we can obtain an expression for the total heat flux as follows:

$$E_\lambda(\lambda) = \int_0^{2\pi} \int_0^{\pi/2} I_{\lambda,e}(\lambda,\theta,\phi)\cos(\theta)\sin(\theta)\,d\theta\,d\phi \tag{2.2}$$

$$E = \int_0^{\infty} E_\lambda(\lambda)\,d\lambda \tag{2.3}$$

where $E$ $(W \cdot m^{-2})$ is referred to as the total emissive power.

For a diffuse emitter, the intensity of the emitted radiation is independent of the direction. This means that $I_{\lambda,e}(\lambda,\theta,\phi) = I_{\lambda,e}(\lambda)$ and thus can be removed from the integrand in Equation 2.2. This also provides a reasonable approximation for many surfaces in engineering applications. Performing the resulting integration and substituting it in Equation 2.3 yields:

$$E = \int_0^{\infty} \pi I_{\lambda,e}(\lambda)\,d\lambda \tag{2.4}$$

When trying to solve for the total emissive power, it is useful to begin by analyzing black bodies. A black body is an idealized body that is both a perfect absorber and a perfect emitter. That is, it absorbs all incident radiation irrespective of its wavelength or direction and it emits more energy than any other body at a particular wavelength and temperature. Its emission is also diffuse in nature, which means that the emitted radiation intensity is independent of the direction. For such a black body, the spectral intensity $I_{\lambda,bb}$ is:

$$I_{\lambda,bb}(\lambda,T) = \frac{2hc_0^2}{\lambda^5\left(e^{(\frac{hc_0}{\lambda kT})} - 1\right)} \tag{2.5}$$

where $h = 6.626 \times 10^{-34}\,\mathrm{J \cdot s}$ is the Planck constant, $k = 1.381 \times 10^{-23}\,\mathrm{J \cdot K^{-1}}$ is the Boltzmann constant, $c_0 = 2.998 \times 10^{8}\,\mathrm{m \cdot s^{-1}}$ is the speed of light in a vacuum and $T$ is the black body temperature in Kelvin.

By substituting $I_{\lambda,bb}(\lambda,T)$ from Equation 2.5 in Equation 2.4 and performing the integration, an expression for the total emissive power of a black body is obtained:

$$E_{bb}(T) = \sigma T^4 \tag{2.6}$$

where $\sigma = 5.670 \times 10^{-8}\,\mathrm{W \cdot m^{-2} \cdot K^{-4}}$ is the Stefan-Boltzmann constant. Equation 2.6 is called the Stefan-Boltzmann law, which describes the total radiation emitted over all wavelengths and in all directions as a function of the black body temperature.

The concepts discussed thus far illustrated the behaviour of idealized black bodies, but they can be further generalized to describe the behaviour of real surfaces. This is achieved by defining material emissivity and absorptivity, which are thermo-optical properties that describe the deviation of the behaviour of real bodies from the idealized assumptions.

Emissivity is the ratio of the radiation emitted by the real surface to the radiation emitted by a black body at a specific temperature:

$$\varepsilon(\theta, T) = \frac{\int_0^\infty \varepsilon_{\lambda,T} E_{\lambda,bb}(\lambda, T)\, d\lambda}{E_{bb}(T)} < 1 \tag{2.7}$$

Similarly, the absorptivity is the ratio of the radiation absorbed by the real surface to the radiation absorbed by a black body:

$$\alpha(\theta, T) = \frac{\int_0^\infty \alpha_{\lambda,T} E_{\lambda,bb}(\lambda, T)\, d\lambda}{E_{bb}(T)} < 1 \tag{2.8}$$

The emissivity and absorptivity are assumed independent of $\phi$, but not independent of $\theta$. For emissivity, this leads to different values for diffuse and spectral emission. Similarly, the absorptivity can be specular or diffusive (i.e. dependent on the angle of incidence or not). For spaceflight applications, it is common to use both the diffuse emissivity for the infrared wavelength region and diffuse absorptivity for the spectral range of the Sun [13]. The emissivity and absorptivity are also dependent on the temperature and the wavelength. For a given wavelength and angle of incidence, it holds that:

$$\alpha(\theta, \lambda) = \varepsilon(\theta, \lambda) \quad \forall\theta\, \forall\lambda \tag{2.9}$$

Using the introduced concept of emissivity, the total emissive power for a real surface (commonly referred to as a grey body) can be written as follows:

$$E(T) = \varepsilon_\lambda \sigma T^4 \tag{2.10}$$

When this radiation reaches a surface, part of it will be absorbed, part of it will be reflected and part of it will be transmitted through. The sum of these contributions must be unity as per the law of conversation of energy:

$$\alpha(\lambda, \theta) + \rho(\lambda, \theta) + \tau(\lambda, \theta) = 1 \tag{2.11}$$

The reflectance $\rho$ consists of both diffuse and specular reflectance, which depend on the material properties. The lunar surface can be considered as a purely diffuse reflector while it can be useful to consider both reflectance types for man-made objects. The transmittance $\tau$ is only relevant for transparent or semi-transparent materials, which is not the case in spaceflight applications where most materials used are opaque, and so $\tau$ can be assumed zero.

By combining the effects of heat emission and absorption, the rate of radiative heat transfer between two surfaces can be obtained as follows:

$$q_{ij} = -\alpha_j \varepsilon_i \sigma F_{ij} (T_j^4 - T_i^4) \tag{2.12}$$

where $q_{ij}$ is the heat flux ($W \cdot m^{-2}$) from the emitting surface $i$ to the receiver surface $j$. Therefore $\alpha_j$ is the absorptivity of the receiver surface, $\varepsilon_i$ is the emissivity of the emitter surface, $\sigma$ is the Stefan- Boltzmann constant and $F_{ij}$ is the view factor between the two surfaces.

The view factor, which is also sometimes called the configuration or shape factor, is defined as the fraction of the radiation leaving surface $i$ that is intercepted by surface $j$ [14]. It provides a geometric relation between any arbitrary surfaces, taking into consideration their shape and orientation in space as shown in Figure 2.2. For the two surfaces $A_i$ and $A_j$ that are diffuse emitters and reflectors (with uniform radiosity), the view factor can be calculated as follows:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos(\theta_i)\cos(\theta_j)}{\pi R^2} \, dA_i \, dA_j \tag{2.13}$$

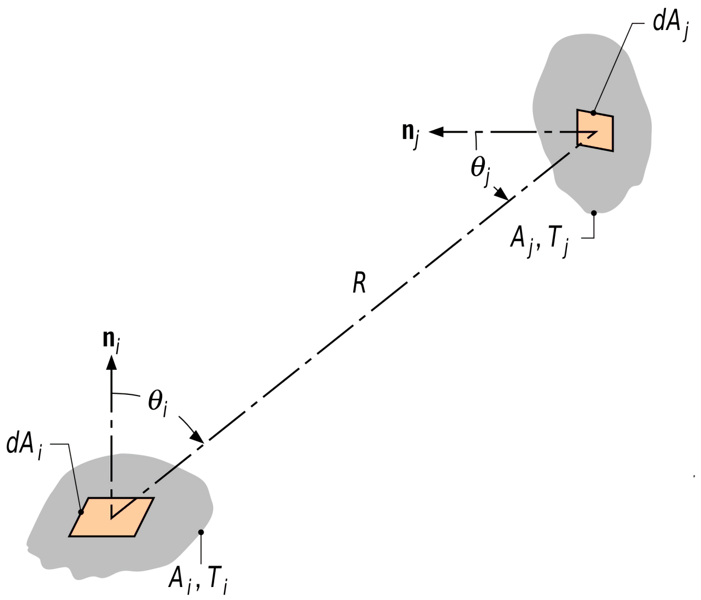


Figure 2.2.: View factor used in calculating the radiative heat transfer between elemental surfaces $dA_i$ and $dA_j$. Adapted from [14].

where $R$ is the length of the line connecting the elemental surfaces $dA_i$ and $dA_j$ and the angles $\theta_i$ and $\theta_j$ are the polar angles between this line and the surface normals $n_i$ and $n_j$ respectively. It is important to note that, in general:

$$F_{ij} \neq F_{ji} \tag{2.14}$$

To obtain one view factor from another, the following relation should be used instead, which is termed the reciprocity relation:

$$A_i F_{ij} = A_j F_{ji} \tag{2.15}$$

## 2.2. Radiation Modelling

When modelling heat transfer by radiation for large and complex scenes, the computation of view factors becomes impossible to do analytically. Additionally, the view factor computation discussed in the previous section becomes invalid if specular surfaces are involved and the radiation exchange factors must instead be determined numerically.

A solution to this problem can be obtained using ray tracing, which would allow us to avoid explicitly determining the view factors [13]. Ray tracing can be used to create an accurate description of the geometric relationships between the different surfaces in a scene. This is a statistical approach and so the results are approximate in nature, with the amount of rays used determining the accuracy of the final solution.

### 2.2.1. Monte-Carlo Methods

Ray tracing is considered a Monte-Carlo method. Monte-Carlo methods are computational algorithms that rely on repeated random sampling to produce an approximate solution of a problem. This problem is usually too difficult or impossible to solve using other methods. In computer graphics, ray tracing is used to solve the rendering equation [15], where the light reaching any point on a surface is dependent on all other points in the scene. The nature of the equation is therefore infinitely recursive with an infinite dimensionality. This means that no analytical solutions exist except for trivial cases.

This is where Monte-Carlo methods have an advantage. They can be used to perform numerical integration and their convergence rates are independent of the dimension of the integration domain, making them suitable for approximating multi-dimensional integrals when the space dimension gets large [28]. For a given multi-dimensional definite integral:

$$I = \int_{\Omega} f(x)\, dx \tag{2.16}$$

The Monte-Carlo approximation can be obtained by sampling points $X_1, X_2, ..., X_N$ on $\Omega$ and calculating the mean:

$$I \approx I_N = \frac{1}{N} \sum_{i=0}^{N} f(X_i) \tag{2.17}$$

Despite their robustness, Monte-Caro methods suffer from a slow convergence rate, which is $\mathcal{O}(N^{-1/2})$, where $N$ is the number of samples taken.

### 2.2.2. Ray Tracing

Originally introduced by Appel [1] in 1968 and later improved upon by Whitted [35] to use recursive ray tracing, this method was mainly used for image rendering, especially in video games and movies. However it has since proved useful in scientific simulations and visualizations.

Ray tracing as a rendering technique is used to produce a two-dimensional image of a scene. This is achieved by generating a number of rays and casting them into the scene. Any ray used for ray tracing can be described by an origin $\vec{o}$ and a direction $\vec{d}$ in space. The ray is then described by the following parametric equation:

$$R(t) = \vec{o} + t\vec{d} \tag{2.18}$$

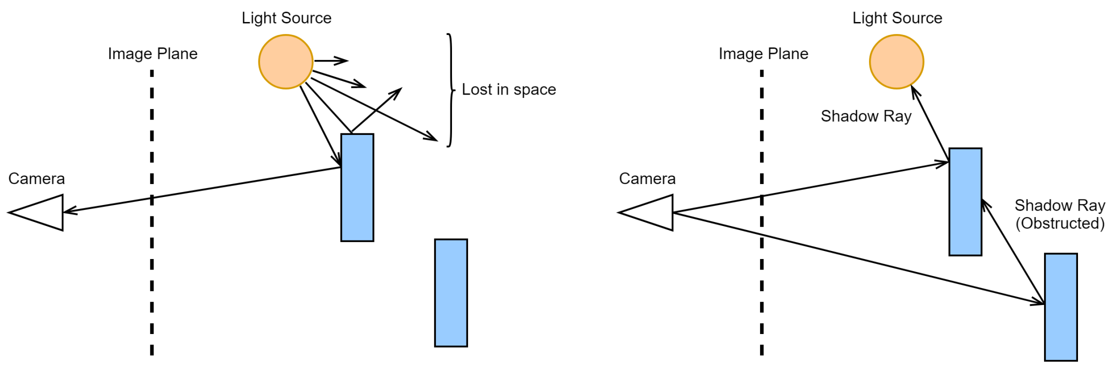
where $t$ determines the distance the ray has travelled along its direction. As they travel, the rays interact with the underlying geometry of any objects in the scene, causing them to be scattered and absorbed in varying degrees depending on the optical properties of the objects. The contributions of all the rays reaching the viewing point are then accumulated and used to produce the final image.

In essence, ray tracing is based on a simplified model of real-life photons. They are emitted from an origin and travel in a perfectly straight path until they intersect with an object. Theoretically, if no (opaque) object exists in their path, the photons would continue to travel indefinitely in space. Practically, the rays are given a maximum range to ensure they terminate once they leave the scene bounds. If a ray intersects a scene object, the material properties of the intersected object determines how much of the ray is absorbed, reflected or transmitted through in the case of transparent materials.

When such an intersection occurs, after a part of the ray's energy is absorbed by the object, the rest is scattered again into the scene. This is achieved by launching secondary rays from the intersection point in order to simulate optical reflections. This is a recursive process where if one of the reflected secondary rays intersects with an object, then it would in turn produce more rays. This process can be repeated until either the primary ray is reflected for a fixed maximum number of times, or if the amount of energy left in the scattered ray is small enough to be neglected.

When tracing a three-dimensional scene, the rays are usually collected on a two-dimensional surface. This surface can either be an image plane representing a camera's point of view into the scene, similar to a photograph, or it can represent a physical surface in the scene where all incident light from the rest of the scene is gathered. The first definition is usually of interest to video game developers, while the latter definition is of more importance for scientific simulations. Regardless of the nature of the surface, it can be simply stored as a single data buffer in memory.

There are two main variations used in implementing ray tracing: forward ray tracing and backward ray tracing. The two methods differ based on the source from which the rays are generated as shown in Figure 2.3.



**(a)** Forward ray tracing **(b)** Backward ray tracing

Figure 2.3.: Representation of the two ray tracing techniques.

### Forward Ray Tracing

Forward ray tracing is more or less what happens in nature where rays are generated directly from a light source. Rays would then propagate around the scene, possibly intersecting with different objects. Eventually, a ray would either reach the image plane (and in extension the camera) where its contribution would be noted, or it would drift away and out of the scene. The contribution depends on the properties of the intersected object and the light source.

However, since light is emitted equally in all directions, this causes a significant amount of rays to be lost in space. And since it is not possible to determine if a ray would reach the scene or not beforehand, those rays will need to be traced fully, only to be terminated at the scene borders. Even if an intersection with an object occurs, there is no guarantee that the resulting reflected ray would find its way to the image plane.

At this point it becomes clear that forward ray tracing requires substantial computational time and resources to produce desirable results. This is especially true of outdoor scenes where few surfaces exist for rays to interact with. A large number of rays would need to be traced to compensate for the ones that are lost and create an accurate representation of the scene. The number of rays needed would scale with the size of the scene and the required output accuracy.

#### Backward Ray Tracing

Backward ray tracing attempts to approximate nature by reversing the direction of the rays. Instead of being launched from a light source, rays are launched from the image plane itself. Each pixel in the image plane generates a single ray, called a primary ray. The rays are typically launched in the view direction of the camera, which lies behind the plane.

If a ray intersects with an object, a secondary ray is launched from the intersection point. This is usually referred to as a shadow ray. The shadow ray is launched directly towards the light source. If it reaches the light source, the contribution of the ray is calculated and accumulated onto the image plane pixel that launched the ray in the first place. If the shadow ray is obstructed by another object on its way to the light source, that would mean that the original intersection point was in shadow and so no light contribution is added to the image plane.

Since only one ray is being launched from each image pixel position, backward ray tracing performs much faster than forward ray tracing. It is easily able to achieve real-time performance, even when using large and complex scenery. The number of rays launched would not scale when using larger scenes, it would only scale with the image resolution. This is why it is used in applications such as video games.

Despite the significant performance speed-up, backward ray tracing would not be suitable for scientific applications. Its simplified approach simply cannot provide the necessary level of detail needed to accurately represent natural phenomena, which is the deciding factor in scientific simulations. This makes forward ray tracing the more suitable option.

#### Acceleration Structures

As the number of objects in the scene increases and as the underlying geometry becomes more complex, performing ray intersection tests becomes more computationally prohibitive if a naive implementation is used. This is true for both forward and backward ray tracing methods. This is because every ray will have to be intersected with every triangle in the scene, resulting in a complexity of $\mathcal{O}(n_{rays} \cdot n_{tris})$. This is far from optimal.

This can be overcome by using acceleration structures, which are also known as bounding volumes. These are geometric structures that encapsulate a scene object, such that intersection tests are performed on these structures first. If an intersection occurs, then the actual ray intersection tests with the object triangles underneath are performed. This dramatically improves performance and allows for more complex scenes to be traced.

