



THE UNIVERSITY OF  
**SYDNEY**

## **COPYRIGHT AND USE OF THIS THESIS**

This thesis must be used in accordance with the provisions of the Copyright Act 1968.

Reproduction of material protected by copyright may be an infringement of copyright and copyright owners may be entitled to take legal action against persons who infringe their copyright.

Section 51 (2) of the Copyright Act permits an authorized officer of a university library or archives to provide a copy (by communication or otherwise) of an unpublished thesis kept in the library or archives, to a person who satisfies the authorized officer that he or she requires the reproduction for the purposes of research or study.

The Copyright Act grants the creator of a work a number of moral rights, specifically the right of attribution, the right against false attribution and the right of integrity.

You may infringe the author's moral rights if you:

- fail to acknowledge the author of this thesis if you quote sections from the work
- attribute this thesis to another author
- subject this thesis to derogatory treatment which may prejudice the author's reputation

For further information contact the University's Director of Copyright Services

**[sydney.edu.au/copyright](https://sydney.edu.au/copyright)**

# The use of primitives in the calculation of radiative view factors

by

Trevor Walker B.Eng (Chem), B.Sc (Comp)



THE UNIVERSITY OF  
**SYDNEY**

A thesis submitted in fulfilment of the requirements for the degree of

**Doctor Of Philosophy**

School of Chemical and Biomolecular Engineering

University of Sydney

August, 2013

## Preface

The research presented here is my own original work and has not been submitted to any other institution for the award of a degree.

The research presented in this thesis has resulted in the following journal publications and conference presentations.

### ***Publications:***

Walker, T., S.-C. Xue, and G.W. Barton, *Numerical Determination of Radiative View Factors Using Ray Tracing*. Journal of Heat Transfer, 2010. **132**(7): p. 6.

Walker, T., S.-C. Xue , and G.W. Barton, *A robust monte carlo based ray-tracing approach for the calculation of view factors in arbitrary three-dimensional geometries*. Computational Thermal Sciences, 2012. **4**(5): p. 425-442.

### ***Conferences:***

Walker, T., S.-C. Xue , and G.W. Barton, *Numerical determination of radiative view factors*, Chemeca 2010: The 40<sup>th</sup> Annual Australasian Chemical and Process Engineering Conference, September 26-29, 2010, Adelaide, Australia.

Walker, T., S.-C. Xue , and G.W. Barton, *A robust monte carlo based ray-tracing approach for the calculation of view factors in arbitrary 3d geometries*, ICHMT International Symposium on Advances in Computational Heat Transfer, July 1-6, 2012, Bath, England.

Trevor Walker  
29/08/2013

## Acknowledgments

I would like to thank my associate supervisor Dr Shicheng Xue who provided introduction and guidance in the areas of computational fluid dynamics and numerical modelling.

Associate Professor Leon Poladian from the department of Mathematics and Statistics at the University of Sydney for his review of some of the early mathematics presented in this research.

Mutsuo Satio and Makoto Matsumoto from Hiroshima University and Mark Moraes from D E Shaw Research for their insight into the dSFMT and Philox pseudorandom number generators.

Finally, I offer my deepest gratitude to my supervisor Professor Geoff Barton whose interest and support enabled this thesis to be completed. He not only inspired me to take time from industry to pursue these research interests but also provided the guidance needed to see me through my candidature.

Trevor Walker



## Abstract

Compilations of radiative view factors (often in closed analytical form) are readily available in the open literature for commonly encountered geometries. For more complex three-dimensional (3D) scenarios, however, the effort required to solve the requisite multi-dimensional integrations needed to estimate a required view factor can be daunting to say the least. In such cases, a combination of finite element methods (where the geometry in question is subdivided into a large number of uniform, often triangular, elements) and Monte Carlo Ray Tracing (MC-RT) has been developed, although frequently the software implementation is suitable only for a limited set of geometrical scenarios. Driven initially by a need to calculate the radiative heat transfer occurring within an operational fibre-drawing furnace, this research set out to examine options whereby MC-RT could be used to cost-effectively calculate any generic 3D radiative view factor.

Initially, a suite of geometric ‘primitives’ (i.e. a sphere, cylinder, frustum, disc, annulus, rectangle and triangle) and a methodology by which these primitives can be combined and manipulated to construct complex 3D geometries was introduced. As an alternative to using well-established finite element methods (FEMs), these primitives permit more efficient memory usage, higher accuracy (particularly in cases where part or all of the 3D geometry involves curved surfaces) and an intuitive formulation for constructing arbitrary 3D geometries. The functionality required to launch uniformly distributed rays from the surface of each primitive in the suite, and identify valid ray-primitive intersections, were also developed.

A robust C++ based program (called RayFactor) was developed to calculate diffuse radiative view factors for any 3D geometry that could be described by a combination of the available primitives. Using the computational test system for this research, an Apple Mac Pro 4.1, RayFactor was refined by benchmarking against a selection of geometries for which analytical view factors was available in the literature. Using these benchmark examples, RayFactor was quantitatively assessed in terms of its convergence characteristics, statistical results distribution, and computational run-time to ensure that the underlying numerical methods and computational design philosophy for MC-RT were sound.

With a CPU-based version of RayFactor fully tested, a range of options were examined to improve computational speed and efficiency. These included: (i) code structure optimisation targeting Intel’s multi-core processors; (ii) implementation and comparison of a range of pseudorandom number generators (PSRNGs) in terms of speed, quality of the number sequences delivered, and

memory consumption; with the dSFMT generator providing optimal performance on the CPU platform; (iii) implementation and testing of both coarse and fine grain vectorisation methods, leveraging the OpenMP API and the SIMD architecture of modern Intel processors; with problem vectorisation efficiencies of approximately 98% being achieved; and (iv) implementation of a range of algorithmic options to provide ‘fast’ numerical approximations of transcendental functions (such as sine and cosine) repeatedly used within RayFactor.

Despite the extensive optimisation options explored and implemented for the CPU-based version of RayFactor, computational run-time was still felt to be an issue. Thus, OpenCL, a modern heterogeneous computing framework, was utilised to develop a general-purpose graphic processing unit (GPGPU) based MC-RT implementation, called RayFactorCL. The highly vectorisable nature of MC-RT was here exploited by targeting the parallel processing architecture and capabilities of GPGPUs. Two consecutive generations of NVidia GPGPUs, the GTX580 and the GTX680, were installed in the Mac Pro 4.1 test system and extensively tested. Several alternative approaches to ‘work-load partitioning’ were examined. It was determined that GPGPU-based computations were most efficient when such partitioning was conducted at the geometric object level rather than at the ray level. Pseudorandom number generation was revisited in light of the new GPGPU-based architecture, with the Philox counter-based PSRNG, specifically developed for use on GPGPU platforms, being adopted. An overall performance increase (in terms of run-time reduction) of over 32 times was realised when comparing the GPGPU-based RayFactorCL to the earlier CPU-optimised RayFactor, demonstrating that remarkable improvements can be achieved using modern ‘commodity’ hardware provided that care is exercised to match the numerical problem to the choice of hardware and software.

The object representational method employed in RayFactorCL was further examined with a comparative study of finite element methods (FEMs) against geometric primitives being conducted. The advantages and disadvantages of both methodologies were explored in the context of solution accuracy, computational speed and computing resource requirements, with geometric primitives being found to have significantly higher performance than the triangular elements used in typical FEMs for 3D objects with curved surfaces, such as the sphere and cylinder. However, it was apparent that object representation using geometric primitives may not be suitable in all circumstances, and therefore a hybrid method, in which the most advantageous features of both primitives and FEMs were combined, was developed and tested. In essence, this hybrid model mapped finite element meshes to the surfaces of geometric primitives and once ray-primitive intersection was identified,

performed a nearest neighbour search with the intersection point and the associated finite elements. Performance improvements (in terms of run-time reduction) of up to 50% were achieved compared to a standard bounding volume approach, and when using the proposed hybrid representational method, and up to 260% compared with a 'pure' FEM-based approach.

Finally, a fully conjugate heat transfer model of an operational fibre-drawing furnace was developed using the commercial CFD simulation software PolyFlow. Radiative view factors for all internal surfaces within the furnace were calculated using RayFactor. These were then manipulated (due to the limited ability of PolyFlow to include radiative heat transfer) to allow ready data exchange between the two software packages, thus permitting calculation of the radiative heating profile along the length of the polymer preform and the drawn fibre. This heat transfer model was validated against experimental temperature profile data with excellent agreement being achieved, and used in parametric studies to predict fibre drawing behaviour over a range of temperatures and draw ratios. A modified version of this furnace model will be used in planned research into the high-speed production of sub-micron meta-material fibres where the structured metal core is enclosed within a glass sheath. Due to the higher (than necessary for polymeric fibres) furnace draw temperatures, thermal radiation will be the dominant contributor to heat transfer, making the accurate prediction of furnace view factors a key model parameter.

Few projects follow the initial research script, and this one was certainly no exception. What began as an urgent need to calculate the radiative heat transfer within an operational fibre drawing furnace, ended up exploring the computational capabilities (and challenges) of modern heterogeneous computing platforms. Along the way, the advantages (and it has to be said, the disadvantages) of combining geometric primitives and Monte-Carlo methods to enable the rapid determination of complex three-dimensional radiative view factors were explored in some detail.

## Table of Contents

|   |           |
|---|-----------|
| PREFACE .....   | I         |
| ACKNOWLEDGMENTS.....  | II        |
| ABSTRACT .....  | III       |
| LIST OF FIGURES .....   | X         |
| LIST OF TABLES.....   | XIII      |
| NOMENCLATURE .....  | XIV       |
| <br>  |           |
| <b>1 INTRODUCTION</b> .....                                     | <b>1</b>  |
| <br>  |           |
| <b>1.1 RADIATIVE HEAT TRANSFER</b> .....                        | <b>1</b>  |
| <b>1.2 THE BLACKBODY</b> .....                                  | <b>3</b>  |
| 1.2.1 SPECTRAL EMISSIVE POWER OF A BLACKBODY.....               | 4         |
| 1.2.2 WAVELENGTH OF MAXIMUM EMISSION .....                      | 6         |
| 1.2.3 TOTAL EMISSIVE POWER.....                                 | 6         |
| <b>1.3 EMISSION FROM REAL SURFACES</b> .....                    | <b>7</b>  |
| 1.3.1 EMISSIVITY .....  | 7         |
| <b>1.4 THE VIEW FACTOR</b> .....                                | <b>9</b>  |
| <b>1.5 CALCULATION OF VIEW FACTORS</b> .....                    | <b>11</b> |
| 1.5.1 CALCULATION FROM FIRST PRINCIPLES .....                   | 11        |
| 1.5.1 CALCULATION BY NUMERICAL INTEGRATION.....                 | 13        |
| 1.5.2 CALCULATION BY PROBABILISTIC METHODS.....                 | 14        |
| <b>1.6 MONTE CARLO RAY TRACING</b> .....                        | <b>15</b> |
| 1.6.1 THE MONTE CARLO METHOD.....                               | 15        |
| 1.6.2 RAY TRACING AND MC-RT .....                               | 16        |
| 1.6.3 RANDOM NUMBER GENERATION .....                            | 18        |
| 1.6.1 VIEW FACTOR SMOOTHING .....                               | 19        |
| 1.6.2 THE USE OF GEOMETRIC PRIMITIVES IN MC-RT .....            | 21        |
| 1.6.3 ADOPTION AND PROBLEM COMPLEXITY SCALING OF MC-RT .....    | 22        |
| 1.6.4 CURRENT ISSUES IN MC-RT FOR RADIATIVE HEAT TRANSFER ..... | 23        |
| <b>1.7 THESIS AIMS AND STRUCTURE</b> .....                      | <b>24</b> |
| <br>  |           |
| <b>2 PRIMITIVES</b> .....                                       | <b>26</b> |
| <br>  |           |
| <b>2.1 CONCEPT OF A PRIMITIVE</b> .....                         | <b>26</b> |
| <b>2.2 AFFINE TRANSFORMATION</b> .....                          | <b>28</b> |

|            |   |           |
|------------|---|-----------|
| 2.2.1      | COMPOSING AFFINE TRANSFORMATIONS.....                                       | 29        |
| 2.2.2      | SCALING.....  | 29        |
| 2.2.3      | TRANSLATION .....   | 30        |
| 2.2.4      | ROTATION.....   | 30        |
| <b>2.3</b> | <b>RAY DEFINITION .....</b>   | <b>32</b> |
| 2.3.1      | RAY DIRECTION .....   | 32        |
| 2.3.2      | RAY TRANSFORMATION .....  | 35        |
| 2.3.3      | POST-TRANSFORMATION PRIMITIVE SURFACE AREA .....                            | 37        |
| <b>2.4</b> | <b>RAYFACTOR PRIMITIVES.....</b>  | <b>37</b> |
| 2.4.1      | RECTANGLE .....   | 38        |
| 2.4.2      | DISC.....   | 39        |
| 2.4.3      | ANNULUS.....  | 40        |
| 2.4.4      | SPHERE.....   | 41        |
| 2.4.5      | CYLINDER.....   | 43        |
| 2.4.6      | THE FRUSTUM.....  | 45        |
| <b>2.5</b> | <b>THE USE OF PRIMITIVES IN RADIATIVE VIEW FACTOR CALCULATION .....</b>     | <b>47</b> |
| <b>3</b>   | <b><u>NUMERICAL DETERMINATION OF PRIMITIVE BASED VIEW FACTORS</u> .....</b> | <b>48</b> |
| <b>3.1</b> | <b>HIGH-LEVEL DESIGN CONSTRAINTS .....</b>                                  | <b>48</b> |
| <b>3.2</b> | <b>DEVELOPMENT ENVIRONMENT .....</b>  | <b>48</b> |
| 3.2.1      | COMPUTER SYSTEM.....  | 48        |
| <b>3.3</b> | <b>RANDOM NUMBER GENERATION .....</b>                                       | <b>50</b> |
| <b>3.4</b> | <b>RAYFACTOR BENCHMARKING METHODOLOGY .....</b>                             | <b>51</b> |
| <b>3.5</b> | <b>RAYFACTOR VALIDATION FOR THE C-77 CASE .....</b>                         | <b>56</b> |
| 3.5.1      | SOLUTION DISTRIBUTION .....   | 56        |
| 3.5.2      | SOLUTION CONVERGENCE .....  | 58        |
| 3.5.3      | SOLUTION RUN-TIME.....  | 59        |
| <b>4</b>   | <b><u>OPTIMISATION OF RAYFACTOR FOR THE CPU</u> .....</b>                   | <b>61</b> |
| <b>4.1</b> | <b>FLOATING POINT PRECISION.....</b>  | <b>61</b> |
| <b>4.2</b> | <b>MACHINE EPSILON .....</b>  | <b>63</b> |
| <b>4.3</b> | <b>RANDOM NUMBER GENERATION .....</b>                                       | <b>65</b> |
| 4.3.1      | TESTED GENERATORS .....   | 65        |
| 4.3.2      | PRNG SUMMARY AND SELECTION.....   | 67        |
| <b>4.4</b> | <b>VECTORISATION.....</b>   | <b>69</b> |
| 4.4.1      | SINGLE INSTRUCTION MULTIPLE THREAD (SIMT) .....                             | 71        |

|            |  |            |
|------------|--|------------|
| 4.4.2      | SINGLE INSTRUCTION MULTIPLE DATA (SIMD) .....                    | 72         |
| 4.4.3      | SIMT AND SIMD PERFORMANCE IMPROVEMENT.....                       | 74         |
| <b>4.5</b> | <b>ALGORITHMIC OPTIMISATION .....</b>                            | <b>78</b>  |
| 4.5.1      | SIMD RANDOM NUMBER RECYCLING.....                                | 78         |
| 4.5.2      | RAY DIRECTION ALIGNMENT .....                                    | 78         |
| <b>4.6</b> | <b>OPTIMISATION OF ELEMENTARY FUNCTIONS .....</b>                | <b>81</b>  |
| 4.6.1      | FAST RECIPROCAL SQUARE ROOT.....                                 | 81         |
| 4.6.2      | FAST SQUARE ROOT.....  | 83         |
| 4.6.3      | FAST SINE AND COSINE .....                                       | 84         |
| <b>4.7</b> | <b>SPACE-PARTITIONING STRATEGIES.....</b>                        | <b>88</b>  |
| <b>4.8</b> | <b>CONCLUSIONS .....</b>   | <b>90</b>  |
| <b>5</b>   | <b><u>GENERAL PURPOSE COMPUTING WITH OPENCL</u> .....</b>        | <b>91</b>  |
| <b>5.1</b> | <b>OPEN COMPUTE LANGUAGE .....</b>                               | <b>91</b>  |
| 5.1.1      | THE OPENCL PLATFORM MODEL.....                                   | 92         |
| 5.1.2      | THE OPENCL EXECUTION MODEL .....                                 | 93         |
| 5.1.3      | THE OPENCL MEMORY MODEL .....                                    | 93         |
| 5.1.4      | ATOMIC OPERATIONS AND OPENCL BARRIERS .....                      | 95         |
| <b>5.2</b> | <b>TEST SYSTEM GPUS .....</b>                                    | <b>96</b>  |
| <b>5.3</b> | <b>GENERAL-PURPOSE COMPUTATION ON THE GPU.....</b>               | <b>97</b>  |
| 5.3.1      | GPU WORK-ITEM EXECUTION CONTROL.....                             | 99         |
| 5.3.2      | MAPPING THE OPENCL MEMORY MODEL TO THE GPU.....                  | 99         |
| 5.3.3      | MAPPING THE OPENCL EXECUTION MODEL TO THE GPU .....              | 101        |
| 5.3.4      | SINGLE VERSUS DOUBLE-PRECISION FLOATING POINTS ON THE GPU.....   | 104        |
| <b>5.4</b> | <b>OVERVIEW OF RAYFACTORCL .....</b>                             | <b>104</b> |
| 5.4.1      | HOST PROGRAM.....  | 104        |
| 5.4.2      | THE RAYFACTORCL KERNEL.....                                      | 107        |
| 5.4.3      | RANDOM NUMBER GENERATOR.....                                     | 110        |
| <b>5.5</b> | <b>PERFORMANCE COMPARISON OF RAYFACTOR AND RAYFACTORCL .....</b> | <b>112</b> |
| 5.5.1      | BENCHMARKING COMPUTE DEVICES.....                                | 112        |
| 5.5.2      | BENCHMARKING RESULTS .....                                       | 112        |
| <b>5.6</b> | <b>CONCLUSIONS .....</b>   | <b>114</b> |
| <b>6</b>   | <b><u>COMPARISON WITH FINITE ELEMENT METHODS</u> .....</b>       | <b>115</b> |
| <b>6.1</b> | <b>THE TRIANGLE ELEMENT .....</b>                                | <b>116</b> |
| 6.1.1      | RAY-TRIANGLE INTERSECTION ALGORITHM .....                        | 116        |

|            |  |            |
|------------|--|------------|
| 6.1.2      | TRIANGLE RAY LAUNCHING STRATEGY .....  | 122        |
| <b>6.2</b> | <b>COMPARISON OF TRIANGULAR FEMS AND GEOMETRIC PRIMITIVES.....</b>           | <b>122</b> |
| 6.2.1      | RAY LAUNCHING SPEED .....  | 123        |
| 6.2.2      | INTERSECTION TIME.....   | 124        |
| 6.2.3      | COMPARISON FOR BENCHMARKING GEOMETRIES.....                                  | 125        |
| <b>6.3</b> | <b>ACCELERATING FEM-BASED MC-RT ON A GPU-BASED COMPUTING ENVIRONMENT....</b> | <b>131</b> |
| 6.3.1      | SPACE PARTITIONING .....   | 131        |
| 6.3.2      | PROPOSED PRIMITIVE-FEM OBJECT ACCELERATION.....                              | 133        |
| <b>6.4</b> | <b>CONCLUSIONS .....</b>   | <b>140</b> |
| <b>7</b>   | <b>CASE STUDY – A FIBRE DRAWING FURNACE .....</b>                            | <b>141</b> |
| <b>7.1</b> | <b>OPTICAL FIBRE FABRICATION.....</b>  | <b>141</b> |
| <b>7.2</b> | <b>OVERVIEW OF FURNACE OPERATION .....</b>                                   | <b>142</b> |
| <b>7.1</b> | <b>MODELLING OF THE DRAWING FURNACE.....</b>                                 | <b>142</b> |
| 7.1.1      | POLYFLOW MODELLING .....   | 143        |
| 7.1.2      | MODELLING THE FURNACE IN RAYFACTOR .....                                     | 151        |
| 7.1.3      | INTERFACING POLYFLOW AND RAYFACTOR TO MODEL RADIATIVE HEAT TRANSFER .....    | 151        |
| 7.1.4      | EMISSIVITY .....   | 154        |
| 7.1.1      | CALCULATING RADIATIVE VIEW FACTORS WITHIN THE FURNACE .....                  | 155        |
| <b>7.2</b> | <b>NON-DEFORMING PREFORM CASE .....</b>                                      | <b>155</b> |
| <b>7.3</b> | <b>DEFORMING PREFORM CASE.....</b>   | <b>161</b> |
| <b>7.4</b> | <b>CONCLUSIONS .....</b>   | <b>165</b> |
| <b>8</b>   | <b>CONCLUSIONS AND RECOMMENDATIONS .....</b>                                 | <b>166</b> |
| <b>9</b>   | <b>REFERENCES .....</b>  | <b>168</b> |

## List of Figures

|   |     |
|---|-----|
| Figure 1: Blackbody spectral emissive power at a selection of temperatures .....                                    | 5   |
| Figure 2: Configuration for radiative heat exchange between two infinitesimal surface elements .....                | 9   |
| Figure 3: Geometry for disc to parallel unequal coaxial disc (C-41).....  | 11  |
| Figure 4: Selection of 1000 ray starting points in object space .....   | 34  |
| Figure 5: Histogram overlaid with the Gaussian PDF for 600 runs of the C-77 case with a ray density of $10^4$ ..... | 57  |
| Figure 6: Convergence of RayFactor results on the analytical solution for the C-77 case .....                       | 58  |
| Figure 7: Sample standard deviation with increasing ray density .....   | 59  |
| Figure 8: RayFactor run-times with increasing ray density .....   | 60  |
| Figure 9: Slightly left skewed distribution for C-77 case with an error tolerance of $1 \times 10^{-3}$ .....       | 64  |
| Figure 10: Performance and power draw of an under-clocked and over-clocked processor core [77] .....                | 70  |
| Figure 11: Data level parallelism with SIMD .....   | 73  |
| Figure 12: Efficiency of SIMT and SIMD RayFactor implementations .....  | 77  |
| Figure 13: Recycling of random numbers (RN) in an SIMD implementation .....   | 78  |
| Figure 14: Error in the calculation of the reciprocal square root for RSQRT and RSQRT + 1 NR.....                   | 83  |
| Figure 15: Construction of sine and cosine from the function defined on the range $0: \pi/4$ .....                  | 86  |
| Figure 16: Absolute error for the polynomial approximation of the sine and cosine functions .....                   | 88  |
| Figure 17: The OpenCL Platform model.....   | 92  |
| Figure 18: Transistor distribution of a typical CPU and GPU.....  | 98  |
| Figure 19: GPGPU relevant components of NVidia GTX 680 compute unit.....  | 102 |
| Figure 20: Planar representation of a triangle element.....   | 118 |
| Figure 21: Sphere approximation with a mesh consisting of (a) 8 (b) 128 (c) 512 and (d) 2048 triangles.....         | 126 |



|   |     |
|---|-----|
| Figure 22: Triangle mesh approximation of a cylinder and annulus with (a) 20 (b) 160 (c) 600 and (d) 1400 elements .....  | 126 |
| Figure 23: Error in calculated radiative view factor for the C-122 geometry using triangle based FEMs and geometric primitives.....   | 128 |
| Figure 24: Error in calculated radiative view factor for the C-77 geometry using triangle-based FEMs and geometric primitives. ....   | 129 |
| Figure 25: Run times relative to geometric primitives for the calculation of radiative view factors for the C-77 geometry. ....   | 130 |
| Figure 26: Construction of a bounding volume hierarchy for a two-dimensional geometry. ....   | 132 |
| Figure 27: A sphere modelled using a primitive and 512 finite elements (shown as centre points) and its representation in memory where $n$ is the total number of primitives and $W$ is the total number of finite elements (stored as centre points) lying on the surface of $n$ primitives..... | 134 |
| Figure 28: Overview of the ray intersection algorithm for the proposed FEM acceleration method.....   | 136 |
| Figure 29: Overview of the ray launching algorithm for the proposed FEM acceleration method.....  | 137 |
| Figure 30: Run times relative to using geometric primitives for the calculation of radiative view factors for the C-122 geometry using the proposed acceleration method, a standard bounding volume implementation and a 'brute force' FEM approach.....  | 139 |
| Figure 31: Run-times relative to geometric primitives for the calculation of radiative view factors for the C-77 geometry using the proposed acceleration method, a standard bounding volume implementation and a 'brute force' FEM approach.....   | 139 |
| Figure 32: Schematic diagram of fibre drawing furnace.....  | 143 |
| Figure 33: Experimentally measured temperature profile along the furnace wall. ....   | 146 |
| Figure 34: View factors from three positions ( $z_p = 0.0095, 0.0895, 0.1695$ ) on the preform to the furnace wall with ray density $\rho = 10^3$ rays per unit area....  | 156 |
| Figure 35: View factors from three positions ( $z_p = 0.0095, 0.0895, 0.1695$ ) on the preform to the furnace wall with a ray density $\rho = 10^4$ rays per unit area.   | 157 |

|   |     |
|---|-----|
| Figure 36: View factors from three positions ( $z_p = 0.0095, 0.0895, 0.1695$ ) on the preform to the furnace wall with a ray density $\rho = 10^5$ rays per unit area.   | 157 |
| Figure 37: View factors from three positions ( $z_f = 0.0095, 0.0895, 0.1695$ ) on the furnace wall to the preform surface and to itself; ( $\rho = 10^5$ per unit area; cylindrical preform).                  | 158 |
| Figure 38: Net radiative heat flux profiles along a stationary cylindrical preform within a furnace with a uniform heating wall temperature; two different thermal boundary conditions are used for the irises. | 159 |
| Figure 39: Comparison of experimental and furnace model results for preform surface temperature.  | 160 |
| Figure 40: Temperature profile of a vertical cross-section of the furnace for non-deforming draw.   | 161 |
| Figure 41: Flowchart of method employed for converging deforming preform heat transfer model.   | 162 |
| Figure 42: Convergence of $T_\sigma$ temperature profile over four successive iterations.   | 163 |
| Figure 43: Temperature profile for a vertical cross-section of the fibre drawing furnace for a draw ratio $Dr = 4$ .  | 164 |
| Figure 44: Temperature profile for a vertical cross-section of the fibre drawing furnace for a draw ratio $Dr = 50$ .   | 164 |

## List of Tables

|  |     |
|--|-----|
| Table 1: Ray directional functions required for various surface emitters .....   | 33  |
| Table 2: Selected technical specifications for a Mac Pro 4.1 .....   | 49  |
| Table 3: Primate Labs Geekbench 2 scores for a selection of Apple Inc. Computers<br>.....  | 50  |
| Table 4: Selected benchmarking geometries .....  | 52  |
| Table 5: Summary of IEEE 754 binary floating-point types .....   | 62  |
| Table 6: Summary of PRNGs considered for use in RayFactor.....   | 68  |
| Table 7: Measured relative speed for SIMD and non-SIMD RayFactor versions on<br>the Mac Pro 4.1 test system for the C-77 benchmarking case .....                                     | 76  |
| Table 8: Methods for calculating the reciprocal square root on modern<br>processors [81].....  | 82  |
| Table 9: Methods for calculating the square root on modern processors [81].....  | 84  |
| Table 10: Comparison of the number of processor cycles to calculate the sine and<br>cosine functions using system hardware and the implemented polynomial<br>approximation .....     | 87  |
| Table 11: Comparison of OpenCL specifications for the Intel Xeon E5520 CPU and<br>the NVidia GTX 580 and GTX 680 GPUs.....   | 97  |
| Table 12: Physical memory of an NVidia GTX680 with representative<br>performance figures.....  | 100 |
| Table 13: Comparison of RayFactorCL relative run-times for direct and indirect<br>recording of work-item results. ....   | 109 |
| Table 14: Relative run-times for various kernel designs in RayFactorCL.....  | 110 |
| Table 15: Relative performance for each benchmarking geometry on CPU and<br>GPU based systems.....   | 113 |
| Table 16: Relative ray generation performance of objects in an OpenCL<br>implementation.....   | 123 |
| Table 17: Relative ray Intersection performance for objects in an OpenCL<br>implementation.....  | 124 |
| Table 18: Relative run-times for RayFactor (a CPU based environment using 2 x<br>2.26 GHz Intel E5520 and RayFactorCL (a GPU-based environment using a<br>NVidia GTX580 system)..... | 155 |

## Nomenclature

|              |   |
|--------------|---|
| $A$          | Area ( $m^2$ ); Point vector  |
| $b$          | Base of a number  |
| $c$          | Ray directional vector  |
| $c_0$        | Speed of light in a vacuum ( $c_0 = 2.998 \times 10^8 \text{ m.s}^{-1}$ )   |
| $C_1$        | First radiation constant ( $C_1 = 3.7418 \times 10^{-16} \text{ W.m}^2$ )   |
| $C_2$        | Second radiation constant ( $C_2 = 1.4378 \times 10^{-2} \text{ m.K}$ )   |
| $C_3$        | Wien's displacement constant ( $C_3 = 2897.8 \text{ }\mu\text{m.K}$ )   |
| $Dr$         | Draw ratio  |
| $d$          | Squared distance between two points   |
| $E$          | Emissive power ( $\text{W.m}^{-2}.\mu\text{m}^{-1}$ )   |
| $eps$        | Machine epsilon   |
| $f$          | The frequency of electromagnetic radiation ( $\text{s}^{-1}$ )  |
| $F_{ij}$     | Radiative view factor   |
| $g$          | Gravitational constant ( $g = 9.81 \text{ m.s}^{-2}$ )  |
| $G$          | Irradiation ( $\text{W.m}^{-2}$ )   |
| $h$          | The universal Plank constant ( $h = 6.62569 \times 10^{-34} \text{ J.s}$ ); heat transfer coefficient ( $\text{W.m}^{-2}.\text{K}^{-1}$ ) |
| $I$          | Intensity of radiation ( $\text{W/m}^2.\text{s}.\mu\text{m}$ )  |
| $J$          | Radiosity ( $\text{W.m}^{-2}$ )   |
| $k$          | The Boltzmann constant ( $k = 1.3805 \times 10^{-23} \text{ J.K}^{-1}$ )  |
| $M$          | Affine transformation matrix  |
| $\mathbf{n}$ | Surface normal vector   |
| $N$          | Number of elements (i.e. objects, rays)   |
| $O(x)$       | Big O notation where $x$ is a function describing limiting behaviour of a method  |
| $p$          | Precision of a number   |
| $P$          | Point vector; Fraction of program parallelisable  |
| $q$          | Heat transfer rate ( $\text{W}$ )   |
| $r$          | Radius  |
| $R$          | Distance separating two points ( $\text{m}$ )   |
| $S$          | Ray starting point; Fraction of program parallelisable  |
| $t$          | Time ( $\text{s}$ )   |

|              |   |
|--------------|---|
| $T$          | The absolute temperature of a surface ( $K$ )           |
| $u$          | Barycentric coordinate                                  |
| $U$          | Overall heat transfer coefficient ( $W.m^{-2}.K^{-1}$ ) |
| $v$          | Barycentric coordinate                                  |
| $\mathbf{V}$ | Velocity vector ( $m.s^{-1}$ )                          |
| $W$          | Width   |
| $WG$         | OpenCL work group                                       |
| $WIG$        | OpenCL work items per work groups                       |

### ***Greek Letters***

|               |  |
|---------------|--|
| $\alpha$      | Absorption coefficient ( $cm^{-1}$ )   |
| $\beta$       | Thermal expansion coefficient ( $K^{-1}$ )   |
| $\gamma$      | Variance; Surface tension ( $N.m^{-1}$ )   |
| $\delta$      | Kronecker delta; Optical thickness   |
| $\varepsilon$ | Emissivity   |
| $\epsilon$    | Efficiency   |
| $\eta$        | Dynamic viscosity ( $Pa.s$ )   |
| $\theta$      | Azimuth angle ( $0 \leq \theta < 2\pi$ )   |
| $\kappa$      | Thermal conductivity ( $W.m^{-1}.K^{-1}$ )   |
| $\lambda$     | The wavelength of electromagnetic radiation ( $m$ )  |
| $\xi$         | A uniformly distributed random number ( $0 \leq \xi \leq 1$ )                                    |
| $\rho$        | Density ( $kg.m^{-3}$ ); Ray density ( $rays.unit^{-2}$ )  |
| $\sigma$      | Stefan-Boltzmann constant ( $\sigma = 5.67 \times 10^{-8} W.m^{-2}.K^{-4}$ ); standard deviation |
| $\phi$        | Zenith angle ( $0 \leq \phi \leq \pi$ )  |

### ***Subscripts***

|         |                  |
|---------|------------------|
| $c$     | Conduction       |
| $cores$ | Processing cores |
| $e$     | Emission         |
| $f$     | Fiber            |
| $g$     | Gas phase        |

|                |  |
|----------------|--|
| <i>h</i>       | Point/time at which a ray intersects an object |
| <i>i</i>       | The $i^{\text{th}}$ surface or component; In   |
| <i>j</i>       | The $j^{\text{th}}$ surface or component       |
| <i>local</i>   | OpenCL local memory                            |
| <i>o</i>       | Out  |
| <i>p</i>       | Preform  |
| <i>PE</i>      | OpenCL processing element                      |
| <i>private</i> | OpenCL private memory                          |
| <i>r</i>       | Radiation                                      |
| <i>w</i>       | Wall   |
| $\infty$       | Bulk or ambient conditions                     |

# Introduction

---

## 1.1 Radiative Heat Transfer

Radiative heat transfer is important across a wide range of engineering systems. In all such cases, the ability to accurately predict three-dimensional (3D) radiative heat transfer can be critical in both system design and analysis.

Radiative heat transfer, unlike the conductive and convective mechanisms of heat transfer, can occur in the absence of either a solid or fluid transmission medium. It is generally accepted that radiative heat transfer occurs via the emission and absorption of electromagnetic radiation with wavelengths in the range 0.1 to 100 $\mu\text{m}$  with the propagation of thermal radiation as electromagnetic waves being described by the following equation.

$$c = f\lambda \quad (1)$$

Where  $c$  is the speed,  $f$  is the frequency and  $\lambda$  is the wavelength. The emission of electromagnetic radiation occurs due to the energy released as a result of atomic oscillations and translations. These are maintained by the internal energy of the material, which is a strong function of temperature. Therefore, as the temperature of the matter increases, so too will the intensity of its thermal radiation.

Radiative heat transfer occurs with all forms of matter and is a volumetric phenomenon in which radiation emerging from a finite volume of matter is the integrated effect of local emission throughout the volume [1]. In most solids and liquids, when radiation is emitted from a molecule within its volume, it is subsequently absorbed and re-emitted by its neighbours. This cycle of emission, absorption and re-emission results in any thermal radiation emerging from the

volume originating from within a short distance of the exposed surface. This allows radiative heat transfer to be viewed largely as a surface phenomenon for many materials of engineering interest.

The rate at which radiant energy is emitted with a wavelength  $\lambda$  and a direction defined by the spherical coordinates  $\theta$  and  $\phi$  is referred to as the radiation intensity,  $I_{\lambda,e}$ . The radiative intensity is defined in terms of the unit area of the emitter,  $dA_1$  and the normal to the direction of emission and may be expressed mathematically as a function of the wavelength and direction as shown in equation (2).

$$I_{\lambda,e}(\lambda, \theta, \phi) = \frac{dq}{dA_1 \cdot \cos \phi \cdot \sin \phi \cdot d\phi \cdot d\theta \cdot d\lambda} \quad (2)$$

If the spectral and directional distributions of the radiation intensity are known, then the quantity of radiation emitted per unit area of the emitter (i.e. the emissive power) may be determined. Considering monochromatic radiation, the rate at which it is emitted in all directions (i.e. the spectral emissive power) may be calculated using the intensity of the emitted radiation,  $I_{\lambda,e}$ , as shown in equation (3).

$$E_\lambda(\lambda) = \int_0^{2\pi} \int_0^{\pi/2} I_{\lambda,e}(\lambda, \theta, \phi) \cdot \cos \phi \cdot \sin \phi \cdot d\phi \cdot d\theta \quad (3)$$

The rate of which radiation is emitted at all wavelengths, in all directions (i.e. the total emissive power) may be calculated by integrating the spectral emissive power over all wavelengths as shown in equation (4).

$$E = \int_0^\infty E_\lambda(\lambda) d\lambda \quad (4)$$

Similarly, if the spectral and directional distribution of the radiation incident on a surface is known, the quantity of radiation intercepted by the surface (i.e. the irradiation) may be determined. Again considering monochromatic radiation with a wavelength  $\lambda$ , the rate at which radiation is received from all directions



(i.e. the spectral irradiation) may be calculated using the intensity of the incident radiation,  $I_{\lambda,i}$  as shown in equation (5).

$$G_{\lambda}(\lambda) = \int_0^{2\pi} \int_0^{\pi/2} I_{\lambda,i}(\lambda, \theta, \phi) \cdot \cos \phi \cdot \sin \phi \cdot d\phi \cdot d\theta \quad (5)$$

The rate at which radiation is incident on a surface at all wavelengths from all directions may then be determined by integrating the spectral irradiation over all wavelengths in a similar fashion to the total emissive power.

$$G = \int_0^{\infty} G_{\lambda}(\lambda) d\lambda \quad (6)$$

As suggested by the equations for calculating the emissive power and irradiation, the nature and temperature of the emitting surface govern the characteristics of the spectral and directional effects on radiation emission. The non-ideal nature of radiation emission can result in quite complex and time-consuming analysis for systems containing ‘real’ surfaces.

Rather than analysing complex directional or spectral emission of radiation, it may be acceptable in many engineering situations to make the simplification that the objects in a given system behave as ideal emitters and absorbers of radiant heat. Such an idealized surface would emit radiation uniformly in all directions (i.e. exhibit a diffuse distribution) and absorb all incident radiation. Such an idealized surface is known as a blackbody and by using its emission and absorption characteristics, radiative heat transfer analysis can be greatly simplified.

## 1.2 The Blackbody

A blackbody is an idealized volume which emits and absorbs the maximum possible amount of radiation at a given temperature in all directions over all wavelengths. Blackbodies are perfect emitters and absorbers of radiation and, therefore, are useful as a reference when describing radiative heat transfer for non-ideal surfaces.

Unsurprisingly, real surfaces never fully exhibit the ideal behaviour of a black body, although such behaviour may be closely approximated by a cavity whose inner walls are maintained at a uniform temperature and radiation is only permitted to enter and leave the cavity via a small aperture. The blackbody behaviour of this particular idealized geometry may be explained in terms of the following:

1. All radiation entering the cavity will experience multiple reflections at the walls of the cavity and will ultimately be entirely absorbed.
2. All radiant heat emitted at the walls of the cavity can only leave through the aperture, as the walls are isothermal and therefore heat transfer between them would be in violation of the second law of thermodynamics.
3. Radiation emitted at the walls of the cavity will undergo multiple reflections before finding a path out of the cavity. These multiple reflections will result in the exiting radiation having a highly scattered directional distribution as it leaves through the cavity aperture.

Although radiative heat transfer systems seldom include geometries which behave as a blackbody (such as an idealised cavity), adopting the blackbody approximation (or variations on it) allows the emission and absorption of radiative heat to be readily characterised.