Acceleration structures typically used are simple geometrical shapes such as spheres or axis-aligned cuboids since their intersection tests are trivial. More complex shapes can be used if necessary, although this can be avoided by dividing a scene object into a number of segments and wrapping each segment in a different structure. It is important for a structure to fit a scene object as tightly as possible in order to avoid false positive intersection results, which would cause the underlying object triangles to be unnecessarily tested.

The acceleration structures are usually placed into a tree data structure. This would allow for an even better performance to be achieved. Trees are non-linear data structures that store information in a hierarchy of nodes. They are traversed recursively and allow faster searches than linear data structures. By placing the various acceleration structures into such a tree, the amount of intersection tests performed is reduced to $\mathcal{O}(n_{rays} \log n_{tris})$. Commonly used data structures for ray tracing are k-d trees, Octrees and Bounding Volume hierarchies (BVH).

## 2.3. General Purpose Computing on Graphics Processing Units (GPGPU)

The graphics processing unit (GPU) has traditionally been used for rendering computer graphics and image processing at interactive frame rates in real-time. These applications are computationally intensive and massively parallelizable, which is reflected in the architectural design of the GPU, especially in contrast to the central processing unit (CPU).

A CPU is made up of a few cores which are optimized for processing data sequentially, while a GPU is made up of thousands of smaller and more efficient cores specifically designed for performing many tasks simultaneously. These GPU cores run slower than CPU cores because of the difference in design philosophy. While the basic principle of the CPU is to run the system and any applications as quickly as possible, the GPU is more concerned with throughput than latency. This is the consequence of the human visual system which operates on six orders of magnitude slower than modern processors, meaning that the latency of individual operations are unimportant as long as the whole dataset is processed in parallel [25].

### 2.3.1. GPU performance

In other words, the CPU excels at performing complex computations on a small set of data, while the GPU excels at performing simple computations on a large set of data. For scientific computations with large datasets and parallelizable algorithms, the GPU has the potential to achieve a significant performance speed-up when compared to the CPU, as shown in Figure 2.4 below.

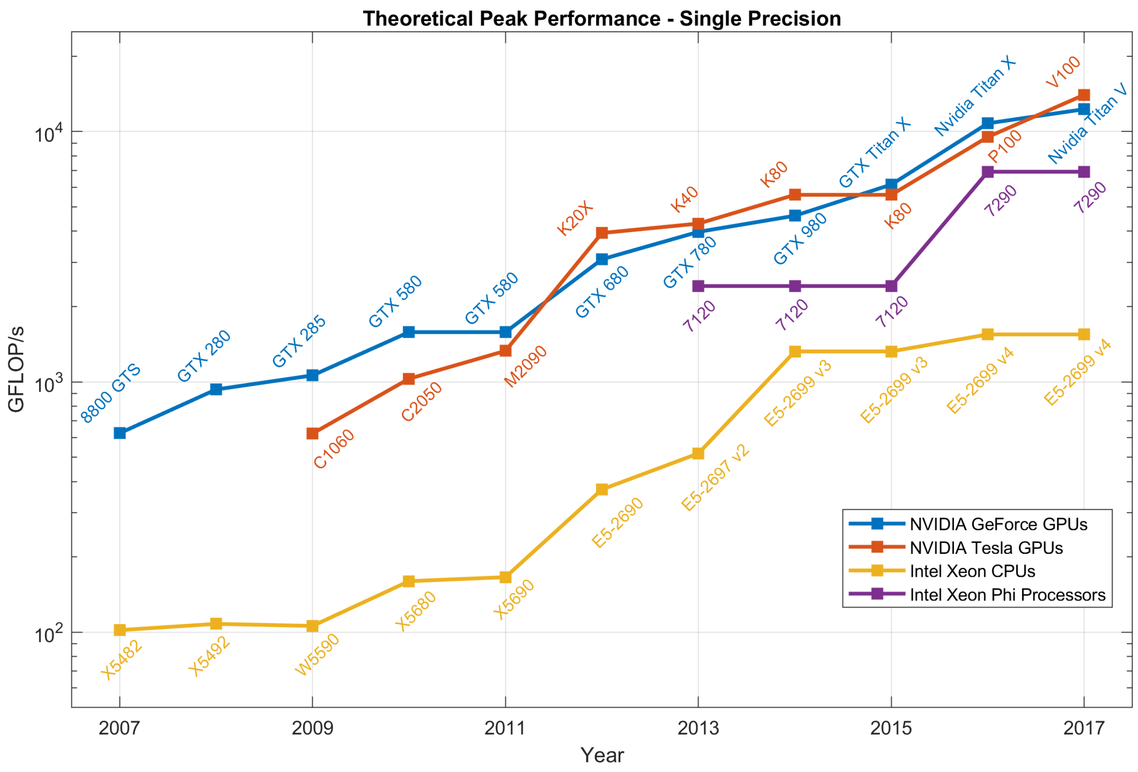


Figure 2.4.: Comparison of single-precision theoretical peak performance of modern NVIDIA GPUs and Intel processors [29].

The figure shows a comparison between different commercial NVIDIA GPUs and Intel CPUs produced in the last decade with respect to their theoretical maximum performance. The metric used is GFLOPs, which stands for Giga Floating Point Operations Per Second. Only values for single precision floating point arithmetic are provided since that is what is used in this work. GPU performance is for the most part about a full order of magnitude greater than CPUs. This edge in performance makes GPUs highly attractive for running parallel algorithms.

The GPU pipeline is made up of a set of stages, or programs, that each perform a specific task such as vertex processing, primitive assembly or rasterization. Each of these GPU programs operate on a single program, multiple data (SPMD) model. This means that the rendering (or compute) elements are split up and processed simultaneously by the different GPU threads, where each of the threads operate on a different set of input. The stream of elements in the scene would thus go through this pipeline one stage at a time such that each element is independent from the rest.

Traditionally, these programs were essentially fixed black boxes and could not be modified by the user, limiting the GPU's applications beyond graphics rendering. However, this changed in the early 2000s with the introduction of the programmable pipeline which gave users the ability to customize the operation of many of the pipeline stages by writing the respective programs themselves [18]. This increased flexibility paved the way for general purpose computing on the GPU.

### 2.3.2. CUDA

Even with the introduction of the programmable pipeline model, the general purpose computing abilities of the GPU were still rather limited. The user faced restrictions in memory access, input data formats, floating point support and debugging tools [30]. The user also had to rely on APIs designed specifically for graphics rendering such as DirectX or OpenGL, as well as shader languages such as HLSL or GLSL.

To overcome these limitations, NVIDIA released CUDA in 2007. CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and programming model developed for general computing on GPUs [6]. It is based on the industry standard C programming language. In CUDA, both the CPU (referred to as host) and the GPU (referred to as device) are used together. Code on the host-side is responsible for allocating and managing memory on both the host and the device, as well as launch functions (referred to as kernels) on the device. Code on the device-side performs the actual computation where the kernels are executed on the different GPU threads in parallel. The results can then be transferred back to the host for output or further processing.

CUDA introduces many features that improve the GPU's general purpose computing abilities such as arbitrary read and write memory access which is necessary for any scatter algorithm. It provides a per-block software managed cache (shared memory) which allows fast communication between threads, as well as a uniform memory model that allows memory to be shared between the CPU and GPU, improving performance and reducing programming complexity. As such, CUDA is now widely used in the scientific community in a variety of fields such as computational fluid dynamics, medical imaging and environmental science [30].

## 2.4. OptiX

OptiX is a general purpose ray tracing engine with a programmable pipeline developed by NVIDIA [26]. The basic philosophy of OptiX is to build a framework around a set of operations which are programmable by the user. It does not impose any high-level rendering mechanisms, allowing the user to construct a specialized program to fit their problem.

The OptiX engine is made up of two APIs, a host-side and a device-side API. The host API is responsible for setting up the application on the CPU. This is done by creating a *Context* object, which is used for connecting graphical devices, building the scene as well as configuring and launching ray programs. The API is written in C, with a C++ interface available if preferred. The device-side API is responsible for executing the ray tracing kernel, which perform ray generation, traversal and intersection. It uses the CUDA driver API. Figure 2.5 shows the control flow of the OptiX pipeline. It consists of a number of user-defined programs written in the device API and compiled using the host API. The different programs shown in Figure 2.5 are summarized as follows [26]:

**Ray generation** programs are the main entry point into the ray tracing pipeline. When the ray tracing is started after a call to *rtContextLaunch*, the program is instantiated across all the different GPU threads available. Each thread would generate a ray in a manner defined by the user. Every ray would carry information such as its starting position, its direction and a user-defined structure containing any number of properties. This provides a generic approach that gives the user freedom and flexibility. Every ray type is associated with a different ray generation program, allowing further customization.

**Intersection** programs carry out intersection tests between the rays and the scene geometry. If a ray successfully intersects with an acceleration structure, the intersection program is invoked. The program must check whether any of the geometry primitives were intersected. Geometry primitives can be anything from triangles to spheres to fractal geometries.

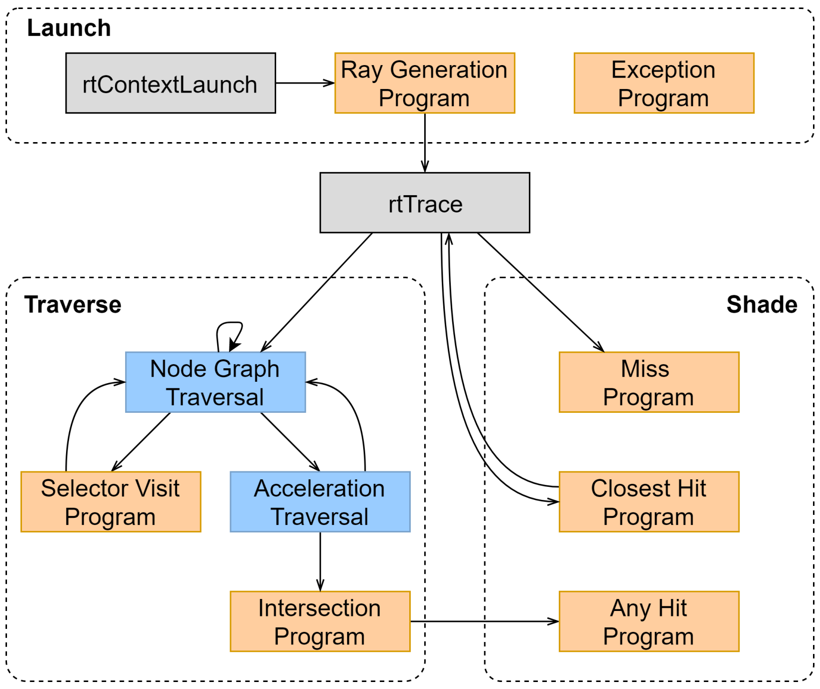


Figure 2.5.: The OptiX pipeline. Adapted from [26]. Grey boxes represent host-side function calls, orange boxes represent device-side CUDA programs written by the user and blue boxes represent internal OptiX algorithms.

**Bounding box** programs are used to compute the bounds for any chosen primitive type (not shown in Figure 2.5).

**Closest hit** programs are called the first time a ray intersects with a primitive.

**Any hit** programs are called every time a ray intersects with a primitive, even if it is occluded by other primitives.

**Miss** programs are called when a ray does not intersect any primitives in the scene.

**Exception** programs are called when the kernel execution encounters an unexpected condition, usually denoting an error.

**Selector visit** programs provide an adaptive graph traversal mechanism. This allows different parts of the scene to be traversed at lower or higher precision than others.

OptiX is a mature product with an active support community and a steady stream of performance improvements. In addition to that, it provides a versatile and fast ray tracing capability, lightweight scene management and an abstraction of the low-level implementation behind a clean interface. All of this makes it an ideal candidate to be used in this work.

## 2.5. Interprocess Communication

In order to create a more efficient and flexible interface between the TherMoS components, the old text file interface was replaced with a more capable interface. This new interface makes use of Interprocess Communication (IPC) to achieve data transfer and synchronization. The specific details regarding the difference between the interfaces and the new changes introduced will be discussed in Chapter 5. In this section a brief summary of IPC is provided for readers without a background on the topic.

IPC is the name given for any mechanism that allows the exchange of data between different processes. These mechanisms can be useful when dealing with a complex application that divides its tasks between various components, which could be completely independent programs written in different languages. IPC would provide these programs with the necessary means for sharing information, allowing them to run in parallel in an efficient and modular fashion. A summary of the main established IPC methods is as follows [11][34]:

**Pipes** are used for unidirectional communication between two processes. A pipe transfers a limited amount of data in a first in, first out (FIFO) fashion. When writing to a full pipe, the writer process is blocked until the pipe is able to receive more data. Similarly, the reader process is blocked when reading from an empty pipe until data is written to it. This mutual exclusion of access allows for automatic synchronization between the writing and reading processes.

**Sockets** are used for bidirectional communication between processes on the same machine or on different machines connected to the same network. The data sent over a socket is split into smaller chunks called packets. These packets are transmitted using a protocol, such as UDP or TCP/IP, which determines how the data is addressed, transferred and received. Synchronization is also a characteristic of the protocol used.

**Message Queues** are used for sharing information between different processes that do not necessarily need to be related or aware of each other. They allow asynchronous multiplexing of data from multiple processes. Data to be exchanged is packaged in a predefined message structure which is inserted into a message queue maintained by the operating system. Any process can write a message to the queue, which can then be read by one or multiple reading processes. A message is associated with a type that the reader processes can use to read the message in a FIFO fashion from the set of messages sharing the same type.

**Shared Memory** allows multiple processes to share a block of virtual memory space. A single process requests the allocation of a shared memory segment from the operating system, then all processes request a one-to-one mapping between that shared memory and a segment of their own local memory address space. From the perspective of each process, accessing this mapped memory is no different from accessing the rest of its memory addresses. However, the mapping would allow any write operations performed by a process to that mapped memory to be automatically seen by all the other processes. Further steps must be taken to synchronize data access between the processes since shared memory does not have a built-in synchronization capability.

**Memory-Mapped Files** are used to map a physical file to the virtual memory address space of several processes, in a similar manner to shared memory. Once the file is created on disk by a single process, all processes request a one-to-one mapping of the file or a section of it to their own memory address space. Consequently, any changes performed by a process automatically update the file content as well as the data seen by the other processes.

# Part II.

# Implementation

# 3. Radiative Heat Transfer on the GPU

In this chapter the modelling of the radiative heat transfer on the GPU is described. The implementation is based on the work by S. Nogina [19] and uses the OptiX API to calculate the radiation. The computation is divided into two main parts, solar radiation and infrared radiation. Both parts are computed using ray tracing, with the output being a distribution of heat fluxes across the simulated scene. The ray tracing code is written in C++ and uses OptiX version 5.0.0. The chapter is divided into the following sections:

- Section 3.1 presents an overview of the ray tracer and its internal components as part of the larger TherMoS tool.
- Section 3.2 provides a description of the scene layout used and how it is represented in OptiX.
- Section 3.3 describes the solar radiation model.
- Section 3.4 describes the infrared radiation model.
- Section 3.5 explains the modelling of ray reflections for both radiation types.
- Section 3.6 details the exchange of data between the TherMoS ray tracer and solver components.

## 3.1. Model Overview

TherMoS is used to compute a dynamic thermal profile of a moving sample object on the Moon. Each simulation is performed over a period of time where the Sun is moved along its orbit at every iteration. Similarly, the sample object is moved along a path on the lunar surface. The sample object and the region of terrain around it are simulated together to account for the heat transfer between them, in addition to the heat absorbed from the sun.

TherMoS performs this simulation using two main components: the ray tracer component and the solver component. The ray tracer is responsible for calculating the heat fluxes on the lunar surface due to solar radiation and infrared radiation, while the solver, which is written in MATLAB, is responsible for solving the thermal system and computing the surface temperatures. The inner workings of the solver component are not the subject of this work and its implementation details will not be discussed.

In the first iteration of the simulation, an initial thermal profile is used by the ray tracer to compute the heat fluxes across the scene. The heat fluxes are then sent to the solver where they are used to compute the new surface temperatures. These temperatures are then sent back to the ray tracer to be used as input to the next iteration. This process is repeated until the desired time period is simulated. A flowchart depicting the interaction between the two components during a single iteration of the simulation is provided in Figure 3.1 below.