### 1.2.1 Spectral Emissive Power of a Blackbody

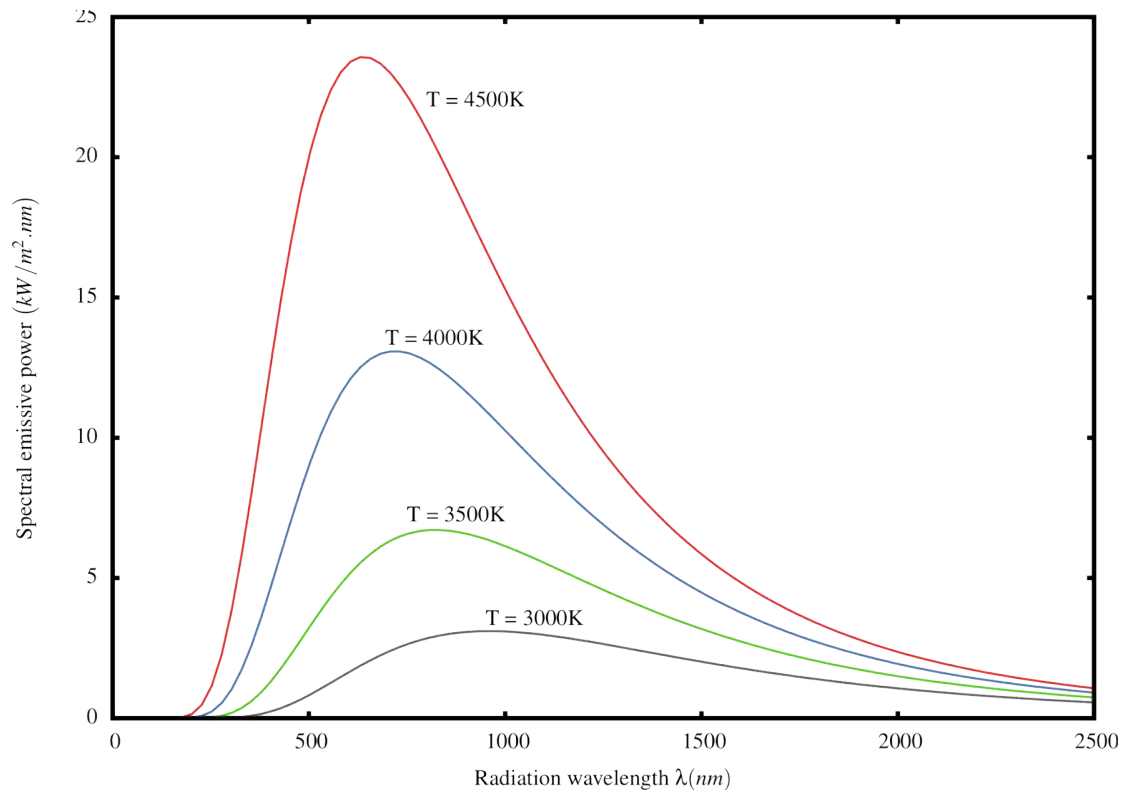
The spectral distribution for the intensity of radiation emitted from a blackbody was first determined in 1901 and is known as the Planck distribution, as shown in equation (7).

$$I_{\lambda,b}(\lambda, T) = \frac{2 \cdot h \cdot c_0^2}{\lambda^5 \left[ \exp\left(\frac{h \cdot c_0}{\lambda \cdot k \cdot T}\right) - 1 \right]} \quad (7)$$

Given that a blackbody is a diffuse emitter and therefore  $I_{\lambda,e}(\lambda, \theta, \phi) = I_{\lambda,b}(\lambda)$ , its spectral emissive power can be calculated by substituting the Planck distribution into the spectral emissive power equation (3). The resulting formula for the spectral emissive power of a blackbody is shown in equation (8).

$$E_{\lambda}(\lambda, T) = \pi I_{\lambda}(\lambda, T) = \frac{C_1}{\lambda^5 \left[ \exp\left(\frac{C_2}{\lambda T}\right) - 1 \right]} \quad (8)$$

The spectral emissive power of a blackbody at a selection of temperatures is shown in Figure 1. As a blackbody is a perfect emitter, the curves shown here provide an upper bound for the spectral emissive power of any real surface at the same temperature.



**Figure 1: Blackbody spectral emissive power at a selection of temperatures**

From Figure 1 the following emissive characteristics of a blackbody become immediately apparent:

1. The emitted radiation varies continuously with wavelength;
2. The peak emissive power increases with temperature;
3. As the temperature of the body increases, so the wavelength at the point of peak emissive power decreases.

Of particular importance on this curve is the wavelength at which maximum emissive power occurs,  $\lambda_{max}$ .

### 1.2.2 Wavelength of Maximum Emission

The wavelength at which a blackbody will have its maximum emissive power may be calculated using Wein's displacement law. This states that the wavelength of maximum emissive power,  $\lambda_{max}$ , is displaced to shorter wavelengths with an increasing temperature. The mathematical form of this law is shown in equation (9) and indicates that the product of  $\lambda_{max}$  and the (absolute) temperature is constant.

$$\lambda_{max}T = C_3 \quad (9)$$

Where  $C_3$  is Wien's displacement constant. It is of interest to note that Wein's displacement law may be obtained by differentiating equation (8) with respect to  $\lambda$  and solving for the case where this derivative is equal to zero. Alternatively, if we integrate equation (8) with respect to  $\lambda$  over the range of all wavelengths, we obtain another important quantity, the total emissive power.

### 1.2.3 Total Emissive Power

Although presented later here, the formula to calculate the total emissive power of a blackbody actually predates the discovery of the Planck distribution, being determined experimentally in 1879 by Joseph Stefan and verified theoretically in 1884 by Ludwig Boltzmann [1]. It may be calculated as a function of absolute temperature as shown in equation (10).

$$E_b = \sigma T^4 \quad (10)$$

This equation is today known as the Stefan-Boltzmann law and may be used to calculate the emissive power of a blackbody over all wavelengths, in all directions, per unit time and area. As mentioned previously, equation (10) can be obtained by substituting equation (8) into equation (4).

Given the simplicity in calculating the radiative characteristics of a blackbody, the appeal of using such an approximation for radiative heat transfer analysis is clearly apparent. Unfortunately, most real surfaces have non-ideal emission and absorption characteristics. However, the blackbody approximation can still be used as a reference point, if the radiative behaviour of a surface can be described in terms of its emissivity.

### 1.3 Emission from Real Surfaces

#### 1.3.1 Emissivity

In reality no real surface exhibits the ideal emission/absorption characteristics of a blackbody. However, for many real surfaces, it is adequate to describe their behaviour in terms of a surface radiative property known as emissivity.

Emissivity is defined as the ratio of the radiative emission from the real surface to that of a blackbody. It therefore lies in the range  $0 < \varepsilon \leq 1$  where a black body has an emissivity of 1 and any real object will have an emissivity less than 1.

As the emission of thermal radiation from a real surface is dependent on the temperature of the emitter, the (two spherical coordinate) direction of the emission and the wavelength of the emitted radiation, the emissivity is a function of four parameters. When dependence on all four parameters is considered, the emissivity is referred to as the spectral-directional emissivity and may be defined as shown in equation (11).

$$\varepsilon_{\lambda,\theta}(\lambda, \theta, \phi, T) \equiv \frac{I_{\lambda,e}(\lambda, \theta, \phi, T)}{I_{\lambda,b}(\lambda, T)} \quad (11)$$

For some materials, we may simplify our treatment of emission by assuming that the emissivity is spectrally independent i.e. emissivity is constant with respect to wavelength. This is then known as the directional emissivity and represents the 'spectral average' of  $\varepsilon_{\lambda,\theta}$  as defined in equation (12).

$$\varepsilon_{\theta}(\theta, \phi, T) \equiv \frac{I_e(\theta, \phi, T)}{I_b(T)} \quad (12)$$

Similarly, if appropriate for a particular analytical situation, one could instead make the assumption that emission characteristics are independent of the emission direction and use the spectral hemispherical emissivity, as defined in equation (13).

$$\varepsilon_\lambda(\lambda, T) \equiv \frac{E_\lambda(\lambda, T)}{E_{\lambda,b}(\lambda, T)} \quad (13)$$

By making the further simplification that emission is independent of both emission angle and the wavelength, one can define total hemispherical emissivity, which represents the average emissivity over all wavelengths and directions, as defined in equation (14).

$$\varepsilon(T) \equiv \frac{E(T)}{E_b(T)} \quad (14)$$

A final simplification that can be made for a real surface (relative to the corresponding blackbody) is that its emission is constant over all temperatures, and wavelengths. This is known as the ‘grey body’ assumption. Despite its limitations, this simplification is commonly used in engineering analysis due to the far greater availability of grey body emissivity factors compared to spectral or directional emissivity data. Although the grey body simplification is reasonable for many real materials, one must always be aware that it is in fact an approximation and that the emission characteristics of all real surfaces will depart from diffuse, temperature independent emission to a greater or lesser extent.

Once the emissivity of a given material is known, one may calculate the radiant heat emitted from an object by simply multiplying its emissivity by the corresponding blackbody body value. For example, if we have the spectral emissivity  $\varepsilon_\lambda(\lambda, T)$ , we may calculate the spectral emissive power from equations (8) and (13). Similarly if we have the total hemispherical emissivity  $\varepsilon(T)$ , or the grey body emissivity  $\varepsilon$ , we may calculate the total emissive power using equations (10) and (14) or simply multiplying equation (10) by the grey body emissivity.

Clearly, the characterisation of the radiative behaviour of real surfaces is a challenging endeavour that has been addressed by many researchers over many years. In addition to the quantification of the radiant heat emitted from an object, however, knowledge of how the emitted radiation is distributed to the surrounding environment is required to complete any radiative heat transfer analysis. This distribution is dictated by the geometric arrangement of the

various elements within a radiative heat transfer system. For any two elements, this may be characterised using a parameter known as the view factor.

### 1.4 The View Factor

The view factor (also known as the configuration, exchange, or shape factor) is defined as the fraction of thermal radiation leaving one surface  $i$  which is incident on a second surface  $j$ . The view factor,  $F_{ij}$  accounts for both the geometric configuration of surface  $j$  relative to  $i$  and the directional distribution of the radiation leaving surface  $i$ . Formulation of the view factor may be demonstrated by considering two infinitesimal surfaces  $dA_i$  and  $dA_j$  as shown in Figure 2.

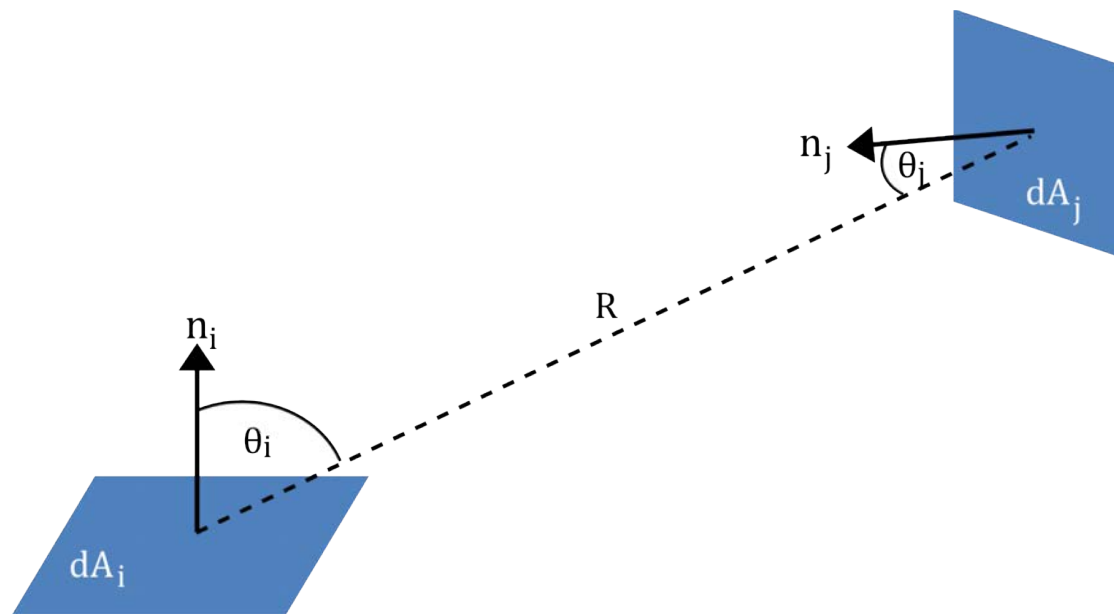


Figure 2: Configuration for radiative heat exchange between two infinitesimal surface elements

The rate at which radiation leaves  $dA_i$  and is incident on  $dA_j$  may be expressed in terms of the intensity of the radiation leaving surface  $i$  ( $I_i$ ), the straight line distance between the two finite surfaces ( $R$ ), and the angles that this line makes with each surface normal ( $\theta_i, \theta_j$ ) as shown in equation (15).

$$dq_{i \rightarrow j} = I_i \frac{\cos \theta_i \cos \theta_j}{R^2} \cdot dA_i \cdot dA_j \quad (15)$$

Assuming that surface  $i$  is a diffusely emits and reflects radiation, the total radiation leaving  $i$  may be represented in terms of its radiosity  $J_i$  as shown in equation (16).

$$dq_{i \rightarrow j} = J_i \frac{\cos \theta_i \cos \theta_j}{\pi R^2} \cdot dA_i \cdot dA_j \quad (16)$$

Making the additional assumption that the radiosity is uniform over the surface  $i$  the total rate at which radiative energy is leaving  $i$  and being intercepted by  $j$  may be calculated by integrating over the surfaces  $A_i$  and  $A_j$  as shown in equation (17).

$$q_{i \rightarrow j} = J_i \int_0^{A_i} \int_0^{A_j} \frac{\cos \theta_i \cos \theta_j}{\pi R^2} \cdot dA_i \cdot dA_j \quad (17)$$

As the view factor  $F_{ij}$  is the ratio of all radiation leaving surface  $i$  and being subsequently intercepted by surface  $j$  to the total amount of radiation leaving surface  $i$ , we may present the view factor as the ratio of the two as shown in equation (18).

$$F_{ij} = \frac{q_{i \rightarrow j}}{A_i \cdot J_i} \quad (18)$$

Substituting equation (17) into equation (18), the equation for the view factor for a grey, diffuse emitter is obtained as shown in equation (19).

$$F_{ij} = \frac{1}{A_i} \int_0^{A_i} \int_0^{A_j} \frac{\cos \theta_i \cos \theta_j}{\pi \cdot R^2} \cdot dA_i \cdot dA_j \quad (19)$$

Evaluation of any required view factor is straightforward provided that  $R$ ,  $\theta_i$  and  $\theta_j$  can be expressed in terms of the geometrical parameters that define the two participating surfaces, and the necessary integration of equation (19) can be performed.

Because of the importance of radiative heat transfer in a wide variety of applications, compilations of analytical or tabulated results (often in terms of a set of relevant dimensionless geometrical parameters) are available in the



literature [2]. In some cases, an unknown view factor can be generated from known factors by making use of view factor algebra [3]. However, for many complex radiative heat transfer geometries, analytical or tabulated view factors are not available. In such cases, view factors must be calculated from first principles, for example by solving equation (19) for a diffuse emitter, or by using a suitable alternative calculation methodology.

## 1.5 Calculation of View Factors

The analysis of many geometries does not lend itself to the simple application of compiled view factors. As one might suspect from equation (19), it can become quite challenging to calculate closed-form view factors for all but the simplest geometries.

The following section will provide a demonstration of the potential complexity in calculating closed-form view factors for the simple geometry of parallel, co-axial discs using equation (19). Subsequently, a more promising method which is at the heart of this thesis called Monte Carlo Ray Tracing (MC-RT) will be introduced.

### 1.5.1 Calculation from First Principles

Consider a geometry containing two parallel coaxial discs of unequal radius as shown in Figure 3. The view factor from  $A_1$  to  $A_2$  may be calculated analytically by solving equation (19) for any two given radii ( $r_1, r_2$ ) and the distance separating the two discs ( $h$ ).

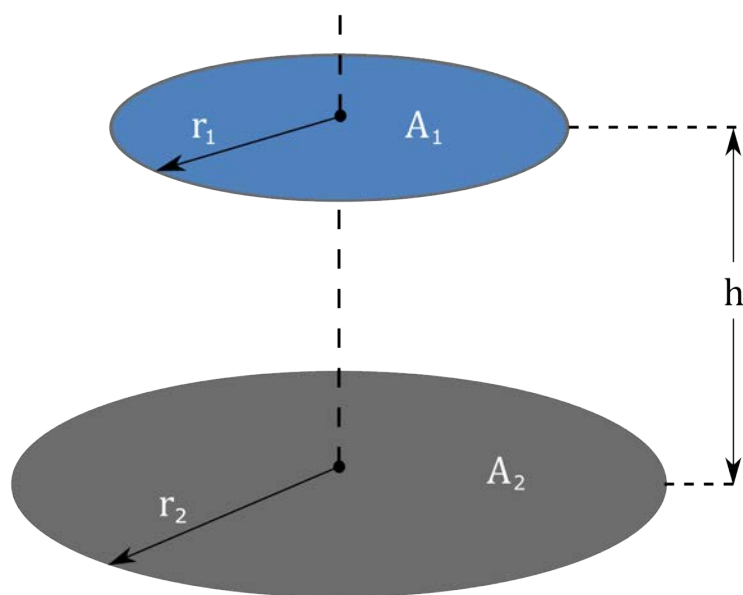


Figure 3: Geometry for disc to parallel unequal coaxial disc (C-41)

Given that, as the discs are parallel, the two angles  $\theta_i$  and  $\theta_j$  formed by the line  $R$ , which joins a point on  $A_1$  to a point on  $A_2$  will be equal. Furthermore, the cosines of each of these angles will be equal to the distance separating the discs divided by the length of  $R$  and therefore equation (19) may be expressed as shown in equation (20).

$$F_{ij} = \frac{1}{A_i} \int_0^{A_i} \int_0^{A_j} \frac{(h/R)^2}{\pi \cdot R^2} \cdot dA_i \cdot dA_j \quad (20)$$

In order to evaluate this integral, the variable  $R$  must be cast in such a way that it represents changes in the integration variables. This is achieved through the use of polar coordinates as shown in equation (21).

$$\begin{aligned} R^2 &= (\tilde{r}_i \cdot \cos \tilde{\theta}_i - \tilde{r}_j \cdot \cos \tilde{\theta}_j)^2 + (\tilde{r}_i \cdot \sin \tilde{\theta}_i - \tilde{r}_j \cdot \sin \tilde{\theta}_j)^2 + h^2 \\ &= \tilde{r}_i^2 + \tilde{r}_j^2 + h^2 - 2 \cdot \tilde{r}_i \cdot \tilde{r}_j \cos(\tilde{\theta}_i - \tilde{\theta}_j) \end{aligned} \quad (21)$$

$$dA_i = \tilde{r}_i \cdot d\tilde{r}_i \cdot \tilde{\theta}_i$$

$$dA_j = \tilde{r}_j \cdot d\tilde{r}_j \cdot \tilde{\theta}_j$$

Substituting these definitions back into equation (20) results in the quadruple integral shown in equation (22):

$$F_{ij} = \frac{1}{A_i} \frac{h^2}{\pi} \int_0^{r_i} \int_0^{r_j} \int_0^{2\pi} \int_0^{2\pi} \frac{1}{R^4} (\tilde{r}_i \cdot d\tilde{r}_i \cdot \tilde{\theta}_i) (\tilde{r}_j \cdot d\tilde{r}_j \cdot \tilde{\theta}_j) \quad (22)$$

Integrating for one of the polar angles and constructing the new variable definition  $\hat{\theta} = \tilde{\theta}_i - \tilde{\theta}_j$  results in a triple integral, which can then be integrated for  $\hat{\theta}$  as shown in equation (23).

$$\begin{aligned}
F_{ij} &= \frac{2\pi h^2}{A_i \pi} \int_0^{r_i} \int_0^{r_j} \int_0^{2\pi} \frac{\tilde{r}_i \cdot \tilde{r}_j \cdot d\hat{\theta} \cdot d\tilde{r}_i \cdot d\tilde{r}_j}{(\tilde{r}_i^2 + \tilde{r}_j^2 + h^2 - 2 \cdot \tilde{r}_i^2 \cdot \tilde{r}_j^2)^2} \\
&= \frac{4 \cdot \pi \cdot h^2}{A_i} \int_0^{r_i} \int_0^{r_j} \frac{\tilde{r}_i \cdot \tilde{r}_j \cdot (\tilde{r}_i^2 + \tilde{r}_j^2 + h^2)}{[(\tilde{r}_i^2 + \tilde{r}_j^2 + h^2)^2 - 4 \cdot \tilde{r}_i \cdot \tilde{r}_j]^{3/2}} \cdot d\tilde{r}_i \cdot d\tilde{r}_j
\end{aligned} \tag{23}$$

Now by using changes in variables and the designations  $S_i = r_i^2$  and  $S_j = r_j^2$ , one can simplify the previous integral and allow for two successive integrations over the  $S_i$  and  $S_j$  to take place as shown in equation (24).

$$\begin{aligned}
F_{ij} &= \frac{h^2}{A_i} \int_0^{r_i^2} \int_0^{r_j^2} \frac{S_i + S_j + h^2}{[(S_i + S_j + h^2)^2 - 4 \cdot S_i \cdot S_j]^{3/2}} \cdot dS_i \cdot dS_j \\
&= \frac{1}{2 \cdot r_i^2} \left[ h^2 + r_i^2 + r_j^2 - \sqrt{\{h^2 + (r_i - r_j)^2\} \{h^2 + (r_i + r_j)^2\}} \right]
\end{aligned} \tag{24}$$

Simplifying this view factor equation by introducing normalization variables ( $R_i = r_i/h$  and  $R_j = r_j/h$ ) and the substitution variable ( $T = 1 + R_i^2 + R_j^2$ ) the analytical solution for the view factor between two parallel coaxial discs is obtained.

$$F_{ij} = \frac{1}{2 \cdot R_i^2} \left( T - \sqrt{T^2 - 4 \cdot R_i^2 \cdot R_j^2} \right) \tag{25}$$

No novelty is claimed for this analysis. However, the point can be made that even for a relatively simple geometry, a certain level of applied mathematical sophistication is required to obtain a solution.

### 1.5.1 Calculation by Numerical Integration

In addition to solving the view factor equation from first principles, numerical integration methods such as contour double integration may be used [4]. Rather than developing a closed-form parameterised formula as presented in equation (25), numerical integration methods will solve the view factor equation as a definite integral, providing an explicit value for  $F_{i-j}$ . It should be noted that the

view factor set describing the radiative heat transfer for the case study presented in Chapter 7 was originally calculated using numerical integration techniques and served as a benchmark for view factors calculated by MC-RT methods which will be presently introduced in subsequent sections.

While numerical integration does reduce the difficulty of solving the view factor equation, it still requires some algebraic manipulation before it can be applied. For example, when considering the view factors for the geometry shown in Figure 3, one must first develop equation (22) before any numerical integration (by Gaussian quadrature) may be attempted. Furthermore, in order to numerically integrate multi-dimensional integrals such as the view factor equation, it is typically required that the integral be phrased as a series of iterative one-dimensional integrals, which results in a progressive growth in computational run-time with the dimensionality of the integral.

While several efficient algorithms have been developed for the numerical integration of view factors [5, 6], these algorithms are typically for very specific, idealised geometries such as the coaxial parallel discs example shown in Figure 3.

Although extensive catalogues of both numerical and closed-form view factor solutions have been presented in the literature [2], these solutions can be complicated leaving them prone to round-off and/or truncation error and like the numerical integration algorithms, are typically not general enough to be used for the complex radiative geometries often encountered in 'real' heat transfer systems. In such cases, probabilistic methods are often utilised.

### **1.5.2 Calculation by Probabilistic Methods**

As the complexity of the geometry and/or emission characteristics increases, so the difficulty in applying deterministic approaches (such as first principles or numerical integration) increases exponentially, quickly rendering these methods intractable in many cases. Furthermore, due to the nature of radiative heat transfer any modifications to the geometry and/or the material properties during the design process will typically require the solution to be fully reworked.

Due to the inherent complexity in solving the view factor equation for all but the simplest geometries, Monte Carlo based methods have widely been employed to solve radiative heat transfer problems [7-15]. Although Monte Carlo methods may be used to solve the view factor integral itself (via Monte Carlo integration),

when analysing radiative heat transfer it is most powerful when coupled with a ray-tracing technique.

## **1.6 Monte Carlo Ray Tracing**

As the name implies, Monte Carlo Ray Tracing (MC-RT) is the use of a ray-tracing technique within the context of a Monte Carlo simulation. MC-RT is an incredibly versatile probabilistic method for the calculation of radiative view factors, and as such its use and implementation on contemporary computer hardware/software platforms is the focus of this thesis. The following sections will provide a high-level overview of both Monte Carlo methods and ray-tracing, with more specific details of its application to the calculation of radiative view factors being presented in Chapters 2 and 3.

### **1.6.1 The Monte Carlo Method**

Monte Carlo methods form a class of numerical algorithms that rely on repeated random sampling of the behaviour of individual elements in order to statistically characterise the behaviour of a population as a whole [16]. It is often concisely described (somewhat unflatteringly) as the technique of solving a problem by putting in random numbers and getting out random answers [17]. The statistical nature of Monte Carlo methods mean that models do not always steadily converge on a single result but instead fluctuate around the solution with the magnitude of these fluctuations progressively reducing with an increase in the number of samples used.

Monte Carlo based methods have been in use for many years, with the experimental determination of  $\pi$  by French mathematician Buffon in 1768 being cited as an early example [18]. The work of von Neumann, Metropolis and Ulam during the Manhattan project in the late 1940s provided a formal foundation for Monte Carlo methods where they were used to predict the average behaviour of nuclear processes by repeatedly simulating the behaviour of individual neutrons [8]. Today, Monte Carlo methods are applied to a host of diverse numerical problems such as financial forecasting and climate modelling where they have proven to be exceptionally useful in circumstances where it is difficult to obtain closed-form expressions or infeasible to apply deterministic algorithms. In this context, the calculation of radiative view factors is a good example.

The complexity in analysing radiative heat transfer may be greatly reduced by implementing a Monte Carlo strategy. Here, rather than examining the radiative heat transfer process as a whole as required by the view factor equation, analysis is limited to a single discrete packet or ‘photon’ of radiative energy. When

considering just a single photon, radiative transport may be represented as a series of interactions between the photon and its surroundings (i.e. as a Markov chain process) with the photon's behaviour at each interaction being governed by a probability density function (PDF). Through implementing such a Monte Carlo strategy, radiative heat transfer can be quantified by repeatedly sampling the behaviour of many (often in the many millions) independent photons to obtain the average behaviour of emitted radiation. Not only does this strategy drastically simplify the mechanics required to analyse radiative heat transfer problems, but the deconstruction into a series of independent samples also has important ramifications for implementations on modern vectorised computer hardware (which will be explored later in Chapters 4 and 5).

When considering surface-surface radiative exchange problems, a Monte Carlo strategy can be coupled with a computational technique called ray-tracing to provide limitless bounds on the geometries that are open to analysis.

### **1.6.2 Ray Tracing and MC-RT**

The term ray-tracing was first coined to describe the general technique of modelling the path taken by rays of light as they interacted with optical systems such as camera lens and microscopes. Over the years, ray-tracing has steadily evolved and today has largely come to represent the computational technique in which the path of a photon is traced from a point (such as in a camera) and through a virtual three-dimensional geometry for the purposes of generating photorealistic images. In this application, rather than trace the ray from source to destination, it is instead traced from the destination (i.e. the camera) to the light source. This is done for efficiency reasons as it ensures rays that do not reach the camera from the emission source are not considered in the analysis.

The application of ray-tracing in the field of computer graphics has been around for over four decades, with the first image-rendering ray tracing algorithm being presented by Appel in 1968 [19]. Due to the intense interest (and strong financial incentive) of the computer graphics community in ray-tracing, an extensive body of research has been developed on its application [20-23] particularly in the areas of computational run-time reduction [24-30].

Although much of the modern ray-tracing knowledge/expertise has been developed in the field of computer graphics, it is readily adaptable to the analysis of radiative heat transfer problems. Here, rather than trace a ray of light from destination to source, a photon of radiation is traced from emitter to absorber. By coupling ray-tracing with Monte Carlo methods, radiative heat transfer can be

quantified by sampling a statistically significant number of rays, which each exhibit the physical characteristics of the emitted electromagnetic radiation.

The use of MC-RT for the calculation of radiative heat transfer is not a new concept, with comprehensive surveys of its application available [8]. In its formative years, MC-RT was used primarily for the direct simulation (DSMC-RT) for both surface-surface exchange [14, 31-36] and participating media systems [7, 17, 37-40] (it should be noted that consideration of participating media can result in a large increase in the solution run-time, particularly if the medium is optically thick [16]). Here, each ray is assigned a discrete energy level and a wavelength and dissipates this energy as it travels through a geometry undergoing successive absorption, reflection and/or transmission processes.

Howell [7] outlined a basic DSMC-RT framework for the direct computation of radiative exchange between two surfaces and applied it to the exchange between a differential element,  $dA_1$ , and to an infinite plane,  $A_2$ . In this method, the total emitted energy per unit time for the emitting element  $dA_1$  is calculated using the total hemispherical emissivity with the total energy being evenly distributed across  $N$  'samples'. In this context, these samples may be looked at as bundles of energy (i.e. photons) with each bundle being of a different wavelength. Conceptually, this leads to a different number of photons per bundle as a photon's wavelength is dependent on its energy (and vice-versa).

Following the allocation of energy to each bundle, a random launch direction is selected by assuming the directional distribution of a grey diffuse emitter. However, it is also noted that more complex emission characteristics could be implemented at an increased computational cost. If the launch direction results in the bundle hitting  $dA_2$  then a wavelength is assigned to the bundle according to the emission properties of  $dA_1$  and the bundle is either absorbed or reflected depending on the absorptivity of  $dA_2$  and the bundle's wavelength. All bundles which are reflected by  $dA_2$  back to  $dA_1$  are neglected and therefore it may be assumed in this two surface problem that all reflected bundles are 'lost' to the environment and require no further computation.

More recently, MC-RT has been used in the calculation of three-dimensional view factors for arbitrary geometric systems [12, 35, 41-43]. Here, the basic framework of the simulation is essentially identical to that used for DSMC-RT, with the minor difference that each ray is not assigned an energy level, a difference that allows certain optimisation options to be employed on modern hardware, such as the Graphical Processing Unit (GPU), which will be examined in Chapter 5. However, before further consideration may be given to the

calculation of radiative view factors, one must examine one of the most critical aspects of any Monte Carlo implementation, the random number generator (RNG).

### **1.6.3 Random Number Generation**

The most critical aspect of any Monte Carlo implementation is the fast generation of ‘high-quality’ random numbers. In the context of MC-RT treatment of radiative heat transfer, billions of random numbers are required for the determination of ray starting points, ray launch directions and material characteristics. Due to the sheer quantity of random numbers required for MC-RT it is often impractical to feed ‘true’ random numbers streams (such as those derived from atmospheric noise) into the simulation. Instead these are provided by pseudo random number generators (PRNGs).

PRNGs use a deterministic algorithm to generate a stream of numbers that (closely) approximate a truly random sequence. Ideally, these sequences are uniformly distributed, uncorrelated, reproducible, exhibit long periods (i.e. the amount of numbers generated before the sequence repeats itself) and are easily split into many independent streams for the purposes of problem vectorisation [44].

As literally billions of random numbers are required even for MC-RT treatment of relatively simple geometries, it is essential that an efficient random number generator of long period is implemented. Many algorithms have been proposed for the generation of pseudo random numbers [45-50] with several general reviews of their use and suitability available in the open literature [44, 51-56].

As each PRNG provides its own balance of speed, quality and ‘memory consumption’ (which are typically conflicting attributes), no one PRNG can be considered as the ‘best’ across all numerical applications and computer hardware architectures. It is therefore typically recommended that for a given application, multiple PRNGs are considered and tested to ensure high overall performance (as indeed will be presented in Section 4.3 for the current research).

While the selection of the PRNG greatly impacts the rate at which an MC-RT simulation can converge on the true value of a radiative view factor (from both a generation speed and sequence quality perspective), it will not eliminate statistical variations in the final solution. While these variations can be reduced through increasing the number of samples (at the cost of increasing the run-



time), these variations can also be reduced through the use of view factor smoothing techniques.

### 1.6.1 View Factor Smoothing

As view factors calculated from MC-RT are a result of random statistical sampling, they will have inherent stochastic errors. Only when the number of rays emitted from a surface,  $N_b$  approaches infinity will the MC-RT view factor  $\bar{F}_{ij}$  be equal to the 'exact' view factor  $F_{ij}$ . As it is generally impractical to increase the number of rays fired from a surface to extreme levels, alternative methods, such as view factor smoothing, may be adopted to minimise the discrepancy between  $\bar{F}_{ij}$  and  $F_{ij}$ .

The simple premise behind view factor set smoothing is to force compliance with reality. From the very definition of a view factor, it is clear that the sum of all the view factors from a given surface must equal unity as follows [1]:

$$\sum_{j=1}^N F_{ij} = 1 \quad (26)$$

This is known as the summation rule and is clearly satisfied when  $\bar{F}_{ij}$  is substituted for  $F_{ij}$  as every ray that is released as part of an MC-RT simulation is inevitably absorbed at some point by some other surface.

In addition, the second law of thermodynamics requires that the net heat transfer between two surfaces at the same temperature to be equal to zero, and this can be described by the reciprocity rule [1]

$$A_i \cdot F_{ij} = A_j \cdot F_{ji} \quad (27)$$

As  $\bar{F}_{ij}$  and  $\bar{F}_{ji}$  are determined independently from different statistical samplings, it cannot be guaranteed that the reciprocity rule is satisfied for any given view factor in a MC-RT simulation. Smoothing algorithms utilise this reciprocity requirement by aiming to increase the overall accuracy of a view factor matrix by enforcing 'reality compliance' through matrix modification.

Two main classes of smoothing algorithms exist. The first class iteratively modifies the view factor matrix to satisfy the reciprocity rule without limiting the size of the correction. The second class aims to find the smallest vector of correction factors that could be added to a set of view factors to satisfy the required reality rules.

The first class of algorithm, originally presented by van Leersum [57], modified the view factor matrix iteratively to enforce the two ‘reality rules’ until convergence was achieved. This method treated each view factor identically, regardless of their relative accuracy. Lawson [58] subsequently built on van Leersum’s method by taking into account the relative accuracy of the unsmoothed view factor set by proportioning the correction to the size of the unsmoothed view factor. Taylor et al [59] took a different approach by instead calculating the ‘upper triangle’ of the view factor matrix and using the reciprocity and summation rules to calculate the ‘lower triangle’ and diagonal of the view factor matrix, respectively. Although this class of smoothing algorithm is simple to implement, they do not guarantee that the modified view factor matrix will be more accurate than the original matrix [60].

Thus, Vercammen and Froment [43] introduced a second class of algorithm in order to smooth Hottel’s exchange areas as determined by a Monte Carlo method, but treated view factor smoothing as a constrained-least-squares optimisation problem. In this approach, a correction vector is added to the unsmoothed view factor matrix to yield a smoothed matrix with the overall aim of minimising the correction vector. This approach was later refined by Larsen and Howell [61] and Loehrke et al [60], with modifications including the weighting of elements in the correction vector to ensure that the smaller, generally less accurate, view factors were modified in preference to the larger, generally more accurate, view factors [62]. By 1995, the run-time for large MC-RT simulations to a given accuracy was effectively halved through the use of view factor smoothing with Loehrke reporting an accuracy improvement comparable to that obtained by doubling the total number of photons released in the simulation [60].

Although these various algorithms produced quite substantial increases in the accuracy of MC-RT view factor sets, they could still produce negative view factors in the final smoothed matrix as a non-negativity constraint could not be imposed on the least-squares minimisation problem without making it analytically intractable [62]. Daun et al [62] solved the negative smoothed view factor problem in 2005 by implementing a constrained maximum likelihood (CML) estimation rather than the least-squares optimisation utilised by previous

authors. When considering an identical geometry to that analysed by Loehrke et al [60] but with CML smoothing, Daun et al observed accuracy increases consistent with the previous approach. However, these authors emphasised that the CML smoothing problem is roughly half the size of the least-squares problem, and that CML is more accurate than least-squares optimisation when relatively few rays  $N_b < 1 \times 10^3$  are used [62].

In addition to view factor smoothing the accuracy and computational run-time required to calculate radiative view factors by MC-RT simulation may be improved by revisiting the methods by which geometries are represented within the simulation.

### 1.6.2 The Use of Geometric Primitives in MC-RT

Although a finite element representation of surfaces is ideal for analysing heat transfer by conduction, it is not always the best option for radiative heat transfer analysis. Despite the fact that the necessary polygon intersection routines are relatively fast (an issue to be explored in Chapter 6), it may take thousands of such (finite element) polygons to give an accurate representation of tightly curved surfaces. An alternative is not to discretise the surfaces, but rather to instead treat each surface as a whole – with the complete geometry being described in terms of a suite of generic ‘primitive’ shapes. This representation method borrows heavily borrowed from the field of modern computer graphics, and its use in the calculation of radiative view factors will be detailed in Chapter 2.

As an example, Vueghs et al [12] combined the use of finite elements for conductive heat transfer and a primitive based MC-RT for thermal radiation to calculate the heat transfer flows within a satellite. A geometric representation of a sphere was used to enclose an alternative finite element mesh representing the same sphere. When a ray was launched into the geometry to model radiative heat transfer, intersection was first tested with the sphere as a whole (i.e. with the primitive). If the ray did not intersect the sphere, then the computation was deemed to have finished. However, if the ray met the ‘primitive’ sphere, then its intersection point was mapped onto a structured mesh placed over the sphere and all finite elements describing the sphere were tested for intersection. This hybrid technique was also applied to calculating the view factor between two concentric spheres exhibiting specular reflection. It was found that not only were run-times reduced, but errors arising from the approximation of the surface curvature were effectively eliminated.

Walker et al [14] have greatly extended this concept of geometric primitives in radiative heat transfer. They have developed and presented a framework for modelling complex three-dimensional geometries using a limited set of primitive objects (such as a sphere, a cylinder, a (truncated) cone, and a flat surface), to which affine transformations could be applied to generate any desired geometry. Using such primitive objects permitted easy generation of geometries without the requirement for any finite element modelling. This work will later be explored throughout this thesis.

### **1.6.3 Adoption and Problem Complexity Scaling of MC-RT**

The primary disadvantage of MC-RT simulations is that they are computationally expensive, and indeed in the early years of their implementation they were impractical for all but the simplest simulations. As a result, alternative, more established methods (such as direct numerical integration of the problem as posed in terms of analytical geometry) were the accepted way of calculating radiative heat transfer at this time. However, such approaches scaled quite poorly with increasing problem complexity, and in many cases such approaches were totally impractical for more challenging geometries.

Campbell [63] compared the solutions obtained by finite element based methods and MC-RT, and concluded that while the former can generally calculate a solution, MC-RT has a clear potential advantage when handling more complex geometric scenarios. His comparison of the two methods lead to Campbell noting that at the time of publication (1967) there was no one computational method capable of handling all radiative heat transfer problems encountered in science/engineering, and therefore it was advantageous to have access to both methodologies.

In 1968, Howell [16] extended his previous work [7] and provided a thorough review of general Monte Carlo analysis and its application to a range of heat transfer problems. In this paper, he concisely sums up the issue of implementing a MC-RT solution by stating that anyone analysing a radiative exchange problem has to ask themselves: "Is it better to program the solution of the integral equations by finite difference techniques, with the possibility that convergence will not be attained, or by Monte Carlo based methods, which, though computationally demanding will give an answer sooner or later?".

In addition, Howell revisited the surface exchange problem presented in his earlier paper [7], further demonstrating the ease with which emissivities of greater complexity than the simple grey diffuse emitters may be incorporated

into a Monte Carlo framework with little increase in implementation difficulty, although implying that such increases in complexity would also increase the simulation run-time. In broad agreement with Campbell [63], Howell states that many specialised techniques had been developed which surpass a generic Monte Carlo based approach in terms of accuracy and run-time, however the range of problems to which they may be applied is limited (and in some cases, very limited). It was concluded that a Monte Carlo program increases in complexity in roughly direct proportion to the complexity of the problem, while the alternatives, such as finite difference based methods, broadly increase in proportion to the square of the problem's complexity. The situation has not changed radically from when this conclusion was initially drawn, and today one of the key advantages of MC-RT solutions is that they still scale in a much more computationally friendly manner for complex radiative systems than the alternative methods.

#### 1.6.4 Current Issues in MC-RT for Radiative Heat Transfer

As stated in the preceding section, the primary issue for MC-RT methods is the time they take to converge on a solution. As demonstrated by Campbell [63], MC-RT simulations have two key properties governing their behaviour: (i) that the statistical error approaches 0 as the number of rays launched approaches infinity, and (ii) that there is no propagation of statistical error<sup>1</sup>. It is within the context of these two properties that the way forward for MC-RT may be addressed.

Considering the first property, the issue is one of ensuring that an MC-RT simulation has high performance i.e. it converges quickly. The 'performance' of Monte Carlo solutions may be quantified in terms of the run-time  $t$  and the variance  $\gamma$  of the results as shown in equation (28).

$$\text{Performance} \propto \frac{1}{\gamma^2 t} \quad (28)$$

Here the performance is judged by the statistical error of the solution, which from the previous point (i) can be increased by increasing the number of rays<sup>2</sup>, used and reducing the time it takes to simulate these rays. While the first

---

<sup>1</sup> This means that the average result of 10 runs using 1000 bundles each will have the same accuracy as a result of a single run using 10,000 bundles.

<sup>2</sup> The standard deviation is proportional to  $1/\sqrt{N_{\text{rays}}}$ , therefore to double the accuracy we must 'shoot' four times as many rays.

property permits error reductions by increasing the number of samples, the second property allows these samples to be taken independently allowing vectorised computer hardware to be readily exploited for the calculation of radiative view factors.

While there are suitable convergence acceleration techniques, such as view factor smoothing available, MC-RT performance is ultimately decided by the selection of a suitable high quality PRNG, geometric object representational methods, and the exploitation of modern, high performance hardware such as the graphical processing unit (GPU). This thesis will aim to address these key areas as described in the following section.

## **1.7 Thesis Aims and Structure**

The flexibility of Monte Carlo Ray Tracing makes it an invaluable tool for the calculation of radiative view factors in complex three-dimensional geometries. However, its high computational cost can limit the size and types of problems that may be feasibly analysed using this method. This thesis has as its central aims a comprehensive re-examination of the performance of MC-RT within the context of modern computing hardware/software, and the development of an efficient, extensible framework for the calculation of radiative view factors, by exploring alternative object representational methods such as the concept of a geometric primitive and modern heterogeneous computing.

These two thesis aims will be addressed over the course of the following chapters.

**Chapter 2** introduces the concept of the geometric primitive and the mechanics by which primitives can be manipulated to construct complex 3D geometries for the calculation of radiative view factors.

**Chapter 3** describes the creation and optimisation of a primitive based MC-RT program called RayFactor. Both coarse and fine grain vectorisation methods are discussed while various algorithmic optimisations are explored in reference to their impact on computational accuracy and speed.