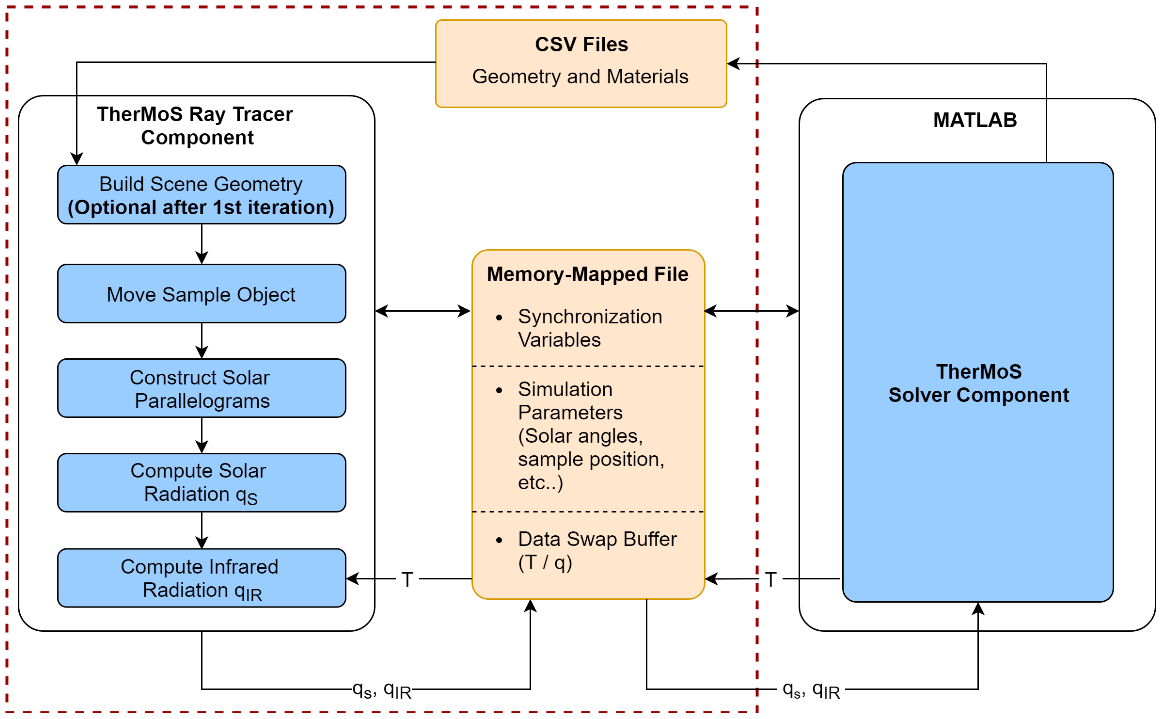

CSV Files
Geometry and Materials
TherMoS Ray Tracer Component
Build Scene Geometry (Optional after 1st iteration)
Move Sample Object
Construct Solar Parallelograms
Compute Solar Radiation $q_S$
Compute Infrared Radiation $q_{IR}$
Memory-Mapped File
Synchronization Variables
Simulation Parameters (Solar angles, sample position, etc..)
Data Swap Buffer (T / q)
MATLAB
TherMoS Solver Component
T
T
$q_s$, $q_{IR}$
$q_s$, $q_{IR}$

Figure 3.1.: Overview of a single iteration of TherMoS with focus on the ray tracer. The ray tracer (left) and its interaction with the solver (right) are the subject of this work and are highlighted in red.

The interaction between the ray tracer component and the solver component is achieved using *.csv* files and a memory-mapped file. The *.csv* files contain geometry and material data necessary to construct the scene (terrain and sample object). This scene is constructed once at initialization, but can also be reconstructed during the simulation when different geometry is to be simulated. The memory mapped file is used for transferring simulation results back and forth every iteration. It is also used for synchronization between the two components.

The ray tracer component is split into five main subcomponents. The first subcomponent is responsible for reading and building the scene geometry in OptiX. The second subcomponent is responsible for moving the sample object along the lunar surface. The third subcomponent constructs the solar light source (parallelogram) responsible for generating the solar rays for tracing. Such a separate light source is not required for infrared rays since they are generated directly from the terrain and sample triangles.

The last two subcomponents are responsible for simulating the radiative heat transfer via ray tracing. The solar subcomponent requires no input besides the scene data and solar light source, while the infrared component also requires the temperature values generated by the TherMoS solver in the previous simulation iteration. The solar and infrared subcomponents produce the heat fluxes $q_S$ and $q_{IR}$ for every triangle in the scene.

## 3.2. The Scene

The scene is made out of two meshes, a large patch of lunar terrain and a sample object placed on the patch surface. The sample object could be a planetary exploration rover, an astronaut or any kind of man-made object to be placed on the Moon. It could be either static or dynamic. The meshes are made up of triangle primitives and are assigned different materials such that each primitive is associated with unique values. The materials are used to represent the thermo-optical properties of the underlying geometry, specifically the optical absorptivity, emissivity and reflectivity.

Since simulating the entire lunar surface is computationally prohibitive, it is instead divided into a number of patches. The use of patches is acceptable since radiation from far away regions of the Moon would not reach the sample both due to self-occlusion and the curvature of the surface. Patch dimensions can be up to $100\,\mathrm{km} \times 100\,\mathrm{km}$ in size, with the resolution of a patch affecting the overall ray tracing accuracy. If the sample object moves during the simulation, the terrain patch can be replaced with a new patch further down the sample's direction of motion. This requires the scene to be rebuilt in OptiX.

Ideally, a high resolution mesh would be used to represent the entire patch to improve the accuracy of the results. However, this would in turn increase the simulation time. A compromise is reached by using different resolutions across the patch. The region around the sample, which is usually the centre, is subdivided into a finer resolution while the rest of the mesh is kept at a lower resolution as shown in Figure 3.2. This level-of-detail technique is used to increase the ray tracing accuracy around the sample, which is necessary due to its small size in comparison to the surface patch. If the difference in size between the sample and terrain triangles is too large, the sample triangles would be exposed to unphysically large levels of radiation. For the same reason, the difference in resolution within the terrain mesh should be chosen carefully. A size ratio of 1:100 to 1:1000 between the terrain mesh triangles is used in this work.

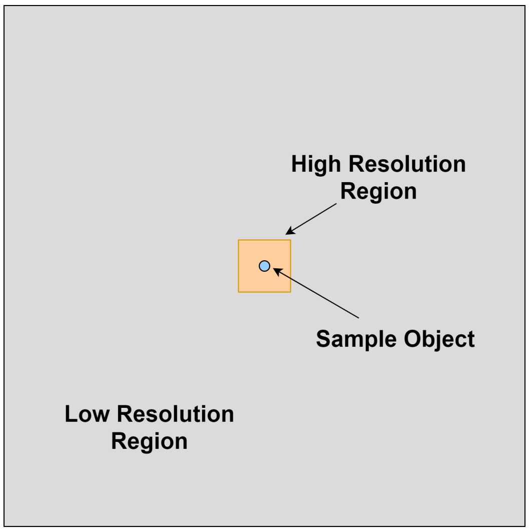


Figure 3.2.: Top view of a typical scene layout. The terrain patch contains a region of high resolution at the centre, inside which the sample object can move freely. Figure not to scale.

### 3.2.1. Acceleration Structures

To improve performance, an acceleration structure is assigned to both the terrain patch and the sample object. The type of acceleration structure used would necessitate a tradeoff between build time and traversal speed. Figure 3.3 shows the different acceleration structure types provided by OptiX. At first look, the TrBvh (Treelet Reordering Binary Volume Hierarchy) structure might seem the best option, providing high performance in construction and traversal. For rendering applications such as video games with fast moving cameras and demand for real-time response, this is optimal. However, in this simulation that is not the case.

The terrain patches are not exchanged often due to their large size, allowing the sample to move freely for many iterations before another patch is needed. On the other hand, the sample does move frequently and is positioned and rotated differently at every iteration. If the transformations are applied to the underlying geometry, it will invoke an acceleration structure rebuild. This would make the use of TrBvh attractive. However, a more efficient solution is to apply an OptiX transform node to the sample. This would require only a 4x4 matrix to be adjusted every simulation iteration, instead of changing the entire sample geometry.

All of this means that the terrain acceleration structure is rebuilt sporadically, while the sample acceleration structure is never rebuilt. Therefore for this case the SBvh (Split Bounding Volume Hierarchy) is a better alternative to TrBvh since build time is inconsequential and the traversal performance is 9% higher than TrBvh [7].

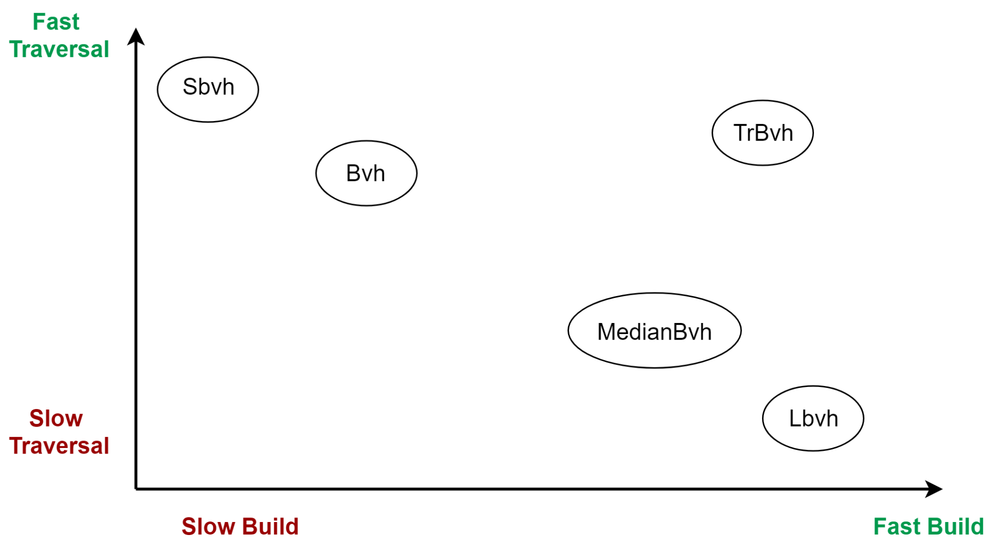


Figure 3.3.: Comparison of the different acceleration structure types provided by OptiX. Adapted from [7].

### 3.2.2. Scene Graph

The OptiX engine provides a lightweight scene management system based on an object model with dynamic inheritance [26]. The node graph system is geared towards flexibility and efficiency, supporting instancing, level-of-detail and nested acceleration structures.

An OptiX scene is represented as a directed graph, where the traversal of rays starts at the top graph node and follows the hierarchy down the bottom. The scene graph used in this work is shown in Figure 3.4. The scene graph is split into two branches, one branch represents the sample and the other represents a lunar terrain patch. The different node types used in the scene representation are briefly summarized as follows:

**Group** is a generic graph node that is usually used for separating distinct scene sections. It is associated with an acceleration structure. It is usually used to provide a top level starting point for the scene traversal structure.

**Geometry Group** encapsulates scene object geometries and materials. They are leaves of the graph and are also associated with an acceleration structure.

**Acceleration Structure** is connected to groups and geometry groups and allows for more efficient ray traversal.

**Transform** is a node consisting of a matrix that defines the rotation and translation used to perform an affine transformation on the underlying geometry.

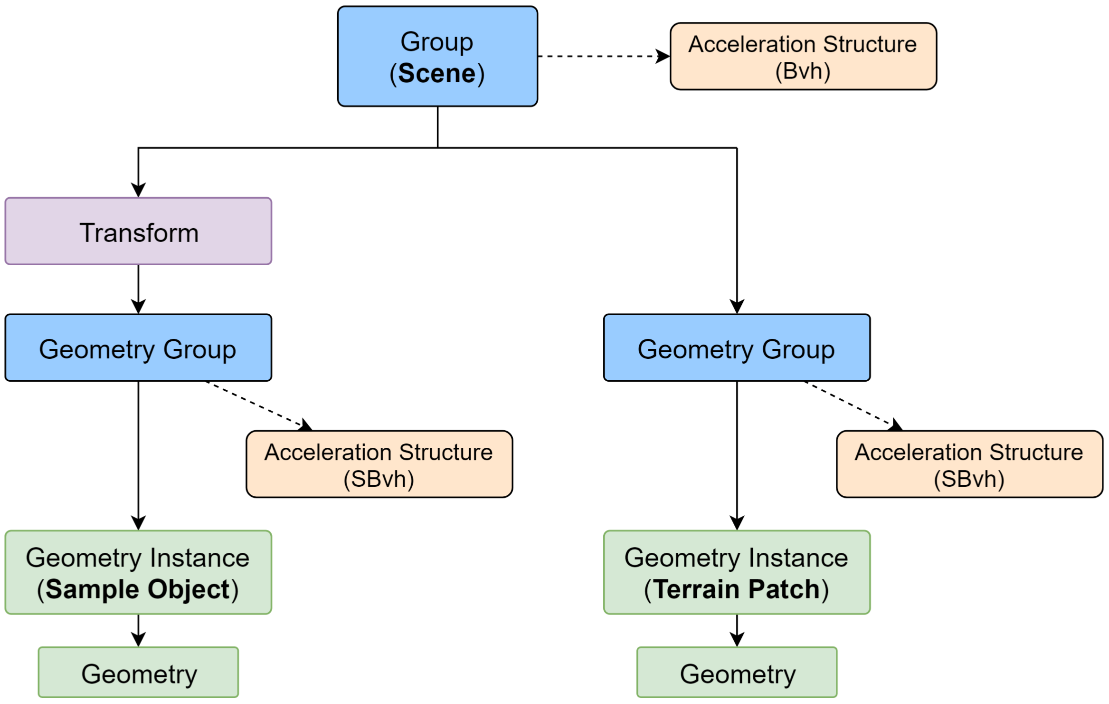


Figure 3.4.: Scene Hierarchy in OptiX. Template adapted from [21].

The decision of how to construct the scene graph using geometry groups will affect the efficiency of the ray traversal. Using a single geometry group to encompass all the different geometry instances in a scene is more efficient as the different objects are treated as one single unified geometry with a single acceleration structure. While efficient, this representation would suffer when using dynamic objects.

Alternatively, each scene object instance can be placed in its own geometry group such that each object is associated with its own distinct acceleration structure. Having such multi-level acceleration structures increases the complexity of the scene and so slightly reduces performance. However this approach is much more suited towards dynamic scenes since it allows different objects to be transformed or completely reconstructed without rebuilding the acceleration structure over the entire scene. Only the acceleration structures associated with the modified objects must be rebuilt.

Therefore a balancing between ray traversal efficiency and scene construction efficiency is necessary. In this work, due to the frequent movement of the sample and infrequent replacement of terrain patches, the second scene representation was used.

## 3.3. Solar Radiation

This section explains the implementation details of the solar radiation model. Section 3.3.1 describes how the sun is modelled as a flat solar surface, specifically a parallelogram. This parallelogram is used to launch solar rays into the scene. Section 3.3.2 presents an algorithm for generating this solar parallelogram. Finally, Section 3.3.3 presents the implementation details of radiative heat transfer due to direct solar radiation via *solar rays*.

### 3.3.1. Sun Modelling

The distance between the Sun and the Moon is relatively equal to the distance between the sun and the earth (astronomical unit), which varies between approximately 147 million kilometres and 152 million kilometres, depending on the position of the earth along its elliptical orbit. Even though the sun emits electromagnetic waves radially in all directions, such a large distance allows us to consider the incident solar rays on the lunar surface to be parallel [13].

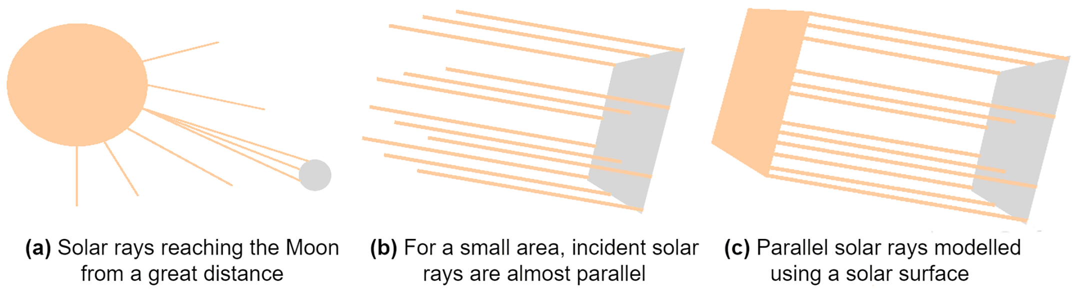

**(a)** Solar rays reaching the Moon from a great distance **(b)** For a small area, incident solar rays are almost parallel **(c)** Parallel solar rays modelled using a solar surface

Figure 3.5.: Solar radiation model [19].

To calculate the origin, it is not practical to use such a solar distance considering how small the simulated lunar terrain patch is in comparison. The variation in size is even more pronounced in the case of the rover or astronaut samples. Simulating such a wide spatial resolution ranging from the order of centimetres to hundreds of millions of kilometres would result in numerical errors, jeopardizing the accuracy of the results.

To work around this, we can rely on the fact that the incident ray directions are considered parallel, allowing the sun to be placed anywhere along the ray direction provided it is sufficiently far enough to not be occluded by the terrain. Using the Manhattan distance[1] of the terrain patch's bounding box provides such a suitable distance.

However, using this smaller modified solar distance makes it unsuitable to consider the sun to be a point light source, as is usually assumed in rendering applications. This would destroy the parallel nature of the solar rays. Instead, the sun can be modelled as a planer surface pointing towards the scene. The solar surface would be centred around the modified solar distance, and is oriented based on the solar azimuth and elevation angles.

The original method to calculate the solar surface shown in Figure 3.5 is described in [19]. The method relied on computing the intersection points between the solar surface (referred to as light plane), and lines going through a selected region of the lower terrain plane and extending in the opposite direction of the solar rays towards the sun. This would result in a solar parallelogram as shown in Figure 3.6.

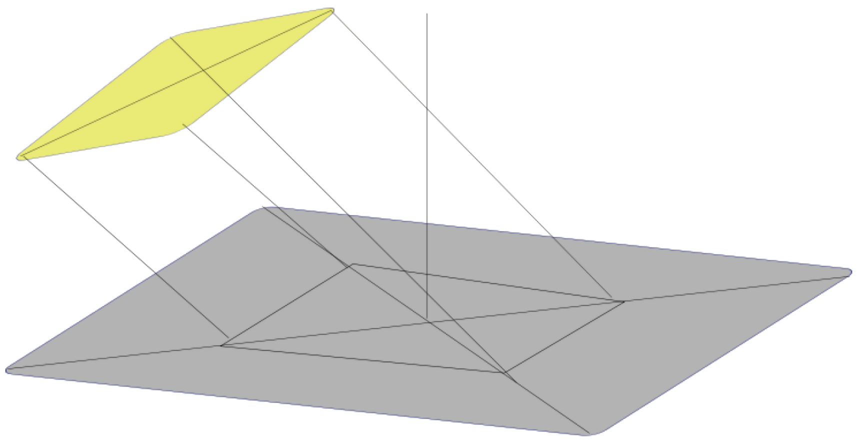


Figure 3.6.: Sun modelled as a parallelogram generated from the lower plane of a terrain patch [19].

However, this method only takes into account the lower plane of the terrain patch bounding box, ignoring any elevation. This is not a suitable assumption considering the

[1]The distance between two points measured along axes at right angles.

significant variation in the elevation of the lunar surface, which varies from about 11 kilometres above the mean Moon reference radius to more than 9 kilometres below it [23]. This means that the terrain patch might only be partially covered, with portions of the terrain at high elevation or at the edge not receiving any light from the solar surface.

This can be solved by using all eight corners of the terrain patch bounding box to create the solar surface. However the result would not necessarily be a parallelogram. Instead it would be an irregular hexagon since eight points would be projected onto the light plane instead of four. Only if the solar vector is perpendicular to any side of the bounding box would the resulting projected hexagon devolve into a rectangle.

Dealing with a hexagonal solar surface would still be possible. One option would be to triangulate the hexagon and use each triangle to generate a subset of the solar rays. This however would require uniformly sampling triangles, which is slightly more expensive than sampling a parallelogram.

Another issue with this method is that maintaining regular sampling of ray origins requires the use of more rays than in the case of a parallelogram. A parallelogram can be implicitly sub-divided into a grid, where an equal subset of rays can be launched from each grid cell. This would ensure that all regions of the terrain receive rays, even when using a small number of rays. This is more difficult to achieve with triangles, requiring either explicit triangle subdivision until small enough triangles are produced and subsequently launching rays from each of these high level triangles, or the use of quasi-Monte Carlo methods such as a 2D Sobol sequence to produce a more uniform distribution than a standard pseudorandom number generator. Naturally this would be more computationally expensive.

An alternative to the discussed methods would be to fit a parallelogram around the generated hexagonal surface. This solution would be faster to construct every iteration and would allow more efficient sampling. A disadvantage of this method is that some rays will to be lost into space since the fitted surface would in many cases be larger than the actual projected surface. Nevertheless, this is preferred to having parts of the terrain patch being out of the rays' reach. This is the approach used in this work.

### 3.3.2. Solar Parallelogram Generation

Finding a parallelogram to fit a set of points is an optimization problem and is solved using the rotating calipers method [31]. This method was used to calculate the diameter of a convex polygon in $\mathcal{O}(n)$ time by finding all antipodal vertex pairs of the polygon and choosing the pair with the largest distance. However it has since been used in many other applications, including generating a minimum-area bounding rectangle [33]. This method can be further generalized to generate a minimum-area bounding parallelogram.

Given a list of 3D vertices making up a terrain patch, a summary of the procedure for calculating the solar parallelogram $A$ is as follows:

1. Compute terrain patch axis-aligned bounding box. This significantly reduces the number of points needed to generate the parallelogram.
2. Project the corners of the bounding volume onto the infinite solar plane defined by the elevation and azimuth angles. The projected points represent the possible extremes of the parallelogram.
3. Map the 3D intersection points to 2D by finding an orthonormal basis for the solar plane. Reducing the dimensions allows for simpler calculations.
4. Compute a convex hull of the point set. This allows us to ignore points inside the parallelogram which would not contribute to the final surface bounds.
5. Use the method of rotating calipers to fit a minimum-area parallelogram around the projected set of 2D points.
6. Map the parallelogram corners back to 3D space using the orthonormal basis.

The parallelogram $A$ is defined by three vectors, $A_{min}$ denoting the position of the lower left corner point, $A_{span_x}$ defining the span of the parallelogram along its local x-axis, and $A_{span_y}$ defining the span of the parallelogram along its local y-axis.

When simulating large terrain patches, the resulting solar parallelogram would in turn have a large surface area. This means that a significant number of rays must be launched in order to provide a high resolution coverage of the scene, which is necessary to produce enough intersections with the sample which is orders of magnitude smaller than the terrain patch. Otherwise the sample would be hit by too few rays to produce a physically accurate heat transfer between it and the solar surface.

However, this increase in number of rays would quickly begin to degrade performance. A solution would be to use a secondary, smaller parallelogram that would specifically target the high resolution region in the centre and the sample within it [20]. This would allow the sample to receive enough solar radiation while maintaining a lower ray density across the rest of the terrain.

### 3.3.3. Solar Ray Generation

To generate the solar rays, two solar parallelograms are used as shown in Figure 3.7. The primary parallelogram is used to target the terrain patch, while a much smaller secondary parallelogram is used to target the high resolution region in the centre of the terrain that was introduced in Section 3.2. This is necessary to ensure that both the high resolution region and the sample object within it receive enough solar rays. Therefore, the solar ray generation procedure described below is repeated twice, once for each parallelogram.

To launch the solar rays, the solar *Ray Generation* program is invoked in OptiX by a call to the appropriate $rtContextLaunch$ function, depending on the dimension of the launch configuration. Since the source of solar rays is a 2D surface, it is intuitive to use a 2D launch of $N_s \times N_s$ rays, where $N_s$ is the number of solar rays to be launched along one dimension. The rays will then be divided across the threads of any active device(s) and traced in parallel.

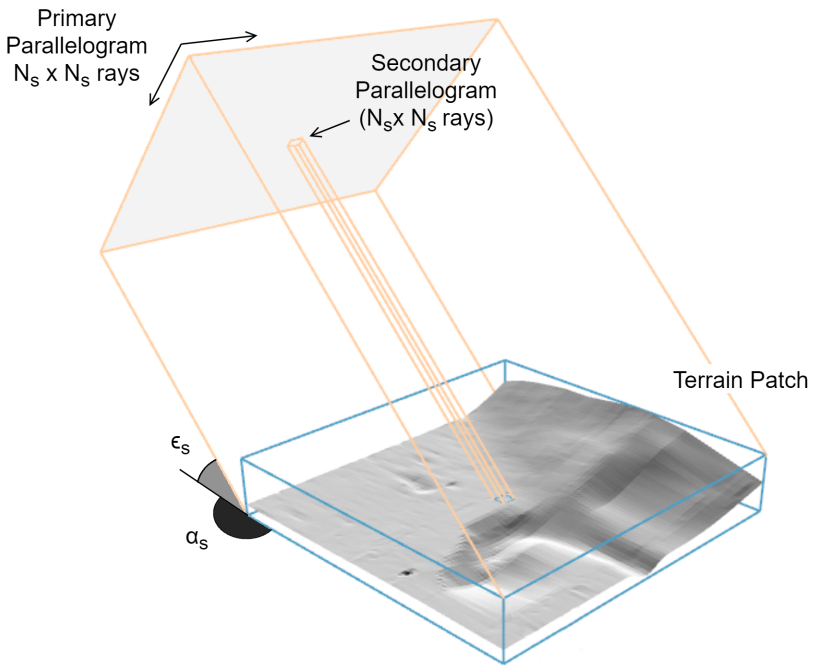


Figure 3.7.: Solar ray generation using a primary and secondary parallelograms.

The origin of each ray $r_{ij}$ can then be calculated as follows:

$$\vec{o_{ij}} = A_{min} + u_{ij} \cdot A_{span_x} + v_{ij} \cdot A_{span_y}$$

where $u$ and $v$ are the normalized coordinates $[0, 1]$ along the spans of the parallelogram $A$. They are calculated as follows:

$$u_{ij} = \frac{i + \hat{p_i}}{N_s}$$

$$v_{ij} = \frac{j + \hat{p}_j}{N_s}$$

where $\hat{p}$ is a small perturbation added in order to provide an element of randomness. It is a function of the ray tracing iteration and the ray indices.

The ray direction is parallel to the solar parallelogram normal and points towards the scene. It is the same for all solar rays and is calculated from the solar azimuth $\alpha_s$ and elevation $\epsilon_s$ angles as follows:

$$\vec{d_{ij}} = -\big(\sin(\alpha_s) \cdot \cos(\epsilon_s) \qquad \cos(\alpha_s) \cdot \cos(\epsilon_s) \qquad \sin(\epsilon_s)\big)$$

The final value that is needed to complete the ray is the payload. It represents the amount of power being carried by the ray. It is calculated using the solar constant $S$, which is the average amount of solar radiation at 1 AU (astronomical unit) per unit area and is equal to $1367\,\mathrm{kW} \cdot \mathrm{m}^{-2}$. This payload is equal for all solar rays and is precomputed as follows:

$$P_{ij} = \frac{area(A) \cdot S}{N_s^2}$$

After constructing and launching the ray in the *Ray Generation* program, OptiX performs the necessary tracing and intersection tests behind the scenes using the supplied $Intersection$ program. When a ray intersects a terrain primitive for the first time, which is the kind of intersection we're interested in, the $ClosestHit$ program is invoked.

The handling of this event is similar for both the terrain and the sample. The program is supplied with the identifier of the triangle that the ray intersected with and so the rate of heat transfer reaching the triangle $T_k$ per unit area (heat flux) can be calculated as follows:

$$q_{T_k} = \frac{\alpha_{T_k}}{Area(T_k)} \cdot P_{ij}$$

where $\alpha_{T_k}$ is the triangle absorptivity.

A false positive closest hit might occur on the terrain due to rays hitting terrain triangles from the back. This occurs because the terrain is simulated in patches, and a patch is not a closed surface. The sample on the other hand is a closed surface and does not need any special treatment. To solve the issue on the terrain, a simple back face culling test is performed. The scalar product of the terrain face normal and the incident ray direction is tested for its sign. The winding of the terrain vertices (clockwise vs. counter-clockwise) determine which condition to test against. If the test fails, the ray is hitting the triangle from the back and the calculation is aborted.

It is important to make sure that rays from the primary parallelogram do not contribute energy to triangles within the high resolution centre. Similarly, rays from the secondary parallelogram should not contribute energy to triangles in the low resolution region. Otherwise triangles would be over-illuminated due to receiving energy from two rounds of solar ray generation.

## 3.4. Infrared Radiation

Heat transfer due to the thermal radiation emitted by the scene objects is modelled using *infrared rays*. Unlike solar rays, there is no need for an external ray emitting surface. Instead, the scene meshes themselves act as emitters, where rays are launched into the scene from each primitive surface depending on its temperature. This is shown in Figure 3.8 below.

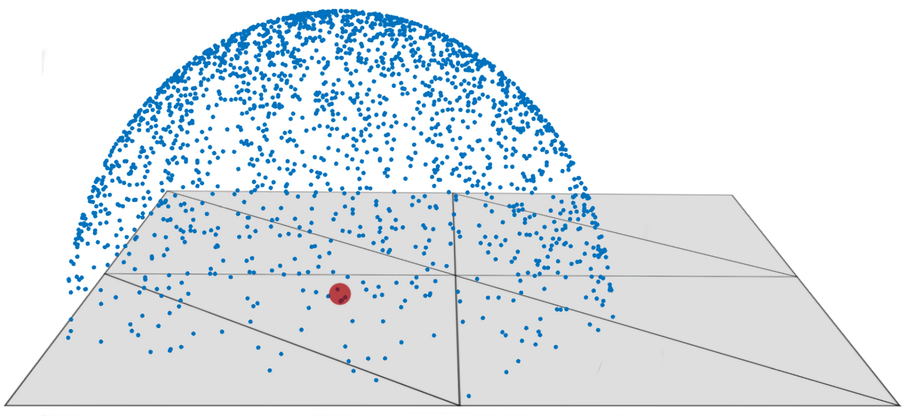


Figure 3.8.: Infrared rays launched from a random triangle position.

To construct an infrared ray, a mesh triangle needs to be sampled. Mesh triangle indices and vertices for both the terrain and sample are stored in buffers that can be accessed from all the OptiX programs such as the *Ray Generation* and *Closest Hit* programs.

In the *Ray Generation* program, a random point along the triangle primitive surface is selected every time an infrared ray is processed. This determines the origin of the $i$th infrared ray launched from the triangle and is calculated with the following equation [24]:

$$\vec{o_i} = (1 - \sqrt{r_1})V_0 + \sqrt{r_1}(1 - r_2)V_1 + \sqrt{r_1}r_2V_2$$

where $V_0$, $V_1$ and $V_2$ are the triangle's vertex positions and $r_1$ and $r_2$ are random numbers between 0 and 1.

The ray direction is obtained using a cosine-weighted hemisphere sampling. This is based on Lambert's cosine law, which states that the radiance of perfectly diffuse emitters or reflectors is proportional to the cosine of the angle between the viewing direction and the surface normal [32].

This means that infrared rays in the normal direction have higher contribution than rays parallel to the surface. Using a cosine-weighted function would bias the sampling towards the normal as shown in Figure 3.9. This would allow us to cast less rays than with a uniform distribution. The sampling is achieved by first uniformly sampling a point on a disk and then projecting it on the hemisphere [8]. The result is transformed into world coordinates using an orthonormal basis obtained from the triangle's normal.

$$\vec{d_{ij}} = \left(\sqrt{r_1}\cos(\phi) \qquad \sqrt{r_1}\sin(\phi) \qquad \sqrt{1 - r_1\cos(\phi)^2 - r_1\sin(\phi)^2}\right) \cdot ONB$$

where $\phi = 2\pi r_2$, $r_1$ and $r_2$ are random numbers between 0 and 1 and $ONB$ is the orthonormal basis. The normal needed for the orthonomal basis is simply calculated on the fly using the triangle's vertices.

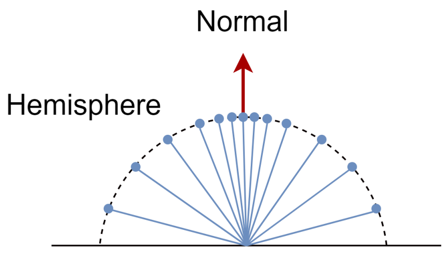


Figure 3.9.: Cosine hemisphere sampling. More points are sampled perpendicular to the surface than parallel to it.

Finally, the ray payload emitted from triangle $T_k$ is calculated using the Stefan-Boltzman law as follows:

$$P_i = \varepsilon_{T_k} T^4 \cdot \frac{area(T_k)}{N_{IR}}$$

where $\varepsilon_{T_k}$ is the triangle emissivity and $N_{IR}$ is the number of infrared rays per triangle. Since infrared rays are used to model heat loss from the mesh triangles, the energy of each triangle must be reduced by the total amount of energy lost through each ray in order to maintain energy conservation.

As with solar rays, after an infrared ray intersects with another triangle in the scene, its energy is partially absorbed and the rest is emitted back into the scene via specular and diffuse reflection rays.