**Chapter 4** introduces the computer system selected for this research and presents a suite of geometries that were used to verify and validate the quality and accuracy of radiative view factors as calculated by RayFactor.

**Chapter 5** further exploits the vectorisable nature of MC-RT by presenting a RayFactor implementation within the framework of a general-purpose graphic processing unit (GPGPU). In order to interact with the GPGPUs used, a modern heterogeneous computing framework called OpenCL is adopted. Here, it is demonstrated that significant performance improvements can be obtained even using modern ‘commodity’ hardware.

**Chapter 6** compares the use of geometric primitives for object representation to established Finite Element Methods (FEMs). The advantages and disadvantages of both methods are explored in the context of accuracy, speed and computing resources, while a framework within which both primitives and FEMs can be used to complement each other is discussed.

**Chapter 7** utilises the RayFactor software to calculate the radiative view factors for an operational fibre-drawing furnace. Here, a fully conjugate heat transfer model of the furnace is developed and verified against experimental temperature profile data while further parametric studies are undertaken to predict fibre drawing behaviour over a range of temperatures and draw ratios.

**Chapter 8** summarises the findings of this research and discusses outstanding areas that warrant further research and development.

---

The starting point for any MC-RT simulation is object representation. Each surface (or volume) of a geometry treated by MC-RT must have a suitable virtual representation, flexible enough to represent arbitrary three-dimensional geometries and capable of both ray launching and ray intersection calculations. The object representational method can have a significant impact on both the computational run-time and accuracy on the simulation and is therefore an important part of any MC-RT. This chapter introduces the use of primitives, an alternative representational method to well established finite element methods.

## 2.1 Concept of a Primitive

Finite element methods (FEMs) are well developed and widely used in many branches of science and engineering (e.g. computational fluid dynamics). The main strength (and potential weakness) of FEMs is that any surface can be described by a fine mesh of regularly shaped (often triangular) elements. However, in modern graphical environments (such as video games and computer generated graphics), it is more intuitive to construct 'scenes' from a set of geometric primitives rather than to use a finite element mesh. In undertaking this thesis, the issue of the relative advantages of using a FEM and primitives to determine 3D view factors was seen as a key consideration and one that will be returned to in Chapter 6.

Geometric primitives are objects just like the (typically triangular) objects used in FEMs. Objects such as a sphere, a cylinder, a cone, and a rectangular surface are common examples of primitives used in graphical descriptions. However, primitives are not limited to such familiar objects. Indeed, any object that can be explicitly described in terms of a surface equation and a ray-surface intersection algorithm (simply a method by which the intersection of a straight line and the surface can be calculated) is a suitable candidate for inclusion in a MC-RT methodology.



In the context of radiative heat transfer calculations, primitive objects can be used in conjunction with a FEM (typically for combined radiative/conductive models) or else can provide a complete stand-alone representation of the required 3D geometry. As an example of the former combined approach, Vueghs et al. [12], combined the use of geometric primitives and a FEM to increase the speed and accuracy of the radiative portion of a combined conductive/radiative heat transfer model of a satellite in orbit. Geometric primitives were used to map the surfaces being described using finite element meshes. Each time a ray was launched into the geometry, intersection (between ray and surface) was first tested with a sphere primitive describing the satellite as a whole, rather than considering potential intersection with each of the many finite elements making up the satellite surface. If the ray did not intersect the sphere, then the computation was deemed to have finished. However, if the ray met the sphere as a whole, then its intersection point was mapped to the finite element mesh and all elements located in the general area of the intersection point was tested for possible intersection. This combined technique (a FEM plus primitives) was also employed by these authors in their calculation of view factors between two concentric spheres with specular reflection. They reported a reduction in computational time, as well as the effective elimination of errors in reflective component arising from the approximation of surface curvature.

This work clearly demonstrated a role for geometric primitives in the calculation of radiative view factors within a MCRT framework. However, modelling arbitrary 3D geometries requires the development of a comprehensive suite of primitives (i.e. basis shapes such a sphere, rectangle and cone) that can be linked together to form the required heat transfer environment, together with the means for launching rays from one surface and intersecting with another. This realisation was crucial in defining the scope of the research forming this current thesis.

However, when considering the use of primitives for arbitrary 3D geometries, it is impractical to require the user to customise these various algorithms for each new configuration encountered. Indeed, this eliminates one of the prime strengths of the MC-RT approach, its inherent flexibility. Alternatively, primitives may be implemented with an affine transformation framework. Such an approach allows essentially any 3D geometry to be described by a small number of 'standardised' primitives (e.g. a unit radius sphere centred at the origin).

## 2.2 Affine Transformation

The surface equations of standardised geometric objects provide the basis upon which complex geometries may be formed. However, this is only possible through the use of affine transformations that allow a generic, standardised object to be converted into a similar ‘real world’ object possessing the required size, orientation and position. These various transformations preserve distance ratios and collinearity, and can be readily applied through the use of an appropriate transformation matrix  $M$  (and reversed using the inverse matrix  $M^{-1}$ ). For example, to transform the three-dimensional point  $P$  into  $\hat{P}$  one would multiply the original point with a transformation matrix as follows.

$$\hat{P} = M \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix} \quad (29)$$

In order to transform  $P$  back to  $\hat{P}$ , one would simply perform a similar calculation using the inverse transformation matrix  $M^{-1}$  as shown in equation (30).

$$P = M^{-1} \begin{pmatrix} \hat{P}_x \\ \hat{P}_y \\ \hat{P}_z \\ 1 \end{pmatrix} \quad (30)$$

The ‘1’ appended to the end of the point co-ordinates is used to apply the translation transformation which is stored in the fourth column of  $M$ . If a directional vector was to be transformed rather than a point, a ‘0’ would be used in place of the ‘1’ as directional vectors are always defined relative to the origin and therefore are exempt from translation transformations.

Through the use of such affine transformations, it is possible to formulate all ray algorithms required for a primitive-based MC-RT framework in terms of standardised objects that exist in what may be described as ‘object space’. For example, a ray’s starting point may be determined on the surface of a unit sphere (i.e. in object space), and transformed to a starting point on an actual sphere (i.e.

in the actual ‘world space’) by pre-multiplying the point by the appropriate transformation matrices.

### 2.2.1 Composing Affine Transformations

When multiple transformations are performed on a given primitive, a compound transformation matrix may be computed. Therefore, regardless the number of transformations required to move between object space and the desired world space configuration, only a single matrix needs to be stored and a single matrix multiplication performed per object in the run-time loop.

The compound transformation matrix may be calculated by pre-multiplying a transformation matrix by each subsequent transformation. For example if transformation  $M_1$  was performed followed by transformation  $M_2$  the compound transformation matrix would be calculated as follows:

$$M = M_2 M_1 \quad (31)$$

A single transformation matrix using this method may represent any number of sequential transformations. Similarly the compound inverse transformation matrix may be determined by post-multiplication by each subsequent transformation as shown in equation (32).

$$M^{-1} = M_1^{-1} M_2^{-1} \quad (32)$$

This simplification significantly reduces the computational load as billions of rays may be required to accurately model a complex 3D geometry each requiring transformation prior to ray-primitive intersection calculations.

The elementary affine transformations implemented in RayFactor will now be described.

### 2.2.2 Scaling

Affine scaling, scales a primitive object about the origin by scaling factors  $S_x$ ,  $S_y$  and  $S_z$  along the  $x$ ,  $y$  and  $z$  coordinate axes, respectively. The transformation and inverse transformation matrices have the form expressed in equation (33).

$$M = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M^{-1} = \begin{pmatrix} 1/S_x & 0 & 0 & 0 \\ 0 & 1/S_y & 0 & 0 \\ 0 & 0 & 1/S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (33)$$

Reflection of a primitive about a given axis can be achieved with a scaling transformation by specifying a negative scaling factor for that axis. If a scaling factor is defined as -1, a 'pure' reflection will result from the application of the transformation.

When applying a scaling transformation care must be taken to ensure that the scaling factors are not too large, especially when the implementation uses single precision floating point number as degenerate scaling can occur. Degenerate scaling will result in a primitive being treated as infinitely small when the inverse transformation is applied, as the reciprocal of the scaling factor in the inverse transformation matrix will evaluate to 0 within the computer.

### 2.2.3 Translation

Translation allows a primitive to be moved (or translated) by amounts  $T_x$ ,  $T_y$  and  $T_z$  along the x, y and z-axes, respectively, from its initial position at the origin. The transformation and inverse transformation matrices for translation are shown in equation (34).

$$M = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M^{-1} = \begin{pmatrix} 1 & 0 & 0 & -T_x \\ 0 & 1 & 0 & -T_y \\ 0 & 0 & 1 & -T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (34)$$

As previously mentioned, the translation transformation is only a valid transformation for points and should not be applied to direction vectors.

### 2.2.4 Rotation

Rotation allows a primitive object to be rotated about an arbitrary axis through a given angle  $\theta$ . In its most elementary form, the primitive will be rotated around the x (x-roll), y (y-roll) and z (z-roll) coordinate axis. These elementary rotations are listed in equation (35) where  $c = \cos \theta$  and  $s = \sin \theta$ .

$$\begin{aligned}
M_{x-roll} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
M_{y-roll} &= \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
M_{z-roll} &= \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\end{aligned} \tag{35}$$

To calculate the inverse rotation transformation, one simply calculates the transformation matrix using the negative of the rotation angle.

Often it is required that multiple elementary rotations are applied to achieve the desired geometric positioning, leading to the computation of multiple transformation matrices. However, Euler's theorem states that any rotation (or sequence of rotations) about a point is equivalent to a single rotation about some axis through that point [64]. Therefore, instead of using a sequence of x, y or z-rolls, one can more efficiently choose to perform a single rotation about an appropriate axis.

Maillot [65] formulated the transformation matrix for rotation about an arbitrary axis  $u$  which is shown in equation (36).

$$M = \begin{pmatrix} c + (1-c)u_x^2 & (1-c)u_xu_y - su_z & (1-c)u_xu_z + su_y & 0 \\ (1-c)u_xu_y + su_z & c + (1-c)u_y^2 & (1-c)u_yu_z + su_x & 0 \\ (1-c)u_xu_z + su_y & (1-c)u_yu_z + su_x & c + (1-c)u_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{36}$$

Here, the inverse transformation is again calculated using the transformation matrix with the negated rotation angle.

The Maillot rotation transformation, rather than the elementary rotation transformation matrices, is implemented in RayFactor to provide maximum flexibility and efficiency when dealing with such transformations.

## 2.3 Ray Definition

This concept is central to any ray-tracing implementation, as although transformations are specified in terms of the primitives it is the ray on which transformations are actually applied. A ray is completely specified in terms of its starting point  $S = (S_x, S_y, S_z, 1)$ , and its direction,  $c = (c_x, c_y, c_z, 0)$ , giving the ray equation as follows:

$$r = S + ct \quad (37)$$

Here, the variable  $t$  is representative of the ‘time’ that the ray has taken to travel from its starting point to a given point along its length, and plays an important role in verifying valid ray intersections (see Section 2.4).

Within the context of a Monte Carlo simulation, the selection of appropriate ray starting points and directions is of critical importance. The formulation of the ray starting point is dependent on the geometric object the ray is being launched from and, therefore, its selection will be covered in detail for each primitive in Section 2.4. The direction, however, is dependent on the emission properties (diffuse, directional, etc) and not the geometric configuration and, therefore, may be discussed separately from each specific primitive.

### 2.3.1 Ray Direction

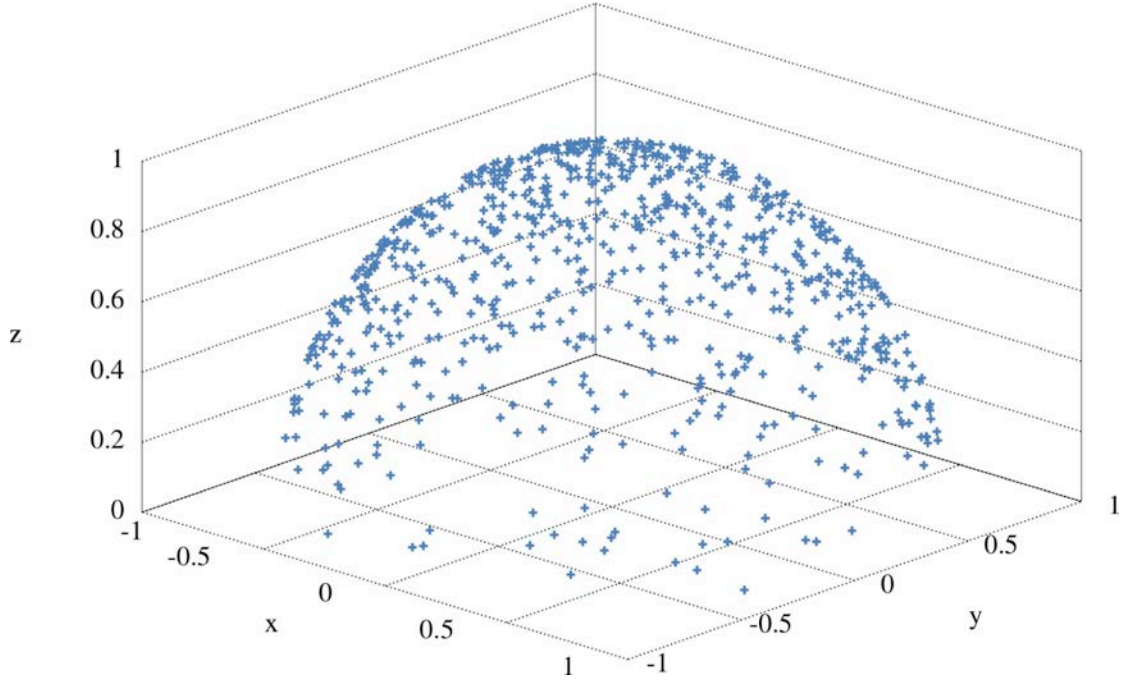
The ray launch direction is dependent on the properties of the surface being considered, and therefore the calculation of the ray direction can vary. With alternative numerical methods, such as integration by finite difference, more complex emission characteristics can result in an increase in solution complexity proportional to the square of this complexity [16]. However, the complexity of an MC-RT solution increases in an essentially linear fashion with the complexity of the emission characteristics. Table 1 demonstrates the functions required to calculate the ray direction for various types of emitters within a MC-RT solution for two independent, uniform random numbers  $\xi_1$  and  $\xi_2$  defined in the range  $0 \leq \xi < 1$ .

**Table 1: Ray directional functions required for various surface emitters**

| Ray directional functions assuming independence of azimuthal angle |  |
|--|--|
| Diffuse emitter  | $\xi_1 = \sin^2(\phi)$ $\xi_2 = \frac{\theta}{2\pi}$   |
| Directional grey emitter   | $\xi_1 = \frac{2}{\varepsilon_T} \int_0^\phi \varepsilon(\phi) \sin(\phi) \cos(\phi) \cdot d\phi$ $\xi_2 = \frac{\pi}{\sigma T^4} \int_0^\lambda i_\lambda \cdot d\lambda$   |
| Directional non-grey emitter                                       | $\xi_1 = \frac{2}{\varepsilon_T \sigma T^4} \int_0^\phi \int_0^\infty \varepsilon(\lambda, \phi) i_\lambda \sin(\phi) \cos(\phi) \cdot d\lambda \cdot d\phi$ $\xi_2 = \frac{\pi}{\varepsilon_T \sigma T^4} \int_0^\lambda \varepsilon(\lambda) i_\lambda \cdot d\lambda$ |

It should be noted that in order to reduce the MC-RT run-time when analysing for cases with complex emitters, numerical integration is required in the pre-processing stage to obtain probability density functions (PDFs) which allow the ray wavelength and emission zenith angles to be calculated directly as a function of a uniform random number rather than performing integration each time a ray is launched.

Although ray directional vectors need to be selected such that a ray is fired away from the surface at its starting point, these vectors are first formulated in object space on the surface of a unit hemisphere whose base lies in the x-y plane for all primitives as shown in Figure 4.



**Figure 4: Selection of 1000 ray starting points in object space**

When considering a grey diffuse emitter, the ray directional vector may be calculated simply using the formula shown in Table 1. If it is desired that the vector be described in Cartesian coordinates (as required by RayFactor), one could simply convert the azimuthal and zenith angles from polar coordinates using standard conversion formulae (which from a computational perspective would require three calls to trigonometric functions). However, a more efficient method to calculate the directional vector in Cartesian coordinates is to first calculate the azimuthal angle ( $\theta$ ) and a pseudo-zenith angle ( $\eta$ ):

$$\begin{aligned}\theta &= 2\pi \xi_1 \\ \eta &= \sqrt{\xi_2}\end{aligned}\tag{38}$$

The Cartesian coordinates may then be calculated using a single call to *sincos*, a function that calculates the sine and cosine of an angle in a single step, costing around the same time as a single call to the cosine function.

$$\begin{aligned}c_x &= \eta \cos \theta \\ c_y &= \eta \sin \theta \\ c_z &= \sqrt{1 - \xi_2}\end{aligned}\tag{39}$$



Once the ray directional vector has been calculated in object space, it must be transformed into world space by orientating it with the surface normal at the ray's starting point in world space. To obtain the surface normal in world space, the normal is first calculated in object space as described for each primitive in Section 2.4. The normal is then transformed from object space to world space using the transpose of the inverse transformation matrix as shown in equation (40).

$$N = M_T^{-1} N_{object} \quad (40)$$

Although the ray direction could be orientated with the surface normal using the affine rotation matrix previously described, this would require the calculation of the surface normal in polar coordinates and a number of trigonometric function calls to construct the rotation matrix. Alternatively, the Rodrigues rotation formula [66] can be applied which allows the construction of a rotation matrix using only the  $x, y$  and  $z$  coordinates of the surface normal  $(n_x, n_y, n_z)$  at the ray starting point.

$$M_{rotation} = \begin{pmatrix} n_z & 0 & n_x \\ 0 & n_z & n_y \\ -n_x & -n_y & n_z \end{pmatrix} + \frac{1}{1 + N_z} \begin{pmatrix} n_y^2 & -n_x n_y & 0 \\ -n_x n_y & n_x^2 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad (41)$$

Using the Rodrigues rotation matrix, one may transform the directional distribution from object space to world space by pre-multiplying by equation (41). Following the selection of the ray's starting point and its direction, the ray is then ready to be tested for intersection against objects in the surrounding environment.

### 2.3.2 Ray Transformation

The heart of the MC-RT method is the transformation (and inverse transformation) of the ray. As previously discussed, although transformations are specified in terms of primitive objects, it is actually the ray to which the transformations are applied.

The general procedure when launching a ray from object  $i$  is to first generate the ray in object space. This ray is then transformed from object space to world space using the transformation matrix of object  $i$  as shown in equation (42).

$$r_{world} = M_i \begin{pmatrix} S_x \\ S_y \\ S_z \\ 1 \end{pmatrix} + M_i \begin{pmatrix} c_x \\ c_y \\ c_z \\ 0 \end{pmatrix} t \quad (42)$$

Once the ray has been transformed, into world space, intersection tests between the ray and other objects can be conducted. In order to test a ray for intersection with an object,  $j$ , the ray must now be transformed from world space into the object space of  $j$  using the inverse transformation matrix as shown in equation (43).

$$r_{object-j} = M_j^{-1} \begin{pmatrix} S_x \\ S_y \\ S_z \\ 1 \end{pmatrix} + M_j^{-1} \begin{pmatrix} c_x \\ c_y \\ c_z \\ 0 \end{pmatrix} t \quad (43)$$

When the ray is inversed transformed it becomes specified relative to object  $j$ , meaning that testing for intersection between the inverse transformed ray and the primitive object is equivalent to testing intersection between a ray and object specified in the space coordinate space. This allows the use of generic intersection calculations, which are dependent on the type of primitive and not the actual geometric configuration of the object in the system being modelled. For example, a sphere with a radius of 2 centred at the point  $(-1, 3, 2)$  will employ the same intersection routine as a sphere with radius 5 centred at the point  $(10, 10, 10)$ .

To test additional objects for ray intersection, the steps subsequent to obtaining the ray in world space are repeated with the ray being transformed from world space into the object space of each primitive and intersection tested using the generic intersection methods for each primitive type. The calculations required to both generate the rays and test intersection in object space for each primitive employed in RayFactor are detailed in Section 2.4.

### 2.3.3 Post-Transformation Primitive Surface Area

When conducting a Monte Carlo simulation, it is often desirable to specify the number of samples used in terms of a sampling density rather than an absolute quantity. This allows the simulation sample rate to be decoupled from the actual geometric configuration of the system being analysed.

RayFactor adopts this approach by allowing a ray density (i.e. the number of rays per unit area) for each object (or the entire system) to be specified. However, in order to know the total number of rays to launch at run-time, this ray density must be converted to an absolute quantity for each object. Given every object starts as a primitive of known surface area and is transformed to the desired geometric configuration using an affine transformation, we may calculate the surface area of the transformed object using the determinate of the transformation matrix as shown below.

$$A_{transform} = A_{primitive}|M| \quad (44)$$

Through equation (44), the absolute number of rays to be launched from each object may be calculated by multiplying the area post-transformation,  $A_{transform}$ , by the ray density.

## 2.4 RayFactor Primitives

A variety of primitives were selected for implementation within the RayFactor program. These primitives are the rectangle, disc, annulus, sphere, cylinder and frustum and were selected as they can be manipulated to represent almost all objects commonly found in real-world engineering geometries. Although this initial set of primitives may seem somewhat limited, it could be extended with relative ease due to the computational structure of RayFactor and the nature of the MC-RT framework employed.

The following sections provide descriptions of each primitive used and present methods for generating random points on the surface of each object (a ray starting point) and detecting ray-object intersection. It is with the basis provided in the following sections that a robust primitive-based MC-RT algorithm for computing 3D view factors may be built.

### 2.4.1 Rectangle

This is the simplest primitive in RayFactor and represents a rectangle lying in the x-y plane ( $z = 0$ ) with bounds on the x and y coordinates such that  $-1 < x < 1$  and  $-1 < y < 1$ . The surface equation of the rectangle primitive is shown in equation (45).

$$F(x, y, z) = z = 0 \quad (45)$$

The surface equation of this primitive is simply the equation of the x-y plane, however by bounding the x and y coordinates one may consider the treatment of a discrete surface rather than an infinite plane (which would be impractical to launch rays from).

#### **Ray Starting Point**

Given the simple nature of the rectangle primitive, the x and y starting positions may be calculated from two random numbers  $\xi_1$  and  $\xi_2$  using just two multiplications and subtractions as presented in equation (46).

$$\begin{aligned} S_x &= 2\xi_1 - 1 \\ S_y &= 2\xi_2 - 1 \\ S_z &= 0 \end{aligned} \quad (46)$$

The vector normal to the surface of a rectangle primitive at any point along the surface has the coordinates (0,0,1).

#### **Ray Intersection**

Intersection between a ray fired from another object and the x-y plane occurs when  $S_z + c_z t = 0$  and the 'time' at which intersection occurs  $t_h$ , may be simply calculated as:

$$t_h = -\frac{S_z}{c_z} \quad (47)$$

Prior to calculating the intersection time, it must be ascertained that the ray is not moving parallel to the plane. This is indicated when  $c_z = 0$  and if left unchecked will result in a divide by zero error within RayFactor at run-time.

If the intersection time is greater than 0 (within machine error) the intersection point is calculated from the ray equation as shown in equation (48).

$$P_h = S + c \cdot t_h \quad (48)$$

If the x and y coordinates of the ‘hit point’ are within the bound of the rectangle primitive ( $-1 < x < 1$  and  $-1 < y < 1$ ) then the intersection is accepted as valid.

#### 2.4.2 Disc

The disc primitive represents a circle lying in the x-y plane, the surface equation of this primitive being shown in equation (49).

$$F(x, y, z) = x^2 + y^2 - r^2 \quad (49)$$

Although the implicit equation of a disc allows for the radius to be set as a parameter, the RayFactor implementation uses a fixed radius of 1. This permits various computational optimisation options to be employed but has no impact on the overall flexibility of the disc primitive.

#### **Ray Starting Point**

One might expect that the ray starting point for the disc primitive may be calculated by choosing a random azimuthal angle,  $\theta \in [0, 2\pi]$ , and radius,  $r \in [0, 1]$ . However, this is incorrect and would lead to a non-uniform distribution of points with crowding around the disc centre.

As the aim to uniformly distribute the ray launching points over the surface area of the disc, and the area of the disc increases proportionally with the square of the radius, one must take the square root of the randomly generated radius and calculate the ray starting point as shown in equation (50).

$$\begin{aligned}
S_x &= \sqrt{r} \cos \theta \\
S_y &= \sqrt{r} \sin \theta \\
S_z &= 0
\end{aligned} \tag{50}$$

Here  $\theta$  is calculated as shown in equation (38). This formulation allows a ray starting point to be determined (within RayFactor) with a square root, two multiplications and a single call to the *sincos* function.

Like the rectangle primitive, the vector normal to the disc surface at any ray launch position is (0, 0, 1).

### **Ray Intersection**

Similarly to the rectangle primitive, the intersection time  $t_h$  is calculated using equation (47). If the hit time is greater than 0, the distance of the intersection point from the origin (0, 0, 0) is calculated using equation (51).

$$D = (S_x + c_x \cdot t_h)^2 + (S_y + c_y \cdot t_h)^2 \tag{51}$$

If  $D$  satisfies the condition  $D < 1$ , indicating that the point of intersection falls within the unit circle on the x-y plane, then the intersection is considered to be valid.

### **2.4.3 Annulus**

The generic annular primitive is similar to the generic disc with the additional functionality of having specified inner and outer radii and thus the ray starting point must be selected in such a way that it lies on the annulus surface.

### **Ray Starting Point**

The ray starting point on the surface of the annular primitive requires the selection of the azimuthal angle  $\theta$  (equation (38)), followed by the selection of a random radius which lies between the two annulus radii,  $r_o$  and  $r_i$ . This random radius  $r_\xi$  may be calculated using equation (52).

$$r_\xi = \sqrt{(r_o^2 - r_i^2)\xi_1 + r_i^2} \quad (52)$$

Subsequently, the Cartesian coordinates for the ray starting point may be calculated using  $r_\xi$  as the radius in equation (50).

### **Ray Intersection**

Ray intersection with the annular primitive is detected in much the same way as for the disc primitive. That is to calculate the intersection time as shown in equation (47), then to calculate the distance of the intersection point from the origin in the x-y plane ( $D$ ) as shown in equation (51). However, here we consider the intersection valid only if  $r_i < D < r_o$ .

#### **2.4.4 Sphere**

This primitive represents a unit sphere, centred on the origin. The implicit form of the sphere primitive is displayed in equation (53).

$$F(x, y, z) = x^2 + y^2 + z^2 - r^2 \quad (53)$$

Although the implicit equation permits the use of a radius parameter, like with the disc primitive, the RayFactor implementation uses a fixed radius of 1 for similar reasons.

### **Ray Starting Point**

There are numerous methods by which random points on the surface of a sphere may be selected. Several of these are outlined by Marsaglia [67] including the ‘rejection method’, one of the fastest approaches for generating a random point of the surface of a sphere from a uniformly distributed random number.

The rejection method is essentially an exercise in picking random points inside a cube and throwing away points which do not also lie within a sphere bounded by the cube. From the ratio of a sphere’s volume to that of a bounding cube, we may calculate that the probability that a random point will lie within the volume of the sphere as  $\pi/6$ . Therefore on average  $6/\pi$  ( $\sim 1.9$ ) attempts are made before each random point is generated.

At each attempt, two independent uniformly distributed numbers,  $\xi_1, \xi_2 \in [-1,1]$ , are generated and the acceptance criteria shown in equation (54) is calculated.

$$\xi_1^2 + \xi_2^2 < 1 \quad (54)$$

If (54) is not satisfied the point would be rejected and two new random numbers selected. Once random numbers are selected such that equation (54) is satisfied, the random point is projected onto the sphere of the sphere and the Cartesian coordinates of the point can be calculated using equation (55).

$$\begin{aligned} S_x &= 2 \cdot \xi_1 \cdot \sqrt{1 - \xi_1^2 - \xi_2^2} \\ S_y &= 2 \cdot \xi_2 \cdot \sqrt{1 - \xi_1^2 - \xi_2^2} \\ S_z &= 1 - 2(\xi_1^2 + \xi_2^2) \end{aligned} \quad (55)$$

Although this method takes on average 1.9 attempts for each point generated, it does not require the use of transcendental functions (cos, sin etc.) and therefore still executes several times faster than alternative methods.

Although the rejection method was utilised as the method for generating ray starting points on a sphere's surface in RayFactor's initial serial implementation, it is not an ideal option moving towards highly vectorised implementations (see Section 4.4.2 for more details). For example, on modern Intel processors capable of processing packed data arrays of up to 8 elements in a single processor cycle, an average of 22.1 attempts per starting point generated would be required.

Thus, subsequent versions of RayFactor opted for a determinate method for the generation of ray starting points a sphere. In this method, an azimuthal angle is calculated as shown in equation (38) while using a second independent random number the Cartesian coordinates for the ray starting point can be calculated as shown in equation (56).



$$\begin{aligned}
S_x &= \sqrt{1 - z^2} \sin \theta \\
S_y &= \sqrt{1 - z^2} \cos \theta \\
S_z &= 2\xi_2 - 1
\end{aligned} \tag{56}$$

Although this method does require computationally expensive sine and cosine functions, its form leaves it receptive to extensive optimisation through the implementation as fast sine and cosine approximations, which will be explained in detail in Section 4.6.3.

### **Ray Intersection**

Substituting the ray equation (37) into the surface equation of the sphere (53), we get the equation needed for ray-sphere intersection below.

$$|c|^2 t_h^2 + 2(S \cdot c)t_h + (|S|^2 - 1) = 0 \tag{57}$$

This is a quadratic equation of the form  $At^2 + 2Bt + C = 0$  and the time of intersection may be calculated as shown in equation (58).

$$t_h = -\frac{B}{A} \pm \frac{\sqrt{B^2 - AC}}{A} \tag{58}$$

However, before any attempt to calculate the intersection time, one first calculates the discriminate  $B^2 - AC$ . If the discriminate is negative, the ray does not intersect the sphere and the intersection calculations are exited early. If the discriminate is equal to 0, the ray grazes the surface of the sphere with just one point of intersection. If the discriminate is positive, there are two intersection points, one as the ray enters the sphere and one as it exits. When this occurs, we calculate both intersection times and accept the valid intersection as being the one with the earliest non-negative time.

### **2.4.5 Cylinder**

This primitive represents a cylinder with a radius and height of 1, with its base lying in the x-y plane. The implicit form of the cylinder primitive is shown in

equation (59) with the specified height of 1 being obtained through a bound on the  $z$  coordinate such that  $0 < z \leq 1$ .

$$F(x, y, z) = x^2 + y^2 - r^2 \quad (59)$$

Like other primitives discussed previously, the RayFactor implementation uses a fixed radius of 1 and relies on affine transformations to obtain any alternative radius.

### **Ray Starting Point**

To generate a random ray starting point on the cylinder primitive, a random height along the cylinder ( $S_z$ ) is first selected using a uniform random number  $\xi \in [0,1]$ . An azimuthal angle is then selected as described in equation (38) and the  $x$  and  $y$  coordinates calculated from this angle using a standard polar to Cartesian coordinate system conversion. Equation (60) lists the complete set of equations for the generation of a ray starting point on the surface of the cylinder primitive.

$$\begin{aligned} S_x &= \sin \theta \\ S_y &= \cos \theta \\ S_z &= \xi \end{aligned} \quad (60)$$

The vector normal to the surface of the cylinder at the ray starting point is dependent on the starting point location and specified using the coordinates at this point as  $(S_x, S_y, 0)$ .

### **Ray Intersection**

Similarly to the sphere primitive, to check a ray-cylinder intersection we may substitute the ray equation (37) into the surface equation (60) of the cylinder and obtain a quadratic equation describing the intersection as shown in equation (61).

$$(c_x^2 + c_y^2)t_h^2 + 2(S_x c_x + S_y c_y)t_h + (S_x^2 + S_y^2 - 1) = 0 \quad (61)$$

This can then be solved for  $t_h$  using the quadratic equation (58). Like the sphere primitive, we may exit intersection calculations early if the discriminate is negative. Once we have identified that there is at least one valid intersection, we must calculate the  $z$  coordinate of the intersection point using the ray equation and perform the additional check that  $0 \leq z \leq 1$ .

#### 2.4.6 The Frustum

The frustum primitive represents a portion of a cone lying between two parallel planes. Like the cylinder primitive, the frustum has its base in the  $x$ - $y$  plane with its base radius and height each unity, and a top radius  $r_t$  that may be varied between 0 (giving a generic cone) and 1 (giving a generic cylinder). The implicit form of the frustum is shown in equation (62).

$$F(x, y, z) = x^2 + y^2 - (1 + (r_t - 1)z)^2 \quad (62)$$

Although the base radius is fixed at unity in the RayFactor implementation, the top radius  $r_t$  may be specified to support a wide range of configurations.

##### **Ray Starting Point**

The selection of a ray starting point on a frustum is the most complex of the supported primitives within RayFactor, as measures need to be taken to ensure that all points are distributed uniformly over a surface whose area varies along the  $z$ -axis and also with a changing top radius  $r_t$ . The approach used here was to first select a random area fraction,  $\xi_1$ , and convert this to a  $z$  coordinate under which the fraction of the frustum's total surface area is equal to  $\xi_1$ . This  $z$  coordinate may be calculated as follows:

$$S_z = \frac{-1 + \sqrt{1 - \xi_1 + \xi_1 r_t^2}}{r_t - 1} \quad (63)$$

Following the selection of a  $z$  coordinate, the radius of the frustum at this height may be calculated as shown in equation (64).

$$r_\xi = 1 + (r_t - 1)S_z \quad (64)$$

Using a random azimuthal angle calculated by equation (38), and the radius of the frustum at  $S_z$ , the x and y coordinates of the ray starting point may be calculated as shown in equation (65).

$$\begin{aligned} S_x &= r_\epsilon \cos \sqrt{\theta} \\ S_y &= r_\epsilon \sin \sqrt{\theta} \end{aligned} \tag{65}$$

The vector normal to the surface of the frustum at the ray starting point is somewhat more involved to calculate compared to the previous primitives due to the variable slope of the side of the frustum. Equation (66) displays the formulation of the frustum surface normal.

$$N = (S_x, S_y, 1 - S_z - r_t - S_z r_t^2) \tag{66}$$

Due to the flexibility of the frustum primitive, it was used to model a range of geometries from the cone ( $r_t = 0$ ) through to the cylinder ( $r_t = 1$ ) in the initial version of RayFactor. However, as one may suspect from the material presented for the frustum and cylinder primitives, calculations may be greatly simplified by using a dedicated primitive for the cylinder.

### **Ray Intersection**

In order to test for ray-frustum intersection, we substitute equation (37) into the frustum surface equation (62) to obtain a quadratic equation of the form  $At^2 + 2Bt + C = 0$ . The quadratic coefficients for this intersection are shown in equation (67).

$$\begin{aligned} A &= c_x^2 + c_y^2 - ((r_t - 1)c_z)^2 \\ B &= S_x c_x + S_y c_y - c_z S_z (r_t^2 - 2r_t + 1) + c_z (r_t + 1) \\ C &= S_x^2 + S_y^2 - ((r_t - 1)S_z + 1)^2 \end{aligned} \tag{67}$$

This can then be solved for  $t_h$  using the quadratic equation (58) with a familiar ability to exit intersection calculations early if the discriminate is negative. Once we have identified that there is at least one valid intersection, we must calculate

the  $z$  coordinate of the intersection point using the ray equation and perform the additional check that  $0 \leq z \leq 1$  to ensure that the intersection takes place within the defined bounds of the frustum primitive.

## **2.5 The use of Primitives in Radiative View Factor Calculation**

The concept of the primitive and the mathematics by which they can be integrated into a MC-RT framework have been presented in this chapter. However, in order to become a viable tool for radiative heat transfer analysis, a software implementation must be developed and carefully optimised to ensure accurate results can be produced in a reasonable amount of time.

The following chapter will detail the implementation of the concepts presented in this chapter into a robust primitive-based MC-RT application called RayFactor.

## Numerical Determination of Primitive Based View Factors

---

In order to calculate radiative view factors using a primitives-based MC-RT computational environment the concepts introduced in Chapter 2 must be implemented in robust computer code and the results benchmarked against known solutions to confirm the implementation's accuracy and gauge its speed. This chapter discusses these aspects for a software implementation called RayFactor.

### 3.1 High-Level Design Constraints

The development of RayFactor was initially driven by a research need to numerically calculate view factors within an operational fibre-drawing furnace (discussed in detail later in Chapter 7). However, the decision was made early on to develop robust, portable code for calculating view factors in arbitrary 3D geometries, rather than simply building the fastest possible codebase for modelling the furnace geometry. This wider objective imposes numerous programming design constraints which ultimately affect the overall run-time performance to some (greater or lesser) extent.

Furthermore, it was decided that RayFactor be targeted for commodity computer hardware rather than any specific hardware. The consequences of this constraint may not seem significant, however it dictates the technologies that may be utilised and limits the assumptions that can be made during code development and optimisation.

### 3.2 Development Environment

#### 3.2.1 Computer system

An Apple Inc. Mac Pro 4.1 in a 2 x 2.12 GHz Quad Core Intel Xeon E5520 Nehalem processor configuration was selected as the development environment for

RayFactor. The technical specifications of interest for this configuration of the Mac Pro 4.1 are outlined in Table 2 below.

**Table 2: Selected technical specifications for a Mac Pro 4.1**

| Attribute       | Value                |
|-----------------|----------------------|
| Processors      | 2 (8 Cores)          |
| Processor speed | 2.26 GHz             |
| Processor type  | Quad Core Xeon E5520 |
| Architecture    | Nehalem              |
| L1 Cache        | 32KB x 8             |
| L2 Cache        | 256 KB x 8           |
| L3 Cache        | 8 MB x 2             |
| RAM             | 8 GB                 |
| RAM Type        | 1066 MHz DDR3 EEC    |

The primary motive for the selection of an Apple Inc. system was the high performance and stability UNIX based operating system, 'Mac OS X'. Additionally, all required software development tools and environments were provided with the operating system including the integrated development environment (IDE) 'XCode', performance analyser and visualizer 'Instruments' and automation tool for testing called 'Automator'.

The Mac Pro 4.1 model was selected, as it was the highest performance Apple Inc system available at the time and the only Apple system where 3<sup>rd</sup> party GPUs could be easily installed (this was important for the work to be presented in Chapter 5). Secondary to these requirements, the Mac Pro had the ideal performance characteristics. At the time of purchase, it was one of the most powerful 'off the shelf' computing systems available and by the end of the research it was expected that it would be representative of computational power available to the general consumer. The assumption of the Mac Pro 4.1 representing 'general' computing power was verified when comparing the Primate Labs GeekBench 2 scores for the Mac Pro 4.1 used and the contemporary Mac Mini (the lowest powered Apple Inc. system) as shown in Table 3. The Primate Labs Geekbench software is the industry standard for measuring the average performance of a computer and works by running a series of stress tests quantify test integer, floating point, memory and bandwidth performance.

**Table 3: Primate Labs Geekbench 2 scores for a selection of Apple Inc. Computers**

| Computer                         | Processor  | Geekbench 2 Score |
|----------------------------------|--|-------------------|
| Mac Pro 4.1 (2009)<br>(selected) | 2 x 2.26 GHz Quad Core Intel<br>Xeon E5520         | 11,803            |
| Mac Pro 4.1 (2009)               | 2 x 2.93 GHz Quad Core Intel<br>X5570              | 14,904            |
| Mac Pro 5.1 (2011)               | 2 x 3.06 GHz Hex Core Intel<br>Xeon X5675          | 22,137            |
| Mac Mini 6.1 (2011)              | 2.6 GHz (3.4 GHz Boost)<br>Dual Core Intel Core i7 | 11,758            |

The top processor configurations for the Mac Pro 4.1 and 5.1 have been included in Table 3 to show the performance range for the Mac Pro between 2009 and 2011. Of particular interest is the fact that a 50% increase in performance between the two models is obtained by increasing the core count (by 50%) rather than the processor clock speed (increased by only 4.5%). This is the current trend in the computer hardware industry and will be exploited during the optimisation of RayFactor, as discussed in chapter 5.

### 3.3 Random Number Generation

The generation of random numbers using deterministic methods (i.e. computer based generation) is an area of great importance for any Monte Carlo simulation, and has led to the development of a large selection of pseudorandom number generators (PRNG) [46-49, 68, 69].

For all classes of Monte Carlo simulation (of which MC-RT is no exception) the key to obtaining an accurate result in the fastest possible time frame is the rapid generation of 'high quality' random numbers. Given the dual objectives of having random numbers that are generated quickly and are of high quality with long period (time it takes before the sequence is repeated) and uniform distribution, the selection of a suitable random number generator for use in RayFactor can be viewed as an optimisation problem.

The selection and performance characteristics of the PRNG used in RayFactor is discussed in detail in Chapter 4. However, it is important to note that the selection of the PRNG will not only influence the computational run-time of a MC-RT but also the accuracy of the results such those presented in this chapter.



### 3.4 RayFactor Benchmarking Methodology

To test the code which performs ray formulation, intersection and transformation algorithms, the view factors for a selection of geometries were calculated using RayFactor and compared against the analytical solutions available in literature<sup>3</sup> [2].