## 3.5. Ray Reflection

Once the contribution of a direct solar or infrared ray is applied to the intersected triangle, the remaining energy which was not absorbed by the triangle is emitted back into the scene. This is used to model both diffuse and specular reflection and is achieved by launching secondary rays directly from the triangle surface. The secondary rays are centred around the original intersection point and their direction depends on the reflection type as shown in Figure 3.10.

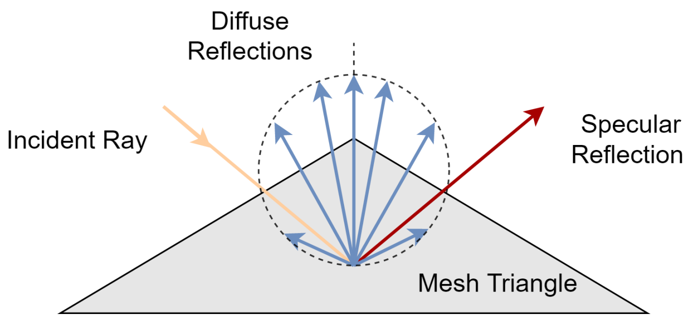


Figure 3.10.: Incident ray being reflected specularly in a mirror direction and diffusely in a cosine-weighted fashion.

For specular reflection, a single ray is launched at an angle that mirrors the incident ray angle. For diffuse reflection, a number of rays are launched in a random direction over the hemisphere using a cosine weighted sampler in a similar fashion to infrared rays as discussed in Section 3.4. The number of diffuse rays used depends on the desired performance and accuracy. It should be noted that using too few diffuse reflection rays would be worse than not using any reflection at all. Since it would skew the results and in many cases cause triangles to receive an unphysical amount of energy as the entire reflected energy is distributed among too few directions.

Because of the stone nature of the lunar surface, only diffuse reflection is used for the terrain, while both reflection types are made available for the sample object. The diffuse and specular reflectivity can be set on a per-triangle basis. This can be useful for example to model a sample with different parts being made of different materials.

If a reflection ray intersects with another triangle in the scene, the whole process is repeated recursively either until a set maximum number of reflections is reached, or if the reflected ray carries a negligible amount of energy.

## 3.6. Data Exchange

The transfer of data between the two main TherMoS components, the heat transfer ray tracer and the solver, is achieved in three ways: command line arguments, text files and a synchronized memory-mapped file.

Since the TherMoS solver is considered to be the parent component, it is responsible for launching the ray tracer component. As such, it provides the ray tracer with simulation specific settings as command line arguments. These arguments include the number of solar rays, number of infrared rays, number for reflection rays as well as geometry and material data filenames.

The data files are read by the ray tracer during initialization. Since performance during the setup stage is not an issue, regular text files are used for input. The *.csv* file format is used due to its simplicity and universality. Both the terrain and the sample are associated with three *.csv* files each. The first file contains information about the vertex coordinates. The second file contains information that associates each mesh triangle with a set of vertices. And the third file contains per-triangle thermo-optical properties (or materials).

The final data transfer mechanism occurs during the simulation at the beginning of each ray tracer iteration. Since this data exchange is so frequent, a fast method is essential to not degrade the overall ray tracer performance. The information exchanged include settings that vary at each iteration such as the solar angles and sample position. The results of both the ray tracer and the TherMoS solver are also exchanged, where the solver provides the ray tracer with the face temperatures, and the ray tracer provides the solver with the face heat fluxes. This is achieved using a memory-mapped file and a more detailed explanation of the implementation is provided in Chapter 5.

# 4. Multi-GPU Ray Tracing

The implementation presented so far only considered the use of one GPU to calculate the radiative heat transfer via ray tracing. In this chapter, the ray tracer is extended to run on multiple GPUs by dividing the workload and performing the computation in parallel. The aim is to reduce the ray tracing time and therefore improve the overall performance of TherMoS. Parallelization is achieved using the Message Passing Interface (MPI).

MPI is a message passing standard for distributed memory systems that allows different processes to work together towards achieving a common goal. It provides the communication and synchronization mechanisms necessary for such an effort. Detailed description of MPI is beyond the scope of this work, but the reader is encouraged to consult the introductory material by B. Barney [2] for more information.

This chapter is structured as follows:

- Section 4.1 describes how such a parallelization would work with the OptiX model and the problems it posed for this implementation.
- Section 4.2 explains how the different GPUs are bound to the ray tracer processes using MPI.
- Section 4.3 details the ray tracing load distribution among the GPUs.
- Section 4.4 describes how data is communicated between the ray tracer processes.

## 4.1. Multi-GPU Ray Tracing with OptiX

OptiX natively provides multi-GPU ray tracing capabilities where a single OptiX *Context* object can make use of multiple graphics hardware devices [21]. This is done automatically and without further user input. OptiX handles the ray scheduling and distribution behind the scenes, and provides the results in an output buffer.

However, this only works as desired when working with an algorithm where each launch index writes to a distinct memory buffer location. This is the case for a gather algorithm such as backward ray tracing, where only a ray launched from a specific buffer location will ever write back to it. On the other hand, with a scatter algorithm such as forward ray tracing, where any ray could write to any buffer location, this would lead to wrong results because of race conditions. It would not be possible to avoid such race conditions by using atomic operations since write operations from threads across multiple GPUs are not serialized, unlike operations across a single GPU. To put simply, OptiX does not provide automatic support for multi-GPU scatter algorithms and so they must be handled by the user instead.

Another issue is the decrease in performance due to limited bandwidth as described in [21]. If a single context is used to manage multiple devices, its performance will be limited by PCIe bandwidth because buffers for ray generation and intersections are transmitted over the PCIe bus to the different GPU devices.

All things considered, the use of one context to leverage multiple GPUs would therefore not be viable for our application. In this work, each GPU is therefore managed by a different context. Since OptiX is not thread safe, running multiple contexts in a single process across different threads would not work. To work around this, every context is run on its own process in parallel on a distributed memory system. This will allow us to avoid both race conditions and data transfer latencies, resolving both issues mentioned above.

## 4.2. Parallelization Setup

Given that multiple ray tracing processes need to work in tandem, the MPI standard is used to create, manage and synchronize the different instances. As many MPI processes are launched as there are available GPUs. The different processes all undergo the same initial setup procedure before the ray tracing tasks can be distributed among them.

First, each process must select a GPU and bind it to the OptiX context. It is possible to perform this GPU selection by separating the processes into subsets such that each subset contains processes belonging to a single shared memory node. This is shown in Figure 4.1. In MPI, this is achieved by splitting the world communicator with the *MPI_COMM_TYPE_SHARED* flag. The processes in each of the resulting shared communicators can then use their new local rank to select a GPU from the available devices identified by OptiX.

After that, the processes read the input files simultaneously and build the scene. It is important to note that while the purpose of the parallelization is to divide the workload between all available GPUs, every process must build the entire scene for itself. Because even though each GPU will be casting rays from a different section of the ray generation areas, the rays can still hit any triangle in the scene due to object occlusion and ray reflections.

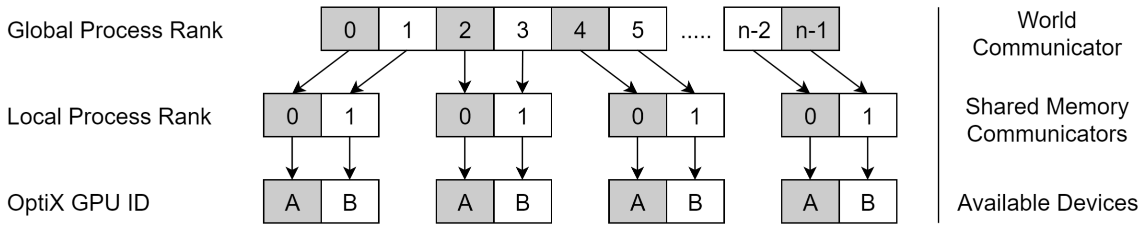


Figure 4.1.: Assignment of GPUs to the different MPI ranks.

## 4.3. Load Distribution

When working with parallel computing, it is important to divide the computational workload as equally as possible between the different processes used. This would minimize a process's idle time, keeping it as busy as possible and so increasing the parallelization efficiency. Without proper load balancing, some processes would end up waiting for others to finish, wasting time that could otherwise be spent performing the next set of computations.

Fortunately, ray tracing is an embarrassingly parallel problem, meaning that the problem can be easily divided into smaller tasks that can be performed in parallel. The condition being that little or no dependencies exist between the different tasks, allowing computation to proceed unhindered by inter-process communications. This is exactly the case for ray tracing where a single ray can be traced completely independently of other rays, such that the only communication necessary is the summation of all ray contributions at the end. This is the case for both forward and backward ray tracing.

The load distribution of the ray generation surfaces is shown in Figure 4.2. As can be seen, the load distribution among the processes (and so among the GPUs) is easily achieved. Each surface is divided into equal sections such that every process only launches rays from the specific section assigned to it.

For solar rays, each process is assigned a column-wise section of the main and secondary solar parallelograms. This is done by manipulating the $A_{min}$ and $A_{span_x}$ variables for both parallelograms. For infrared rays, each process is supplied with a different offset into both the terrain and sample primitive indices buffers. This allows good cache utilization since each process works on a contiguous section of the scene objects.

It should be noted that the existence of the central high resolution region does lead to some imbalance in the workload. Since it reduces the surface area of the two central sections, causing less rays to be processed by the GPUs to which those sections were assigned. However, the size of this region is much smaller in comparison to the whole ray generation surfaces and so its effect on the load balance is minimal and can be neglected.

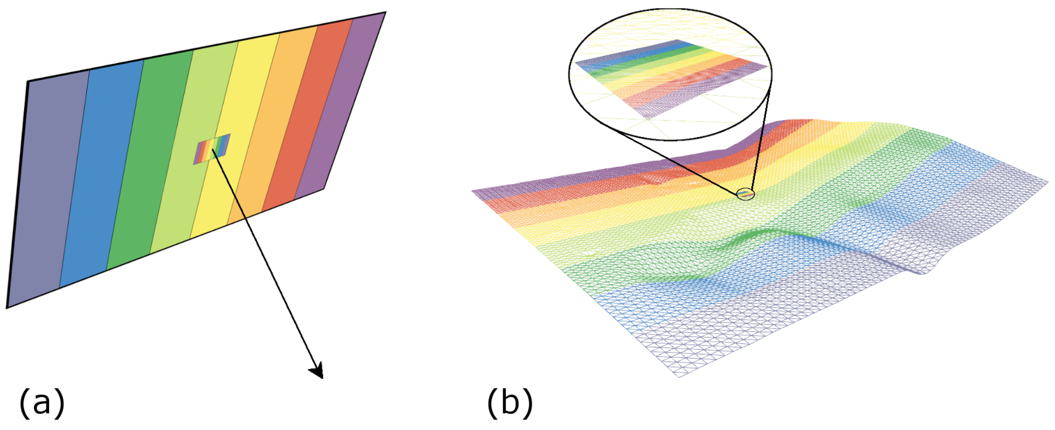


Figure 4.2.: Load distribution of ray generation from (a) solar parallelograms and (b) terrain patch mesh.

## 4.4. Data Communication

During a single ray tracer run, the only data transfer necessary between the various processes is the collection of the heat flux values computed by each process. This occurs twice, once for solar rays and once for infrared rays. Each of these results is stored in a separate buffer before being output. A single processor therefore acts as a receiver, gathering the local heat flux buffers from all the other processes, labelled as senders, and performing an element-wise summation on the data.

In parallel applications, the time spent on performing communications relative to the time spent on computations affects the overall efficiency of the parallelization. GPU ray tracing is fast, even more so when utilizing a multi-GPU setup. As a result, the time spent on transmitting the results between the different ray tracer instances would make up a noticeable fraction of the total simulation time. This fraction increases as the resolution the scene objects is increased, since more data will need to be transmitted due to the higher triangle count.

With blocking communication (Figure 4.3), each process would have to wait until all other processes have completed their computation to perform the data transfer. Process computation time may vary due to load imbalance. In a ray tracing application, even though the ray generation can be distributed equally, the actual ray tracing time can vary. This occurs for a variety of reasons, the most important of which is a heterogeneous scene.

In our case, mountains, craters and other geographic features of the terrain patch would affect ray intersections. A ray hitting a crater at a low angle might spawn reflection rays which would repeatedly bounce between either sides of the crater. On the other hand, a ray hitting a flat plain would reflect directly into space, ending its tracing. When launching a significant number of rays, these small interactions add up. This would cause some processes to finish earlier than others. This variation in computation time is depicted in the figure, albeit the effect is exaggerated for clarity.

After the data transfer starts, the processes would also need to wait (block) until their buffers can be safely reused, which may not take a negligible amount of time for high resolution scenes.

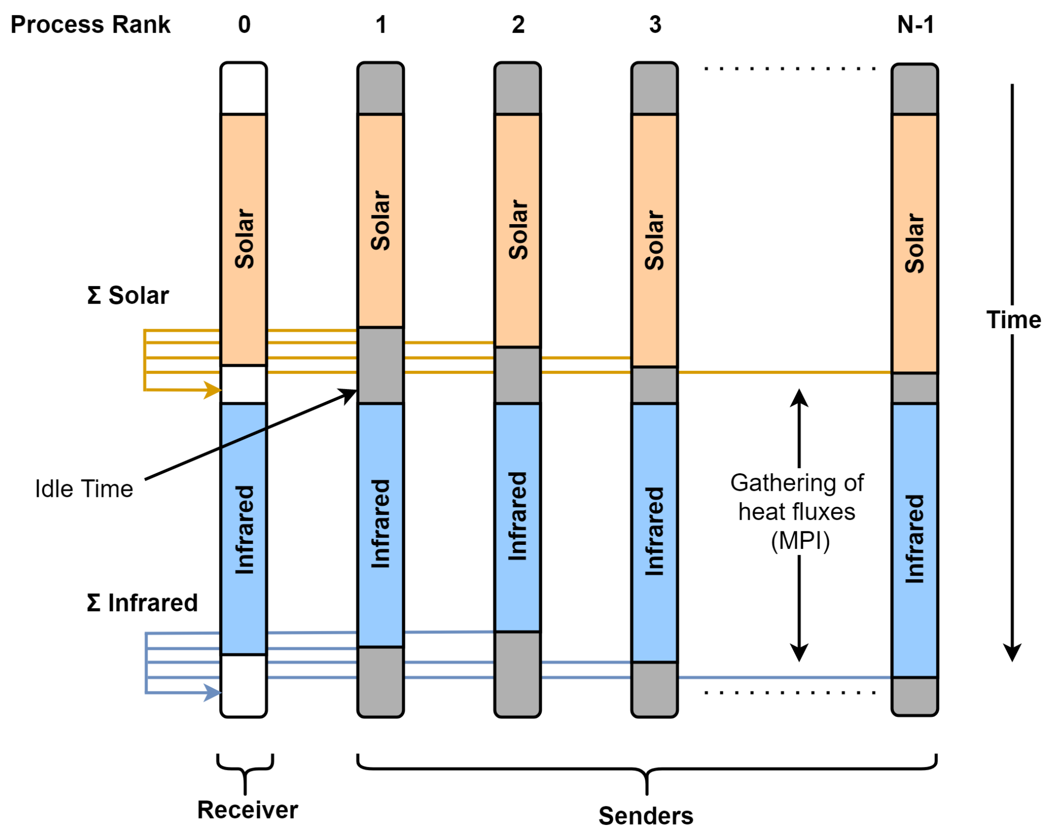


Figure 4.3.: Blocking communication scheme between the different ray tracing processes. Data gathering is delayed until all processes finish their computation, causing some processes to wait idly. Therefore, this scheme is not used in this work.

A common solution to this issue is to overlap communication with computation as shown in Figure 4.4. This is achieved by using non-blocking communication routines. These routines do not block the process from running until the communication is completed. Instead, they return control immediately to the process after they have been initiated, even if the data was not transmitted yet. This allows the process to continue performing further computations instead of waiting idly. In this work, the non-blocking communication scheme is used.

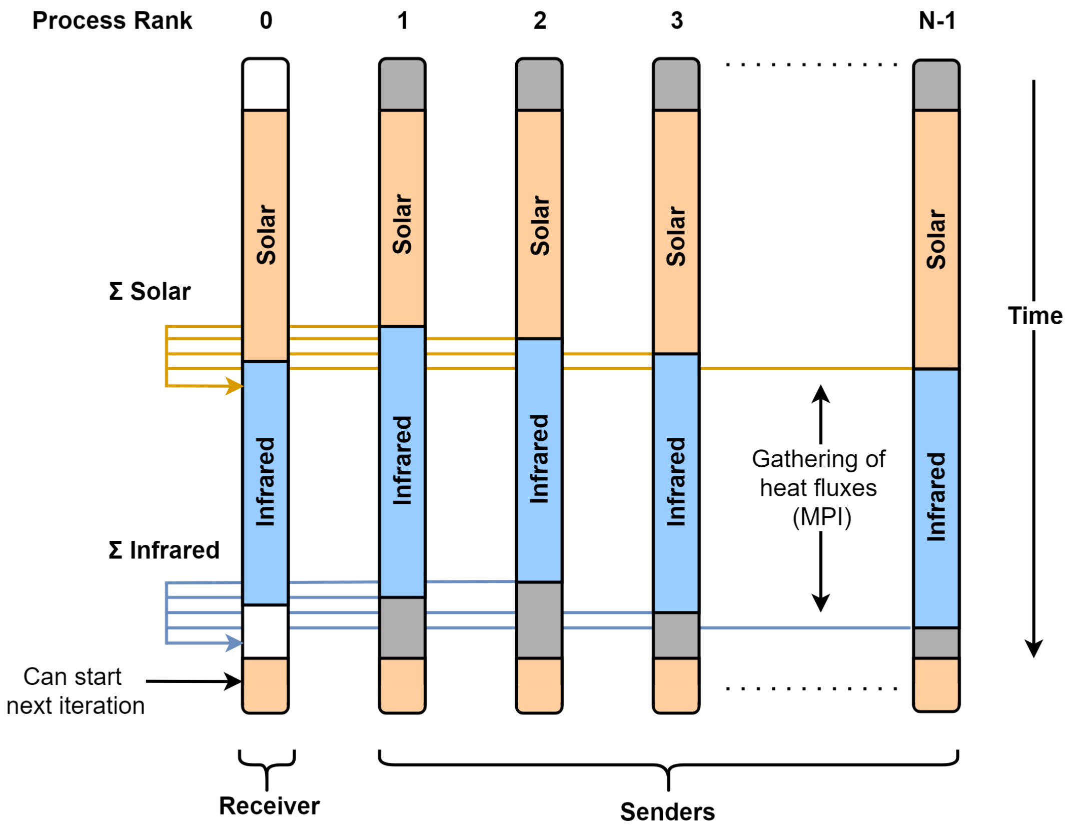


Figure 4.4.: Non-blocking communication scheme between the ray tracing processes showing the advantage over blocking communications: avoiding process idle time using computation-communication overlap. Next iteration can start earlier than with blocking communication, although processes would first need to wait until the TherMoS solver is done. This is the scheme used in this work.

Once both computation phases are complete, the processes wait for any data transfer to complete if necessary. The local buffers on the receiver process (rank 0) would then contain the contribution of all other processes. They can then be copied to the memory-mapped file data buffer to be read and used by the TherMoS solver component.

# 5. Ray Tracer - MATLAB Solver Interface

This chapter discusses the implementation of the new interface between the ray tracer and solver components of TherMoS. This would replace the existing interface which relied on communication via text files. It would also allow the ray tracer process(es) to persist between calls, unlike the previous implementation which relaunched the ray tracer component at every time step. Therefore, the purpose of reworking the existing setup is to improve performance and to provide a flexible and easily extensible interface. The new interface achieves this using a memory-mapped file through the Boost.Interprocess library. This would allow faster data transfer and provide the means for synchronizing the TherMoS components.

The chapter is divided into the following sections:

- Section 5.1 explains the existing setup and how it can be improved upon.
- Section 5.2 provides a brief evaluation of the interprocess communication (IPC) mechanisms presented previously in Section 2.5 and explains the decision for selecting memory-mapped files.
- Section 5.3 details the use of the memory-mapped file to create the new interface, and how it is used to perform the communication between the TherMoS components.
- Section 5.4 discusses how the file is used to synchronize the two components. This section is technical in nature and is aimed at readers with little or no background on the subject.

## 5.1. Overview

As mentioned previously, the heat transfer ray tracer does not operate on its own. It is part of a larger simulation, where the heat transfer values computed by the ray tracer are used to compute the thermal profile of the scene in the TherMoS solver. Therefore it is important to provide a clean and efficient interface between the two components. Each component should be able to operate as a black box, with minimal interaction with the other. This allows design flexibility where each component can be modified separately without worrying about breaking the operation of the other. This would also allow the ray tracer to be used with different solvers with little extra effort.

The data exchange interface should also not add significant overhead to the overall simulation performance. In the original implementation, data was exchanged between the components using *.csv* files. Additionally, the ray tracer was launched at every simulation time step, performed its calculation and then terminated. This meant that at every iteration the operating system is asked to create a new process which then reloaded the scene from disk, rebuilt the OptiX acceleration structures and finally wrote the output back to disk after the ray tracing was performed.

This whole process added unnecessary overhead to the total ray tracer running time. To overcome this, the ray tracer is launched only one time at the beginning of the simulation. Therefore, the scene is read and the acceleration structures are built once at startup. Further scene reconstruction can occur when the small high resolution area in the terrain is moved. However this occurs rather infrequently and has a negligible impact on performance. Since the two TherMoS components will be running in parallel, the TherMoS solver would need a way to control the operation of the ray tracer. Thus it is important to use synchronization mechanisms to allow them to work in tandem. This is achieved using mutexes (mutual exclusion objects) and condition variables.

Another way to improve performance is to move away from *.csv* files to avoid costly disk I/O operations. Geometry and material data would still be read from *.csv* files to construct the scene. However, data exchange of component results at every time step is moved to a memory-mapped file instead.

## 5.2. Evaluation of IPC Methods

In terms of efficiency in transmitting data, shared memory and memory-mapped files outperform the rest of the discussed methods [11]. The reason being that once the shared memory is created and mapped, no system calls are needed for the data exchange and so accessing the shared memory has similar overhead to any other memory addressing, which is quite fast. The gap in performance becomes much more pronounced as the data size increases. On the other hand, shared memory and memory-mapped files do not provide built-in synchronization mechanisms. This means that the user must coordinate data accesses between the processes to avoid any race conditions, increasing the complexity of the system.

Between them, shared memory has slightly lower latency than memory-mapped files. However, this edge would become negligible if the data exchange time is a small fraction of the overall running time, where the majority of which is spent performing the actual computation. The choice then becomes a matter of preference. Memory-mapped files are chosen in this work as they would offer easier debugging since the files can be made to persist after the application finishes. It would also allow flexibility in the future if the intermediate data exchanged needs to be retained.

## 5.3. Memory-Mapped File

Before the solver is ready to launch the ray tracer process, it first creates a mappable object that represents the physical file on disk. This object is used to create a one-to-one mapping of any desired region within the file. In this case, the entire file is mapped. This mapping will be associated with a memory address in the address space of the calling process. Modifying any section of that memory address will automatically modify the equivalent section in the memory-mapped file on disk.

This memory-mapped file only needs to be created once, but the mappable object should be created by all processes wishing to access the file. Once the ray tracer has performed its own mapping, it would be able to see any changes made to it. Each component can then read and write to its respective mapped memory space as it would normally access its assigned random-access memory, except such actions would be equivalent to transferring and receiving data from the other component.

The structure of the file is divided into three parts as shown in figure 3.1. The first part contains data relating to synchronization. Namely a single mutex, two condition variables and a flag to notify the ray tracer that it is time to terminate the simulation. The second part is made up of a header containing the different simulation data. Since the TherMoS solver is the main component driving the simulation, it is the one that provides the ray tracer with the per-iteration simulation settings such as solar elevation and azimuth angles and the 4x4 sample transformation matrix. The third part is made up of a data swap buffer. The solver writes to the buffer the per-face temperature values for both the terrain and the sample, which are needed to calculate the infrared radiation. The ray tracer would in turn write back the solar and infrared per-face heat fluxes for both the terrain and the sample. Since the file is used for data exchange and there is no need to store any of this data once it has been read, it is sufficient for the data swap buffer to only be large enough to transfer data in either direction.

## 5.4. Synchronization

At the beginning of the simulation, the TherMoS solver provides the ray tracer with the initial per-face temperature values across the scene. Once the ray tracer uses these temperatures to compute the total heat transfer, it returns its results back to the solver. This process is then repeated at every time step. Therefore, it is important to synchronize the two components such that they are aware when it is their turn to wait, and when to compute. The synchronization is also necessary to avoid race conditions when reading from and writing to the memory mapped file.

### 5.4.1. Solver-side Synchronization

The TherMoS solver is written in MATLAB, and since the Boost.Interprocess library is C++ based, it is necessary to use MEX files. MEX files stand for MATLAB executable and can be used to write dynamically linked subroutines that can be executed by the MATLAB interpreter [17]. This provides a suitable interface for MATLAB scripts to interact with any C++ code, including any Boost library. The MEX C++ interface used for synchronization on the solver-side will from here on now be referred to as the *Synchronizer*.

After creating the memory-mapped file, the solver spawns the ray tracer processes using MPI. This is done by passing an *mpiexec* launch command to the system. The launch command varies depending on the operating system, but in all cases it would decide how many ray tracer processes are to be launched and where to find the newly created memory-mapped file.

The Synchronizer then writes to the file the initial temperatures needed by the ray tracer. In each simulation time step, the ray tracer computes first. It uses the temperatures from the previous iteration and computes the current heat fluxes which are in turn used to compute the current temperatures. So for the first time step, initial scene temperatures are needed. These are computed analytically in the TherMoS solver component.

Once the first set of data is written to the file, the two components perform a simple handshake. This is used to verify that a connection has been successfully established between them. The handshake is achieved by simply locking the mutex on the solver side and waiting on the ray tracer notification via a condition variable which would unlock the mutex, completing the handshake process. All waiting done in the synchronization is timed. Meaning that once a component has been waiting for more than a set amount of time, the waiting function would return control to the caller function regardless whether the mutex was unlocked on the other side or not. This is used to prevent a component from waiting indefinitely if the other side failed to unlock the mutex due to a catastrophic error. The timeout period is application-specific and is set manually such that it is longer than the expected time needed by a component to perform one iteration.

When the handshake is successful, the solver enters the synchronization loop. Here it would wait for the ray tracer to finish computation and read its results. The MEX synchronizer would then pass control back the MATLAB solver scripts along with the heat flux buffers. An important thing to note here is the memory management aspect when it comes to MEX files. MATLAB expects data to be passed to and from MEX files using an array of pointers to *mxArray* variables. If those variables were allocated in memory using the built-in *mxCalloc* function, then the pointers can be safely used in the MATLAB scripts to access data buffers allocated in the MEX file. However, since the memory was allocated under-the-hood by the Boost.Interprocess library instead of using the built-in MATLAB memory functions, pointers to that memory cannot be used and an intermediate extra copying step is needed. The data would have to be copied from the memory-mapped region provided by Boost.Interprocess to a set of temporary *mxArray* variables. The addresses of these temporary buffers will then be sent to the MATLAB scripts.

Once the MATLAB solver scripts perform their computation, they return control to the synchronizer which writes the results to the memory-mapped file. Finally it notifies the ray tracer that it has finished and resumes waiting. This process is then repeated until the simulation ends.

### 5.4.2. RayTracer-side Synchronization

On the ray tracer side, only one of the processes (rank 0) is tasked with exchanging data through the memory-mapped file. Since the results of the different ray tracer processes are already being gathered on a single process, it is only necessary for that process to synchronize its data transfer with the TherMoS solver. Therefore it acts as an intermediary between the solver and the rest of process pool, controlling the execution flow of the other processes in a Master-Slave model.

After being launched, all ray tracer processes perform their usual scene initialization steps. The slave processes then enter an infinite loop and wait for a signal from the master to continue or terminate the simulation. This is done using a simple MPI send command. Meanwhile, the master process creates a mapping of its own to the memory-mapped file, and establishes a connection with the solver by completing the handshake.

During the loop, whenever the solver notifies the master process of the ray tracer's turn to proceed, the master process reads the input, broadcasts it to the rest of the processes and they all perform their computations. Finally the results are gathered and written by the master and the solver is notified.

# Part III.

# Results and Conclusion

# 6. Results

This chapter presents the performance results of the new multi-GPU implementation of the ray tracer component of TherMoS, as well as the performance results of the new interface between the ray tracer and the solver. The chapter is structured as follows:

- Section 6.1 describes the test bed configuration and the test cases used to obtain the results.
- Section 6.2 shows the typical output of the ray tracer component.
- Section 6.3 presents a performance analysis of the multi-GPU ray tracer implementation
- Section 6.4 presents a performance comparison between the old and new TherMoS interfaces.

## 6.1. Test Configuration

### Testbed

In this work, the TherMoS ray tracer simulation was run on the MAC Cluster hosted by the Leibniz Supercomputing Centre (LRZ). In particular, the *nvd* partition of the cluster was used, which is made up of four homogeneous nodes. The nodes have the following specifications:

- Dual socket Intel SandyBridge-EP Xeon E5-2670
- Two NVIDIA M2090 GPUs
- 128 GB RAM
- FDR infiniband

In total, the cluster partition provides eight NVIDIA GPUs for testing parallel software. The GPUs are based on the Fermi architecture with compute capability 2.0, which limits their support to CUDA 7.5 and in extension limits the use of OptiX to version 3.9.5. However, the code is forward compatible with the newest OptiX version, which is 5.0.0 as of the time of this writing. This means that even better performance speed-up can be expected of this implementation on newer hardware, both due to better GPU performance and more optimized OptiX ray tracing.

The cluster runs on the SuSe 11.1 SP1 operating system and the MPI implementation provided is the Intel MPI Library version 5.1.3.

**Test Scenes**

The testing scenes are each made up of a single patch of lunar terrain and a single sample object. The lunar geographical data is obtained from the Lunar Reconnaissance Orbiter and the triangulation is performed in MATLAB. The sample object is a human astronaut exploring the surface of the Moon.

Each terrain patch is of approximate dimensions $8\,\mathrm{km} \times 8\,\mathrm{km} \times 1.4\,\mathrm{km}$. The patch meshes are varied in resolution by subdividing their triangles repeatedly into smaller ones. Out of the set of meshes used, the lowest resolution mesh (referred to as the *Coarse Mesh*) is made up of 23090 triangles while the highest resolution mesh (referred to as the *Refined Mesh*) is made up of 187650 triangles in total. Each patch also features a small region of relatively higher resolution in the centre as discussed in Section 3.2. This region is made up of 11250 triangles. The astronaut sample is made up of 504 triangles and its movement is confined within this region.

In this implementation, the thermo-optical material properties of the scene objects can be assigned in a per-face fashion. However, for the purpose of analyzing performance it is sufficient to use uniform values across a mesh. The lunar terrain is considered to be a purely diffuse surface while the astronaut spacesuit material reflects incoming radiation both in a diffuse and specular fashion. A total of ten diffuse reflections were launched per ray, with a maximum recursive depth of three. The ray payload cutoff criterion used was $1 \times 10^{-7}\,\mathrm{W}$.

## 6.2. Ray Tracer Output

Every iteration, the TherMoS solver component requests the computation of radiative heat fluxes from the ray tracer component. The ray tracer is provided with new solar angles that are used to compute the solar heat fluxes and new per-face temperatures that are used to compute the infrared heat fluxes. It also transforms the sample object to its new position and orientation. A typical ray tracer output of heat fluxes for a single TherMoS iteration is shown in Figures 6.1, 6.2 and 6.3 below.

Figure 6.1 shows the solar radiation heat flux distribution across a patch of the lunar surface. The sun is positioned at both an elevation and azimuth angles of $45°$ and pointing towards the scene. As is expected, unoccluded flat regions of the terrain receive rather uniform solar radiation. On the mountains, slopes oriented towards the direction of the sun receive the most radiation while slopes pointing away from it receive the less radiation. The least amount of radiation absorbed in the scene is seen by the impact crater to the left. Due to its depth and relatively low solar elevation, solar rays cannot reach the innermost crater triangles at all.