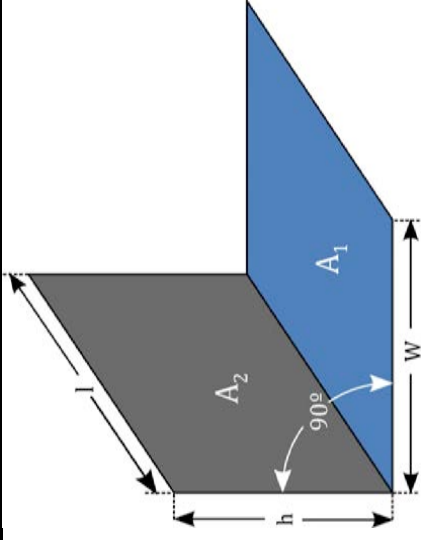
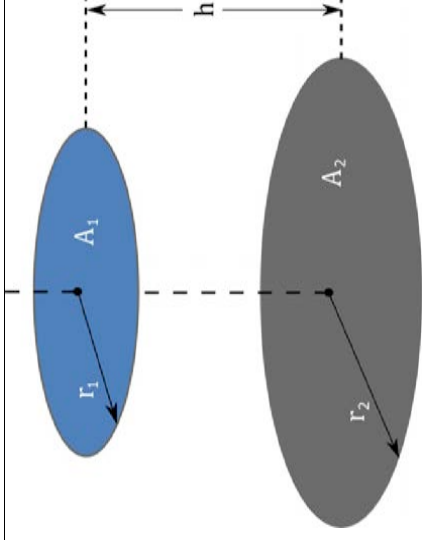
The 7 chosen geometries are shown in Table 4 and were selected to ensure that each primitive implemented (i.e. sphere, cylinder, frustum, rectangle, annulus, disc and triangle) was tested thoroughly, not only for a 'standard' configuration but also for configurations where volumetric primitives (such as the sphere and cylinder) were considered to be bounding (i.e. looking inward).

For each of the geometries selected a series of ray densities (typically from 10 to  $10^8$  rays per unit surface area) were analysed with up to 600 runs being carried out at each ray density to provide statistically significant data to examine convergence behaviour with increasing ray density and the distribution of view factor results at each ray density.

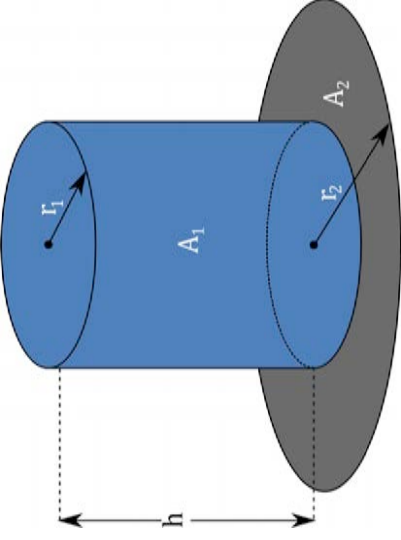
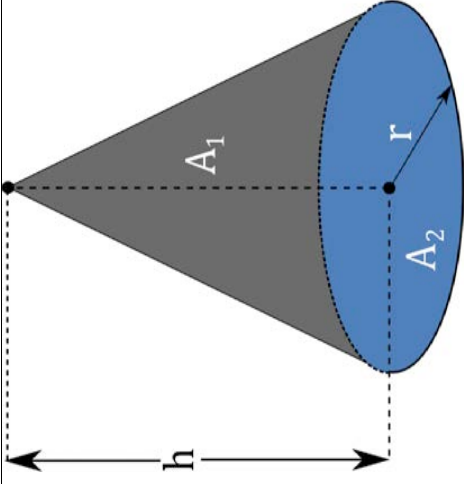
---

<sup>3</sup> RayFactor results for a specific case (A-19) were also verified through personnel communications with J.R. Howell.

Table 4: Selected benchmarking geometries

| Howell<br>Designation | Formula calculating view factor   | Diagram  |
|-----------------------|---|--|
| C-14<br>$F_{1-2}$     | $= \frac{1}{W\pi} \left( W \tan^{-1} \frac{1}{W} + H \tan^{-1} \frac{1}{H} - \sqrt{H^2 + W^2} \tan^{-1} \sqrt{\frac{1}{H^2 + W^2}} \right)$ $+ \frac{1}{4} \ln \left\{ \frac{(1 + W^2)(1 + H^2)}{1 + W^2 + H^2} \left[ \frac{W^2(1 + W^2 + H^2)}{(1 + W^2)(W^2 + H^2)} \right]^{W^2} \left[ \frac{H^2(1 + W^2 + H^2)}{(1 + H^2)(W^2 + H^2)} \right]^{H^2} \right\}$ <p>Where: <math>H = h/l, W = w/l</math></p> |   |
| C-41                  | $F_{1-2} = \frac{1}{2} \left[ X - \sqrt{X^2 - 4 \left( \frac{R_2}{R_1} \right)^2} \right]$ <p>Where: <math>R_x = r_x/h, X = 1 + (1 + R_2^2)/R_1^2</math></p>  |  |

| Howell<br>Designation | Formula calculating view factor   | Diagram |
|-----------------------|---|---------|
| C-47                  | $F_{1-2} = \frac{1}{2} \left( R_3^2 - R_2^2 - [(1 + R_3^2 + H^2)^2 - 4R_3^2]^{\frac{1}{2}} + [(1 + R_2^2 + H^2)^2 - 4R_2^2]^{\frac{1}{2}} \right)$ <p>Where: <math>H = a/r_1, R_2 = r_2/r_1, R_3 = r_3/r_1</math></p>   |         |
| C-52                  | $F_{1-2} = \frac{1}{2(R_2^2-1)} \left( [(R_2^2 + R_3^2 + H^2)^2 - (2R_2R_3)^2]^{\frac{1}{2}} - [(R_2^2 + R_4^2 + H^2)^2 - (2R_2R_4)^2]^{\frac{1}{2}} + [(1 + R_4^2 + H^2)^2 - (2R_4)^2]^{\frac{1}{2}} - [(1 + R_3^2 + H^2)^2 - (2R_3)^2]^{\frac{1}{2}} \right)$ <p>Where: <math>H = a/r_1, R_2 = r_2/r_1, R_3 = r_3/r_1, R_4 = r_4/r_1</math></p> |         |

| Howell<br>Designation | Formula calculating view factor   | Diagram  |
|-----------------------|---|--|
| C-77                  | $F_{1-2} = \frac{B}{8RH} + \frac{1}{2\pi} \left( \cos^{-1} \left( \frac{A}{B} \right) - \frac{1}{2H} \left[ \frac{(A+2)^2}{R^2} - 4 \right]^{\frac{1}{2}} \cos^{-1} \left( \frac{AR}{B} \right) - \frac{A}{2RH} \sin^{-1} R \right)$ <p>Where: <math>H = h/r_2</math>, <math>R = r_1/r_2</math>, <math>A = H^2 + R^2 - 1</math>, <math>B = H^2 - R^2 + 1</math></p> |   |
| C-109                 | $F_{1-2} = \frac{1}{(1 + H^2)^{1/2}}$ <p>Where: <math>H = h/r</math></p>  |  |

| Howell<br>Designation | Formula calculating view factor   | Diagram   |
|-----------------------|---|---|
| C-122                 | $F_{1-2} = \frac{1}{4\pi} \tan^{-1} \left( \frac{1}{D_1^2 + D_2^2 + D_1^2 \cdot D_2^2} \right)^{1/2}$ <p>Where: <math>D_1 = d/l_1, D_2 = d/l_2</math></p> |  |

### 3.5 RayFactor Validation for the C-77 case

Although RayFactor was benchmarked for all the cases shown in Table 4, only the results for a single case will be presented in this thesis for the sake of brevity. The benchmarking results from this particular case are representative (in terms of the speed and accuracy) of the results of all other benchmark cases.

The results to be presented here are from the C-77 case in which the view factors between the outer surface of a cylinder and an annular disc at the end of the cylinder are calculated. When benchmarking software of this nature, the following population characteristics are of interest.

1. Multiple samples at a fixed sample rate have a normal distribution around the known solution.
2. Simulation results converge on the known solution as the sample rate is increased.
3. Computational time increases linearly with the sample rate.

Each of these qualities will be examined for the C-77 case using sample rates (i.e. ray densities) ranging from 10 rays/unit<sup>2</sup> to 100,000,000 rays/unit<sup>2</sup> and 600 runs at each respective ray density.

It should be noted that benchmarking was conducted at numerous points during the development of RayFactor to assess the changes in quality and performance for each new feature or optimisation implemented. The results presented below were generated using the latest version of RayFactor at the time of writing; this particular version has the best ratio of performance to accuracy of all versions of RayFactor and is therefore the most appropriate to examine.

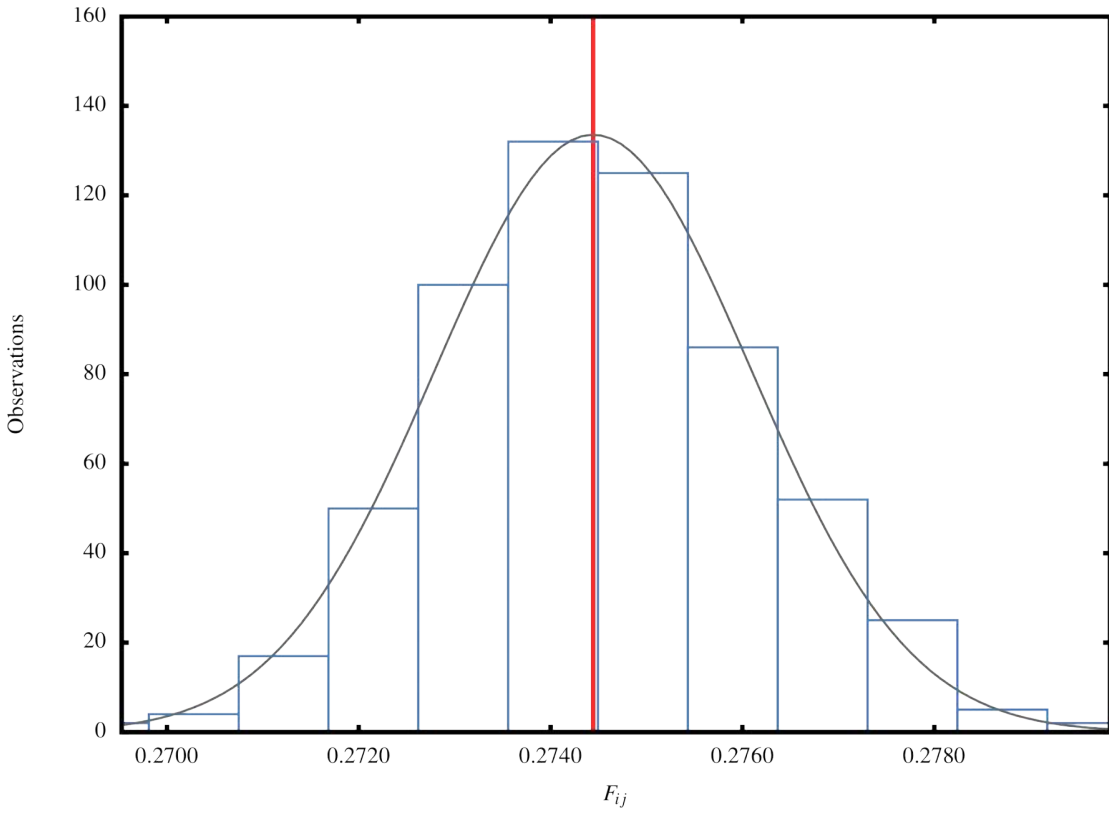
#### 3.5.1 Solution Distribution

A view factor determined by a MC-RT simulation is simply the average of a number of Bernoulli experiments (i.e. does a random ray launched from object  $i$  hit object  $j$ ?). As such, view factors produced by MC-RT should belong to a normal distribution with the population mean equal to the analytical solution.

Production of normally distributed results is of critical importance as deviation from such a distribution is a clear indicator of poor PRNG performance or overly aggressive optimisations (which reduce the overall simulation accuracy). For example, a flattened distribution indicates that the PRNG may not be producing uniform random numbers or the accuracy of in-built functions such as that used

to evaluate square root is too low, while a skewed distribution indicates that the selected program epsilon (a tuneable constant for combating machine error) may be too high (discussed in detail in Section 4.2).

Figure 5 shows the distribution of 600 RayFactor runs for the C-77 case with a ray density of 10,000 rays/unit<sup>2</sup> and a normal curve with a mean equal to the analytical solution (marked by the red line) and a standard deviation equal to that of the 600 samples. It is clear that the distribution of results produced by RayFactor closely matches a normal distribution, an outcome that provides confidence in the accuracy of the results generated.



**Figure 5: Histogram overlaid with the Gaussian PDF for 600 runs of the C-77 case with a ray density of  $10^4$**

During the implementation of vectorisation features in RayFactor, examination of the results distribution uncovered problems in the parallel seeding of the PRNG being used, dSFMT. Correspondence with the creators of dSFMT, Mutsuo Satio and Makoto Matsumoto, who was also a co-creator of the Mersenne twister algorithm one of the most widely used PRNGs today, resulted in a ‘patch’ being developed and implemented to the seeding of the dSFMT function in parallel computing applications.

### 3.5.2 Solution Convergence

As the ray density is increased, the view factors calculated by RayFactor are expected to converge on the analytical solution. Convergence for the C-77 case may be demonstrated by plotting the result of a single run at each density as shown in Figure 6.

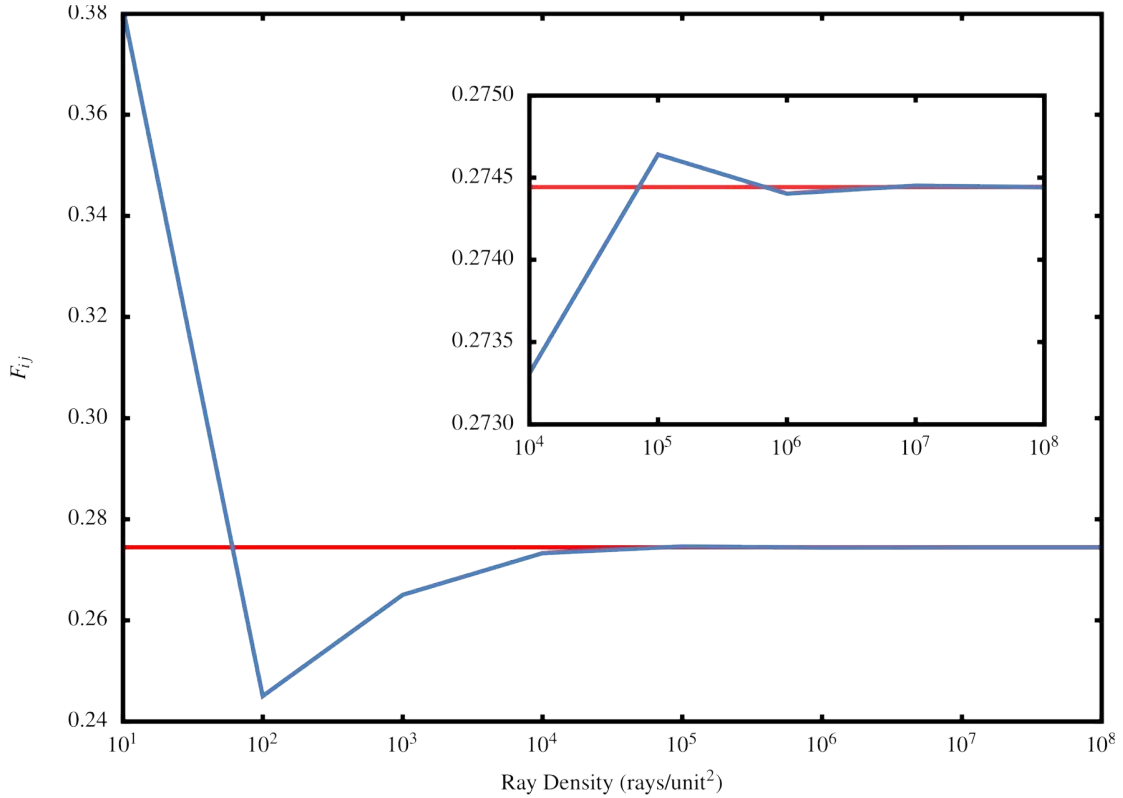
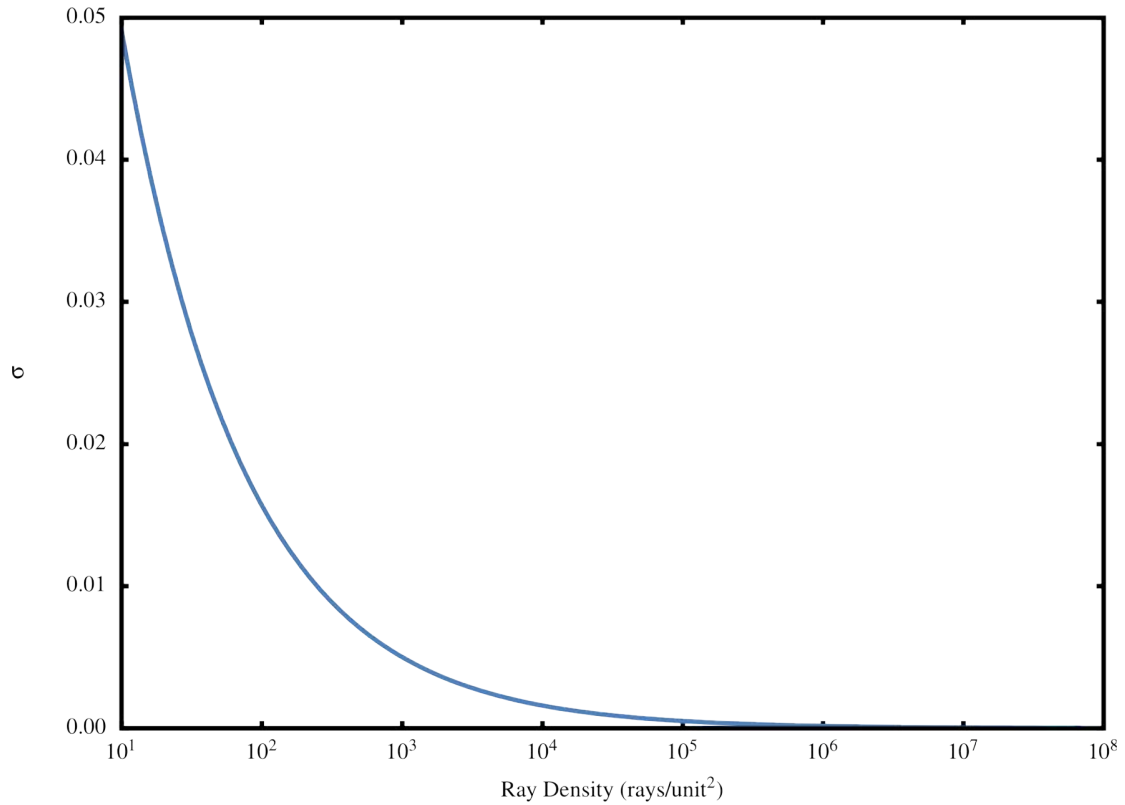


Figure 6: Convergence of RayFactor results on the analytical solution for the C-77 case

In addition to single run convergence, it is also desirable to ensure that the range over which results are produced for any given ray density becomes smaller as the ray density increases. From the central limit theorem, we can expect the convergence rate of a Monte Carlo solution to be proportional to the inverse square root of the number of samples [16]. This relationship is clear when plotting the sample standard deviations at each ray density as shown in Figure 7.



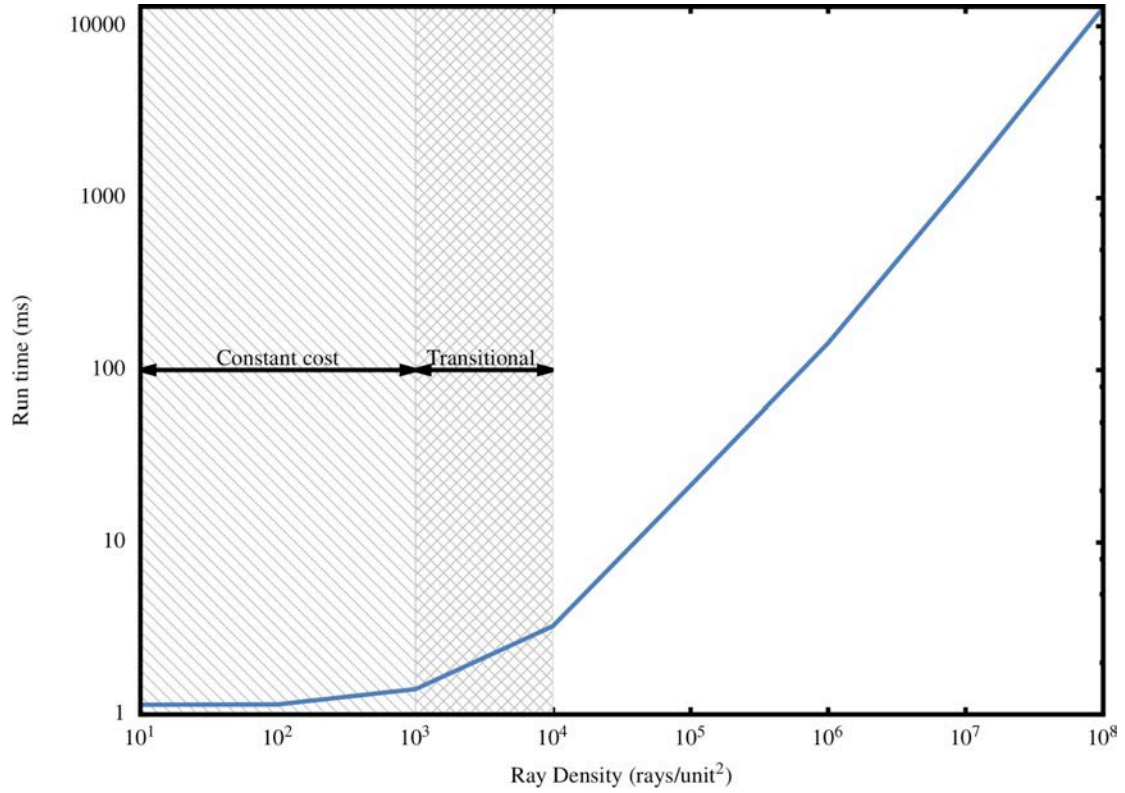


**Figure 7: Sample standard deviation with increasing ray density**

The curve in Figure 7 can be described by the equation  $\sigma = 1.537/\sqrt{\rho}$  which agrees with the anticipated rate of convergence with sample size, further validating the accuracy of RayFactor.

### 3.5.3 Solution Run-Time

The final property of a Monte Carlo method to be demonstrated by RayFactor is a linear increase in computational time with increasing ray density. As a Monte Carlo method is essentially repeating the same operation for a given number of samples (in the case of RayFactor, formulate, fire and calculate the intersection of a ray), the amount of ‘work’ that must be completed is directly proportional to the number of rays fired. Any significant deviation from a linear relationship could indicate errors in the computational implementation such as memory leaks. Figure 8 displays the increase in the average run time with increasing ray density for the C-77 case.



**Figure 8: RayFactor run-times with increasing ray density**

Examining the average run times as a function of ray density in Figure 8, we observe a fairly constant run-time at ray densities lower than  $10^3$ . Here the computation involved in launching and intersecting rays is negligible compared to the ‘fixed costs’ of seeding the PRNG, resulting in a fairly constant run-time. As the ray density increases above  $10^3$ , the computational time transitions to being dominated by the launching and intersecting of rays, and for ray densities greater than  $10^4$  the expected linear relationship between the ray density and run-time is indeed observed.

While of critical importance, the calculation of radiative view factors with high accuracy and desired distributional properties is only one of the requirements of a useable MC-RT implementation. One must also be able to analyse geometry of meaningful complexity in an appropriate amount of time as described by Gustafson’s law [70]. Although the performance of computing hardware is continually increasing, extensive software optimisations are required to exploit these gains due the recent shift in computing paradigm from high clock frequencies to highly vectorised designs. The following chapter details the optimisations implemented in RayFactor to exploit the performance gains afforded by the architecture of contemporary CPUs.

## Optimisation of RayFactor for the CPU

---

Due to the statistical nature of MC-RT, the results of the simulation become increasingly accurate with the number of samples. Whilst this provides a simple pathway by which the uncertainty of results can be reduced, the workload required to obtain acceptable results can be extreme [71].

In order to calculate view factors with high accuracy in a reasonable amount of time, systematic optimisation must be carried out to ensure that all aspects of the codebase are operating as efficiently as possible as well as exploiting the underlying hardware.

However, before such optimisation is undertaken an appreciation of one of the most fundamental aspects of any code base must be gained. This is the hardware representation of real numbers, which on modern hardware are approximated using floating-point representation.

### 4.1 Floating Point Precision

Arithmetic on computer hardware presents the interesting problem of trying to accurately represent a (infinitely defined) real number in a finite amount of memory. One widely used method is floating pointing representation in which numbers are represented by a fixed number of significant digits called the significand (or mantissa) which is scaled using an exponent in a similar fashion to scientific notation as shown in equation (68).

$$Number = significand \times base^{exponent} \quad (68)$$

Floating point number representation is provided on nearly all-modern hardware as specified by the IEEE 754 standard. This standard defines four basic binary

representations (base=2) which are commonly supported in most programming languages and listed in Table 5.

**Table 5: Summary of IEEE 754 binary floating-point types**

| IEEE Name | Common Name | Base | Significand Bits | Exponent Bits | Bits Precision | Decimal digits |
|-----------|-------------|------|------------------|---------------|----------------|----------------|
| binary16  | Half        | 2    | 10               | 5             | 11             | ~3             |
| binary32  | Single      | 2    | 23               | 8             | 24             | ~7             |
| binary64  | Double      | 2    | 52               | 11            | 64             | ~16            |
| binary128 | Quadruple   | 2    | 112              | 15            | 113            | ~34            |

The trade-off involved in the selection of the floating-point data type is that of maximising precision while minimising storage size. Although one might expect that the difference in floating point storage size will not affect performance, for all but pure serial CPU applications a larger data type will reduce the maximum computational throughput. As processors have increased in speed over the years, and their architecture has become increasingly parallel, data transfer speeds have become a bottleneck, a situation that is particularly true for GPGPU programs which will be discussed in detail in Chapter 5.

Furthermore, the speed increases obtained through data level parallelism instructions such as SIMD (see Section 4.4.2) is proportional to the size of the floating-point representation. For example, on a processor with a SIMD width of 128 bits, one could operate on 8, 4, 2 or 1 numbers simultaneously depending on whether a half, single, double or quadruple precision floating point is used, respectively.

Given the limitations of the half precision floating point and the limited support for the quadruple precision floating point, early versions of RayFactor were developed using double precision floating point numbers as is common practice in scientific computing. However, as vectorisation efforts were ramped up, RayFactor was converted to single precision to exploit performance gains at the cost of some precision.

An extensive discussion of floating point representation and arithmetic is out of the scope of this thesis but a detailed commentary may be found in the literature [72]. However, there is one aspect of floating point representation that is of particular importance and worthy of discussion, the machine epsilon.

## 4.2 Machine Epsilon

As floating-point numbers cannot exactly represent real numbers, rounding is performed in order to obtain a floating-point representation of a real number. The upper bound on the relative error due to rounding is known as the machine epsilon. This may be calculated using the precision ( $p$ ) and the base ( $b$ ) of the floating point as shown in equation (69).

$$eps = \frac{b}{2} b^{-(p-1)} \quad (69)$$

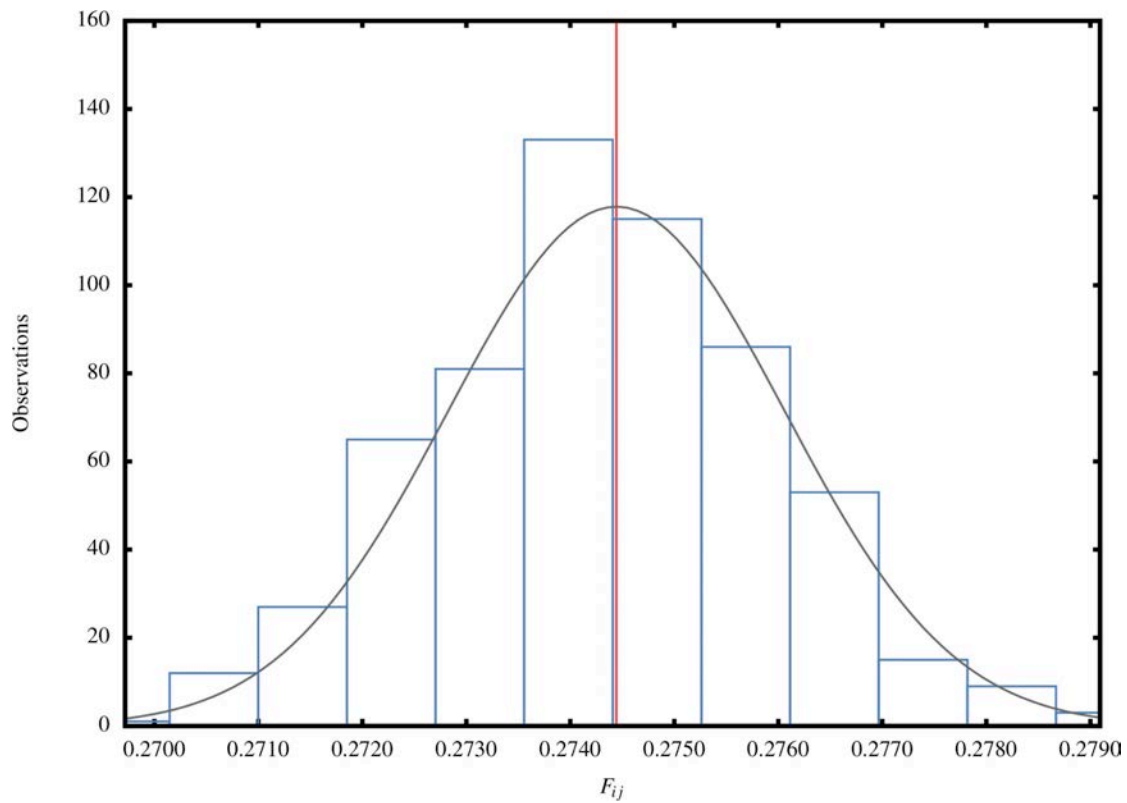
From this equation, we can determine that the upper bound on the relative error for a single precision floating-point number ( $b=2$ ;  $p=24$ ) is  $1.1921 \times 10^{-7}$ , and therefore all calculations will carry this relative uncertainty.

Machine epsilon and floating point precision becomes an issue in RayFactor when calculating ray intersection times. As presented in Chapter 2, ray-primitive intersection is determined by the earliest intersection time greater than 0, but consider launching a ray from the curved surface of a cylinder or sphere. When the zenith angle of the ray direction  $\phi$  is selected, it may have a maximum value of  $\pi$ . The absolute error in the representation of the zenith angle could thus be as large as  $3.74 \times 10^{-7}$ . Although this might not seem very large it is large enough to cause the ray to intersect with the convex surface from which it was fired with an intersection time greater than zero.

The issue of ‘false intersections’ is not only limited to launching rays from primitives with curved surfaces but can occur in numerous circumstances. Another example is launching a ray from a rectangle primitive which has been translated and/or rotated. Limitations in the floating point precision when calculating the ray starting point can result in the ray being fired from slightly behind the surface rather than on the surface, again resulting in a false positive when testing for intersection.

RayFactor accounts for the lack of precision in floating point representation by defining a factor for absolute error in the intersection time calculations. This number acts as a ‘tuneable’ factor for calculation tolerance with intersection times less than this factor being discarded. In the initial stages of RayFactor development, this factor was set to  $1 \times 10^{-6}$ , however as faster, less precise functions were implemented for the calculation of sine, cosine and square root, this factor was

increased to  $1 \times 10^{-4}$  which provides the optimal balance between rejecting false positives and accepting valid intersections for primitives in close proximity to each other. The effects of having an incorrectly selected error factor may be observed if we revisit the view factor distribution for the C-77 benchmarking case with an error tolerance of  $1 \times 10^{-3}$  as shown in Figure 9. Here we can visually detect a slight left skew on the distribution as valid intersections where the cylinder and annulus meet are disregarded, as due to the short distance between the two objects their intersection times are less than  $1 \times 10^{-3}$ .



**Figure 9: Slightly left skewed distribution for C-77 case with an error tolerance of  $1 \times 10^{-3}$**

Although the left skew in Figure 9 is moderate, there is a definite change in the view factor distribution when compared to Figure 5, demonstrating that the choice of error tolerance is worth optimising due to the limited precision of floating point numbers.

However, the precision of number representation is not the only factor in the integrity of view factors calculated by RayFactor. Perhaps worthy of more attention is the random number generator which lies at the heart of every Monte Carlo based method.

### 4.3 Random Number Generation

As discussed in Section 1.6.3, the selection of a pseudorandom number generator (PRNG) is of critical importance in MC-RT. When programming a Monte Carlo based method, particularly for parallel applications [44], the implemented PRNG should be thoroughly examined as no one PRNG is superior in all circumstances, as the quality of a given generator is closely related to the problem being solved [73]. Therefore, it is necessary to examine multiple PRNGs and their effect on result distribution and simulation speed to ensure that poor simulation performance is not realised due to inappropriate PRNG selection.

Several algorithmic classes of PRNGs were considered for use in RayFactor with the classes being linear congruential generators (LCG), the Mersenne Twister (MT) algorithm and the recently developed counter-based Threefry generator.

In addition to the PRNGs considered in this research, many high quality alternatives such as the XorShift (XorS) [45] and the Well Equidistributed long-period Linear (WELL) generator [69] have been developed. Given the focus of this thesis not all PRNGs could be considered, and therefore selection was limited to generators that have been demonstrated to provide high performance and for which suitable well-supported implementations are available.

#### 4.3.1 Tested Generators

##### *Linear Congruential Generators*

Linear Congruential Generators (LCG) are one of the oldest and more commonly used PRNG algorithms in Monte Carlo simulation of radiative transport [73]. Once seeded, these generators produce a sequence of pseudorandom numbers using a recurrence relation requiring only 32 bits of memory to retain the state of the generator. However, they suffer from a short period (typically around  $2^{32}$ ) and show strong serial correlation.

Generators that exhibit a strong serial correlation will generate similar sequences of pseudorandom numbers for close seed values, limiting the options where they can be used for the generation of parallel streams of pseudorandom numbers.

The highly accessible standard library implementations of LCG, `rand` and `drand48`, were examined for use with RayFactor.

### ***Mersenne Twister Generator***

Makoto Matsumoto and Takuji Nishimura [48] developed the Mersenne Twister generator as a PRNG specifically optimised for Monte Carlo simulations. The Mersenne Twister algorithm is a twisted, generalised feedback shift register function based on the Mersenne prime and uses matrix linear recurrence over a binary field to produce fast, high quality pseudorandom numbers.

The Mersenne Twister generators have a large period ( $\geq 2^{19937} - 1$ ) and do not exhibit serial correlation, making them a viable option for generation of parallel streams of pseudorandom numbers. However, the Mersenne Twister generator does require a large amount of memory to retain the state of the generator and can take a long time to start generating a high quality sequence of random numbers if seeded with a poor initial state<sup>4</sup>. Therefore, when using this generator care must be taken to select suitable seed numbers.

Two implementations of the Mersenne Twister generator were examined for use with RayFactor. The 2004 MT19937-64 implementation and the double precision SIMD-orientated Fast Mersenne Twister (dSFMT) [49].

### ***Threefry Generator***

D.E Shaw Research [50] developed the Threefry algorithm based on the cryptographically secure Threefish algorithm in late 2011. The Threefry generator trades the cryptographic strength of Threefish for speed while still maintaining a high level of randomness. The Threefry generator creates a very high quality sequence of pseudorandom numbers passing the full battery of the TestU01 tests of randomness [74] (also known as the crush tests) and requires only 256 bits of memory to retain its state.

The Threefry implementation in the Random123 library published by D.E. Shaw Research was examined for use in RayFactor. Numerous counter-based generators are present in this library including the Philox PRNG which is used in the OpenCL version of RayFactor presented in Chapter 5. However, the Threefry generator was selected as a potential candidate for RayFactor as it is the fastest counter-based PRNG in the Random123 library for CPUs without Advanced Encryption Standard (AES) hardware support [50].

---

<sup>4</sup> Poor initial state refers to using a seed number with low entropy, such as a number whose binary representation is predominately many zeros.



Unfortunately, the Random123 library was published after the development and testing of RayFactor had been completed and therefore its consideration is largely in retrospect.

#### **4.3.2 PRNG Summary and Selection**

The selection of a PRNG for MC-RT is based primarily on two factors, speed and randomness. PRNG speed may easily be quantified in terms of the number of random numbers generated per second but randomness, however, is a more difficult property to define.

Several batteries of statistical tests have been compiled to assess the randomness of a PRNG with the most frequently referenced being the DIEHARD tests [75], and its successor the TestU01 suite [74] commonly referred to as the ‘crush tests’. The later are by far the more stringent of the two test suites, being designed to address the known shortcomings of the DIEHARD tests.

The TestU01 suite is divided into three separate batteries of tests, ‘small crush’ consisting of 15 tests, ‘crush’ consisting of 96 tests and ‘big crush’ consisting of a total of 106 tests [74]. An appreciation of the intensity of the crush tests relative to the DIEHARD tests may be gained by examining the time it takes to complete PRNG testing for each suite. The small crush, crush and big crush tests take around 2 minutes, 1.7 hours and 12 hours, respectively, while the DIEHARD tests take approximately 15 seconds to be completed on a comparable machine [76].

As the TestU01 suite has become the method of choice for testing the randomness of PRNGs [56], the ‘randomness’ of the PRNGs examined for use with RayFactor will be quantified using this option. Table 6 summarises the properties of interest for the PRNGs examined in this research, including their performance on the Mac Pro 4.1.

Here the ‘crush resistance’ has been graded poor, good or excellent, with poor meaning failure of the small crush test, good being limited failure of crush and big crush, and excellent being the passing of all the crush test batteries. The crush resistance of MT19937 has been rated ‘good’ as it only fails 2 out of the 96 crush tests and 2 out of the 106 big crush tests [76], however these tests are measures of linear complexity and failure here is unlikely to effect the results of a MC-RT simulation.

**Table 6: Summary of PRNGs considered for use in RayFactor**

| PRNG          | State Size | Period         | Performance        |          | Crush Resistance |
|---------------|------------|----------------|--------------------|----------|------------------|
|               | bytes      |                | Numbers/s          | Relative |                  |
| rand          | 4          | $2^{32}$       | $9.09 \times 10^7$ | 0.64     | Poor             |
| drand         | 8          | $2^{32}$       | $6.67 \times 10^7$ | 0.47     | Poor             |
| MT19937       | 2496       | $2^{19937}-1$  | $1.43 \times 10^8$ | 1.00     | Good             |
| dSFMT         | 3080       | $2^{216091}-1$ | $1.43 \times 10^8$ | 1.00     | Excellent        |
| dSFMT (array) | 3080       | $2^{216091}-1$ | $1.67 \times 10^8$ | 1.17     | Excellent        |
| Threefry 2x32 | 16         | $2^{128}$      | $1.00 \times 10^8$ | 0.70     | Excellent        |
| Threefry 4x32 | 32         | $2^{128}$      | $1.29 \times 10^8$ | 0.90     | Excellent        |
| Threefry 4x64 | 64         | $2^{128}$      | $1.14 \times 10^8$ | 0.80     | Excellent        |

*NOTE: Threefry variants are designated Threefry AxB where B is the size of the number in bits (32 = single precision floating point; 64 = double precision floating point) while A is the number of random numbers generated per function call.*

Examining the results presented in Table 6, it appears that the optimal selection is the array-based dSFMT as it exhibits the best characteristics across the board with the exception of its state size. However, its implementation carries the requirement that the pseudorandom numbers are generated as an array with a minimum size of 192 elements each time dSFMT is called. This requires excessive memory for storage of the array and extra computations to distribute the array contents, both of which compound to erode the performance bonus. However, using dSFMT in a conventional manner (i.e. a single number is generated at each call) does not carry these constraints and provides the next best performance.

Comparable to dSFMT was MT19937, which demonstrated identical performance, lower memory requirements for state retention, and negligible difference in ‘randomness’ for application in MC-RT. However, dSFMT has been shown to have significantly better recovery than MT19937 from 0-excess state<sup>5</sup> making it a more reliable candidate [49].

The main weakness of dSFMT is the amount of memory required to retain its state. Moving towards vectorisation of RayFactor using an independent sequence

---

<sup>5</sup> 0-excess state is where too many of the bits in the PRNG state are set to 0. When this occurs the PRNG can take a long time to ‘recover’ the state bits to being a blend of 1’s and 0’s. As the pseudorandom numbers produced by the PRNG are a function of the state during this period the sequence of generated numbers will be similar.

technique for parallel generation of pseudorandom numbers, each computing thread (discussed in detail in Section 4.4) would need to store its own copy of the generator state. As the number of threads increases, this will strain processor resources and negatively impact on simulation speed. This makes Threefry an attractive alternative, as it requires nearly 100 times less memory for state retention for only 10% lower performance. Although the period of Threefry is comparatively low, it has been demonstrated to produce at least  $2^{128}$  unique sequences of pseudorandom numbers making it ideal for massively multicore processors.

Despite the advantages of Threefry in parallel applications, its very recent release (November 2011) lead to dSFMT being incorporated in RayFactor.

The selection of a fast, high quality PRNG ensures integrity of simulation results. However, the overall performance improvement from its integration is somewhat limited compared to what is possible through vectorisation. Modern processors are increasingly built using vectorised architectures and without user care the vast performance gains possible will not be realised.

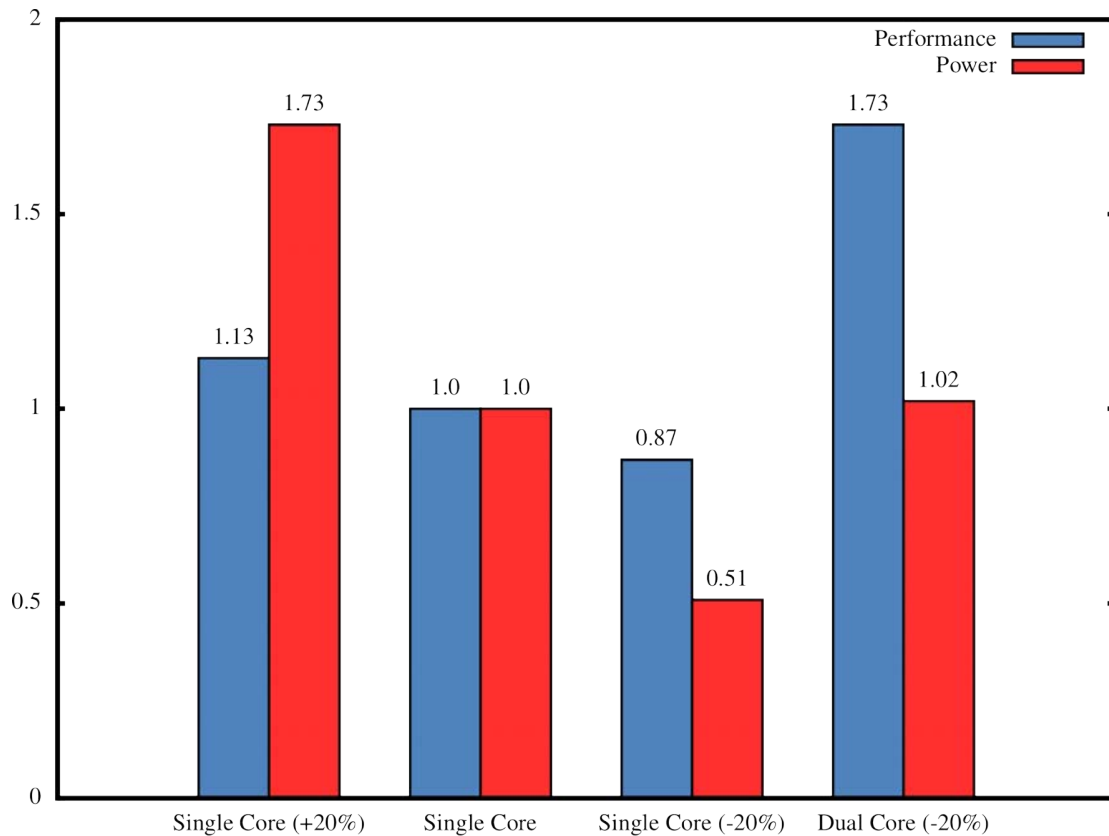
#### **4.4 Vectorisation**

MC-RT is often referred to as an ‘embarrassingly parallel’ algorithm, as it involves a large number of essentially identical sequences of calculations being performed for each ray launched. This algorithm structure lends MC-RT to vectorisation in which rather than performing each calculation sequentially, processing is divided into independent blocks of data and processed in parallel.

In years past, performance increases have been largely realised through dramatic increases in processor clock frequency with the top tier clock speeds jumping from 5 MHz in 1983 to 3 GHz in 2002 [77]. However, as processor clock speeds approached the 3 GHz mark, the associated heat generation and dissipation posed a design constraint on further CPU performance gains through increases in processor clock frequency.

Despite the constraints posed by heat generation, the number of transistors on integrated circuits (transistor budgets) has continued to double approximately every two years in accordance to Moore’s Law. These growing transistor budgets combined with the constraints in increasing processor clock frequency have naturally lead to the development of multicore processors. The introduction and success of multicore processors has not, however, only been due to the growing transistor budgets but also due to the enhanced performance and power

efficiency they afford. This is demonstrated in Figure 10, which shows the performance and power draw of a single core which has had a 20% increase (over-clocked) and decrease (under-clocked) in clock frequency compared to a 20% under-clocked dual core.



**Figure 10: Performance and power draw of an under-clocked and over-clocked processor core [77]**

By combining two 20% under-clocked single cores, a substantial performance increase of 73% is achieved for only a 2% increase in power draw. However the performance increases by moving to multi-core processor designs is not automatic and software must be vectorised to reap the benefits.

Two vectorisation techniques were used in RayFactor, Single Instruction Multiple Data (SIMD) and Single Instruction Multiple Thread (SIMT). In essence, SIMT is a variant of SIMD with the added characteristic that instructions can be either coherent (i.e. instructions being executed are synchronous over all data) or incoherent (i.e. instructions are being executed out of step over all data).

#### 4.4.1 Single Instruction Multiple Thread (SIMT)

Initial vectorisation efforts for RayFactor used a coarse grain SIMT approach using the Open Multiprocessing (OpenMP) Application Programming Interface (API). OpenMP is a cross-platform, shared memory, multiprocessing interface which provides a portable and scalable way to utilise modern multicore CPUs.

In a SIMT approach, the workload is divided across multiple threads (i.e. an independently managed sequence of instructions) which are then executed independently when a processing resource becomes available. For a MC-RT problem, the optimal number of threads will match the number of virtual processor cores available. On the Mac Pro 4.1 test system, which has 8 physical cores and employs hyper-threading (i.e. two virtual cores per physical core), optimal performance is obtained using 16 threads. If too few threads are used, processing resources will be under-utilised and if too many threads are used, time and resources will be wasted in the creation and swapping of the surplus threads. It is therefore important that the number of threads used matches the underlying processor. In its default configuration, OpenMP will use a thread count equal to the number of virtual cores made available by the underlying processor.

In addition to using the correct number of threads, it is also important that the work is divided evenly among all threads. Uneven workloads will result in under-utilisation of the available processing resources resulting in a drop in overall simulation performance.

To reap the full benefits of SIMT, work must be broken up into independent units of work, any dependencies a work unit has on adjacent units will lower the efficiency of running across multiple cores. For MC-RT there are several ways in which the work can be divided to obtain independent work units of which the most obvious choices are on a per object and per ray basis.

Dividing the workload on a per object basis, each thread would be responsible for generating and launching all rays for a single object. In order to achieve full processor utilisation, the number of objects in the geometry being studied must be a multiple of the number of threads. For current commodity CPUs this means the total number of objects should be a multiple of 8 or 12 which seems reasonable. However, for research grade hardware, such as the Intel 80-core Polaris processor, the total number of objects would need to be a multiple of 160 for full utilisation. Although an object count of 160 might not seem like much for a contemporary finite element mesh (FEM) based MC-RT program, primitive

based modelling can permit the modelling of complex systems with relatively few objects. Therefore, workload division on a per object basis is not optimal for primitive based MC-RT and was not pursued in the development of RayFactor.

Rather, the division of workload on a per ray basis was adopted for RayFactor's SIMT implementation. Here, for each object the number of rays to be launched is divided evenly among the threads, and each thread is responsible for the generation and intersection calculations of their allocated rays. This results in even loading across each thread and high resource utilisation as typically millions of rays are launched per object.

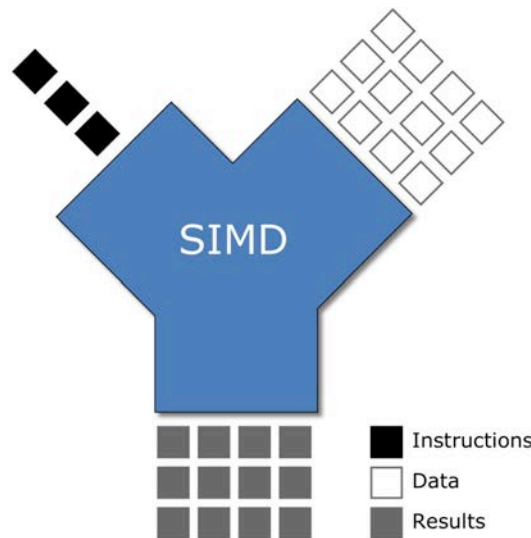
However, when dividing the workload in such a way, care must now be taken in the generation of the random numbers used to construct the rays. As all threads are simultaneously launching rays from a single object, each thread must maintain its own PRNG which if not seeded correctly will result in 'domain collisions' (i.e. the PRNG on each thread generates identical sequences of numbers). Early SIMT implementations of RayFactor encountered this issue, which was discovered to be a bug in the seeding function of the dSFMT PRNG. Correspondence with the authors of dSFMT lead to a fix in which the PRNG of each thread could be uniquely seeded using an array containing the thread number, the clock time in microseconds, and the number of processor ticks from the start of RayFactor execution to the time of PRNG seeding.

SIMT vectorisation using the OpenMP technology proved a valuable starting point for RayFactor vectorisation efforts; however, further vectorisation on each thread may be achieved using a Single Instruction Multiple Data (SIMD) technique with SSE technology.

#### **4.4.2 Single Instruction Multiple Data (SIMD)**

Further to SIMT vectorisation, finer grain, data level parallelism using SIMD instructions was implemented in RayFactor. In essence, SIMD allows the same instruction to be performed on multiple data points simultaneously, as demonstrated in Figure 11.

One of the first SIMD instruction sets available on commodity processors was the AMD 3DNow! extensions which were introduced in 1998 and promptly replaced in 1999 by Intel's Streaming SIMD Extensions (SSE). Since its introduction the SSE instruction set has become widely supported on both AMD and Intel processors.



**Figure 11: Data level parallelism with SIMD**

The real impact of floating-point precision (or more concisely the size of the floating-point representation) on performance becomes apparent when utilizing SIMD instructions. In order to implement SIMD in hardware, additional arithmetic logic units (ALU), floating point units (FPU) and registers, afforded by increasing transistor budgets, must be added to the processor silicon. The number of data points that may be simultaneously operated on is a function of the registry and floating-point width. The SSE registries on the Mac Pro 4.1 system have a width of 128 bits (referred to as the SIMD width) and, therefore, if RayFactor uses a single precision floating-point representation, which requires 32 bits per number, a total of 4 data points can fit in the register and be operated on simultaneously. If a double precision floating-point representation is used, which requires 64 bits of storage per number, only 2 data points may fit in the registers. Therefore, a SIMD implementation using single precision floating-point will have twice the throughput of a SIMD implementation using double precision floating-point representation.

Further SIMD enabled performance gains are available through the Advanced Vector Extensions (AVX), the successor to SSE. Intel proposed AVX in 2008 and it became available in the Intel Sandybridge and AMD Bulldozer processors released in 2011. The SIMD width on AVX enabled processors was increased to 256 bits, which allows a total of 8 single precision floating-points to be operated on simultaneously. The processor on the system used for this thesis was not AVX enabled, and therefore, only SSE instructions were implemented. However, it is important to note that the benefits from using SIMD are growing due to the continual advancement of SIMD technologies on the CPU.

SIMD vectorisation was implemented in RayFactor using SSE instructions to process four rays (determined by the SIMD width) simultaneously in a structure of arrays (SoA) format. In SoA format, each of a ray's parameters, such as the x coordinate of the ray starting point, are packed into arrays. This allows the treatment of 'ray packets' rather than individual rays and using this format any elementary operation (such as add and multiply) may be completed on four rays simultaneously using SSE instructions.

Performing calculations on ray packets does, however, have limitations. Branched calculations cannot be performed efficiently, as each instruction is performed across all four rays. Early exits from intersection calculations now require all rays in a given bundle to satisfy the exit conditions, otherwise intersection calculations must be fully completed. Given that the rays are produced randomly, it is highly unlikely that they will be coherent lowering the chances that the early exit condition will be satisfied for all four rays.

Regardless of the constraints and limitations imposed by both the OpenMP and SSE technologies, impressive performance gains were delivered as discussed in the following section.

#### 4.4.3 SIMT and SIMD Performance Improvement

The maximum expected performance improvements for vectorisation of a fixed size problem may be calculated using Amdahl's law given the fraction of the program that is parallelisable  $P$  and the number of processing cores used  $N_{cores}$  as shown in equation (70).

$$S = \frac{1}{(1 - P) + \frac{P}{N_{cores}}} \quad (70)$$

Amdahl's law permits the prediction of the maximum number of processors that can be used to increase program performance. For example, a program for which 90% of the algorithm is parallelisable ( $P = 0.9$ ) a maximum speed up of 10 times may be achieved using around 1024 processors. Similarly we may use Amdahl's law to predict that for embarrassingly parallel algorithms such as MC-RT where  $P \approx 1$  the maximum speedup will be essentially equal to the number of processors used.

This same result is true for SIMD and in that the maximum speedup possible is equal to the number of floating point numbers that will fit into the SIMD



registries. Given that SIMD is implemented on each thread, we may compound the maximum speedups and state that for a combined SMT and SIMD MC-RT implementation, the maximum speed up through vectorisation is equal to the number of cores multiplied by the number of SIMD data points.

In practice the Amdahl's law speedups are never fully realised due to factors such as implementation overheads and hardware bandwidth, memory and IO limitations. For SMT, overhead work must be completed to spawn the worker threads and collate their results, while for SIMD overhead work must be completed to pack and unpack SIMD data arrays. This work, while necessary to utilise vectorisation techniques, may be viewed as an inefficiency as it does not directly contribute to simulation results. Given our understanding of Amdahl's law, the efficiency ( $\epsilon$ ) of each vectorisation technique and the number of floating points which fit into a SIMD register ( $W_{SIMD}$ ) the speedup for a combined SMT + SIMD implementation (compared to a serial implementation) may be expressed as shown in equation (71).

$$S = \epsilon_{SMT} \times N_{cores} \times \epsilon_{SIMD} \times W_{SIMD} \quad (71)$$

With respect to RayFactor, the primary cause of efficiency reduction for SMT vectorisation is from the spawning of threads and the reduction of the view factors calculated by each thread into a single view factor matrix, where the fraction of the total simulation time required for each of these operations is proportional to the number of objects and inversely proportional to the number of rays.

The primary cause of efficiency reduction for SIMD is the packaging of random numbers (generated as scalar values) into a packed array for SIMD treatment, and extracting the final ray intersection data from the SIMD array. The fraction of the total simulation spent on these two operations is essentially constant. Despite efficiency reductions for SIMD, the net speedup relative to a non-SIMD version is not necessarily less than 1:1 as it permits various computational shortcuts to be taken (discussed further in Section 4.5.1). Table 7 presents relative performance for SIMD and non-SIMD RayFactor implementations for the C-77 benchmarking case for a range of cores/threads.

Given that the Mac Pro 4.1 test machine has 8 processing cores (spread across its two processors), one might expect that maximum performance would be reached using 8 threads. However, Table 7 indicates that an increase in

performance of 30% (i.e. 7.4 to 9.7) for the non-SIMD and 11% (i.e. 29.5 to 32.9) for the SIMD versions is achieved using twice this amount. This is due to hyper-threading, which allows memory or pipeline latency to be covered up by creating two virtual cores for each physical core. Here, as execution on one of the virtual cores stalls waiting for data, physical resources are quickly switched over to the second virtual core where processing continues until it in turn stalls and resources are switched back to the original virtual core.

**Table 7: Measured relative speed for SIMD and non-SIMD RayFactor versions on the Mac Pro 4.1 test system for the C-77 benchmarking case**

| Number of cores     | Non SIMD | SIMD (SSE) |
|---------------------|----------|------------|
| 1                   | 1.0      | 4.3        |
| 2                   | 2.0      | 8.5        |
| 4                   | 4.0      | 17.0       |
| 6                   | 5.7      | 23.3       |
| 8                   | 7.4      | 29.5       |
| 16 (hyper-threaded) | 9.7      | 32.9       |

Using these relative run-times, we may calculate efficiency factors for both the SIMD and non-SIMD versions as a function of the number of physical cores utilised as shown in equation (72).

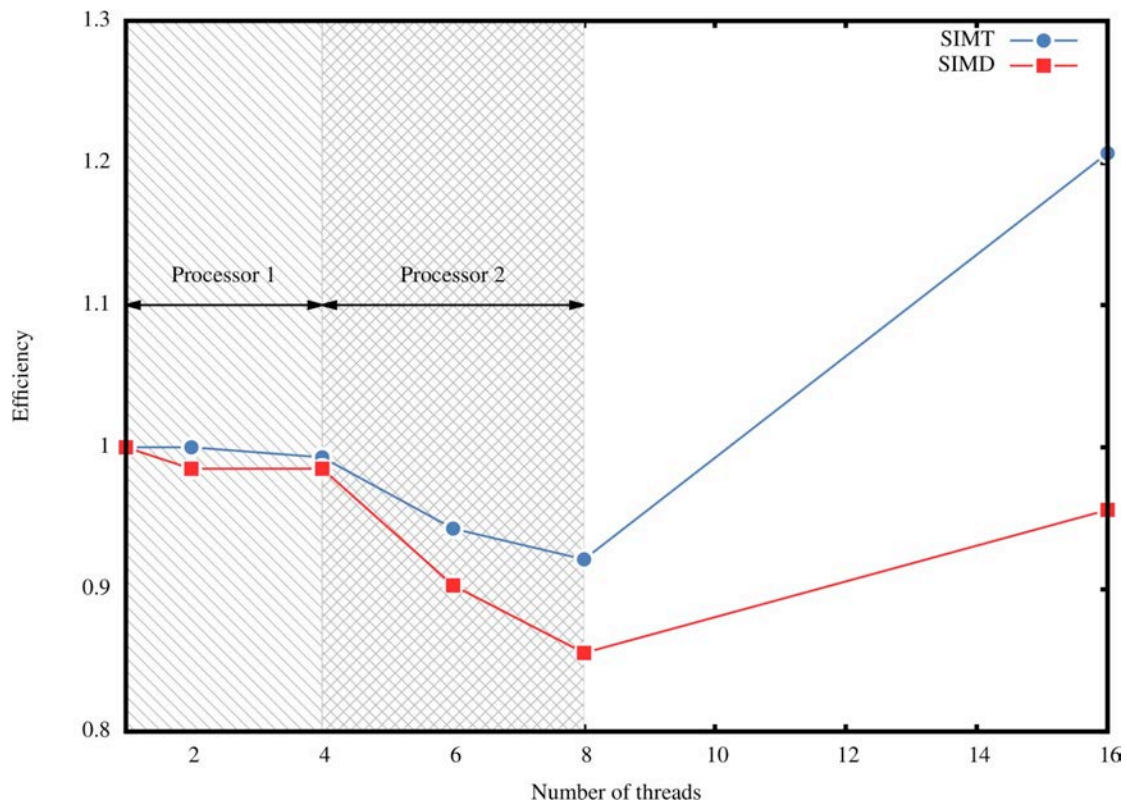
$$\epsilon(N_{cores}) = \frac{t_{parallel} \times N_{cores}}{t_{serial}} = \frac{S}{N_{cores}} \quad (72)$$

The efficiency as a function of the number of physical cores used provides insight into the scalability of RayFactor particularly as it moves from using a single processor ( $N_{cores} \leq 4$ ) to two processors. Figure 12 shows the efficiency as calculated from the run-times of each version on the Mac Pro 4.1 as the ratio of the relative speedup to the number of physical processor cores used.

Here efficiencies of 98.5% and 99% are experienced for the SIMD and non-SIMD versions, respectively, when only a single processor is utilised. As the thread count increases into the region where the second processor is utilised, we see a decline in efficiency, as the slower communication speed between the two processors becomes a bottleneck. The efficiency decline for the SIMD version is

observed to be much sharper than that of the non-SIMD version. This is to be expected, as from Table 7 the throughput of the SIMD version is around 4.2 times that of the non-SIMD version and, therefore, memory and IO limits are approached more closely.

Despite the various limitations, RayFactor vectorisation efforts proved very successful with a 9.7 times speedup for the SIMT version and a 32.9 times speedup for the SIMT + SIMD version. Given the verified scalability of the MC-RT algorithm in RayFactor, later research using hardware-enabled vectorisation was carried out using General Purpose Graphics Processing Units (GPGPU). Typically, speed increases of 10-300 fold [78-80] can be achieved when utilising GPGPUs, however this is not without its challenges and limitations as will be discussed in Chapter 5.



**Figure 12: Efficiency of SIMT and SIMD RayFactor implementations**

Vectorisation techniques allowed RayFactor to run significantly faster by exploiting the underlying hardware. However, once the limit of hardware speedup was reached, further enhancements to the performance of RayFactor had to be achieved through optimisation at an algorithmic level, as discussed in the following section.

## 4.5 Algorithmic Optimisation

### 4.5.1 SIMD Random Number Recycling

In order to fully specify a ray, four random numbers are required to calculate the ray starting point coordinates and directional angles. As discussed, the implementation of SIMD techniques in RayFactor leads to four rays being treated together as a ray packet. Taking a simplified approach, each ray packet would require the generation of 16 random numbers and four operations to pack these numbers into a SIMD array.

Taking advantage of the SIMD architecture, we can reduce the ray generation requirements to the generation of only four random numbers and a single pack operation. Here, each time a random number is used, the elements of the array are shuffled by one place. This results in all four rays using the same random numbers to specify their rays, but the random numbers are used in different positions, as shown in Figure 13, so that each generated ray remains independent of adjacent rays in the packet.

| Ray | 1            | 2            | 3            | 4            |
|-----|--------------|--------------|--------------|--------------|
| RN1 | $\epsilon_1$ | $\epsilon_2$ | $\epsilon_3$ | $\epsilon_4$ |
| RN2 | $\epsilon_2$ | $\epsilon_3$ | $\epsilon_4$ | $\epsilon_1$ |
| RN3 | $\epsilon_3$ | $\epsilon_4$ | $\epsilon_1$ | $\epsilon_2$ |
| RN4 | $\epsilon_4$ | $\epsilon_1$ | $\epsilon_2$ | $\epsilon_3$ |

Figure 13: Recycling of random numbers (RN) in an SIMD implementation

The recycling of random numbers in this manner allows the speed of the SIMD version to be greater than 4 times that of the non-SIMD counterpart as demonstrated in Table 7.

### 4.5.2 Ray Direction Alignment

The ray direction is first calculated as a point on a generic hemisphere whose pole is on the positive z-axis and whose base lies in the x-y plane. In order to be correctly specified for the primitive from which the ray will be fired, the ray directional hemisphere must be rotated so that it is aligned with the surface normal at the ray starting point.

The ray directional hemisphere may be aligned with the surface normal  $(n_x, n_y, n_z)$  by converting the Cartesian coordinates of the surface normal to spherical coordinates as shown in equation (73) and performing an affine rotation of the hemisphere using equation (36).

$$\begin{aligned}\theta &= \tan^{-1}\left(\frac{n_y}{n_z}\right) \\ \phi &= \cos^{-1}(n_z)\end{aligned}\tag{73}$$

While straightforward, this method is computationally very expensive, as it requires execution of slow transcendental instructions ( $\tan^{-1}$ ,  $\cos^{-1}$ ,  $\cos$  and  $\sin$ ). Alternatively, the Rodrigues rotation formula shown in equation (74) may be utilised efficiently to develop the rotational matrix for the ray direction.

$$R = I \cos \theta + [k]_{\times} \sin \theta + (1 - \cos \theta)kk^T\tag{74}$$

The Rodrigues rotation formula requires a rotation angle  $\theta$  and a unit vector along the axis of rotation  $k$ . In the task of aligning the ray directional hemisphere with the surface normal of the primitive at the ray starting point, neither the axis of rotation or the rotation angle are known. Here it is desired to rotate the z-axis to the surface normal  $N$  and, therefore, the axis of rotation is simply the vector perpendicular to  $n$  and the z-axis  $n_{\perp}$ , which may be calculated as shown in equation (75).

$$\begin{aligned}n_{\perp} &= \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \times n \\ &= \begin{pmatrix} -n_x \\ n_y \\ 0 \end{pmatrix}\end{aligned}\tag{75}$$

This leaves the rotational angle  $\theta$  as the remaining unknown parameter. Although this angle could be calculated from the available data, this is inefficient

and would lead to expensive trigonometric calls similar to the first alignment algorithm proposed in equation (73).

Alternatively, we may directly obtain values for  $\cos \theta$  and  $[k]_{\times} \sin \theta$  using the geometric properties of the vectors  $n$ ,  $n_{\perp}$  and  $k$ . Firstly it may be appreciated that the z component of the normal vector is equal to the cosine of the angle it forms with the z-axis (the rotational angle  $\theta$ ) as shown in equation (76)

$$\cos \theta = n_z \quad (76)$$

Secondly, the calculated perpendicular vector  $n_{\perp}$  has a direction along the desired rotation axis  $k$  and a magnitude equal to the sine of the rotational angle  $\theta$  and, therefore, the second component of the Rodrigues rotation formula may be specified as shown in equation (77).

$$\begin{aligned} n_{\perp} &= k \sin \theta \\ [n_{\perp}]_{\times} &= [k]_{\times} \sin \theta \end{aligned} \quad (77)$$

Finally, the component  $kk^T$  must be determined. From equation (77), we may calculate that  $n_{\perp}n_{\perp}^T = \sin^2 \theta kk^T$ , which may be converted to the desired  $kk^T$  by dividing by  $\sin^2 \theta$ . However, calculation of the sine function may be avoided by using the identity  $\sin^2 \theta = 1 - \cos^2 \theta$ , where equation (76) provides the cosine of the rotational angle, to give the final form of  $kk^T$  shown in equation (78).

$$kk^T = \frac{n_{\perp}n_{\perp}^T}{1 - n_z^2} \quad (78)$$

Substituting equations (75) through to (78) into equation (74), the Rodrigues rotation formula may be developed without the use of any computationally expensive transcendental functions as shown in equation (79).

$$\begin{aligned}
R &= I \cos \theta + [k]_{\times} \sin \theta + (1 - \cos \theta)kk^T \\
&= In_z + [n_{\perp}]_{\times} + \frac{n_{\perp}n_{\perp}^T}{(1 + n_z)} \\
&= \begin{pmatrix} n_z & 0 & n_x \\ 0 & n_z & n_y \\ -n_x & -n_y & n_z \end{pmatrix} + \frac{1}{1 + n_z} \begin{pmatrix} n_y^2 & -n_x n_y & 0 \\ -n_x n_y & n_x^2 & 0 \\ 0 & 0 & 0 \end{pmatrix}
\end{aligned} \tag{79}$$

Implementation of the Rodrigues rotational function in place of the initial ray distribution alignment algorithm based on equation (73) led to a reduction in simulation time of approximately 40% for the non-SIMD version of RayFactor.

Despite the fact that transcendental functions were eliminated from the algorithm to align the ray directional distribution with the primitive surface normal, the transcendental functions cannot be completely eliminated from the RayFactor codebase. Given that the instructions to calculate these transcendental functions (and various other elementary instructions such as square root) are up to 100 times slower than addition or subtraction instructions, they are naturally the next target for code optimisation effort.

## 4.6 Optimisation of Elementary Functions

### 4.6.1 Fast Reciprocal Square Root

The calculation of a reciprocal square root is required for the normalisation of the ray directional vectors. This operation is conducted once per ray fired and can consume a significant amount of computational time.

On modern processors, there are two primary pathways by which the reciprocal square root may be calculated. The first is to calculate the square root of a number using the SQRTPS instruction followed by dividing 1 by the calculated square root using the DIVPS instruction. The second is to use the instruction RSQRT which directly computes the reciprocal square root using various approximation shortcuts based on the IEEE 754 floating point layout.

As one might expect, the differentiating factor between the two methods is the accuracy and speed as evident in Table 8 which lists the precision and relative performance of each method for the SSE and AVX instruction sets.

**Table 8: Methods for calculating the reciprocal square root on modern processors [81]**

| Method         | Precision | SSE         | AVX         |
|----------------|-----------|-------------|-------------|
|                |           | Performance | Performance |
| SQRTPS + DIVPS | 24 bits   | 1           | 1           |
| RSQRTPS        | ~11 bits  | 13.5        | 9.1         |
| RSQRT + 1 NR   | ~22 bits  | 5.2         | 17.5        |

NOTE: NR = Newton-Raphson Iteration

In order to maintain the full 24 bits of precision available in single precision floating points, one is forced to use the SQRTPS + DIVPS instructions which take a total of 36 processor cycles to execute. However, if lower levels of precision can be tolerated, a dramatic performance increase may be realised by instead using a RSQRTPS instruction at a cost of only 2-3 processor cycles.

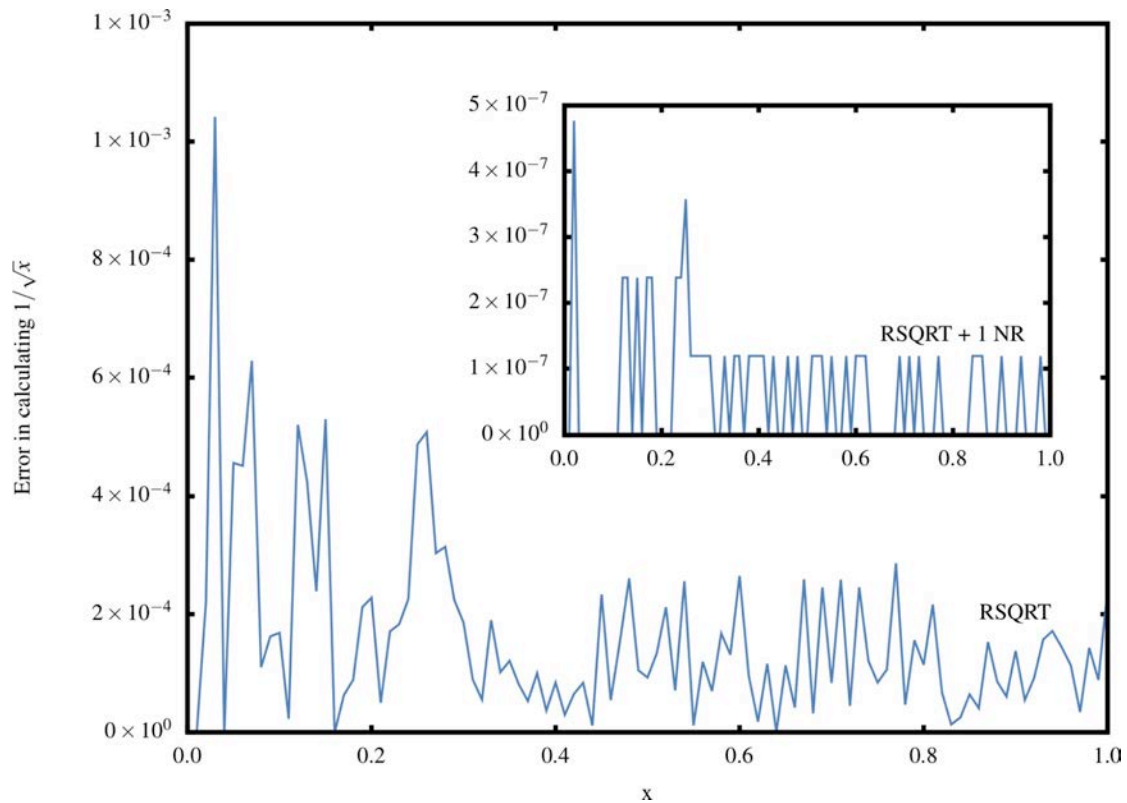
During benchmarking of RayFactor, it was determined that the 11 bits of precision provided by the RSQRTPS instruction was insufficient, resulting in an increase in the number of rays intersecting with the surface of the primitive from which they were launched. While this could to a degree be corrected by increasing the error tolerance in RayFactor, the resulting view factor distribution would be skewed as discussed in Section 4.2.

In order to obtain an optimal balance between the performance of using the RSQRTPS instruction and the precision of using the SQRTPS + DIVPS instructions, the RSQRTPS instruction was used with a single Newton-Raphson iteration which increases the precision to around 22 bits [81]. The Newton-Raphson iteration for the reciprocal square root is simple and may be completed using only four multiplication instructions and one subtraction instruction which can be executed in a single processor cycle each. In this method, the reciprocal square root of  $x$  is initially estimated using RSQRTPS to get  $y_n$ , then a higher precision approximation  $y_{n+1}$  is calculated using equation (80).

$$y_{n+1} = y_n(1.5 - 0.5xy_n^2) \quad (80)$$

The increase in precision is evident when examining the absolute error for the RSQRTPS and the RSQRTPS + 1 NR methods as shown in Figure 14 where the average error is reduced by a factor of approximately 1000 over the range  $0.01 \leq x \leq 1$ .





**Figure 14: Error in the calculation of the reciprocal square root for RSQRT and RSQRT + 1 NR**

Adopting the RSQRTPS + 1 NR method to calculate the reciprocal square root in RayFactor was found to provide sufficient accuracy such that the difference between implementing RSQRTPS + 1 NR and the slower but more precise SQRTPS + DIVPS was virtually undetectable when examining the view factor distribution.

#### 4.6.2 Fast Square Root

In addition to calculating the reciprocal square root, the square root is itself required for numerous calculations throughout RayFactor with the most frequently executed case being from the calculation of the discriminate when testing for ray-primitive intersection.

As covered in Section 4.6.1, the square root of a number may be calculated to the full 24 bits of single floating-point precision using the SQRTPS instruction but at the expense of around 20 processor cycles. Similarly to the calculation of the reciprocal square root, alternative faster, lower precision methods are available. There are numerous alternatives such as the Babylonian method or Taylor

expansion but given the now optimised calculation of the reciprocal square root, the best method is to calculate the square root as shown in equation (81).

$$\sqrt{x} = x \frac{1}{\sqrt{x}} \quad (81)$$

Given the calculation of the square root to 24 bits of precision only requires the SQRTPS instruction, the performance enhancements are not as large as those for the reciprocal square root as demonstrated in Table 9.

**Table 9: Methods for calculating the square root on modern processors [81]**

| Method                 | Precision | SSE<br>Performance | AVX<br>Performance |
|------------------------|-----------|--------------------|--------------------|
| SQRTPS                 | 24 bits   | 1                  | 1                  |
| RSQRTPS + MULPS        | ~11 bits  | 4.7                | 5.9                |
| RSQRTPS + 1 NR + MULPS | ~22 bits  | 2.3                | 4.3                |

NOTE: NR = Newton-Raphson Iteration

The performance increases from using approximations for the square root instead of the SQRTPS instruction may not be as impressive as those for the reciprocal square root, but given that RayFactor is required to perform the reciprocal square root at most once per ray while the square root must be calculated once per ray, per object the overall performance boost for each calculation is comparable. Early benchmarking studies indicated that using RSQRTPS + 1 NR for the calculation of the reciprocal square root and RSQRT + 1 NR + MULPS for the calculation of the square root gave an average RayFactor run-time reduction of approximately 5%

#### 4.6.3 Fast Sine and Cosine

The slowest executing instructions on a modern processor are transcendental functions such as sine and cosine. These functions are performed as scalar operations in the processor's floating-point unit and take up to 100 processor cycles to calculate a single sine or cosine value. This makes the calculation of a sine or cosine 20 times more expensive than calculating the square root to full precision. Furthermore, the fact that these instructions are scalar means that expensive

unpacking and packing of a SIMD vector would need to occur in order to use them in a SIMD codebase.

The sine and cosine functions have the corresponding scalar instructions FSIN and FCOS and are used in RayFactor for the generation of the ray starting point and direction. Profiling of an early version of RayFactor (using XCode Instruments) indicated that the FSIN and FCOS instructions were accounting for 23.6% and 20.9% of RayFactor's total run-time, respectively.

Realising that the majority of the FSIN and FCOS instructions are paired, we may make a preliminary optimisation by using the FSINCOS instruction which can calculate both the sine and cosine of a single input number in approximately 110 cycles.

Although the substitution of the FSIN and FCOS instructions by FSINCOS can reduce the time required to calculate the sine and cosine of a given number, the performance is still not ideal. As this is a widespread problem in the game development and scientific computing communities, several fast approximation methods have been developed over the years such as the Goertzels algorithm, table lookup and polynomial curve fits, of which the most common is a Taylor expansion [82].

Given the relative stagnation in memory access speeds compared to the rapid increase in processor clock speed, the Goertzels algorithm and table lookup methods are not strong candidates for a modern processor and therefore polynomial based approximation methods were pursued in this thesis.

RayFactor uses a polynomial approximation method, adapted from the Cephes math library [83] for the calculation of sine and cosine in which three stages are employed, additive reduction, approximation and reconstruction. In the additive reduction stage, the periodicity of the two functions is used to map all input numbers onto the range  $0 \leq x \leq 2\pi$ . Function symmetry is once again used to further reduce the domain of the approximation down to  $0 \leq x < \pi/4$ . This process of additive reduction is commonly expressed in terms of the double angle formulas shown in equation (82).

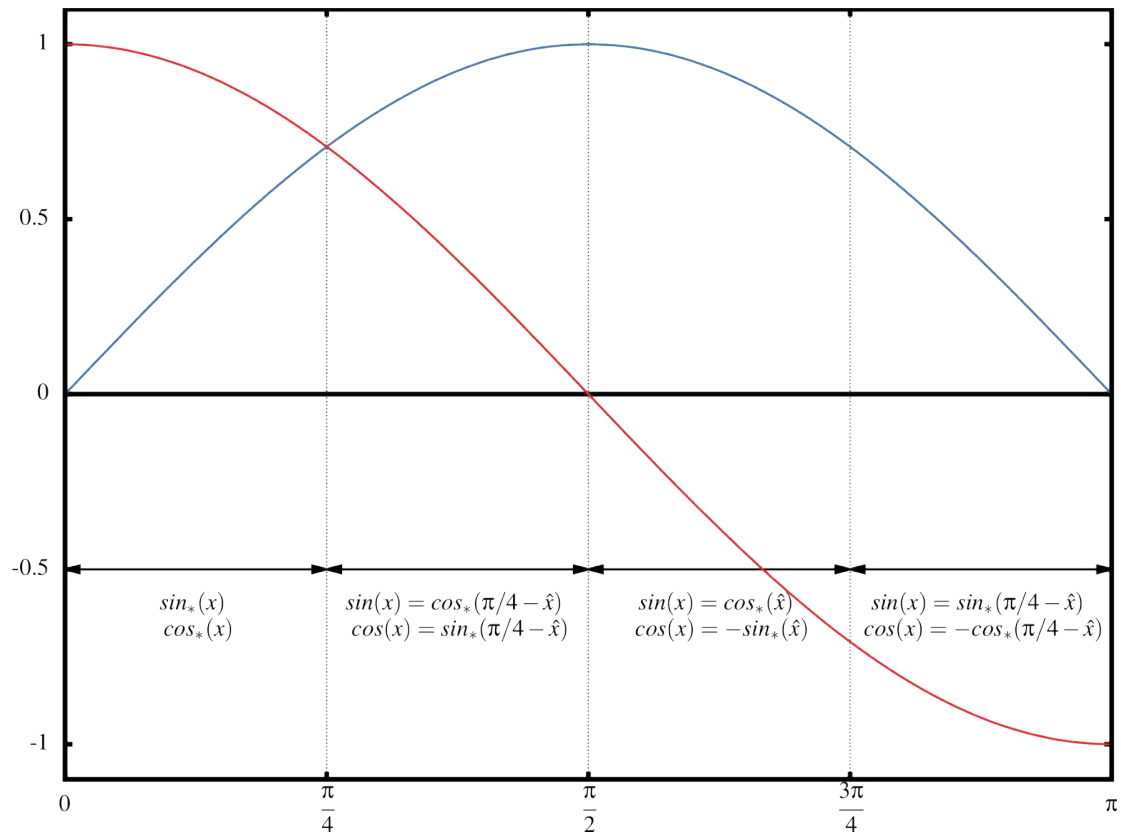
$$\begin{aligned}\sin(\hat{x} + B) &= \sin(\hat{x}) \cos(B) + \cos(\hat{x}) \sin(B) \\ \cos(\hat{x} + B) &= \cos(\hat{x}) \cos(B) - \sin(\hat{x}) \sin(B)\end{aligned}\tag{82}$$

Here the additive angle  $B$  is used to shift the input value  $x$  into the range  $[0: \pi/4]$  and may be calculated as shown in equation (83) where  $k$  is the integer number of times  $x$  may be divided by  $\pi/4$ .

$$\begin{aligned} B &= k\pi/4 \\ \hat{x} &= x - B \end{aligned} \tag{83}$$

Following from equation (82) we may conclude that the full period of the sine and cosine functions may be constructed with the portions of both functions in the range  $0 \leq \hat{x} < \pi/4$  and knowledge of the function's symmetry [84]. This is an important result as it reduces the range over which the functions need to be approximated, which in turn reduces the number of polynomial constants stored and powers evaluated to obtain an accurate fit.

Figure 15 diagrammatically demonstrates how the half-period of the sine and cosine functions may be reconstructed using the reduced range  $[0: \pi/4]$ . The second half-period of the sine and cosine functions may be calculated by mirroring the results from the range  $[0: \pi]$  about the x axis using  $f_{\pi:2\pi}(x) = -f_{0:\pi}(x)$ .