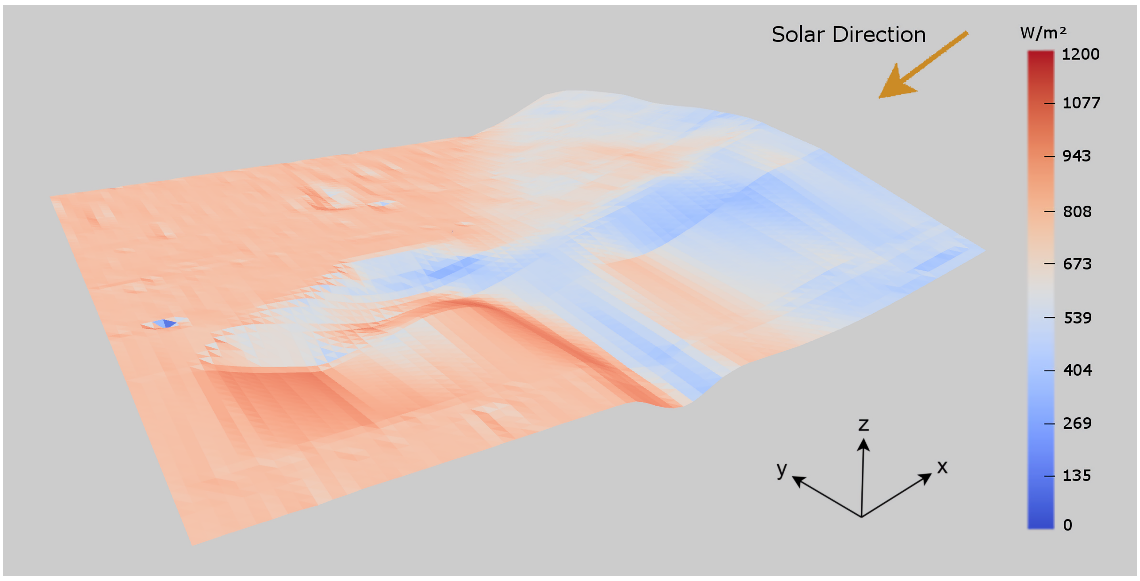


Figure 6.1.: Ray tracing results of solar radiation heat fluxes on a lunar terrain patch. A total of $5600 \times 5600$ rays were launched from both primary and secondary solar parallelograms. The elevation and azimuth angles are set to $45°$.

The infrared radiation heat flux distribution is shown in Figure 6.2. Infrared rays are launched over a hemisphere and model the dissipation of thermal energy from the surface. Flat regions of the terrain surrounded with little or no geological features will dissipate most of their energy into space in a uniform fashion, while in turn receiving little or no infrared radiation from other regions of the surface. This is especially true if neighbouring hills and mountains have low slopes.

On the other hand, radiation emitted from troughs and craters will mostly remain trapped within the inner surface instead of being lost in space. The deeper the troughs and craters are, the more thermal energy they retain, resulting in less net loss of infrared radiation. This is the case because rays leaving one side of the inner surface would simply travel until it hits the opposite side and vice versa. Rays would continue to bounce back and forth between the sides, trapping the thermal energy. As can be seen in Figure 6.2, the lowest heat fluxes are found within such geological features.

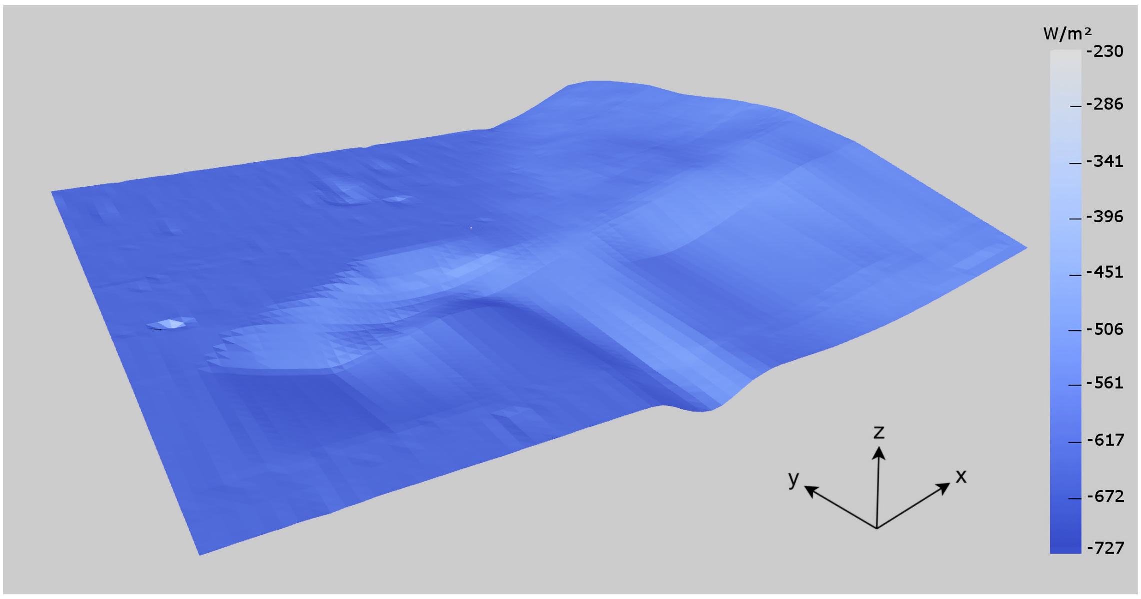


Figure 6.2.: Ray tracing results of infrared radiation heat fluxes on a lunar terrain patch. A total of 5000 rays per triangle were launched.

Figure 6.3 shows a close-up image of the previous two figures, zoomed in on the astronaut in the middle. On the left side, the astronaut is facing the sun and so its front is exposed to more radiation, with the triangles behind him lying in shadow and receiving little or no direct solar radiation. On the right side, the astronaut emits infrared radiation uniformly from its mesh triangles due to its uniform surface temperature distribution. The astronaut is standing on an open flat field and so receives rather equal infrared radiation from the terrain from all directions. The net infrared radiation across its triangles still varies due to self-occlusion and reflections.

To verify the validity of the output, it was compared with data obtained from the ESATAN Thermal Modelling Suite. ESATAN-TMS is a commercial software used for thermal modelling and analysis. It is one of the standard tools used for thermal analysis in the aerospace industry, with users including the European Space Agency (ESA) and the European space industry.

Similarly to the TherMoS ray tracer, ESATAN-TMS performs its radiative heat transfer by tracing both solar and infrared rays along the scene, albeit on the CPU. The validation was carried out using ESATAN-TMS 2017 sp2 where 1000 rays per triangle were launched

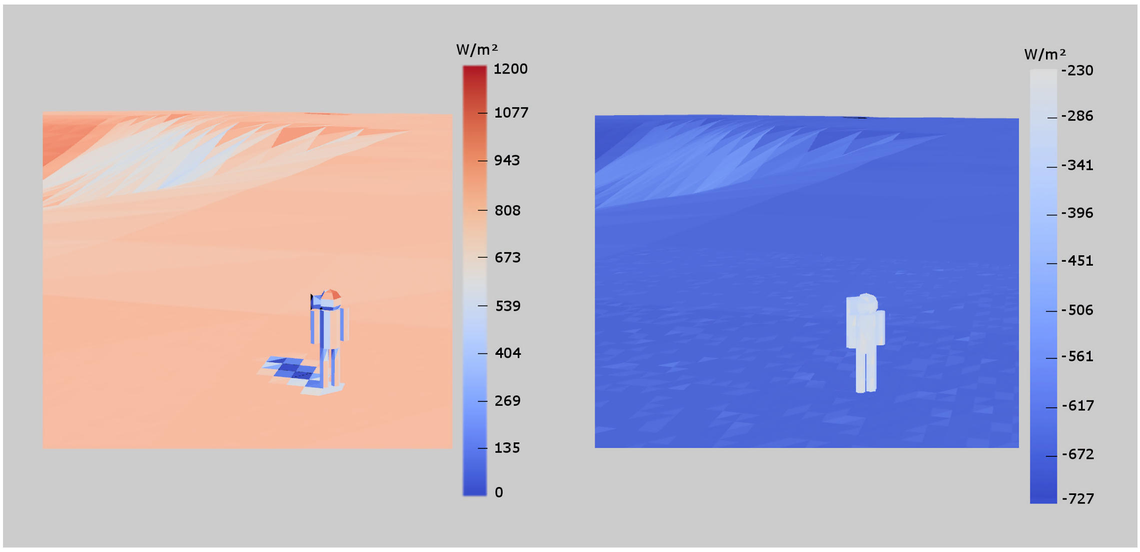


Figure 6.3.: Ray tracing results of solar radiation (left) and infrared radiation (right) heat fluxes on an astronaut sample positioned in the middle of a lunar terrain patch.

for each radiation type to calculate the incoming heat flux. In the ray tracer, 1000 rays per triangle were used for computing the infrared radiation and a total of 5600 $\times$ 5600 rays were launched from the solar parallelogram to compute the solar radiation.

The result of the comparison was a mean deviation of $8.9 \times 10^{-4}$ with a standard deviation of $0.567$ It should be noted that the quality of the results would decrease if:

- Too few rays are launched.
- The sample size is too small in comparison to the terrain patch.
- A large difference in resolution exists between the high-resolution centre region of the terrain and the rest of the terrain surface.

This is especially true for infrared radiation where heat is transferred between the scene triangles. It is important to distribute the triangle energy as uniformly as possible during emission. Consider the case of emitting a single ray from a large terrain triangle in such a direction that it is absorbed by a much smaller terrain or sample triangle. The receiving triangle would absorb an unrealistically large amount of energy, instead of the energy being distributed among it and all of its neighbours. Therefore, a suitable combination of ray count, scene scales and resolutions should be used to avoid such issues. This decision is mostly scene dependent and is currently determined by trial and error.

## 6.3. Parallelization Analysis

The performance of the new multi-GPU implementation is analyzed to determine how effective such a parallelization is in reducing the overall ray tracer simulation time. This is achieved by determining how well the computation of solar and infrared radiation scales with the number of GPUs used.

### Strong Scaling

Strong scaling describes how the solution time varies with the number of computing elements used when solving a problem of a fixed size. Ideally, doubling the amount of computing elements would half the solution time. However, in practice the amount of speed-up achieved depends on and how much of the running time is spent performing sequential actions and thus on how parallelizable the problem is. The strong scaling results are shown in Table 6.1 and Figure 6.4. The solar and infrared ray tracing times are combined into a single *compute* time. For results showing individual tracing times, see Appendix A.

Table 6.1.: Multi-GPU performance of different scene resolution and ray count combinations. The metrics shown are the Speed-up Factor (SF) and the parallelization efficiency ($\eta$).

| | | Solar: $2 \times 5000^2$ Infrared: $1000$ / $triangle$ | | | | | Solar: $2 \times 15000^2$ Infrared: $10000$ / $triangle$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | # GPU | Compute (s) | MPI (s) | Total (s) | SF | $\eta$ | Compute (s) | MPI (s) | Total (s) | SF | $\eta$ |
| Coarse Mesh | 1 | 3.179 | 0 | 3.179 | 1.00 | 1.00 | 16.27 | 0 | 16.27 | 1.00 | 1.00 |
| | 2 | 1.755 | 0.064 | 1.818 | 1.75 | 0.87 | 8.291 | 0.320 | 8.612 | 1.90 | 0.95 |
| | 4 | 1.033 | 0.037 | 1.070 | 2.97 | 0.74 | 4.273 | 0.888 | 5.161 | 3.15 | 0.79 |
| | 8 | 0.678 | 0.214 | 0.912 | 3.49 | 0.44 | 2.280 | 0.555 | 2.835 | 5.74 | 0.72 |
| Refined Mesh | 1 | 7.828 | 0 | 7.828 | 1.00 | 1.00 | 67.41 | 0 | 67.41 | 1.00 | 1.00 |
| | 2 | 4.075 | 0.168 | 4.244 | 1.85 | 0.92 | 33.87 | 1.367 | 35.24 | 1.91 | 0.96 |
| | 4 | 2.183 | 0.306 | 2.489 | 3.15 | 0.79 | 17.08 | 2.481 | 19.54 | 3.45 | 0.86 |
| | 8 | 1.238 | 0.378 | 1.616 | 4.84 | 0.61 | 8.660 | 2.31 | 10.97 | 6.15 | 0.77 |

Two different scenes are analyzed, one with a coarse terrain mesh (23090 triangles) and one with a refined mesh (187650 triangles). Both scenes were traced using a small and large number of rays. The reason for choosing such a setup is to determine the effect on performance by both the ray count and triangle density.

It should be noted that all tracing times are obtained using the SBvh acceleration structure, which outperformed TrBvh in the test scenes used by 10.8%. If build times are irrelevant, as is the case in this implementation, then SBvh is clearly the better choice. In other implementations where scenes have to be rebuilt often, then the build times should be factored in when comparing the structures.

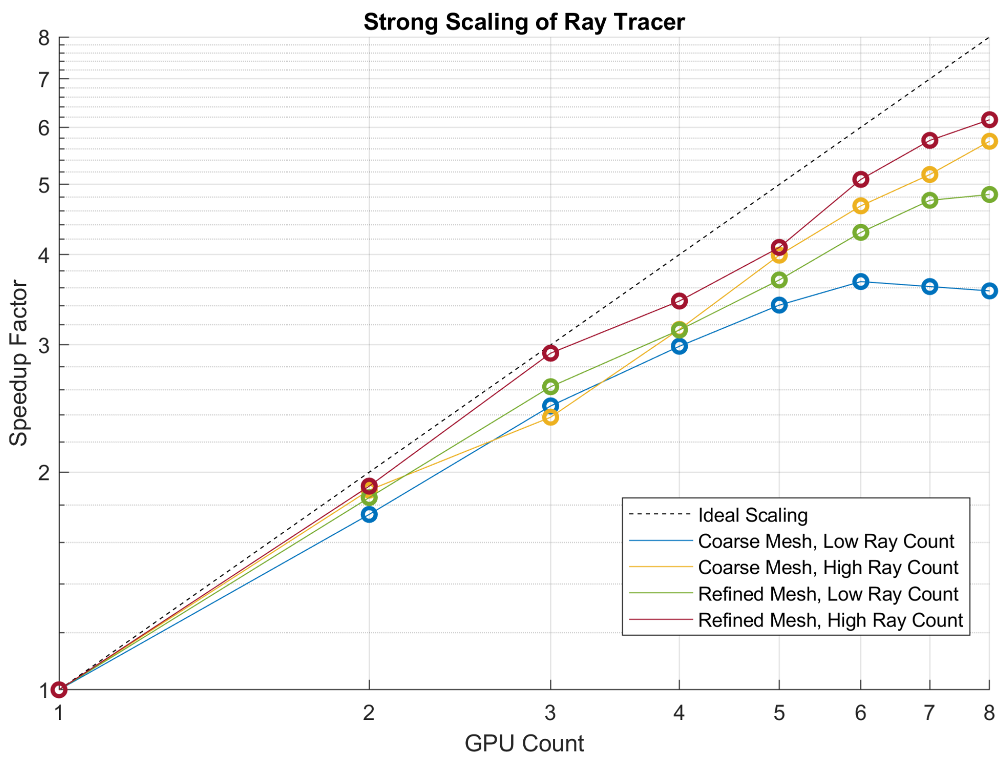


Figure 6.4.: Strong scaling of the ray tracing time, including solar radiation, infrared radiation and MPI communication time.

The results show a rather high computation speed-up, with more than 6 times reduction in the ray tracing time for a high resolution scene with a high ray count. This reduced simulation time from 67.41 seconds to merely 8.66 seconds. This translates to a 77% parallelization efficiency.

It can be seen from the results that when using a low ray count, the effective speed-up achieved is reduced and saturation is reached with 6 GPUs for the coarse mesh and with 8 GPUs for the refined mesh. In these cases, the increased overhead in communication due to using more processes becomes more pronounced, diminishing any further gains obtained from parallelization. Therefore, it is necessary to select an amount of GPUs that is appropriate for the particular simulation being run, and not simply use as many GPUs as there are available.

When looking at the speed-up obtained when using a high ray count for both coarse and refined mesh, it becomes apparent that the triangle density does not have a significant effect on the ray tracing time. Especially when compared to the difference in speed-up due to varying the ray count. This suggests that if better performance is required, it can be more readily obtained by using GPUs with a higher CUDA thread count instead of focusing on optimizing the actual ray tracing programs.

**Weak Scaling**

Weak scaling describes how the solution time varies with the number of computing elements used when solving a problem of fixed size *per element*. Ideally, adding more computing elements would allow the same problem, but with bigger size, to be solved in the same amount of time. The result of such scaling is shown in Figure 6.5 below.

The weak scaling was performed twice. The first time, a constant mesh resolution was used while increasing the number of rays traced, effectively increasing the problem size. The second time the mesh resolution was also increased. This was done by running the coarse mesh with 1 GPU, and then steadily increasing the triangle density by roughly the same amount for every GPU added, until the maximum density of the refined mesh was reached with the maximum number of GPUs available.

The results again confirm the advantage of using a multi-GPU implementation as it easily allows bigger or higher resolution patches to be simulated at relatively the same amount of time, as shown by the orange line. The difference between the two scaling cases shown represents both the increase communication time and scene traversal time due to the increased triangle count. Note that the number of infrared rays used is increased by an equal amount in both cases by taking into account the increasing triangle count. Naturally this overhead becomes more pronounced when simulating very large scenes.