**Figure 15: Construction of sine and cosine from the function defined on the range  $[0: \pi/4]$**

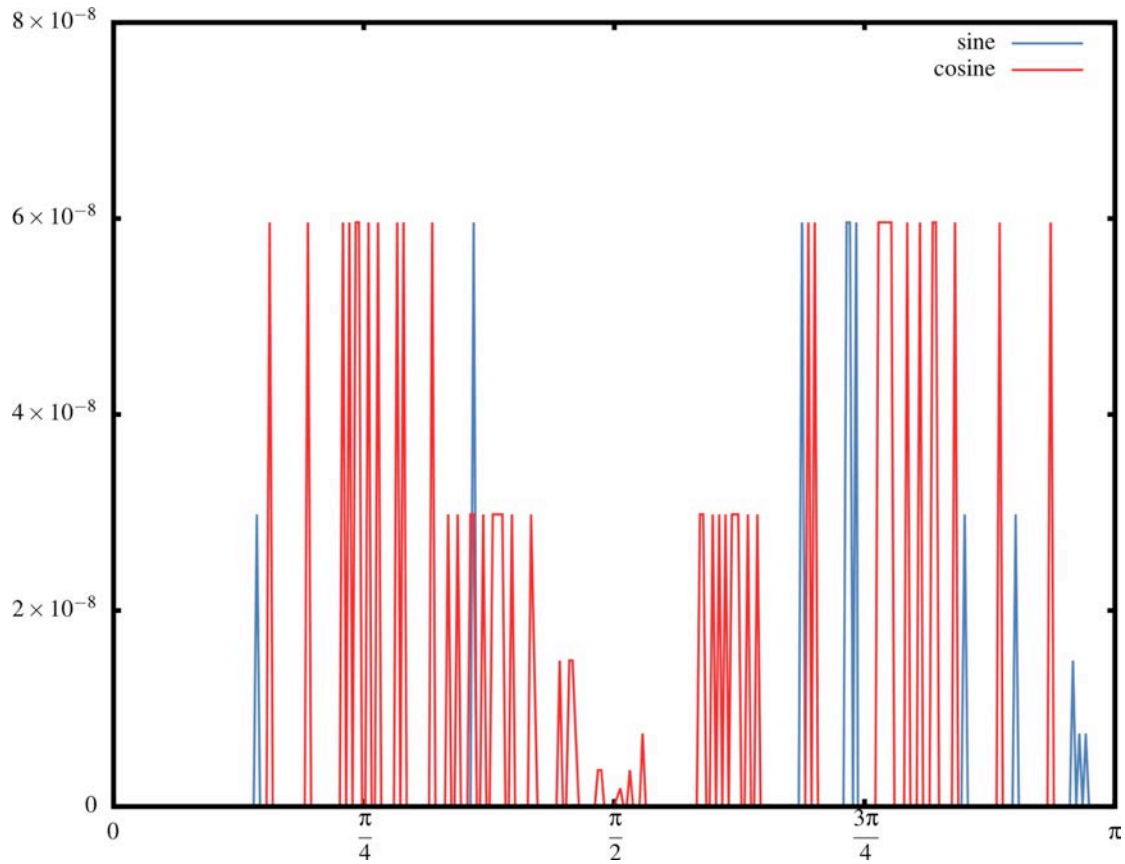
The subsequent step uses a polynomial to approximate the sine and cosine functions over the range of  $0 \leq x < \pi/4$ . Rather than using a Taylor series expansion for the approximating polynomials, a minimax polynomial is used. The coefficients of minimax polynomials are selected (commonly using the Remez exchange algorithm) in such a way that error is uniformly distributed over the function range, whereas the use of a Taylor series expansion will result in a high maximal error due to necessary truncation [85]. Additionally, the polynomial coefficients are optimized for floating point representation and organised using Estrin's algorithm to reduce the number of operations required to evaluate the polynomial [82].

The final step in the fast calculation of sine and cosine is reconstruction. Here the sine and cosine values are produced from the given input value by mapping the polynomial approximations onto the correct region using a series of fast bitwise operations. By using this software based polynomial approximation for the sine and cosine function's, the execution time is greatly reduced compared to the scalar instructions FSIN, FCOS and FSINCOS which are typically carried out on the system hardware.

**Table 10: Comparison of the number of processor cycles to calculate the sine and cosine functions using system hardware and the implemented polynomial approximation**

| Method        | Cycles to Execute |               |
|---------------|-------------------|---------------|
|               | System            | Approximation |
| Sine          | 40-100            | 25            |
| Cosine        | 40-100            | 25            |
| Sine + Cosine | ~110              | 25            |

From Table 10, the performance increase is clear with the calculation of both the sine and cosine functions using the polynomial approximation executing around 440% faster. However, typical of most optimisation efforts, there is a trade-off between performance and accuracy. The absolute error for the calculation of the sine and cosine functions using the polynomial approximation is shown in Figure 16.



**Figure 16: Absolute error for the polynomial approximation of the sine and cosine functions**

Effectively, Figure 16 demonstrates that the sine and cosine approximations have a maximum error of 1 bit resulting in  $\sim 23$  bits of precision. This is in line with the accuracy of the other fast elementary functions implemented in RayFactor and was thus deemed to be acceptable.

#### 4.7 Space-Partitioning Strategies

The time it takes to test for ray-primitive intersections in a MC-RT solution increases linearly with the number of objects being analysed. Although the use of primitive objects (in place of finite elements – see Chapter 6) reduces the number of objects for which ray intersection needs to be tested, for non-trivial geometries considerable run-time reductions could be achieved using a space partitioning strategy.

Space-partitioning strategies are commonly used in physics and graphics engines and divide the simulation space into sub regions, which are placed into a hierarchical tree structure. When a ray is launched, it is tested for intersection

with the root nodes of the space-partitioning tree. If intersection with one of these nodes is determined, ray intersection is then recursively tested with the corresponding ‘child nodes’ until intersection with an element occurs or all the child nodes have been exhausted.

Use of a space-partitioning tree reduces the average run-time complexity for identifying ray-primitive intersection from linear,  $O(N_{objects})$  to logarithmic,  $O(\log N_{objects})$ , where the base of the logarithm will be equal to the number of child nodes used.

A multitude of space-partitioning strategies have been developed including kd-trees [24], Binary Space Partitioning (BSP) trees [86], Octrees [87], R-trees [88] and Bounding Volume Hierarchies (BVH) [27], each with their own advantages and disadvantages.

While predominately studied in the context of computer graphics [29, 89] space-partitioning strategies have also been implemented for the MC-RT simulation of radiative heat transfer. Zeeb et al [90] incorporated a Uniform Spatial Subdivision (USD) with mail-boxing strategy to analyse radiative heat transfer within large arbitrary geometries with non-participating media. Here the geometries analysed contained between 1000 to 5000 surfaces, which were sorted into 4,000-65,000 volumetric nodes (voxels) resulting in computational speedups of 17.7 to 81.4 times.

In more recent studies the use of the BSP strategy in a MC-RT simulation of surface-surface radiative exchange was found to provide speed-ups of up to 51.5 times for a geometry containing over 50,000 elements while speed-ups of 4.6 times were found for geometries containing only 600 elements [91].

While the literature reports impressive speed-ups for the use of space-partitioning strategies, recent changes in computer hardware architecture introduce new challenges to the implementation of these strategies. In order to store the partitioning hierarchy, additional memory is required and with full utilisation of processing resources in a vectorised environment the availability of fast on-chip memory to store these structures is scarce.

Furthermore, most implementations of space-partitioning strategies neglect SIMD hardware. As previously discussed, for full exploitation of the available technology calculations are no longer conducted on single rays but rather on ray packets. For AVX enabled processors, this means that packets containing 8 rays can be treated together. These rays are highly unlikely to be coherent and

therefore multiple traversals of the partitioning hierarchy would be required for a single ray packet. Further to this problem are the constraints involved when incorporating control logic for tree traversal as branching of instructions is not strictly permitted within SIMD.

While efforts to find efficient, SIMD friendly partitioning structures are being undertaken in the field of computer graphics [92-94] research is mainly focused around coherent ray packets making this by no means a solved problem for the simulation of radiative heat transfer.

Due to the research state of this problem, space issues and the relatively small number of elements used to construct the geometries studied in this thesis (typically less than 500), a space partitioning strategy was not implemented in RayFactor. However, this is an interesting area for further research as increasing hardware-enabled vectorisation provides a moving target for the development of efficient SIMD space-partitioning strategies for incoherent ray packets.

#### **4.8 Conclusions**

Numerous optimisation strategies were explored and implemented to enhance the performance of MC-RT on modern CPUs. Due to the embarrassingly parallel nature of MC-RT, optimisations focusing on vectorisation were found to provide the most impressive performance improvements. The following chapter will further explore vectorisation of MC-RT using general purpose Graphics Processing Units (GPGPUs) whose architecture provide vectorisation opportunities far exceeding that of current CPUs.



## General Purpose Computing with OpenCL

---

Of late, impressive performance increases on CPUs have been afforded largely due to their increasingly vectorised architecture. As discussed in Section 4.4, the product of thermal design constraints and increasing transistor budgets has resulted in increasing CPU performance through vectorisation. Although multi-core CPUs are now commonplace, they are still highly optimised for serial processing (and thus capable of performing branching operations efficiently) and contain a relatively few number of complex processing cores.

However, CPUs are no longer the only accessible hardware option for performing general numerical calculations. Since the development of the Brook streaming programming language [95], and its evolution into NVidia's Compute Unified Device Architecture (CUDA) (first released in 2006), Graphics Processing Units (GPUs) have been capable of natively running numerical simulations. GPUs have a fundamentally different design to CPUs and consist of hundreds of small, 'simple' cores, specifically designed for parallel performance, making them an ideal target for embarrassingly parallel algorithms such as MC-RT.

This chapter will introduce a new version of RayFactor, called RayFactorCL, which can access the computational power across multiple computational devices of varying architecture, heterogeneously using the Open Compute Language (OpenCL).

### 5.1 Open Compute Language

OpenCL is a framework for developing and running parallel programs heterogeneously<sup>6</sup>. This framework is a very recent development in high performance computing (HPC) with the first specification being presented in late 2008 [96], and the first implementation being available in Apple Inc's operating

---

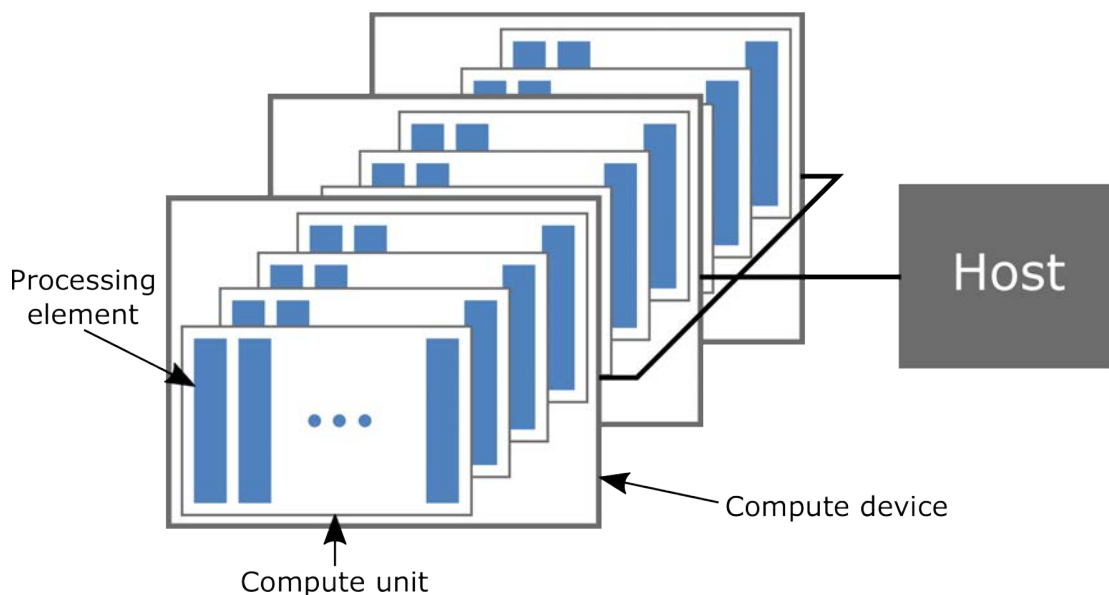
<sup>6</sup> Heterogeneous computing is the act of running a program across multiple devices of different architecture (i.e. CPUs and GPUs). Alternatively, homogenous computing is the act of running a program on a single device architecture.

system OS X 10.6 in late 2009. Since its introduction, OpenCL has received widespread adoption with hardware vendors such as Intel, Advanced Micro Devices (AMD), NVidia and ARM Holdings all providing OpenCL support in their computing devices.

Programs created using OpenCL are both portable and scalable. When running an OpenCL program, the framework detects the available hardware and performs Just In Time (JIT) compilation for each OpenCL device. This means that an OpenCL program can be written once and effectively run on any device supporting OpenCL. However, in order to achieve such portability, a level of hardware abstraction is introduced by the OpenCL specification. This abstraction is implemented through OpenCL's platform, execution and memory models, which are introduced in the following sections.

### 5.1.1 The OpenCL Platform Model

The high level abstraction of a heterogeneous computing system is known as the 'platform model' in OpenCL. This model, presented in Figure 17, states that an OpenCL system consists of a single host responsible for interfacing an OpenCL program with the external environment (e.g. coordinating data transmission) and one or more OpenCL devices on which an OpenCL program (referred to as a 'kernel') is executed.



**Figure 17: The OpenCL Platform model**

An OpenCL enabled device (referred to as a 'compute device') consists of one or more compute units, which may in turn contain one or more processing

elements, the modules responsible for performing the actual computations of a kernel. On the now familiar CPU (i.e. the compute device), each core is a compute unit containing a single processing element. While the distinction between compute units and processing elements may seem unnecessary when considering CPUs, one must remember that OpenCL is device agnostic and therefore the platform model must be capable of describing all computing devices, and with devices such as GPUs (discussed in detail in Section 5.3) this distinction is necessary.

### **5.1.2 The OpenCL Execution Model**

An OpenCL application consists of two parts, (i) the host program, which executes on the host, and (ii) one or more kernels, which execute on the compute devices. The host program is responsible for the initialisation of all compute devices and using the OpenCL API acts as a gateway, transferring data between the kernel and the external program environment (e.g. standard C++ data pre/post-processing functions).

An OpenCL kernel is a function that is executed on a compute device. The ‘real’ work of OpenCL applications is completed in the kernels with the host program simply transferring data (parameters) required by the kernel to the compute devices before the kernel itself is placed in the compute device’s command queue to await execution. When the compute device prepares to execute a kernel, a collection of basic OpenCL work units called work-items (similar to threads on the CPU), are created and organised into independent work-groups. Decomposing the computational domain in this manner provides fine-grained data parallelism, nested within coarse-grained data and task parallelism [97].

The work-items within a single work-group concurrently execute kernel instructions on the processing elements of a single compute unit, while work-groups may execute on different compute units in any order, in parallel or in series. This is the heart of concurrency in OpenCL, and it follows that work-groups cannot be assumed to execute in any particular order or concurrently, and that only work-items within the same work-group can be assumed to share the same compute unit resources. This has ramifications in the sharing of data between work-items as described by the OpenCL memory model.

### **5.1.3 The OpenCL Memory Model**

The OpenCL memory model defines five distinct memory regions: host, global, constant, local and private, which may be defined as follows:

1. **Host memory:** This region is only visible to the host and has no particular influence on the execution of a kernel.
2. **Global memory:** This region is accessible to all work-items of all work-groups. It is the only medium by which data can be shared across work-groups and between the host and a compute device. Global memory may be viewed as the 'bulk' memory of a compute device (similar to RAM for a CPU) as it is typically the largest memory region, however, it provides the slowest read/write access of all the memory regions on a compute device.
3. **Constant memory:** This is a sub-region of global memory that may not be modified during kernel execution (i.e. is read-only). Constant memory must be initialised on the host and transferred to the compute device prior to kernel execution. The read-only nature of this region permits the compute device to optimise for data reads, making it faster than global memory for read-only access and, therefore, the preferable location to store read-only data that will be accessed by all work-items.
4. **Local memory:** This region is local to, and shared by, each work-group. Although, work-items belonging to the same work-group share the work-groups local memory, work-items cannot access the local memory of work-groups to which they do not belong. Depending on the hardware, local memory generally has much lower latency than global memory and is, therefore, typically used as a manual cache for global memory.
5. **Private memory:** This is the most restricted region of memory being private to individual work-items. Work-items may not access the private memory of other work-items regardless of which work-group they belong to. Private memory is the lowest latency memory and is generally located in close physical proximity to the processing core. If a work-item exceeds the amount of private memory physically available, private memory will spill into the higher latency local memory.

Although OpenCL specifies multiple, distinct memory regions, it is at the compute device's discretion as to how each region maps to the physical memory modules of the device. For example, most CPUs map all memory regions to the same physical memory module, a computer's Dynamic Random-Access Memory (DRAM), while a GPU maps each memory region to a physically distinct memory module. The presence of multiple memory regions, and the ability for work-items to concurrently access the memory element, introduces difficulties in maintaining memory consistency. However, in OpenCL memory consistency may be maintained using atomic operations and synchronization points called 'barriers'.

#### 5.1.4 Atomic Operations and OpenCL Barriers

Given that work-items may concurrently write to the same memory element, memory consistency cannot be guaranteed. For example, when calculating a radiative view factor, each work-item will fire a ray from object  $i$  and check to see if it intersects object  $j$ . If two work-items, A and B, concurrently determine that their rays intersect object  $j$ , they will simultaneously proceed to increment the variable holding the number of  $i$ - $j$  intersections. In order to increment a variable, each work-item must first read the variable, add one to it and write this variable back to memory. As this variable is concurrently being read by work-items A and B, both work-items will write the same number to memory resulting in only one of the two intersections being recorded. In order to prevent this loss of information, it must be enforced that only one work-item can increment the variable at a time, that is serially. This may be enforced using atomic operations, however their use should be minimised as they create an execution bottleneck as each work-item waits to gain exclusive access to the variable.

The higher the number of work-items capable of accessing the memory region where data resides, the higher the performance penalty of an atomic operation. If the variable resides in global memory, then in the worst case execution of all work-items becomes serialised as they could all potentially require exclusive access. Alternatively, if the variable resides in local memory, then in the worst case execution is only serialised for work-items within a work-group. In the example of recording the  $i$ - $j$  intersections, the penalties of the atomic operations can be reduced by each work-group maintaining its own intersection counters in local memory, and at a suitable point (e.g. on conclusion of firing rays from the current object), work-groups combining their individual results into global memory. This way, serialisation is limited to the work-group level for each ray and global serialisation will only occur once per object (rather than once per ray). However, this approach introduces a new problem, synchronisation.

Unconstrained, when at least one work-item in a work-group has finished launching its rays from object  $i$ , it will start the task of adding the work-group's local intersection counters to the global counters. However, at this point it is not guaranteed that all work-items have finished recording their intersection data and, therefore, the local counters will be added to the global counters before all intersections have been recorded. This again results in potential information loss and must be prevented using a synchronisation point called a barrier. Barriers may be placed at the local or global level and require all work-items to pass through the barrier before subsequent instructions can be executed. Barriers do not create serialisation points, but they do require execution time while not

strictly performing any additional work. In the example above, by placing a local barrier before local intersection counters are added to the global counters, one can be certain that all work-items have finished processing the current object and therefore no information will be lost.

As demonstrated by the preceding sections, the development of an OpenCL program requires the adoption of a new (and more complicated) programming paradigm. However, by embracing the OpenCL platform, execution and memory models, one may not only produce a highly scalable and portable application, but also one that is capable of heterogeneous computing, able to execute simultaneously on multiple compute devices. In essence, this is an extension to the concept of vectorisation, whereby now the program is not only vectorised across the processing elements of a single compute device but over all capable compute devices installed on a given platform.

Although OpenCL provides abstraction from the underlying hardware, it is not completely opaque, and an understanding of the compute device's architecture is required to achieve optimum performing kernels. Much of the architecture of CPUs has previously been discussed and its mapping to the OpenCL memory models is quite straightforward with typically all memory regions being located in the same physical memory (i.e. DRAM). However, for compute devices such as GPUs this is not case, and careful attention must be paid to the memory regions used. In order to write optimal OpenCL kernels, one must first gain an appreciation of GPU architecture and general-purpose computing on a GPU.

## **5.2 Test System GPUs**

Two different GPUs were installed in the Mac Pro 4.1 test system (see Section 3.2.1), the NVidia GTX 580 and GTX 680. These GPUs were selected as they represent the highest performance consumer GPUs available from each of the last two generations from NVidia (before one moves into dual GPU designs and research units).

The specifications of each GPU, relative to the CPUs installed in the test system, are presented in Table 11. Here, an effort has been made to express each specification in terms of OpenCL nomenclature. Whilst this is straightforward for the GPU, there are some peculiarities with the CPU, specifically in the area of the private and local memory regions. As previously mentioned, the CPU maps all memory regions to the DRAM. However, the CPU has several high bandwidth, low latency memory modules local to its processing elements called the L1, L2 and L3 caches. These modules are similar to the private and local memory

regions on the GPU in that they have drastically higher performance than global memory (DRAM). However, unlike private and local memory they are managed by CPU control units rather than the OpenCL kernel and could contain data from either the private, local or global memory regions. For the purposes of Table 11, L1 cache was taken as representative of the private memory and L2 cache as the local memory. In addition to this, each CPU in the test machine has 8 MB of L3 cache, which like the L1 and L2 caches could be utilised for the storage of any memory region by the CPU.

**Table 11: Comparison of OpenCL specifications for the Intel Xeon E5520 CPU and the NVidia GTX 580 and GTX 680 GPUs.**

| Specification                | Units  | Intel Xeon E5520   | NVidia GTX 580  | NVidia GTX 680     |
|------------------------------|--------|--------------------|-----------------|--------------------|
| Launch date                  | -      | Q1 2009            | Q4 2010         | Q1 2012            |
| Transistors                  | -      | $7.31 \times 10^8$ | $3 \times 10^9$ | $3.54 \times 10^9$ |
| Compute units                | -      | 4                  | 16              | 8                  |
| Processing elements per unit | -      | 1                  | 32              | 192                |
| Total processing elements    | -      | 4                  | 512             | 1536               |
| Clock speed                  | (MHz)  | 2260               | 1544            | 1006               |
| Thermal design power         | (W)    | 80                 | 244             | 195                |
| Memory bandwidth             | (GB/s) | 25.6               | 192.4           | 192.2              |
| Local Memory per unit        | KB     | $256^1$            | 48              | 48                 |
| Private Memory per unit      | KB     | $128^1$            | 128             | 256                |
| Maximum global memory        | GB     | 144                | $1.5^2$         | $2^2$              |

**Notes:** 1. Intel CPUs do not explicitly have local and private memory regions, instead the L1 and L2 caches have been presented, while an additional 8 MB of L3 Cache is available.  
2. This is the amount of memory in the NVidia reference design. Manufacturer version's memory may exceed this.

Further discussion involving general computing on the GPU will be specific to the Fermi and Kepler architectures of the NVidia GTX 580 and GTX680, respectively. However, it should be noted that these architectures are not representative of all GPUs such as those manufactured by Advanced Micro Devices (AMD), and therefore statements made regarding GPU based computing may not be directly applicable for all GPU devices.

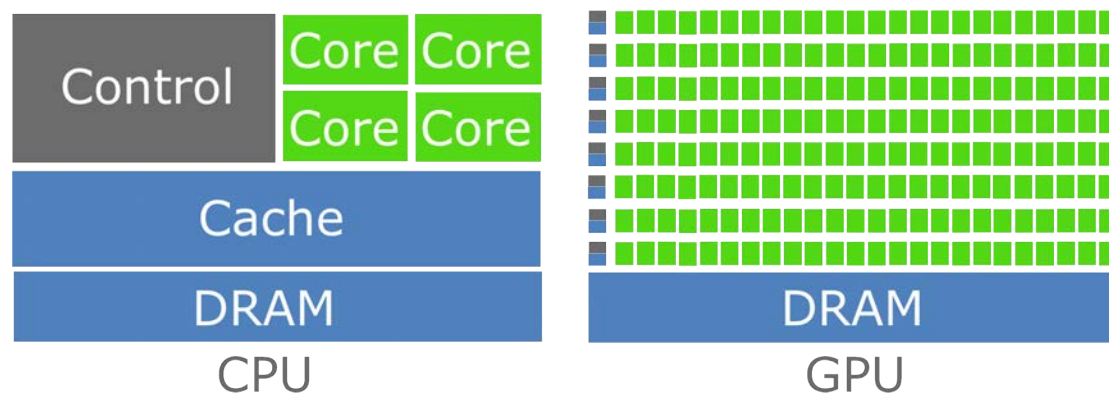
### 5.3 General-Purpose Computation on the GPU

Due to their highly parallel nature, Graphic Processing Units (GPUs) are an ideal platform to run 'embarrassingly parallel' algorithms such as MC-RT. In the early

days of GPU computing, one was required to map numerical problems onto the paradigms used to render graphics to computer screens. However, the development of general-purpose GPU (GPGPU) programming languages, such as CUDA and more recently OpenCL, has made the GPU accessible for traditional numerical calculations.

The GPU is, however, fundamentally different to the CPU. While the CPU has historically been optimised for serial processing of general-purpose tasks, the GPU has been developed for massively parallel processing of specific tasks (i.e. simultaneously colouring the pixels on your computer screen). While the CPU and GPU are slowly converging in their designs (i.e. CPUs are becoming increasingly parallel while GPUs are becoming increasingly general), there is still a marked difference in their architecture and operational characteristics.

The differences in architecture arise from how ‘transistor budgets’ are spent for each device. In order to perform optimally for general tasks, the CPU spends a relatively large portion of its transistor budget on control units and memory cache while GPU designs opt to spend the majority of their budget on processing elements (i.e. processor cores) as demonstrated in Figure 18.



**Figure 18: Transistor distribution of a typical CPU and GPU.**

The reduction in size and complexity of the GPU’s control units, processing elements and memory cache in exchange for an increased number of processing elements, introduces new design considerations for programs targeting the GPU, such as a reduction in cache control and branching performance (due to a lack of control units) and run-time penalties in accessing memory (due to reduced memory caches). These considerations will be discussed in subsequent sections in relation to NVidia GPUs employing the Fermi and Kepler architectures.



### 5.3.1 GPU Work-item Execution Control

On the CPU, each processing element has a sophisticated control unit allowing each work-item to execute its instructions independently of adjacent work-items. On the GPU, however, due to the reduced number and complexity of the control units relative to the number of processing elements, independent control of each work-item cannot be provided, and at the hardware level work-items are placed into groups called ‘warps’ that on current NVidia hardware each contain 32 work-items and share a common control unit called a warp scheduler. This means that work-items within a warp must execute in lock-step with one another, and therefore if a kernel contains a conditional branch and a single work-item must follow this branch, then all other work-items must wait for that work-item to finish executing the branch before the warp may continue executing the main sequence of instructions. This behaviour is similar to that encountered with SIMD instructions on the CPU (discussed in Section 4.4.2). However, for SIMD this behaviour is limited to single operations whereas for a warp it extends to the whole sequence of instructions which make up a kernel.

Another side effect with small, less complex control units is the loss of cache management by the control unit. This has ramifications in the OpenCL memory model, which will be explored in the following section.

### 5.3.2 Mapping the OpenCL Memory Model to the GPU

As described in Section 5.1.3, OpenCL specifies multiple memory regions. On the CPU, these regions all map to the same physical memory with the CPU’s control units managing the movement of data from the slow DRAM to high performance on-chip memory caches as required. Due to the simpler design of a GPU control unit, ensuring that the data required by the work-items is stored in the ‘best’ available memory becomes a responsibility of the kernel rather than the underlying hardware, and may be conducted using the OpenCL memory model. Unlike CPUs, the OpenCL memory region maps to physically distinct memory modules on the GPU, which are summarised in Table 12.

The performance of each memory region may be characterised using its latency (i.e. the time delay between when a control unit asks for a piece of data and the moment it becomes available at the memory module’s output pins) and bandwidth (i.e. the rate at which data can be transferred into or out of a memory module). The higher the performance of the memory region, the more expensive (and therefore the less abundant) it is. For optimum performance, one must

ensure that data is not only placed in a memory region of high performance but also a region in which adequate memory is indeed available<sup>7</sup>.

**Table 12: Physical memory of an NVidia GTX680 with representative performance figures.**

| Memory                     | Size<br>(KB)        | Bandwidth<br>(GB/s) | Latency<br>(cycles) | Managed<br>by |
|----------------------------|---------------------|---------------------|---------------------|---------------|
| Private memory (registers) | 4 <sup>*</sup>      | > 2000              | 1                   | Compiler      |
| Local memory               | 48                  | 1500-2000           | 20-30               | Program       |
| Constant memory            | 48                  | 150                 | 400                 | Program       |
| L2 cache                   | 512                 | 480                 | 100-200             | GPU           |
| Global Memory              | > 2x10 <sup>6</sup> | 192.3               | 400-800             | Program       |

\* Register memory on the Kepler architecture is 256KB per compute unit which is shared amongst all work-items on the compute unit.

In addition to the restrictions on the amount of physical memory available, the private and local memory regions are not allocated on a continuous basis (as one encounters on the CPU), but are instead allocated in ‘blocks’ at the warp level, where the size of a block is specified by the GPU’s hardware. This is called ‘memory granularity’ and means that memory allocated in these regions, for the work-items within a warp will be rounded up to ensure that the size of the allocated memory is a multiple of the GPUs private or local memory allocation block size. For example, if a work-item requires 40 bytes (i.e. ten single-precision floating points) of private memory, theoretically 1,280 bytes would need to be allocated per warp. However, on an NVidia GTX 680 with a private memory allocation unit size of 1024 Bytes, a total of 2048 Bytes would be consumed per warp.

The intricacy of GPU memory interactions is not limited to the allocation of memory in blocks when dealing with private memory. Additionally, due to the reduction in the number of control units (discussed in Section 5.3.1), each warp scheduler concurrently allocates private memory for multiple warps. The number of warps that may be allocated at once is called the ‘warp allocation granularity’ and is dependant on the design of the warp scheduler. For the NVidia GTX 680, the warp allocation granularity is equal to four warps, and therefore private memory is allocated for four warps at a time. This means that if

<sup>7</sup> If a particular memory region runs out of space, it ‘spills’ over into the next region in the memory hierarchy (i.e. private memory would spill to local memory). This results in poor performance, as both memory regions need to be accessed when retrieving data.

the actual number of warps in a work-group is not a multiple of four, then the number of warps will be rounded up to the nearest multiple and private memory allocated on this basis. If the work-group in the previous example contained 160 work-items, there would be 5 warps per work-group. Given the GTX 680 has a warp allocation granularity of 4, private memory would actually be allocated for 8 warps (rounding up to the nearest multiple of 4) resulting in a total private memory allocation of 16,384 Bytes for the work-group compared to the 6,400 Bytes that are actually required prior to accounting for any memory and warp allocation granularity. As demonstrated by this example, memory allocation on the GPU can result in higher than expected memory consumption due to the relatively high granularity of operations within the GPU.

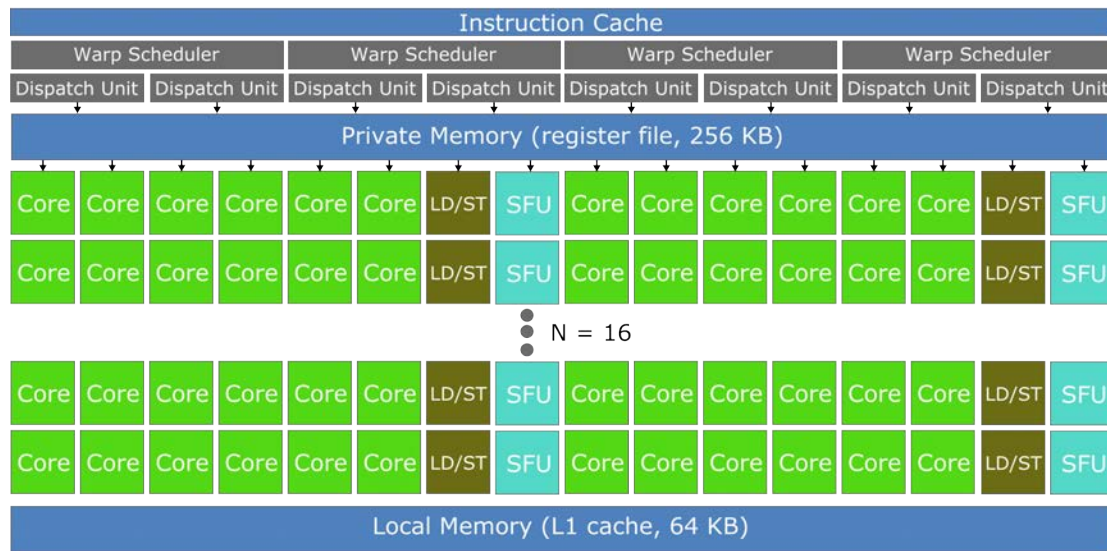
Relative to the CPU, the structure and allocation of memory on the GPU is quite complex, requiring careful examination of memory usage patterns to access the full computational potential of the GPU. However, memory usage is not the only area in which care is required, the application of the OpenCL execution model to the GPU introduces further optimisation constraints.

### **5.3.3 Mapping the OpenCL Execution Model to the GPU**

As discussed in Section 5.1.2 the OpenCL execution model partitions numerical problems into work-groups that each contain one or more work-items. In addition to this partitioning, outside of OpenCL, the GPU hardware allocates work-groups into sub-groups called warps, which contain up to 32 work-items each. In order to fully utilise the GPU hardware, each warp must be fully occupied, as the GPU will commit the same amount of computing resources to process a warp regardless of how many work-items populate it. However, simply ensuring that the work-group size is a multiple of the warp size is not enough to obtain maximum performance on the GPU, one must also account for operational peculiarities that are introduced by the GPU's hardware architecture as shown in Figure 19.

In a similar manner to hyper-threading on the CPU, a warp scheduler will reallocate processing resources when execution of a work-item (more concisely a warp) stalls. When a work-item stalls on the CPU, the current state of memory is transferred from the fast L1 cache of the processing element to slower DRAM, an operation that may take many processor cycles to complete. However, for a GPU compute unit, which can maintain a greater number of work-items, and is far more likely to need to reallocate processing resources due to higher instruction and memory latencies, this approach is far from ideal. Instead, the state (which resides in the private and local memory regions shown in Figure

19) of all work-items in the active work-groups is initialised at the start of work-group execution and maintained until a work-group has been fully processed. This allows the warp schedulers to quickly reallocate processing elements from stalled warps to those ready to execute with nearly no time penalty. However, this strategy introduces an additional constraint on the work-group size. As shown in Figure 19, both the private and local memory is common to all processing elements and, by extension to all work-items in the active work-groups. Therefore, selection of the work-group size must also take into account the amount of private and local memory that will be required by each work-item and ensure that there is sufficient physical memory available in these regions to concurrently store the state for all work-items within a work-group.



**Figure 19: GPGPU relevant components of NVidia GTX 680 compute unit.**

**NOTES:** LD/ST represents the load store unit. SFU represents the special function unit which execute transcendental instructions such as sine, cosine and square root. N = 16 indicates that a total of 16 rows (each row has 12 cores) make up the compute unit.

If an unsuitably large work-group size is selected, the private and/or local memory may ‘spill’ into the more abundant (but slower) global memory, which will increase the simulation run-time. In the case of RayFactorCL, for example, setting the work-group size to 1024 work-items results in a 280% increase in run-time compared to using a smaller work-group size of 128, which allows the private memory of all work-items in a work-group to coexist in the compute unit register file. This limitation on work-group size is characterised through the calculation of the GPU ‘occupancy’, a metric measuring the ratio of work-groups

that can be active on a compute unit ( $N_{WG,active}$ ) to the maximum number of active work-groups a compute unit can manage ( $N_{WG,max}$ ) :

$$Occupancy = \frac{N_{WG,active}}{N_{WG,max}} \quad (84)$$

Here, the number and capacity of the warp schedulers dictate the maximum number of work-groups that may be active at any one time  $N_{WG,max}$ . The number of active work-groups  $N_{WG,active}$ , however, is slightly more complex and must be calculated as a function of the maximum number of active work-groups on the compute unit  $N_{WG,max}$ , the warp size  $N_{warp}$  (where for current NVidia GPUs,  $N_{warp} = 32$ ), the number of work-items per work-group  $N_{WIG}$ , the size of the private and local memory regions on the device,  $D_{private}$  and  $D_{local}$  respectively, and the amount of private and local memory required by each work-group,  $D_{WG,private}$  and  $D_{WG,local}$  as shown below :

$$N_{WG,active} = MIN \left( \begin{array}{c} N_{WG,max} \\ N_{WG,max} \times N_{warp} / N_{WIG} \\ D_{local} / D_{WG,local} \\ D_{private} / D_{WG,private} \end{array} \right) \quad (85)$$

It is important to note that when calculating  $N_{WG,active}$ , the memory required for each work-group in the private and local memory regions must reflect the memory and warp granularity of the GPU hardware, as discussed in Section 5.3.2.

As evidenced by the material presented thus far, the selection of an appropriate work-group is of crucial importance to ensure maximum utilisation of the GPUs available computational power. This is not a straightforward task, and requires optimisation of the kernels to ensure they are not prematurely limited by the latency, memory bandwidth, or instruction bandwidth of the GPU's hardware. Although the selection of an appropriate work-group size greatly impacts the performance of an OpenCL program, it is not the most stringent constraint on GPU utilisation. For this consideration, one must examine the processing of single and double precision floating points on the GPU.

### 5.3.4 Single versus Double-Precision Floating Points on the GPU

The advantages and disadvantages of using either single (32 bit), or double (64 bit) precision floating point representation in regards to the CPU were discussed in Section 4.1. There it was stated that the computational throughput was similar for general instructions with both precision levels, however, the maximum computational throughput using double-precision floating points was reduced by 50% relative to single-precision floating points when using SIMD instructions.

On the GPU, however, the use of double-precision floating points is greatly penalised due to the simplicity of the GPU processing elements. On a CPU, each processing element has parallel execution trains for 32 bit and 64 bit operations, resulting in no throughput penalty for general instructions. However, typical GPU processing elements contain only 32 bit execution trains and therefore, for 64 bit operations, dedicated 64 bit processing elements must be used. The number of these 64 bit processing elements is small compared to the number of 32 bit elements and, therefore, the computational throughput is drastically reduced when using double-precision floating points.

For the Fermi architecture of the GTX 580, the throughput of 64 bit operations is  $1/8^{\text{th}}$  that of 32 bit, while on the more recent Kepler architecture of the GTX 680 the throughput for 64 bit operations is  $1/16^{\text{th}}$  that of 32 bit. Therefore, while the selection of single-precision floating points on the CPU could be debated, for the GPU they are the only sensible option and for this reason single-precision floating point representation was used for RayFactorCL.

## 5.4 Overview of RayFactorCL

The potential of heterogeneous computing and the ability to utilise the vast numerical processing capabilities of the GPU in a portable and scalable manner led to the development of RayFactorCL. Using the algorithms presented in the preceding chapters as a foundation, RayFactorCL was designed to fully utilise all available numerical computing hardware on a platform through the OpenCL API. In conforming to the requirements of the OpenCL specification, RayFactorCL was split into two segments, the host program and the MC-RT kernel.

### 5.4.1 Host Program

As discussed previously, the host program for an OpenCL application is primarily responsible for communicating and allocating workloads to compute devices and the management of data transfer between these devices and the host program environment. However, before any computational work can be conducted, the

host program must ‘survey’ a platform and establish the available computing resources.

### ***Connection to OpenCL Devices***

On initialisation of RayFactorCL, the host program surveys the platform for OpenCL enabled devices and generates a list of all available devices. This list is then processed to remove any devices that do not meet minimum requirements, which for RayFactorCL (due to a dependency on atomic operations) is OpenCL Version 1.1. The kernel is then compiled for each of the remaining devices, which are subsequently sorted using the following performance metric:

$$P_{device} = N_{PE,total} \times f_{PE} \quad (86)$$

By multiplying the total number of processing elements in the device  $N_{PE,total}$ , and the clock speed of the processing elements  $f_{PE}$ , the total number of cycles a device can perform per second can be obtained. While this is not a comprehensive measure of device performance, it is a simple metric to calculate, and the calculation inputs  $N_{PE,total}$  and  $f_{PE}$ , may be obtained solely from OpenCL API device queries with no prior knowledge of the device.

At this point in initialisation, if RayFactorCL is operating in homogenous (i.e. single device) mode, the device with the highest performance is selected and used for computation. However, if RayFactorCL is operating in heterogeneous (i.e. multiple device) mode, a relative performance rating for each device is calculated as the ratio of the device performance to that of the device with the lowest performance. This rating is later used to determine how work is partitioned between devices, although before work is allocated the host must initialise and transmit all required data.

### ***Data Initialisation***

Once the suitable OpenCL devices have been selected and rated, the host creates a three-dimension array containing a pseudo view factor matrix for each device that subsequent computation will be conducted on. This array is referred to as a pseudo view factor matrix as its entries will contain integer intersection counters rather than the actual view factors. Blocks of this array are then transferred to each device along with the primitive object data (which due to the slow transfer speeds can take a significant amount of time). For example, the

transfer of a simple 2x2 pseudo view factor matrix and the data for two primitives takes 10% of the time required to actually perform the simulation for moderate ray densities.

Once all data required by the kernel has been transferred from the host to the devices, the kernels are ready to be launched and begin execution.