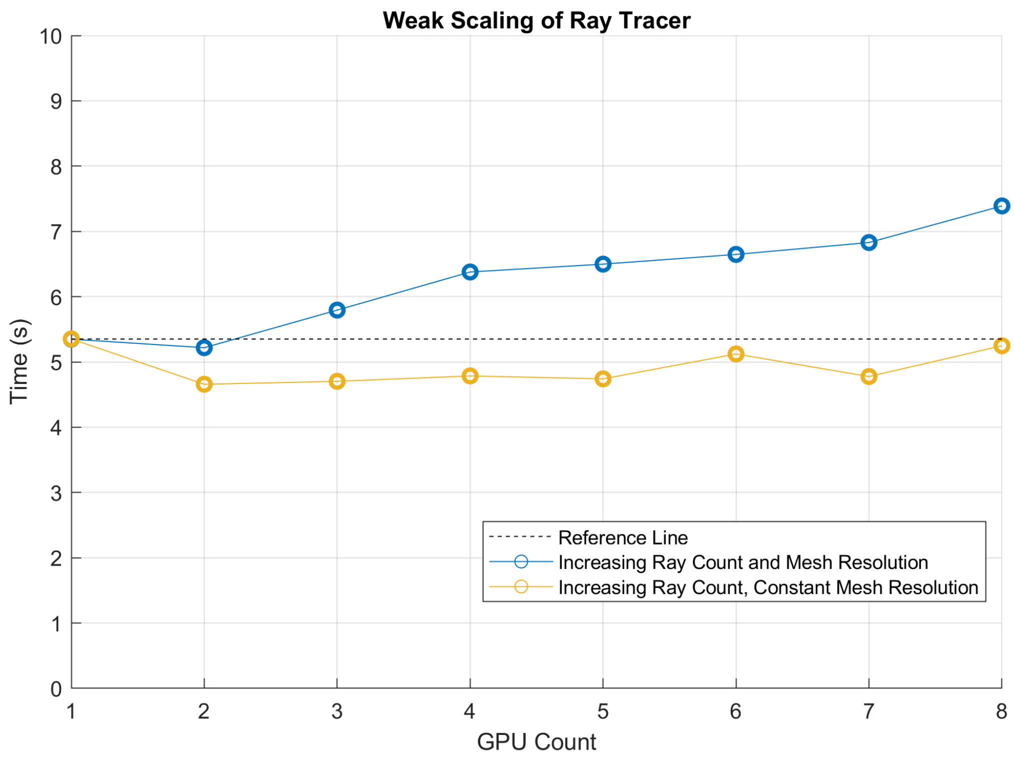


Figure 6.5.: Weak scaling of the ray tracing time, including solar radiation, infrared radiation and MPI communication time.

## 6.4. Interface Performance

With extending the implementation to support multiple GPUs, the ray tracing time was highly reduced. In comparison, this increased the fraction of the time spent performing actions other than the actual computation. In the old interface, this included data exchange with the TherMoS solver component using file I/O, rebuilding the scene every iteration and performing various setup actions such as initializing OptiX and creating its resources.

Consequently, the overall performance improvement was diminished. With the new interface, such overhead is reduced by using fast memory-mapped file data transfer, as well as a synchronized connection between the TherMoS ray tracer and solver components. This allows the ray tracer process to remain open, reducing the amount of initialization needed between each iteration. A comparison between the interfaces with respect to non-computation ray tracer times is shown in Table 6.2. The values shown are for a single iteration.

Table 6.2.: Comparison of non-tracing related simulation times *per iteration* between the old *.csv* interface and the improved memory-mapped file interface.

| | Setup (s) | Build (s) | Data Exchange (s) | Total (s) |
|---|---|---|---|---|
| Old Interface | 0.21 | 0.26 (TrBvh)<br>2.23 (SBvh) | 0.17 - 1.13 | 0.64 - 1.60 (TrBvh)<br>2.61 - 3.57 (SBvh) |
| New Interface | $< 0.001$ | – | 0.02 - 0.13 | 0.02 - 0.13 |

The old interface resulted in a total of 0.64 - 1.60 seconds overhead depending on the size of the scene used. An additional 2 seconds would need to be added on top of that if SBvh was used instead of TrBvh due to its longer build time. Opting not to use SBvh in an attempt to reduce the build time overhead would result in increasing ray tracing time by more than 10% as mentioned in Section 6.3.

In comparison, the new interface results in only a fraction of that overhead. The setup time is negligible since the only setup step that needs to be redone is the solar parallelograms construction. The build time is also negligible as it mostly involves applying a new sample transformation to the scene acceleration structure hierarchy. A complete rebuild is needed when a different terrain patch is loaded, but this happens rather infrequently. Therefore, the new interface allows us to maintain the high speed-up factor achieved using the multi-GPU implementation with minimum additional overhead.

# 7. Conclusion and Future Work

## 7.1. Conclusion

Thermal modelling in aerospace applications is an active field that is constantly providing improved and more accurate models for use in mission planning and equipment design. This is necessary for both increasing the safety of astronauts and creating more efficient and cost-effective hardware. As such models continue to grow in complexity, so does the amount of time needed to solve them. Therefore, it is necessary to make use of the available computing power and resources to aid in the calculations.

This is the main focus of this work; to enhance the performance of an existing thermal simulation tool, TherMoS. TherMoS is used to compute the transient rate of heat transfer of moving objects on the surface of the Moon. It consists of two components, the ray tracer which computes the radiative heat fluxes on the GPU using the ray tracing library NVIDIA OptiX, and the solver which uses these fluxes to compute the surface temperatures. In this work, the ray tracer component is extended to a multi-GPU implementation. The heat transfer computation is divided into two parts. The first part calculates the transfer of heat from the Sun to the Moon via solar radiation, while the second part calculates the emission of heat from the lunar surface and any objects placed on it as infrared radiation.

For large scenes, obtaining an accurate heat profile for the moving object and the surrounding lunar surface requires a large amount of rays to be traced. Since ray tracing is an embarrassingly parallel algorithm, it is well suited for such a parallelization effort. The new implementation makes use of multiple GPUs running on a distributed system using MPI. The rays are divided among the different GPUs and are traced in parallel. The results are then collected and passed on to the TherMoS solver component.

This multi-GPU approach significantly improved performance. Running time speed-up factors ranging from 3.5 to more than 6 were obtained, depending on the complexity of the scenes tested. More complex scenes had bigger room for improvement and therefore saw larger speed-ups. The reduced running time will allow even more detailed scenes to be simulated accurately without being as time intensive. Eight NVIDIA M2090 GPUs were used to obtain these results. However, more powerful GPUs are available on the market, allowing even greater speed-ups to be achieved.

In addition to the multi-GPU implementation, the ray tracer was improved in a number of ways including a more standard approach to generating solar rays, specular reflections, use of OptiX transformation to move the sample instead of rebuilding the scene and the ability to use variable per-face thermo-optical properties.

Finally, the interface between the TherMoS ray tracer and solver components was overhauled to reduce the overhead. The old interface relied on text file communication and repeatedly launched and terminated the ray tracer at every iteration. This was replaced with a new interface using memory-mapped files for communication and synchronization. This reduced both the communication time between the components and allowed the solver and ray tracer to continue running in parallel, avoiding redundant initialization steps from being performed at every time step.

## 7.2. Future Work

There are a number of ways to improve both the performance of the ray tracer and the accuracy of its results. Below are a few suggestions for any future work that builds on the current implementation:

**Overlapping TherMoS ray tracer and solver computation.** In the current implementation, the two components are executed serially. For tracing infrared radiation this is necessary since it requires the solver component to provide it with the newly computed temperature profile. However, solar radiation depends only on the solar angles and not the surface temperatures. It can thus be calculated in the background while the solver component performs its computation. Currently, the solver is quite fast and such an effort may not be worthwhile. However, as the solver model becomes more complex, such computation overlap will become beneficial.

**Adaptive infrared ray generation count.** So far the number of infrared rays generated is constant regardless of the triangle area. This means that large terrain triangles launch as many rays as miniscule sample triangles. A better approach would be to vary the number of rays being launched as a function of the triangle surface area. This would lower the total number of infrared rays launched.

**Improved infrared ray generation model.** Infrared rays are launched in directions sampled in a cosine-weighted fashion over the hemisphere. While this is an improvement over purely random sampling, the majority of infrared rays are emitted into space without intersecting with any of the scene triangles. An improved model that would factor in the geometry of the scene when calculating the directions and use a variable ray payload would cause less rays to be lost in space, reducing the amount of rays needed and so improving performance. This was also suggested in [19].

**Integration with the preCICE coupling library.** preCICE (Precise Code Interaction Coupling Environment) is a coupling library for partitioned multi-physics simulations [3]. It couples programs that are responsible for different parts of the physics involved in a simulation. These programs are treated as black-box solvers and are coupled in a nearly plug-and-play fashion. This allows for flexible and modular design and can be used to standardize the interface between the existing TherMoS components, as well as any new components that might be added in the future.

**Improved mesh triangulation.** The terrain meshes used in the ray tracer are generated in MATLAB from the Moon's elevation data. Each elevation point is connected with triangles, regardless of its value. For highly dense meshes, it might be useful to consider an improved triangulation scheme that would create more triangles in areas of large difference in elevation, while reducing the number of triangles across flat regions with little or no changes in height. This would be done in preprocessing and would help improve the ray traversal speed. However, care must be taken that such a triangulation scheme does not produce triangles with a large difference in area, as this would be problematic as detailed in Section 6.2.

# Appendix

# A. Ray Tracing Times

Table A.1.: Ray tracing times of both solar and infrared radiation. Results are shown for different combinations of terrain mesh resolutions and ray counts.

| | | Solar Time (s) Rays: $2 \times N^2$ | | | | Infrared Time (s) Rays: $N$ / $triangle$ | | | |
|---|---|---|---|---|---|---|---|---|---|
| | # | $N$ | | | | $N$ | | | |
| Mesh | GPU | 5000 | 7071 | 10000 | 14142 | 1250 | 2500 | 5000 | 10000 |
| 23090 Triangles | 1 | 2.489 | 2.733 | 5.27 | 10.35 | 0.805 | 1.292 | 2.412 | 4.644 |
| | 2 | 1.326 | 1.44 | 2.704 | 5.247 | 0.488 | 0.733 | 1.290 | 2.407 |
| | 4 | 0.714 | 0.786 | 1.408 | 2.667 | 0.331 | 0.452 | 0.731 | 1.289 |
| | 8 | 0.442 | 0.315 | 0.766 | 1.389 | 0.254 | 0.460 | 0.454 | 0.731 |
| 46610 Triangles | 1 | 1.820 | 3.127 | 6.031 | 11.83 | 1.677 | 3.167 | 5.770 | 10.96 |
| | 2 | 0.980 | 1.632 | 3.144 | 5.964 | 0.925 | 1.671 | 2.974 | 5.571 |
| | 4 | 0.553 | 0.873 | 1.593 | 3.017 | 0.549 | 0.925 | 1.575 | 2.861 |
| | 8 | 0.345 | 0.503 | 0.861 | 1.572 | 0.364 | 0.550 | 0.876 | 1.524 |
| 93650 Triangles | 1 | 1.980 | 3.393 | 6.554 | 12.85 | 3.576 | 6.169 | 12.08 | 23.73 |
| | 2 | 1.069 | 1.774 | 3.351 | 6.502 | 1.877 | 3.175 | 6.133 | 11.96 |
| | 4 | 0.605 | 0.943 | 1.726 | 3.289 | 1.031 | 1.675 | 3.155 | 6.071 |
| | 8 | 0.367 | 0.539 | 0.924 | 1.696 | 0.600 | 0.925 | 1.667 | 3.124 |
| 187650 Triangles | 1 | 2.166 | 3.727 | 7.228 | 14.19 | 6.711 | 13.11 | 25.83 | 51.49 |
| | 2 | 1.159 | 1.941 | 3.684 | 7.169 | 3.443 | 6.646 | 13.01 | 25.83 |
| | 4 | 0.646 | 1.029 | 1.896 | 3.633 | 1.798 | 3.400 | 6.580 | 12.99 |
| | 8 | 0.394 | 0.584 | 1.012 | 1.866 | 0.977 | 1.778 | 3.370 | 6.576 |

# Bibliography

[1] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[2] Blaise Barney. Introduction to parallel computing. Lawrence Livermore National Laboratory. `https://computing.llnl.gov/tutorials/parallel_comp/`. Accessed March 11, 2018.

[3] Hans-Joachim Bungartz, Florian Lindner, Bernhard Gatzhammer, Miriam Mehl, Klaudius Scheufele, Alexander Shukaev, and Benjamin Uekermann. precice – a fully parallel library for multi-physics surface coupling. *Computers and Fluids*, 141:250—–258, 2016.

[4] Yungus A. Cengel. *Heat Transfer: A Practical Approach*. McGraw-Hill, second edition, 2004.

[5] J. F. Clawson, G. T. Tsuyuki, B. J. Anderson, C. G. Justus, W. Batts, D. Ferguson, and D. G. Gilmore. Spacecraft thermal environments. In *Spacecraft Thermal Control Handbook, Volume I: Fundamental Technologies*, volume 1, pages 21–69, El Segundo, California, dec 2002.

[6] NVIDIA Corporation. Cuda zone. `https://developer.nvidia.com/cuda-zone`, 2018. Accessed February 23, 2018.

[7] Brandon Lloyd David McAllister. Building ray tracing applications with optix. In *SIGGRAPH 2013*, 2013.

[8] Philip Dutre. Global illumination compendium, 2003.

[9] James Gaier and Donald A. Jaworske. Lunar dust on heat rejection system surfaces: Problems and prospects. 880:27–34, 01 2007.

[10] James R. Gaier. The effects of lunar dust on eva systems during the apollo missions. Technical report, NASA/TM-2005-213610, mar 2005.

[11] John Shapley Gray. *Interprocess Communications in Linux: The Nooks & Crannies*. Prentice Hall, jan 2003.

[12] Greg Grewell. Colonizing the universe: Science fictions then, now, and in the (imagined) future. *Rocky Mountain Review of Language and Literature*, 55(2):25–47, 2001.

[13] Philipp Hager. *Dynamic thermal modeling for moving objects on the Moon*. PhD thesis, Technische Universität München (TUM), 2013.

[14] Frank P. Incropera. *Fundamentals of Heat and Mass Transfer*. John Wiley & Sons, sixth edition, 2006.

[15] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[16] Jet Propulsion Laboratory. Surveyor project final report: Part ii. science results. technical report 32-1265. Technical report, NASA, Pasadena, California, 1969.

[17] MathWorks. *MATLAB External Interfaces*, 2017.

[18] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, March 2010.

[19] Svetlana Nogina. Radiation heat transfer with ray tracing engine nvidia optix. Technical report, Technische Universität München (TUM), 2012.

[20] Svetlana Nogina. Improvements on the radiation heat transfer with ray tracing engine nvidia optix. Technical report, Technische Universität München (TUM), 2013.

[21] NVIDIA. *NVIDIA OptiXRay Tracing Engine: Programming Guide*, version 4.1 edition, 2017.

[22] Lucian of Samosata and Emily James Putnam. *Selections from Lucian*. Harper & Bros, New York, 1892.

[23] Lunar Reconnaissance Orbiter. High resolution global topographic map of moon. `https://computing.llnl.gov/tutorials/parallel_comp/`, nov 2011. Accessed January 6, 2018.

[24] Robert Osada, Thomas Funkhouser, Bernard Chazelle, and David Dobkin. Shape distributions. *ACM Trans. Graph.*, 21(4):807–832, October 2002.

[25] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[26] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, July 2010.

[27] Edison Pettit and Seth B. Nicholson. Lunar radiation and temperatures. *Astrophysical Journal*, 71:102–135, March 1930.

[28] Alfio Quarteroni, Fausto Saleri, and Paola Gervasio. *Scientific Computing with MATLAB and Octave*, volume 2 of *Texts in Computational Science and Engineering*. Springer Publishing, Berlin Heidelberg, fourth edition, 2014.

[29] Karl Rupp. Flops per cycle for cpus, gpus and xeon phis. `https://www.karlrupp.net/2016/08/flops-per-cycle-for-cpus-gpus-and-xeon-phis/`. Accessed January 16, 2018.

[30] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[31] Michael Ian Shamos. *Computational Geometry*. PhD thesis, Yale University, may 1978.

[32] Warren J. Smith. *Modern Optical Engineering*, volume PM88. McGraw-Hill, third edition, 2000.

[33] Godfried T. Toussaint. Applications of the rotating calipers to geometric problems in two and three dimensions. In *International Journal of Digital Information and Wireless Communications*, volume 4, pages 372–386. The Society of Digital Information and Wireless Communications, 2014.

[34] Aditya Venkataraman and Kishore Kumar Jagadeesha. Evaluation of inter-process communication mechanisms. 2015.

[35] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.