### ***Kernel Launching***

In order for the kernels to actual execute, the host program must issue an instruction to the command queue of each device, informing the device of the kernel name, the work-group size to use, and the total number of work-items that should execute.

Two approaches were examined for launching the MC-RT kernel. The first instructed a device to run the kernel once, with the kernel looping through and processing all objects in the geometry. The second was to repeatedly instruct the device to run a simpler kernel, once for each object in the geometry with the kernel only processing a single primitive at a time. Each of these approaches had advantages and disadvantages, however the latter approach was ultimately favoured (as discussed in Section 5.4.2), as it had higher performance and provided greater control over workload distribution.

When multiple compute devices are detected, the host program partitions the geometry to each device based on the device's calculated relative performance rating. For example, if two devices were detected and their relative performance ratings were 4 and 1 respectively, and the geometry contained 8 objects, the host would instruct device A to execute the kernel four times to process each of objects 1-4, then instruct device B to execute the kernel once for object 5, then finally instruct device A to further execute the kernel three more times for objects 6-8. While relatively effective, this is a very elementary work partitioning scheme and does not account for the differences in workload per object or any dynamic variations in device performance, opening this aspect of RayFactorCL as a potential area of future research.

Following the execution of the kernel on all devices selected by the host program, the pseudo view factor matrices are retrieved by the host program from the device in preparation for post-processing.



## ***Host Post-Processing***

As multiple devices can execute the MC-RT kernel, each work-item is unaware of how many rays are fired from each object and it, therefore, becomes the task of the host program to produce the view factor matrix. In order to complete this task, the host performs a reduction step in which the corresponding entries of the pseudo view factor matrices from each device are combined and then each row of the resulting matrix is divided by the total number of rays launched from the object corresponding to that row. Once this is complete, the resulting view factor matrix is output to file signifying the completion of the simulation.

### **5.4.2 The RayFactorCL Kernel**

While the host program ensures that compute devices have work to process, it is the kernel that provides the instructions for how to actually complete the work. The MC-RT kernel in RayFactorCL performs the calculations described in Chapter 2, namely the generation of pseudo-random numbers, construction of a random ray and intersection calculations between the ray and all primitives in the geometry. Specific design considerations for the design of the MC-RT kernel will be discussed in the subsequent sections.

## ***Handling of Object Data***

As the kernel does not need to modify the primitive objects, they are stored and accessed directly from the constant memory region of the GPU. This permits hardware-optimised broadcast of primitive data (such as transformation matrices) to the work-items during execution. Early designs of the kernel also performed asynchronous caching of each primitive object in local memory, in an attempt to leverage the lower memory latency. However, this functionality was later removed as it increased run-times by approximately 7% for local memory caching of just the object being processed, and approximately 15% for local memory caching of both the object being processed and the object being tested for intersection, due to additional data handling and work-group synchronisation requirements.

While storage of the primitives in constant memory provides optimal access speeds, it should be noted that although it is not as scarce as private or local memory it still has rather limited capacity. With each primitive requiring 112 Bytes of memory for storage, on contemporary GPUs such as the NVidia GTX 680, the number of objects that may be stored in constant memory is limited to 585. Requirements greater than this will result in primitives ‘spilling’ into the global memory, which will increase the overall run-time.

### ***Handling of the View Factor Matrix***

By virtue of the fact that the view factor matrix must be readable by the host program and writable by all work-groups, it is initialised and stored in global memory. However, to prevent data inconsistencies occurring when multiple work-items attempt to modify the same entry in the view factor matrix, all operations on the view factor matrix are conducted atomically. This creates a serialisation point in which work-items queue to obtain exclusive access to a particular entry in the view factor matrix. For geometries in which view factors are low (less than 0.01), such as those containing numerous evenly distributed primitives, work-items accessing entries in the view factor matrix seldom clash and this is not a significant issue. However, for geometries in which objects will yield moderate or large view factors, serialisation of access to the view factor matrix can significantly increase the run-time.

In order to increase performance while maintaining data consistency, the RayFactorCL kernel implements indirect recording of work-item results by first storing results in local memory and later adding these results to the ‘master’ view factor matrix in global memory. Here, each work-group maintains an array in local memory representing a single line of the view factor matrix. As a work-group starts firing rays from a given object, this array is initialised to zero and then used to store the view factor results from this object for all the work-items in the work-group. When all rays have been fired from this object, each work-group performs a reduction step in which the elements of their local array is atomically added to the view factor matrix in global memory. By doing this, serialisation no longer occurs for all active work-items on the compute device, but is instead limited to the active work-items within a single work-group.

While using an indirect approach to recording results increases the total workload a kernel must perform, it dramatically improves the performance of the RayFactorCL kernel as evident in Table 13, which presents the relative run-times for each approach for the benchmarking cases C-77 and C-109 (shown in Table 4) and a larger geometry denoted W-100 that consists of 55 vertically stacked cylinders.

**Table 13: Comparison of RayFactorCL relative run-times for direct and indirect recording of work-item results.**

| Example | Indirect | Direct |
|---------|----------|--------|
| C-77    | 1        | 3.28   |
| C-109   | 1        | 7.85   |
| W-100   | 1        | 1.02   |

Of the geometries examined in Table 13, the most impressive run-time reduction is observed for C-109. This geometry has the largest view factors ( $F_{1-1} \approx 0.3$ ,  $F_{1-2} \approx 0.7$ ,  $F_{2-1} = 1$ ) and, therefore, the highest incidence of work-items attempting to concurrently access a given entry in the view factor matrix. On the other side of the spectrum, the view factors for geometry W-100 are all quite small ( $F_{i-j} < 0.01$ ), resulting in a lower incidence of concurrent access and, therefore, a less impressive run-time reduction of only 2%.

#### ***Kernel Design and Launching***

Three distinct kernel designs were examined for implementation in RayFactorCL. Design A was a comprehensive kernel that is queued once, and after launch, loops through all objects launching rays from each one as it goes. This architecture resulted in the most complex kernel and the highest private memory footprint of all the kernel designs examined. Design B only launched rays from a single object each time it was launched. This required the kernel to be queued once for each object, increasing the host program's workload. However, it resulted in a reduction in the private memory footprint and the removal of a run-time loop. Finally, Design C extended on Design B, using a specialised kernel for each primitive type. Here, when the host program is queuing a kernel for each object, it examines the type of object and queues a specific kernel for that object type. This removes a step in which the work-item would have to select which function to use to generate a ray. However, it requires that multiple kernels are maintained increasing the complexity of RayFactorCL. The relative run-times for each kernel design using the C-77, C-109 and W-100 geometries is presented in Table 14.

**Table 14: Relative run-times for various kernel designs in RayFactorCL.**

| Kernel Design | A | B    | C    |
|---------------|---|------|------|
| C-77          | 1 | 0.96 | 0.95 |
| C-109         | 1 | 0.72 | 0.76 |
| W-109         | 1 | 0.86 | 0.86 |

As demonstrated by Table 14, run-time reductions are obtained when, moving from a single launch (A) to a multiple launch (B and C) kernel design. Negligible performance difference was observed between B and C and, therefore, kernel Design B was selected as it offered higher maintainability.

#### 5.4.3 Random Number Generator

In order to correctly ‘sample’ a geometry, each work-item must maintain its own Pseudo Random Number Generator (PRNG). Given the private memory constraints, a PRNG implemented on the GPU must have a low memory footprint and, therefore, the issue of PRNG selection had to be revisited.

In Section 4.3.2, dSFMT was selected for pseudorandom number generation on the CPU, requiring a total of 3,088 Bytes to retain its state. Accounting for the warp and memory allocation granularity of the GPU hardware, a memory demand of this magnitude would fail to compile, as it would request more private memory than was physically available on each compute unit. Alternatively, to work-around private memory constraints, the state of the dSFMT PRNG could be stored in the abundant global memory, however the frequency of PRNG use coupled with the lower bandwidth of the global memory region would cripple simulation performance.

In addition to the constraints on private memory availability, one must consider the use case for the PRNG on the device it will run. On the CPU, relatively few independent streams of pseudorandom numbers will be generated due to the low number of available processing elements; however, each stream will contain a significant number of random numbers. This is the perfect use case for the dSFMT PRNG, which can generate pseudorandom number streams of at least  $2^{19937} - 1$  elements, but is very limited in its capability to generate independent streams [49]. The larger number of processing elements in the GPU, on the other hand, requires numerous independent streams of pseudo random numbers with a relatively small number of elements per stream, a less than optimal usage pattern for the dSFMT PRNG.

To contend with the constraints in generating pseudorandom numbers on the GPU, an alternative to dSFMT was sought and found in the crush-resistant, counter-based Philox PRNG from the Random123 library [50]. This PRNG was specifically designed for massively parallel high-performance computation, and as such requires only 24 Bytes of memory to retain state. It can produce at least  $2^{64}$  independent parallel streams<sup>8</sup> each with a period of at least  $2^{128}$ .

The Philox PRNG takes a fundamentally different approach to conventional PRNGs (such as dSFMT) for the generation of pseudorandom numbers. Conventional PRNGs generate a sequence of numbers by successively applying a transformation function to a data structure consisting of earlier numbers in the sequence (i.e. the PRNG's state). This makes conventional PRNGs inherently serial, as generation of successive numbers in the sequence requires knowledge of the previously generated numbers.

Alternatively, counter-based PRNGs such as Philox, generate each number by applying a transformation function to a simple integer counter. This permits a sequence of pseudorandom numbers to be reproduced from any point (or counter value) without having to generate previous numbers in the sequence. By ensuring that the transformation function is bijective, the period for this generator will be equal to  $2^P$  where  $P$  is the size of the integer counter in bits (RayFactorCL used a 128 bit counter). However, if each work-item used a simple counter-based PRNG, they would all generate identical numbers in the event that counter values overlapped, lowering the effective number of simulation samples. This is avoided in the Philox PRNG by using a keyed block cipher as the transformation function. Keyed block ciphers are used extensively in cryptography and map results to a given key, ensuring that the cipher output is unique to the key value. By giving each work-item a unique key such as their OpenCL global identification number<sup>9</sup>, they are guaranteed to produce a unique stream of pseudorandom numbers regardless of counter overlap.

The cipher block functions used in the Philox generator are quite slow to execute relative to conventional PRNG transformation functions, with the standard MT-19937 PRNG achieving approximately 1.8 times the performance on an Intel Xeon CPU [50]. However, due to the significantly lower memory requirements on the Philox generator, it achieves 7.9 times higher performance than the MT-19937 generator on an NVidia GTX 580 GPU [50].

---

<sup>8</sup> This was confirmed with M. Moraes, co-author of the Random123 library, via personal communication on the 23/03/2013.

<sup>9</sup> In OpenCL, each work-item is given a unique global identification number, which is simply a sequential integer.

## 5.5 Performance Comparison of RayFactor and RayFactorCL

Extensive performance benchmarking was performed on RayFactorCL to quantify the performance gains realised using GPGPU computing with the OpenCL framework relative to the CPU optimised version of RayFactor discussed in Chapter 4. This performance benchmarking involved running the test geometries introduced in Section 3.4 in addition to the test geometry W-100 introduced in this chapter and comparing the execution speeds.

### 5.5.1 Benchmarking Compute Devices

Benchmarking was conducted on the Mac Pro 4.1 test system using an NVidia GTX 580 and GTX 680 independently and then in conjunction. Although it is possible to additionally use the test system's CPUs as compute devices, this was not done as the kernel was written specifically targeting NVidia GPUs which would lead to less than optimal performance on the CPU and because heavy loading of the CPU can result in 'GPU starvation' lowering their total throughput [98]. Furthermore, the work partitioning strategy employed would only allocate one out of every 67 objects to the CPU when operating heterogeneously (CPU + GTX 580 + GTX 680) and, therefore, the performance benefits from utilising CPUs as compute devices would be negligible.

### 5.5.2 Benchmarking Results

Typically, when comparing the performance of programs running on a GPU versus a CPU, single threaded CPU programs are used to access the performance of the CPU [99-101]. As demonstrated in Section 4.4, this is inherently wrong as for the past decade CPUs have been manufactured with multiple cores and SIMD support providing large performance gains over serial implementations (32.9 times for RayFactor). In order to obtain a fair comparison of performance between the two classes of device, performance will be presented on a per device basis. Given that the test machine is equipped with two CPUs, this involved limiting execution to a single CPU.

#### ***Run-Timing***

Run-time measurements were taken for both RayFactor and RayFactorCL using the "*cycle.h*" module from the Massachusetts Institute of Technology as part of the FFTW library [102]. This module measures the number of processor cycles between two events, which can subsequently be converted into absolute time using the clock speed of the CPU.

For RayFactorCL, program initialisation time (including the time taken to compile OpenCL kernels and transfer memory between the host program and compute devices) was not included in this comparative study, although it was measured separately. Therefore, the results presented are representative of the time taken to launch and finish executing kernels. Measurements for RayFactor on the CPU were identical with no utility operations being included in the final measurement.

## Results

As previously mentioned, each of the geometries presented in Table 4 as well as the W—100 geometry introduced in this chapter were run with RayFactor on a single Intel Xeon E5520 and with RayFactorCL using a NVidia GTX 580, GTX 680 and a GTX580 + GTX680 configuration. A ray density of  $1 \times 10^6$  rays per unit area was used for all elements in all benchmarking geometries. The relative performance of each implementation and compute device is shown in Table 15.

**Table 15: Relative performance for each benchmarking geometry on CPU and GPU based systems.**

| Case  | Xeon<br>E5520 | GTX<br>580 | GTX<br>680 | GTX580 &<br>GTX680 | Approx.<br>$F_{ij}/F_{ji}$ |
|-------|---------------|------------|------------|--------------------|----------------------------|
| C-14  | 1             | 13.9       | 18.0       | 9.5                | 0.20/0.20                  |
| C-47  | 1             | 14.4       | 19.8       | 11.2               | 0.38/0.13                  |
| C-52  | 1             | 16.9       | 25.6       | 9.0                | 0.15/0.07                  |
| C-77  | 1             | 15.0       | 19.7       | 18.1               | 0.27/0.18                  |
| C-109 | 1             | 11.4       | 10.1       | 8.0                | 0.71/1.0                   |
| C-122 | 1             | 16.3       | 24.2       | 21.0               | 0.06/0.17                  |
| W-100 | 1             | 21.3       | 32.1       | 47.5               | 0.01                       |

From Table 15, it can be seen that RayFactorCL running on either the NVidia GTX 580 or GTX 680 provides significant performance gains (10 – 32 times faster) than RayFactor running on the CPU. It should be noted that the benchmarking geometries with the exception of W-100 provide the worst geometric characteristics for the GPU, specifically large view factors which increase the extent to at which work-item execution is serialised to atomically increment the view factor matrix. This is especially true for geometry C-109, which has the largest view factors and, as a result, the lowest performance increases relative to the CPU.

Geometry W-100 provides the highest performance increases relative to the CPU for both GPUs. As one may expect, this is because with 55 objects this geometry is significantly larger, with each object having relatively small view factors. This geometry is more representative of those that would be analysed in practical engineering problems and, therefore, one should expect performance boosts greater than 20 times in such applications.

### ***RayFactorCL on Multiple GPUs***

Due to the relatively simplistic work partitioning strategy employed in RayFactorCL, geometries C-14 through to C-122 still run solely on the GTX680. This is because in a two GPU configuration RayFactorCL's performance metric calculates that the GTX680 performs twice the number of cycles as the GTX580 and, therefore, allocates both objects to be processed by the GTX680 while the GTX580 sits idle. Although the GTX680 is the only device actually used to produce results, the host program must still perform work partitioning calculations and device synchronisation which results in lower performance gains than if just the GTX680 was used.

For the more realistic W-100 case, utilising the GTX680 and the GTX580 together rather than just the GTX680 alone gains 1.5 times higher performance. While this increase is not insignificant, it could be improved still further by using a more advanced dynamic work partitioning strategy. The results in Table 15 indicate that despite the GTX680 having twice the throughput of the GTX580, as calculated by the performance metric, actual throughput of the GTX680 is only 1.5 times that of the GTX580. This means that the efficiency of the implemented work-partitioning strategy is only 75%, providing ample room for further improvement.

## **5.6 Conclusions**

High performance primitive-based MC-RT simulation for the calculation of radiative view factors targeting both CPU and GPU compute devices has been established in the RayFactor and RayFactorCL programs. However, while heavily utilised in the field of computer graphics, geometric primitives are not commonly used for object representation in radiative heat transfer applications with Finite Element Meshes (FEMs) being the favoured option. FEMs have the ability to approximate any geometry using a single element, such as a triangle. However, they require a significantly larger number of elements to represent a given geometry. Therefore, a comparison of the speed and accuracy of radiative view factor calculation for both primitive and FEM based object representation will be considered in the following chapter.



## Comparison with Finite Element Methods

---

Although surface representation using geometric primitives provides many advantages, its implementation has been largely confined to the field of computer graphics. In contemporary heat transfer software, surfaces are predominately modelled using finite element methods (FEMs). Indeed, finite elements form the basis for a wide range of numerical approaches with an extensive literature available on their use in diverse applications e.g. [89, 103-105].

In the field of heat transfer, FEMs offer high interoperability between radiative, conductive and convective heat transfer solvers making them ideal for conjugate heat transfer models (as will be discussed in the fibre drawing chapter presented in the next chapter). Furthermore FEMs allow for efficient program architecture due to the fact that only a single (typically triangular) object type need be implemented to model any geometry. The use of a single object removes the need for branching operations, which are required when multiple object types are implemented to follow execution paths dependent on object type, resulting in enhanced instruction caching.

However, while object representations with FEMs has definite advantages over geometric primitives they are not without their flaws. FEMs require comprehensive knowledge of available meshing algorithms and finite element types to produce optimal meshes for geometries of interest. Furthermore, unlike geometric primitives, FEMs are only ever approximations of surfaces and will therefore have an inherent error, especially when utilised to model curved surfaces. Although, this error can be minimised or even eliminated<sup>10</sup> through use of an appropriate meshing strategy, it typically comes at the cost of processing a greater number of elements.

---

<sup>10</sup> It is only meaningful to eliminate approximation error relative to the precision of the floating-point representation used in the simulation.

With these considerations in mind, this chapter presents a comparative study of the speed and accuracy using a triangle based FEM mesh and geometric primitives for the two benchmarking geometries C-77 and C-122, and subsequently investigates methods by which both methods may be used in conjunction with each other to enhance the overall performance of the calculation of radiative view factors.

## 6.1 The Triangle Element

The triangular element is the most fundamental element type used for FEM modelling of two and three-dimensional objects. Given the importance of the triangle element in a variety of numeral fields such as computer graphics, and physics and engineering simulations, it has been extensively researched resulting in the availability of extremely high performance ray-triangle intersection algorithms [89, 103, 105, 106]. Given the high performance and low storage requirements of the triangle (only 3 vertices and a normal are required), it makes an ideal candidate for a performance comparison of calculating radiative view factors in geometries represented using either geometric primitives or FEMs.

### 6.1.1 Ray-Triangle Intersection Algorithm

The approach for calculating ray-triangle intersection is substantially less complex than that required to calculate the intersection between a ray and the geometric primitives introduced in Chapter 2. Triangles are fully defined in ‘world’ space and therefore ray-triangle intersections do not require transformation steps during intersection calculations. This reduces the memory footprint of a triangular element as one does not need to store transformation matrices as required by the geometric primitives.

Given that triangles exist completely in world space, performing a ray-triangle intersection is simply a matter of determining if a ray ( $S + ct$ ) passes through a point in the triangle defined by its vertices A, B and C. Using the barycentric coordinates  $u$  and  $v$ , ray-triangle intersection may be determined by solving equation (87).

$$S + ct = A + u \cdot (B - A) + v \cdot (C - A) \quad (87)$$

In order for the intersection to be considered valid, the intersection point calculated by equation (87) must lie within the triangle and be the earliest intersection that the ray experiences, i.e. the ray does not hit any other object before striking this particular triangle. This can be determined by ensuring that the parameters  $t$ ,  $u$  and  $v$  meet the conditions outlined in equation (88).

$$\begin{aligned}
0 < t &\leq t_{best} \\
u &\geq 0 \\
v &\geq 0 \\
(u + v) &\leq 1
\end{aligned} \tag{88}$$

In the context of ray-tracing, numerous computer-based algorithms have been developed to rapidly solve equation (87) with respect to the conditions outlined in equation (88). For the purposes of this research, the ray-triangle intersection algorithm presented by Harvel and Herout [103] was implemented in OpenCL for the triangle primitive. This algorithm builds on the techniques proposed by Wald [89] and Shevtov [106] to achieve a maximum intersection calculation speed-up of 30% over the earlier algorithms.

The Harvel algorithm describes a triangle in terms of three planes rather than its vertices A, B and C, which are each defined by the general equation of a plane using a normal vector  $\mathbf{n}$  and a constant  $d$  as shown in equation (89).

$$x.n_x + y.n_y + z.n_z + d = 0 \tag{89}$$

The three planes used to describe a triangle element are the plane containing the triangle surface, the plane perpendicular to the vector  $(C - A)$  and the plane perpendicular to the vector  $(B - A)$  as shown in Figure 20.

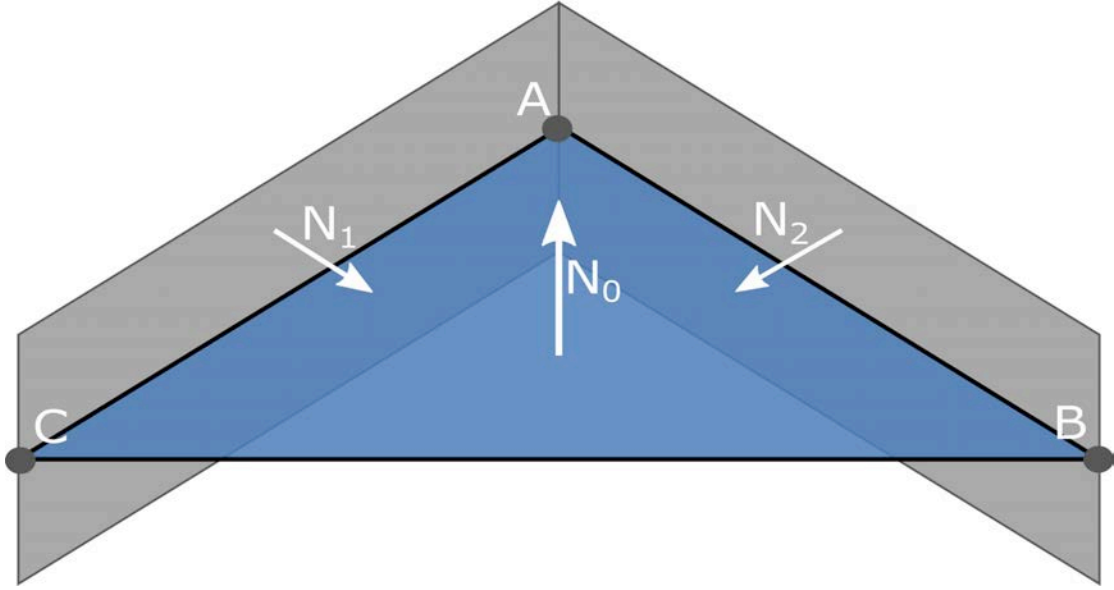


Figure 20: Planar representation of a triangle element

The plane in which the triangle lies may be defined by the normal to the triangle surface  $\mathbf{n}_0$  (this is not a unit normal and when  $\mathbf{n}_0$  is used for calculating ray direction it must be multiplied by the scalar  $1/(n_{0,x}^2 + n_{0,y}^2 + n_{0,z}^2)$ ) and the constant  $d_0$  which can both be calculated using the triangle vertices as shown in equation (90).

$$\begin{aligned}\mathbf{n}_0 &= (B - A) \times (C - A) \\ d_0 &= -A \cdot \mathbf{n}\end{aligned}\tag{90}$$

The plane perpendicular to the side of the triangle  $(C - A)$  may be described by a vector  $\mathbf{n}_1$  that is perpendicular to both the triangle surface normal  $\mathbf{n}_0$  and the vector  $(C - A)$ , and a constant  $d_1$  which may both be calculated as shown in equation (91).

$$\begin{aligned}\mathbf{n}_1 &= \frac{(C - A) \times \mathbf{n}}{|\mathbf{n}|^2} \\ d_1 &= -\mathbf{n}_1 \cdot A\end{aligned}\tag{91}$$

The final plane used to define a triangle element, which is perpendicular to the side of the triangle  $(B - A)$ , may be defined by the vector  $\mathbf{n}_2$  (which is perpendicular to both the triangle surface normal  $\mathbf{n}_0$  and the vector  $(B - A)$ ), and a constant  $d_2$  which may both be calculated as shown in equation (92).

$$\begin{aligned}\mathbf{n}_2 &= \frac{\mathbf{n} \times (B - A)}{|\mathbf{n}|^2} \\ d_2 &= -\mathbf{n}_2 \cdot A\end{aligned}\tag{92}$$

Once these three planes have been calculated, knowledge of the triangle vertices is no longer required to perform ray-triangle intersection calculations. Therefore, by calculating and storing these planes during the pre-processing stage, the total work load in the processing stage is reduced and the memory footprint of the triangle element is not significantly increased as only the vectors  $\mathbf{n}_0$ ,  $\mathbf{n}_1$  and  $\mathbf{n}_2$  and the scalar constants  $d_0$ ,  $d_1$  and  $d_2$  need be stored.

Using the three planes described previously, the system of equations required to find the ray-triangle intersection point  $P$  as described by the barycentric coordinates  $u$  and  $v$  are as shown in equation (93).

$$\begin{aligned}P &= S + ct \\ u &= \mathbf{n}_1 \cdot P + d_1 \\ v &= \mathbf{n}_2 \cdot P + d_2\end{aligned}\tag{93}$$

Here the barycentric coordinates  $u$  and  $v$  express a point on the triangle in terms of a scaled distance along the vectors  $\mathbf{n}_1$  and  $\mathbf{n}_2$ . By substituting the ray equation ( $S = ct$ ) into the plane equation of the triangle, the time at which the ray intersects with the plane containing the triangle may be calculated as shown by equation (94).

$$t = -\frac{\mathbf{n}_0 \cdot S + d}{\mathbf{n}_0 \cdot c}\tag{94}$$

The Havel algorithm, however, recognises that division operations are computationally expensive (approximately 14 times slower than an addition operation on an Intel Nehalem CPU) and therefore defers all divisions until the end of the calculation. This allows unnecessary divisions to be avoided in the event that the ray does not intersect the triangle and there is an opportunity to exit the intersection calculations early. Instead, the denominator and a variant of the numerator are calculated separately as shown in equation (95).

$$\begin{aligned} div &= \mathbf{n}_0 \cdot \mathbf{c} \\ t' &= d - (\mathbf{S} \cdot \mathbf{n}_0) \end{aligned} \tag{95}$$

Without explicitly calculating  $t$  as shown in equation (94), the first condition of equation (88) may be tested using the expression listed in equation (96).

$$sign(t') = sign(div \cdot t_{best} - t') \tag{96}$$

If equation (96) is ‘false’ then intersection calculations may be exited early. However, if it is ‘true’ then the ray intersects the plane containing the triangle in front of its starting point  $\mathbf{S}$  ( $0 \leq t$ ), and it is the earliest intersection found thus far ( $t \leq t_{best}$ ). Assuming that equation (96) is ‘true’ the next step in the algorithm is to check the second condition of equation (88). This is done by calculating the pseudo-barycentric coordinate  $u'$  ( $u' = div \cdot u$ ) as shown in equation (97).

$$\begin{aligned} P &= \mathbf{S} + \mathbf{c}t \\ u' &= \mathbf{n}_1 \cdot P + d_1 \end{aligned} \tag{97}$$

The second condition for a valid intersection,  $u \geq 0$ , may now be tested using the sign of  $u'$  for another opportunity to exit early from the intersection calculations using equation (98).

$$sign(u') = sign(div - u') \tag{98}$$

As with previous checks, if equation (98) is ‘false’ then the ray does not intersect the triangle and intersection calculations may be exited early. However, if it evaluates as ‘true’ the pseudo-barycentric coordinate  $v'$  ( $v' = \text{div} \cdot v$ ) is calculated in the same manner as  $u'$  as shown in equation (99).

$$v' = \mathbf{n}_2 \cdot P + d_2 \quad (99)$$

A third and final check must now be completed using equation (100), which given that equation (98) is true at this point, simultaneously ensures that  $v \geq 0$  and  $(u + v) \leq 1$  providing the last opportunity for an early exit from the intersection calculations.

$$\text{sign}(v') = \text{sign}(\text{div} - u' - v') \quad (100)$$

If equation (100) holds ‘true’, then it can be safely stated that the ray does in fact intersect the triangle and the division deferred from the start of the calculation performed to obtain the actual intersection time and barycentric coordinates of the intersection as shown by equation (101).

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\text{div}} \begin{bmatrix} t' \\ u' \\ v' \end{bmatrix} \quad (101)$$

Although the ray-triangle intersections calculations appear somewhat more complex than those for of the geometric primitives presented in Chapter 2, they require relatively little data and only a single division operation. This results in extremely high performance when checking for ray-triangle intersection as shown later in Section 6.2.2. However, performing ray intersections is not the only workload when performing MC-RT simulations, rays must also be generated and launched from the surface of each object. The following section presents the method implemented for the launching of rays from triangle elements.

### 6.1.2 Triangle Ray Launching Strategy

Two different approaches may be adopted for launching rays from the surface of triangular elements. These methods differ only in the way in which the ray starting point is selected, with the first approach randomly selecting starting points on the surface of the triangle similar to ray starting point selection for geometric primitives, and the second simply using the triangles centre as the ray starting point with each ray differing only in the direction it is fired.

As one may expect, the first approach will yield the most accurate results, however it requires each vertex of the triangle to be stored. This will extend run-times either through the increased data requirements of for the vertices (9 more floating point values are needed) or the additional workload to calculate the planes used in the ray-triangle intersection calculations. For this reason the second approach was taken with the coordinates of each triangles centre point  $P_c$  being calculated during pre-processing using equation (102).

$$P_c = \frac{1}{3}(A + B + C) \quad (102)$$

Calculating the triangle's centre point during the pre-processing phase increases the memory footprint of the triangle by 16 bytes (i.e. 12 bytes to store  $P_{c,x}$ ,  $P_{c,y}$  and  $P_{c,z}$  and one byte of 'padding' to improve cache alignment), however this is significantly less than the 48 bytes required to store the triangle vertices.

As each ray is generated, its starting point is set to the pre-computed centre point of the triangle. The direction of each ray is then calculated using the same methodology used for the geometric primitives, which is outlined in Section 2.3.1. However, the surface normal required to align the ray directional distribution with the launch surface need not be calculated during ray launching (as is done with the geometric primitives), as the triangle surface normal  $\mathbf{n}_0$  has already been calculated during the pre-processing stage using equation (90).

## 6.2 Comparison of Triangular FEMs and Geometric Primitives

In order to assess the value of geometric primitives, it was appreciated that their performance must be compared to that of calculations performed using a 'fast' FEM based approach. This was achieved by assessing the 'raw' and 'compound' performance of each representational method. Raw performance gives insight



into the mechanics of an individual element and can be assessed using metrics such as the rate at which random rays could be generated and intersection calculations performed. Compound performance is a high-level view of each object's representational strategy and measures overall performance in the context of radiative heat transfer geometries; here the metrics used are the total simulation time and the accuracy of the calculated view factors.

### 6.2.1 Ray Launching Speed

As one may expect, this performance metric measures the rate at which random rays can be generated for a given object type. Theoretically, the triangle element stands at a marked advantage over geometric primitives for ray launching. Triangles exist solely in world space and therefore the ray starting point does not need to be transformed as required for a geometric primitive. Additionally, the triangle has many useful properties pre-calculated and available for immediate use, such as the triangle's centre, which is utilised as the ray starting point, and the triangle's surface normal which is required to align the ray directional distribution.

The ray launching speed was measured by developing a series of OpenCL kernels to perform the ray generation algorithms for each object type. The time taken to execute each of these kernels on an NVidia GTX-580 operating under Windows XP with a set number of work-items was measured and the ray generation rate calculated as the total number of rays generated divided by the time taken to complete execution. The rates of ray generation for the geometric primitives relative to the rate of ray generation for the triangle element are shown in Table 16.

**Table 16: Relative ray generation performance of objects in an OpenCL implementation.**

| Object    | Relative Performance |
|-----------|----------------------|
| Annulus   | 0.70                 |
| Cylinder  | 0.72                 |
| Disc      | 0.71                 |
| Frustum   | 0.65                 |
| Rectangle | 0.93                 |
| Sphere    | 0.67                 |
| Triangle  | 1.00                 |

As expected, the highest ray generation rate is observed for the triangle element, which is on average 37% greater than that of the geometric primitives tested.

### 6.2.2 Intersection Time

More important than ray launching to overall simulation speed, however, is the rate at which ray-object intersections can be performed. Multiple intersection calculations per ray launched are required regardless of any space partitioning strategies employed, and therefore efficient intersection routines are paramount to MC-RT performance. Similarly to the testing of ray launching performance, dedicated OpenCL kernels were developed to conduct the intersection calculations for each object type and the time taken to execute these kernels on an NVidia GTX-580 with a set number of work-items was measured. However unlike ray generation, the performance of intersection calculations varies depending on the geometric orientation between the object and the ray. The intersection methods for many objects have early exit conditions, which identify when rays will not intersect an object before the entire intersection calculation is executed, thus terminating calculations early and avoiding unnecessary work. Therefore, in order to examine the full spectrum of intersection calculation performance, both the case in which the ray misses the object (defined as the ‘best’ case in terms of computational speed) and the case where the ray intersects the object (defined as the ‘worst’ case in terms of computational speed) must be examined. These two values may be viewed as the upper and lower limits of intersection performance, as in practice the average rate at which intersection calculations are performed will lie somewhere between these two values. Table 17 lists the ‘best’ and ‘worst’ intersection times for the triangular finite element and the geometric primitives presented in Chapter 2.

**Table 17: Relative ray Intersection performance for objects in an OpenCL implementation.**

| Object    | Best Case | Worst Case |
|-----------|-----------|------------|
| Annulus   | 2.15      | 1.52       |
| Cylinder  | 1.93      | 1.42       |
| Disc      | 2.15      | 1.82       |
| Frustum   | 1.68      | 1.28       |
| Rectangle | 2.14      | 1.79       |
| Sphere    | 2.09      | 1.56       |
| Triangle  | 3.14      | 1.00       |

In the 'best' case, where the ray misses the object and intersection calculations can be exited at the earliest point possible, ray-triangle intersection calculations provide the highest performance with the calculation rate on average 50% higher than for the geometric primitives tested. This is due to the fact that prior to evaluating any early exit conditions, geometric primitives must first inverse transform the ray from world space to object space, a calculation which requires numerous multiplication and addition operations.

In the 'worst' case, where the ray intersects the object, ray-triangle intersection performance is on average 36% lower than that for the geometric primitives. This can be attributed to the simplicity of the primitive's intersection calculations afforded by testing ray intersection in object space. However, low performance of intersection calculations in the event that a ray intersects a triangular element is not necessarily an indicator of poor overall performance. Triangle elements (especially in FEM implementations) generally have a small surface area and therefore the incidence of worst-case ray-triangle intersection will be low. However, as the triangles in an FEM implementation will generally have a small surface area relative to primitives, many more such triangle elements are required to model equivalent surfaces. Within this context, the performance of both triangles and primitives will be examined in the subsequent section.

### **6.2.3 Comparison for Benchmarking Geometries**

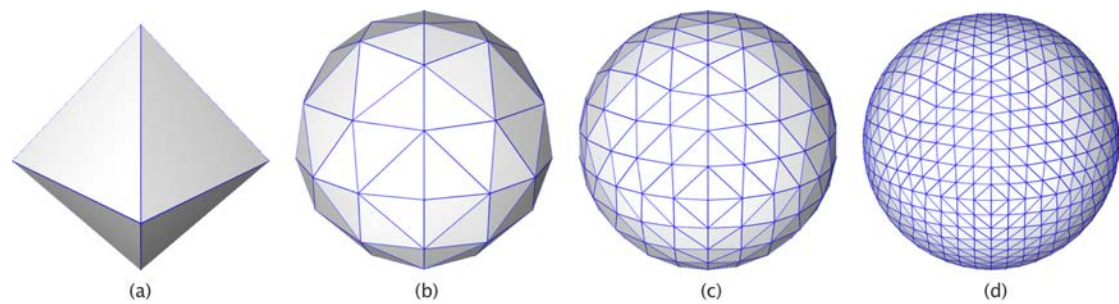
In addition to comparing the raw performance of both triangle elements and geometric primitives, the overall performance in the context of radiative heat transfer geometries must be examined. While the triangle elements have high performance for ray generation and intersection calculations relative to geometric primitives, a large number of triangles are required to adequately represent three-dimensional objects such as spheres and cylinders. To compare the overall performance of each representational method radiative view factors were calculated for the benchmarking geometries (C-77 and C-122) using both a triangular FEM mesh and geometric primitives.

These geometries contain both two-dimensional planar objects, such as a rectangle and an annulus, as well as three-dimensional objects such as a sphere and a cylinder.

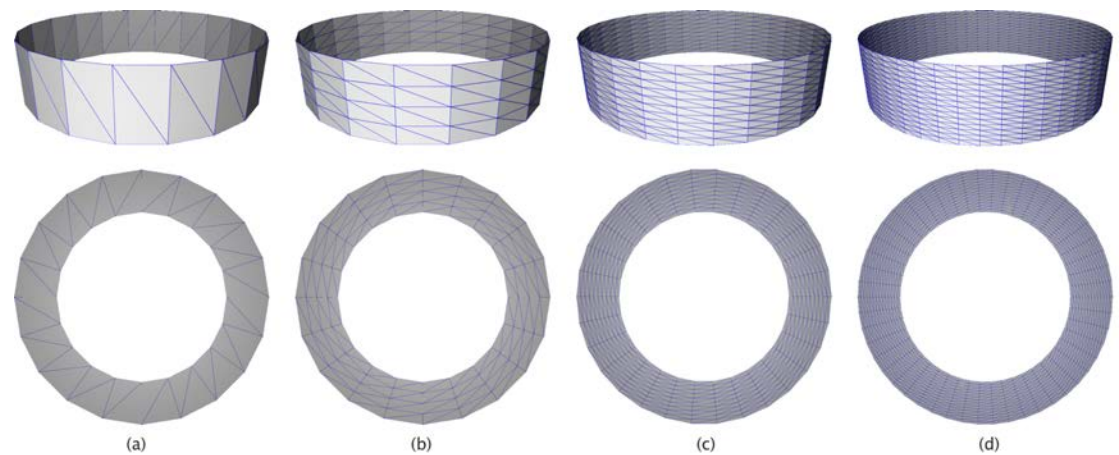
### ***Approximation of Geometric Primitive Using A Triangle Mesh***

Triangle meshes for the geometric primitives in the C-77 and C-122 benchmarking geometries were constructed using elementary meshing algorithms implemented by the author.

The triangle mesh for the sphere in the C-122 geometry was generated by taking two base aligned tetrahedrons and iteratively bisecting the edges of each of the eight triangular faces, creating four smaller triangles for each bisected triangle facet in the process. By repeating this bisection step the number of triangles composing the mesh is increased, improving the accuracy of the sphere approximation. Figure 21 demonstrates the progressively improved triangle mesh approximation by increasing the number of triangles forming the mesh from 8 to 2,048.



**Figure 21: Sphere approximation with a mesh consisting of (a) 8 (b) 128 (c) 512 and (d) 2048 triangles.**



**Figure 22: Triangle mesh approximation of a cylinder and annulus with (a) 20 (b) 160 (c) 600 and (d) 1400 elements**

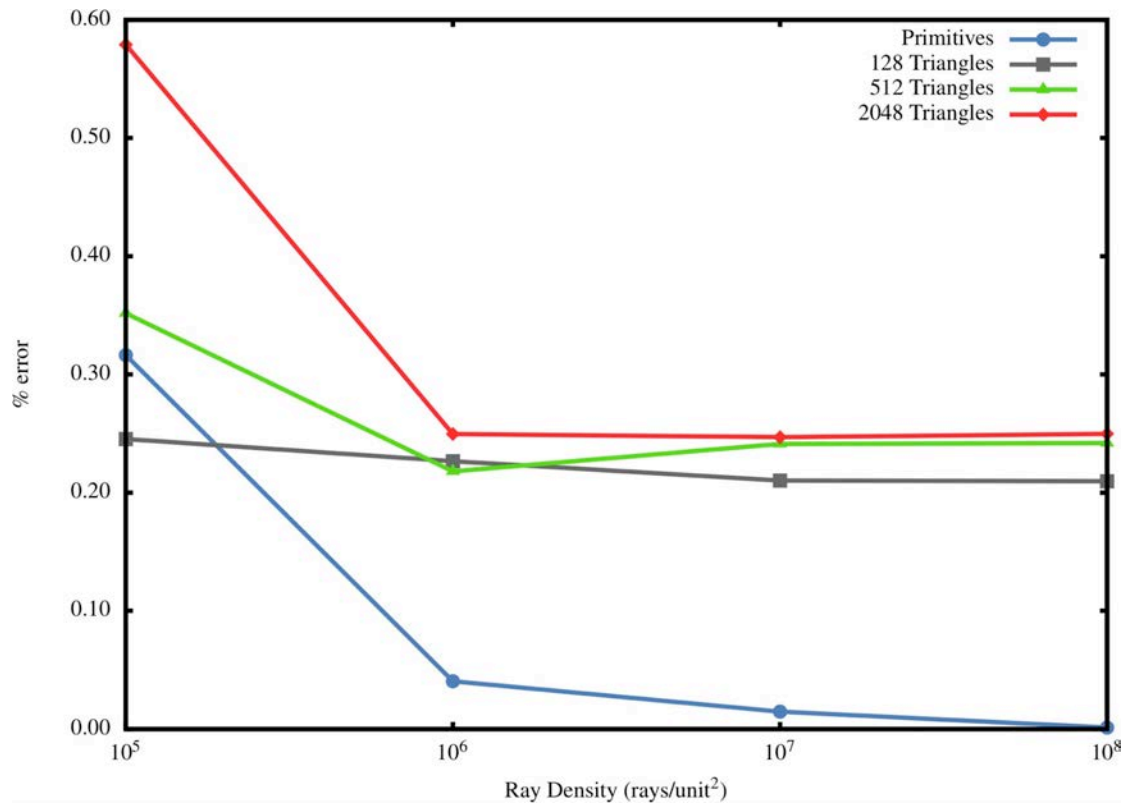
The meshing algorithms implemented for the cylindrical and annular objects were similar in design and simply segmented the upper and lower (or for the annulus inner and outer) circumferences into uniform angular increments. The quadrilateral segments formed by these divisions were then each divided into two triangular elements by joining two opposing vertices. The resulting triangle mesh approximations of the cylinder and annulus objects, which were generated using this algorithm with 20, 160, 600 and 1400 elements, are shown in Figure 22.

As shown in Figure 22, the algorithm to develop FEMs for the cylinder and annulus allow meshing of the cylinder in the axial and angular directions, and of the annulus in the angular and radial directions. As rays are fired from the centre of the triangle elements, computed radiative view factors are sensitive to the number of mesh divisions in either direction.

### ***Simulation Accuracy***

The accuracy of each object's representational method was assessed by comparing the calculated radiative view factors to the analytical solutions for the C-122 and C-77 geometries using ray densities over the range  $1 \times 10^5$  to  $1 \times 10^8$  rays/unit<sup>2</sup>.

In addition to the ray density, the number of triangles used to model each object was varied in an attempt to gauge the sensitivity of view factor error to the number of elements and the structure of the mesh used. View factor error as a function of ray density for the C-122 case, with the sphere modelled using meshes of 128, 512 and 2048 elements is shown in Figure 23. Here it can be seen that the geometric primitives not only have substantially lower error but also converge predictably on the analytical solution with increasing ray density. This is not the case for the FEMs, which for ray densities greater than  $1 \times 10^6$  have an essentially constant error. This is due to the fact that above this ray density, error is a strong function of the number of triangles composing the mesh. Given that rays are only launched from the centre of each triangle element, the total number of unique ray starting points is equal to the number of triangles forming the sphere mesh. Alternatively for geometric primitives ray-starting points are selected at random, meaning that each ray provides a fully unique sample point.

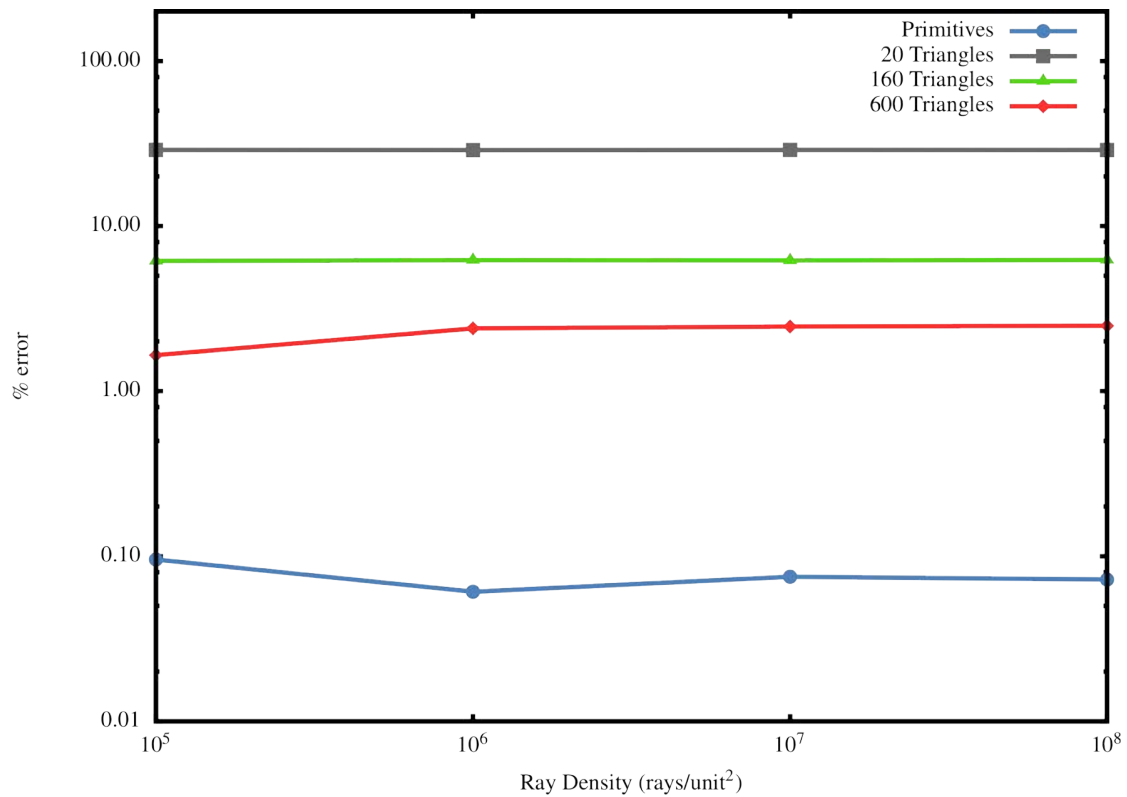


**Figure 23: Error in calculated radiative view factor for the C-122 geometry using triangle based FEMs and geometric primitives.**

Figure 23 indicates a surprisingly weak dependence between the error and the number of triangular elements used to describe the sphere. The reason for this behaviour is unclear but may well be a cumulative effect of the interactions between the various approximation techniques used. In all cases, however, the error for the calculated radiative view factors was less than 0.6%. Certainly, obtaining a solution for the case where 2048 triangles was used to model the sphere was substantially slower than simply using a sphere primitive, a point that will be discussed further in the following section. By comparison, the results for the C-77 geometry (shown on a semi-log plot in Figure 24) show a much stronger relationship between the error and the number of triangular elements used.

Although there are error differences between the two approaches, when considering the results in the context of applied engineering calculations, the error for either method is generally more than acceptable relative to errors introduced by other radiative parameters such as emissivity. Therefore it can be concluded that in terms of accuracy either method can produce solutions to an acceptable level of error for practical engineering calculations. However, accuracy is only one aspect, on which numerical methods should be compared; in

order for solutions to be useful they must not only be accurate but should be obtained as quickly as possible.

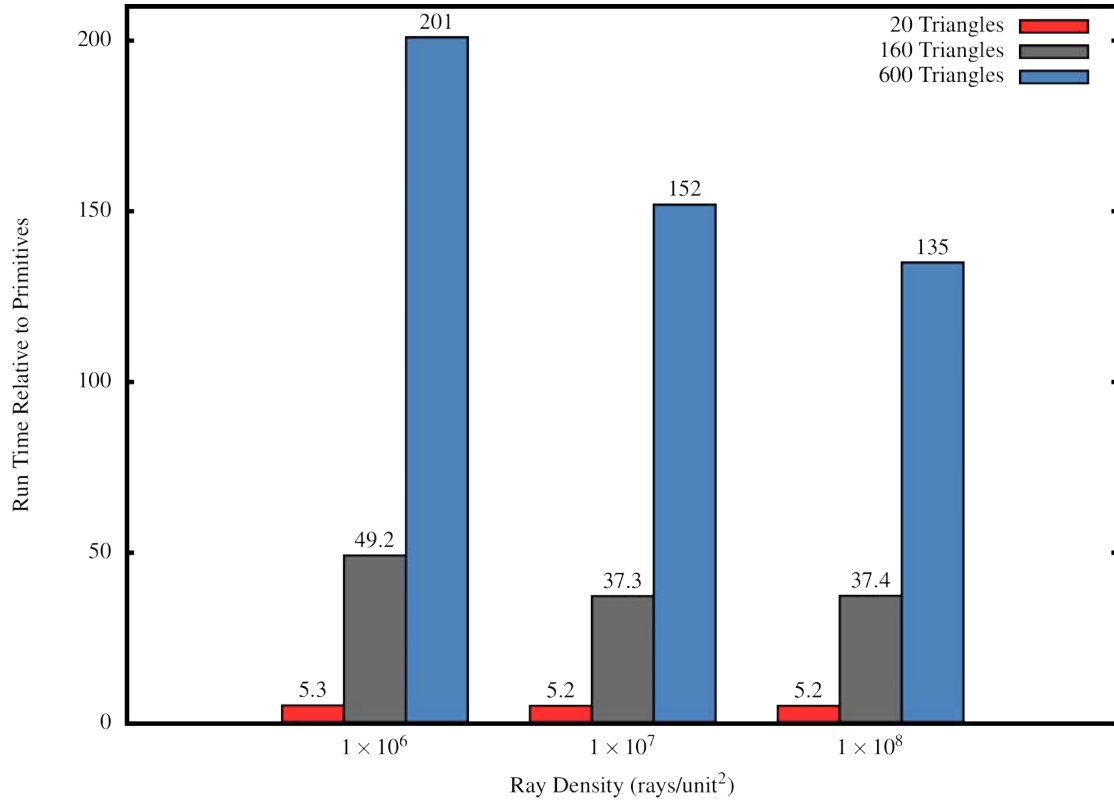


**Figure 24: Error in calculated radiative view factor for the C-77 geometry using triangle-based FEMs and geometric primitives.**

### ***Simulation Speed***

Equally as important as the accuracy of a numerical method is the speed at which it can provide a solution. As put forth by Gustafson's law [70], reductions in simulation run-time not only allow fixed size problems to be solved in the shortest possible time frame, but they also permit the solving of the largest possible problem in a given "reasonable" time frame. Therefore, by increasing the performance of MC-RT, one is ultimately increasing the size and complexity of radiative heat transfer models that can be feasibly solved.

The relative run-times for calculating radiative view factors for the C-77 case using triangular meshes consisting of 20, 160 and 600 elements per object are shown in Figure 25.



**Figure 25: Run times relative to geometric primitives for the calculation of radiative view factors for the C-77 geometry.**

It can be seen here that even when using a relatively coarse mesh of only 20 elements per object, the run-time is approximately 5.2 times greater than when the cylinder and annulus primitives are used. Furthermore, it should be noted that in addition to the substantial run time penalty, the view factors calculated using these 20 triangle meshes carry an error of approximately 29%, much greater than the 0.07 – 0.1% error exhibited by the view factors calculated using geometric primitives. Therefore, not only do geometric primitives provide superior accuracy to FEMs, they also offer an exceptional run-time advantage.

Although the computational time required to perform ray intersection calculations may be reduced from a linear  $O(n)$  to a logarithmic  $O(\log_2 n)$  relationship with the number of objects  $n$ , by implementing a space partitioning strategy (see Section 6.3), the relatively large number of mesh elements required to accurately model even simple objects such as a sphere or cylinder, inhibits the performance of an FEM based MC-RT solution from competing with a geometric primitive based approach.

However, it must be pointed out that FEMs still have several distinct advantages over geometric primitives; notably that they are capable of representing any



arbitrary surface (geometric primitives require a surface to be expressed as an implicit surface equation), and their use affords flexible surface discretisation. The latter feature is particularly important when analysing complex heat transfer models, where it is often desirable to have a meshing pattern that varies with both the local geometry and the dominant local heat transfer modes (i.e. radiative, convective and/or conductive). Traditionally, FEMs have been the only practical option for geometries requiring explicit discretization. However, this thesis proposes a hybrid representational method that draws inspiration from bounding volume space partitioning strategies, so as to provide the discretisation properties of FEMs with a performance level comparable to the use of geometric primitives on a GPU-based system.

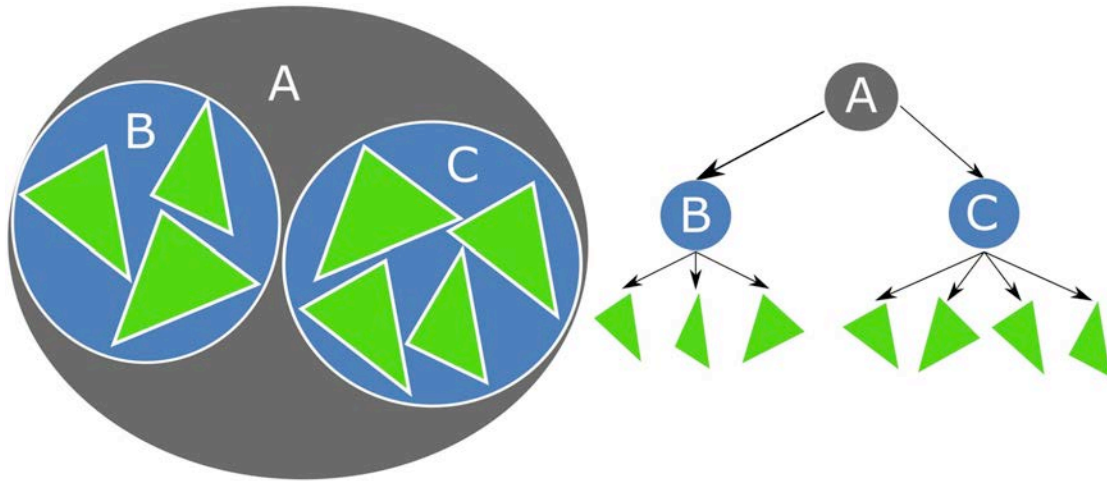
### **6.3 Accelerating FEM-Based MC-RT on a GPU-Based Computing Environment**

#### **6.3.1 Space Partitioning**

One of the most effective ways to accelerating MC-RT for geometries containing a large number of objects is to implement a space-partition strategy. Over the years numerous space partitioning strategies have been developed such as the uniform grid [107], kd-tree [24] and bounding volume hierarchy (BVH) [27] with the optimal partitioning strategy being highly dependent on the characteristics of the partitioned geometry.

Space-partitioning strategies recursively subdivide ‘space’ of interest into two or more sub-volumes and organise these volumes into an efficient data structure such as a binary tree. In the context of MC-RT, sub-dividing space in such a way allows some of the unnecessary ray intersection calculations to be avoided because if a ray does not intersect a sub-volume, it can safely be assumed that it will not intersect any objects within that volume and therefore ray-object intersection calculations need not be conducted. Figure 26 displays the geometric and tree representation for a bounding volume hierarchy for an illustrative two-dimensional system.

The two-dimension geometry in Figure 26 (left) contains eight triangle elements, with bounding volume A (the root node of the BVH) encompassing the entire geometry. A BVH may be constructed by subdividing the area within A into bounding volumes B and C. These subdivisions can be placed in a bounding volume hierarchy as shown in Figure 26 (right) to reduce the computations required to identify ray-triangle intersection.



**Figure 26: Construction of a bounding volume hierarchy for a two-dimensional geometry.**

Although many space-partitioning strategies, such as the kd-tree approach<sup>11</sup>, can reduce the complexity of ray intersection from an  $O(n)$  to  $O(\log_2 n)$  in the best case scenario, they have historically been developed within a CPU-based computing environment, and utilise techniques that are either sub-optimal or not currently supported for a GPU-based environment such as recursive function calls, multiple data referencing and random memory access. Due to the inherent computational power of the GPU, there has been a real incentive to adapt and optimise space-partitioning strategies for the GPU. This has led to the development of several implementations targeting efficient space-partitioning on the GPU [21, 28, 109, 110]. However there are still significant performance improvements to be made, as will now be demonstrated.

Despite the substantially higher computational throughput of a GPU, OpenCL kd-tree implementations targeting both the CPU and GPU have been found to have quite similar performance due to memory bandwidth limitations on the GPU. In fact, for geometries containing less than 30,000 elements, brute force searches (such as those used in RayFactor) were found to provide higher performance than a GPU-optimised kd-tree implementation [109].

Thus, rather than implementing a traditional space-partitioning strategy to enhance the performance of RayFactorCL when processing geometries represented by FEMs, a hybrid Primitive-FEM representational method was developed to obtain run-times closer to those of geometric primitives, while providing the desirable discretisation characteristics of FEMs.

<sup>11</sup> kd-trees have been found to generally provide the highest performance on a CPU [108] This made the kd-tree the de facto space-partitioning strategy.

### 6.3.2 Proposed Primitive-FEM Object Acceleration

While having high computational performance, geometric primitives limit the potential for geometry discretisation, and although FEMs provide high discretisation potential they exhibit lower computational performance on a GPU. While existing space-partitioning strategies can be used to enhance simulation performance, they have been demonstrated to be rather inefficient on the GPU. This has led to an exploration of an alternative acceleration method by which discretisation characteristics of FEMs can be employed but with substantially lower run-times than with ‘straight’ FEM based solutions.

In the proposed acceleration strategy, the geometric primitives act in a similar manner as a bounding volume<sup>12</sup>. However, rather than simply enclosing a FEM, the primitive object is specified (using an affine transformation) such that the elements of the discretization mesh lie on the surface of the primitive. While this approach requires that FEMs can only be described using geometric primitives, and is therefore not suitable for surfaces that cannot be described by implicit equations, it permits several types of optimisation in the ray launching and ray intersection algorithms.

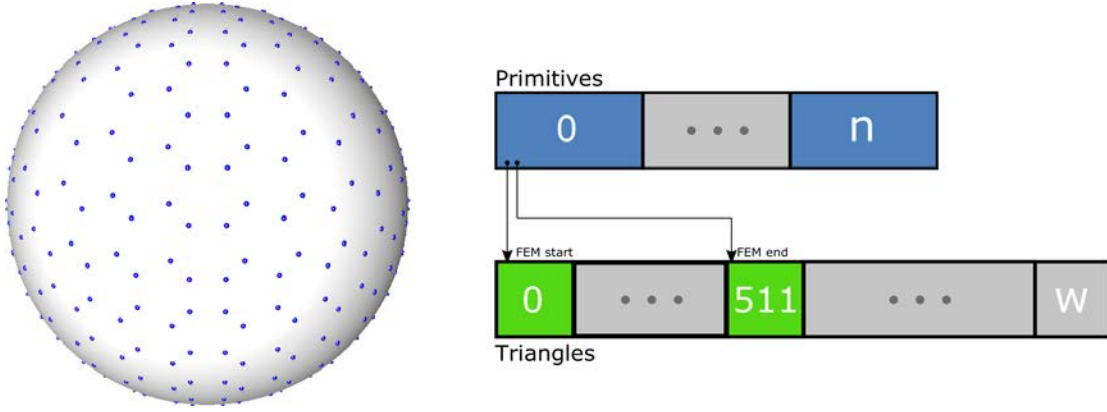
#### ***Object Construction***

Object construction is similar to that previously described in Sections 2.4 and 6.1 for both the primitive and FEM variants of RayFactorCL. However, as each triangle element is known to lie on the surface of a geometric primitive, ray intersection calculations no longer require knowledge of the surface normal, and the vertex planes described in Section 6.1 (the reasons for this will become clear in subsequent sections). Therefore, only the coordinates of the triangle’s centre point need to be stored, reducing the memory required to store FEM data by a factor of four. However, prior to storage, each centre coordinate is first transformed from world space to object space using the inverse transformation matrix of the parent primitive.

As each FEM is associated with a specific primitive object, the constituent FEMs are grouped by primitive type and stored sequentially in a single array. By doing this, each primitive object (stored in a separate array) can determine its ‘associated’ FEMs by storing just two additional integers, the indices of the first and last FEMs for which it is associated as shown in Figure 27.

---

<sup>12</sup> Bounding Volumes and Bounding Volume Hierarchies (BVH) have a distinct difference. A bounding volume is a simple primitive object which encloses one or more objects, while a BVH is a tree-based hierarchy of volumes in which each bounding volume may enclose one or more smaller bounding volumes.



**Figure 27: A sphere modelled using a primitive and 512 finite elements (shown as centre points) and its representation in memory where  $n$  is the total number of primitives and  $W$  is the total number of finite elements (stored as centre points) lying on the surface of  $n$  primitives.**

### **Ray Intersection**

As the primitive object is now defined such that its surface is coincident with its associated FEM, when a ray intersects a primitive it follows that it must also intersect one of the finite elements associated with that primitive. With knowledge that the ray intersects a primitive, the computational problem is no longer one of determining whether the ray intersects an element in the associated mesh, but rather one of determining which element in the mesh is intersected. While the specific element intersected could be identified by conducting ray intersection calculations with each associated mesh element (as done for a traditional bounding volume, and in fact the only way in which the result is guaranteed to be correct), this would require additional intersection computations and memory requirements.

Alternatively, once a ray-primitive intersection is confirmed in the proposed method, the co-ordinates of the intersection point are calculated as shown in Equation (103), and a nearest neighbour search is performed to determine the mesh element with the centre point that lies closest to the ray-primitive intersection point.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} S_x \\ S_y \\ S_z \end{pmatrix} + \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} t \quad (103)$$

To avoid the computationally expensive square root operation when calculating the distance between the intersection point  $H$  and the FEM centre point  $C$ , the squared distance is utilised as shown in Equation (104).

$$d = (H_x - C_x)^2 + (H_y - C_y)^2 + (H_z - C_z)^2 \quad (104)$$

It should be noted that the nearest neighbour search could itself be enhanced through the use of a space-partitioning strategy, however given the expected number of finite elements associated with a given primitive is expected to be less than 30,000, a 'brute force' nearest neighbour search provides the best performance on a GPU [109]. An overview of the proposed acceleration algorithm is shown in Figure 28.

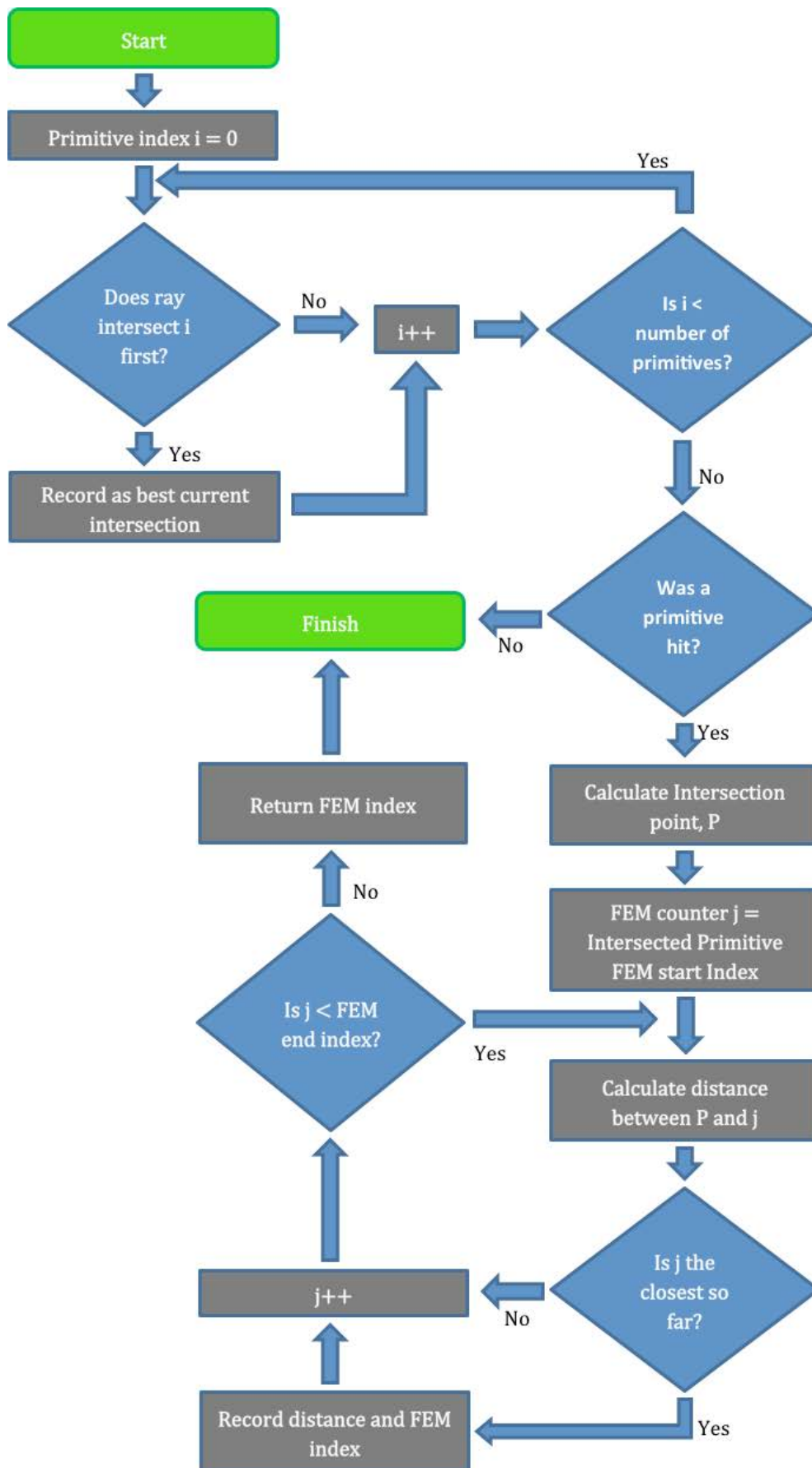
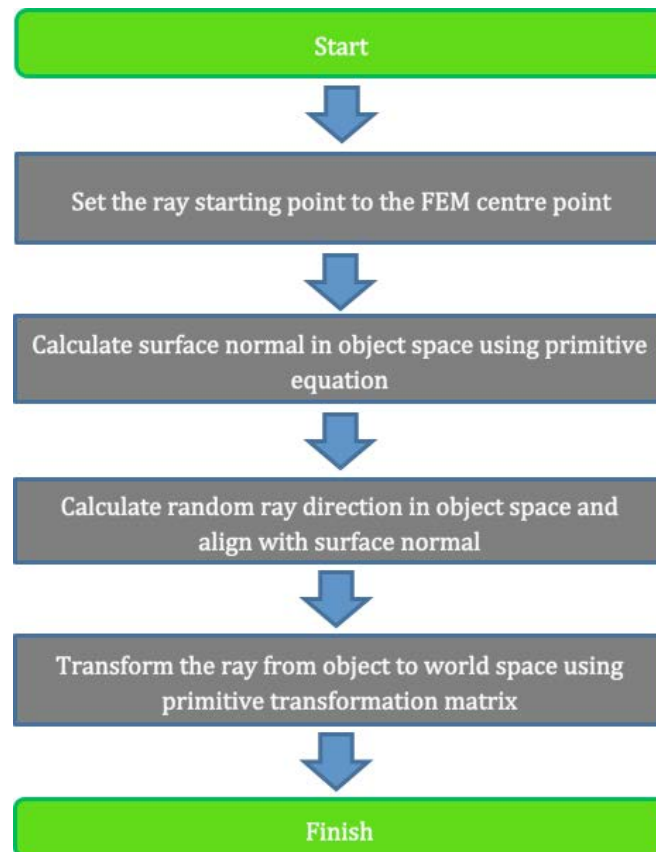


Figure 28: Overview of the ray intersection algorithm for the proposed FEM acceleration method.

### Ray Launching

Ray launching for the proposed acceleration algorithm lends itself strongly to that used previously for the FEMs. Here rays are launched from each FEM centre point, with the associated primitive being used to calculate the surface normal and hence ray orientation at the launch point. A flow chart outlining ray launching is presented in Figure 29.



**Figure 29: Overview of the ray launching algorithm for the proposed FEM acceleration method.**

Although the benchmarking study presented in Section 6.2.3 concluded that the reduction in accuracy experienced when using a FEM representational method was primarily due to limiting the ray starting points to a single location per element (i.e. the triangle's centre coordinates), by adopting the same approach here the number of calculations and data handling required to launch a ray are minimised. For example, if the ray launching algorithm of the geometric primitives was used not only would the determination of a ray's starting point be more computationally expensive (see Table 16), the FEM from which the ray is fired would not be known prior to formulation on the GPU. This means that one could no longer cache a single line of the view factor matrix in local memory as

described in Section 5.4.2, preventing efficient caching of OpenCL work-group results. Furthermore a nearest neighbour search (similar to that in ray intersection) would be required after each ray is formulated to determine which finite element the ray was actually fired from.

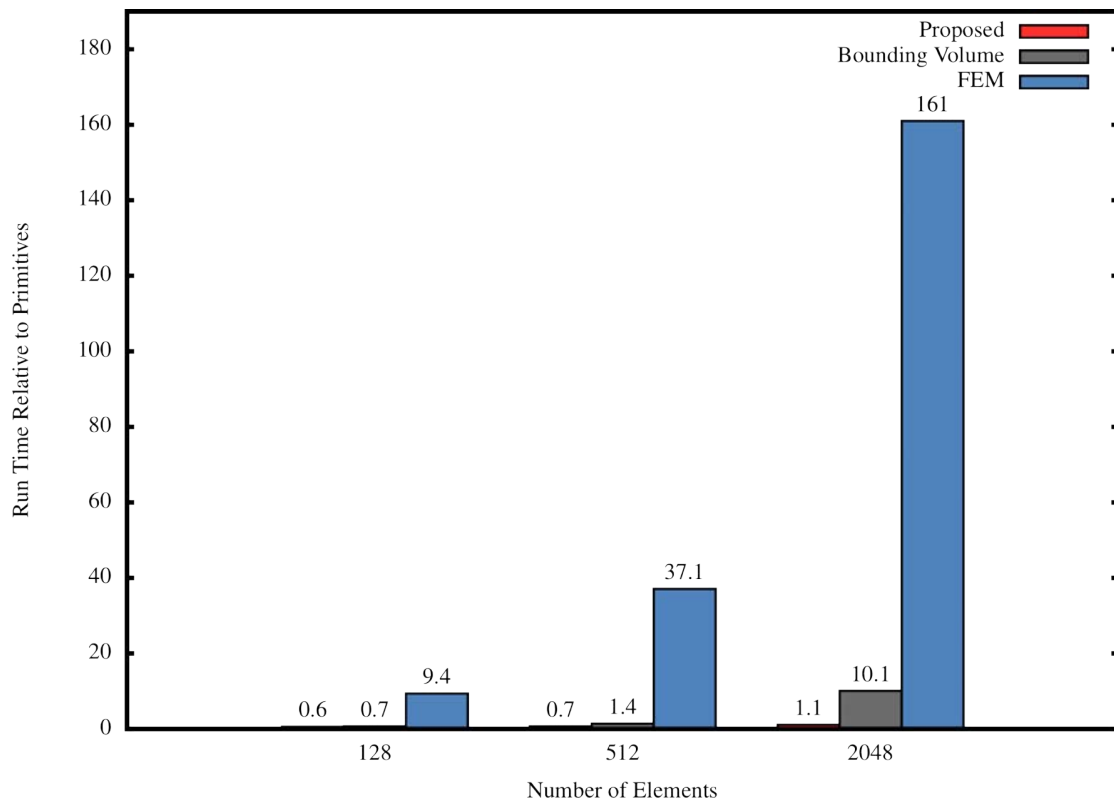
### ***Performance of Proposed Acceleration Method***

The performance of the proposed acceleration method was compared to both a 'brute force' FEM solution (as presented in Section 6.2) and a standard bounding volume implementation [111] in which once the earliest, valid primitive intersection is identified each individual triangle associated with the primitive is tested for intersection (as opposed to simply performing a nearest neighbour search).

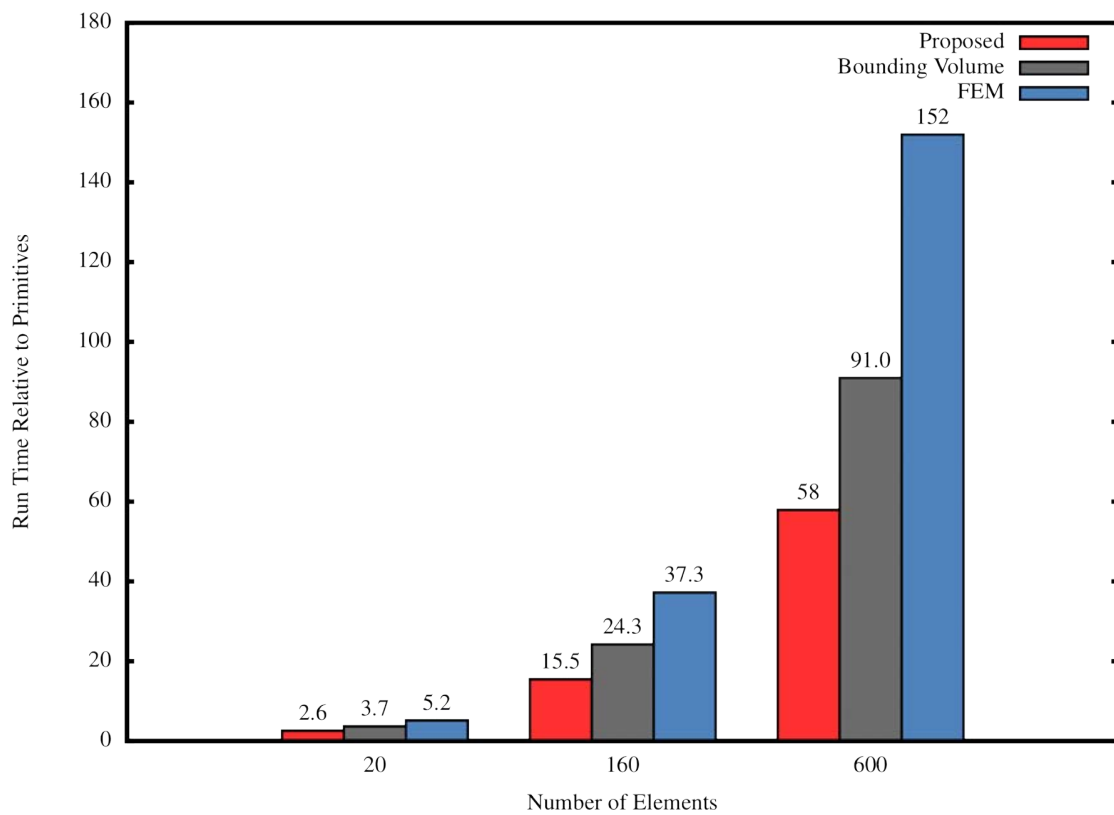
Both the C-77 and C-122 benchmarking geometries were used in this performance assessment. As only two triangles were used to model the rectangle in the C-122 geometry, it provided the highest performance for the proposed acceleration method as a nearest neighbour search using only two coordinates is required when rays launched from the sphere intersect the rectangle, and therefore intersection performance is close to that for a primitive. Coupled with the use of the triangle ray launching method (shown in Section 6.2.1 to have approximately 8% and 49% higher performance than ray launching from the rectangle and sphere primitives respectively), the proposed acceleration strategy has better performance than the straight primitive solution when less than 512 elements are used to model the sphere, as shown in Figure 30.

The C-77 benchmarking geometry, however, provides a more realistic assessment of the proposed acceleration strategy's performance with both the cylinder and the annulus being modelled using equivalent numbers of finite elements (i.e. 20, 160 and 600 triangular elements per object). Under these circumstances, an extensive nearest neighbour search is required when either of the bounding primitives is successfully intersected, and therefore run-times are observed to be 2.6 to 58 times greater (depending on the number of elements used) than for a purely primitive based solution. However, while only obtaining a fraction of the performance of a primitive based solution, the proposed acceleration structure exhibits 2.3 times higher performance on average than the brute force FEM solution, and an average of 1.5 times higher performance than a standard bounding volume implementation.





**Figure 30: Run times relative to using geometric primitives for the calculation of radiative view factors for the C-122 geometry using the proposed acceleration method, a standard bounding volume implementation and a 'brute force' FEM approach.**



**Figure 31: Run-times relative to geometric primitives for the calculation of radiative view factors for the C-77 geometry using the proposed acceleration method, a standard bounding volume implementation and a 'brute force' FEM approach.**

While the proposed acceleration method does provide significant performance benefits over a standard bounding volume implementation, it is important to note that the proposed strategy is not a suitable replacement for conventional bounding volumes under all conditions. The latter offer the flexibility for the associated FEMs to not only be coincident with the surface of the bounding volume, but also to be freely located anywhere within it, making it suitable for objects that cannot be adequately modelled using geometric primitives. Additionally, the use of a nearest neighbour search results in a degree of uncertainty in finite element intersection that is not experienced for bounding volumes where each individual finite element is tested for possible intersection.

While the proposed strategy does have its disadvantages compared to standard bounding volumes, the averaging effect of using a Monte Carlo simulation, the flexibility of employing an affine transformation, and the demonstrated run-time performance improvements, collectively make the proposed acceleration strategy a suitable alternative to bounding volumes for a wide range of radiative heat transfer geometries analysed using a GPU-based computing environment.

## **6.4 Conclusions**

Although FEMs have traditionally been regarded as the de facto representational method for three-dimensional radiative heat transfer geometries, from a computational performance perspective they are not necessarily the optimal choice for MC-RT simulations within a GPU-based computing environment.

Thus, while geometric primitives have been demonstrated to exhibit better performance than FEMs, the discretisation requirements of a particular heat transfer model may restrict their application. To circumvent this limitation, a new bounding volume strategy has been proposed that leverages the advantages of both geometric primitives and finite elements. Benchmarking studies have shown that it conservatively provides 2.3 times higher performance than a 'brute force' FEM strategy and 1.5 times higher performance than a traditional bounding volume approach within a GPU-based environment.

Given the demonstrated high performance GPU-based geometric primitives calculations, the following chapter will demonstrate the use of RayFactor to calculate the radiative heat transfer component for a complex heat transfer model describing an operational fibre drawing furnace.

## Case Study – A Fibre Drawing Furnace

---

It is worth noting that the need to accurately characterise radiative heat transfer within an operational polymer fibre drawing furnace was the primary driving force in the early development of the RayFactor software. As such, initial furnace modelling efforts were undertaken prior to the development of RayFactorCL for a GPU based computational environment, and therefore radiative view factors presented in this chapter have primarily been calculated using software optimised for use on a CPU based platform.

This chapter will detail the development of a fully conjugate heat transfer model for a drawing furnace used in the manufacture of a wide range of polymer optical fibres, and lays the groundwork for on-going research using an upgraded version of this furnace for the high temperature (up to approximately 1000-1100°C) drawing of sub-micron ‘meta-material’ fibres comprising a structured metal core contained within a glass sheath.

### 7.1 Optical Fibre Fabrication

Optical fibres are commonly employed in high-speed, high-bandwidth applications. Traditionally, these fibres are manufactured from high-purity silica, however recently polymer optical fibres (POFs) have emerged as a viable alternative [112-114]. These optical fibres are typically manufactured by feeding a large cylindrical preform into a drawing furnace where it is heated, softened and drawn under tension. During drawing, the heat distribution throughout the preform influences neck-down shape (i.e. the shape of the transition between the preform and the final fibre). When considering the drawing of micro-structured polymer optical fibres (mPOFs), which owe their light guidance properties to a pattern of holes running the length of the fibre, a comprehensive understanding of the heat transfer within the entire drawing furnace is required including the conductive/radiative heat transfer within the fibres themselves [112, 115]. With a suitable heat transfer model of the furnace, one can select furnace-heating characteristics so as to ensure the integrity of the hole structure as the preform

is drawn down to a fibre. Heat transfer to the preform in the furnace occurs by both thermal radiation (both through a quartz window and by re-radiation from the furnace walls) and by convection induced within the furnace. The combination of furnace geometry, a shape-changing preform and multiple heat transfer mechanisms makes this problem a challenging case study for the employment of MC-RT using primitives.

In order to generate a meaningful heat transfer furnace model, the radiative solution from MC-RT was coupled to a commercial computational fluid dynamics (CFD) software package (PolyFlow). Two cases were then considered. In the first, the furnace was operated such that the polymer preform never exceeded its glass transition temperature and, thus, never underwent any reduction in diameter. Here, all relevant view factors were determined by both direct numerical integration and MC-RT. Very good agreement was obtained between experimental temperature measurements (both at the centreline and on the surface of the preform) and results from the furnace heat transfer model. In the second case considered (for which unfortunately no experimental measurements were available from the operational fibre-drawing rig), it was assumed that the preform was raised above its polymeric glass transition temperature for a range of preform draw-down ratios ( $D_r = 4, 10$  and  $50$ ), noting that for a given preform, the larger the value of  $D_r$ , the finer the drawn fibre will be.

## **7.2 Overview of Furnace Operation**

The fibre drawing furnace (shown schematically in Figure 32) consists of a cylindrical section with a height of 0.18m and a radius of 0.032m. When the furnace is operating, a preform (0.006m in radius) enters at the top of the furnace through an adjustable iris while drawn fibre leaves at the bottom through a second iris (typically left partially open). Six external halogen lamps (under on-off control) provide radiant heating through a central quartz 'hot-zone' window, with the rest of the furnace being well insulated. Thus, the preform surface is heated by thermal radiation (both through the quartz window and by re-radiation from the furnace walls) and by convection induced within the furnace.

## **7.1 Modelling of the Drawing Furnace**

Modelling of the fibre-drawing furnace was conducted using PolyFlow, a commercial CFD simulation package specifically designed for simulating transport processes involving viscoelastic materials. PolyFlow was utilised in the modelling of the fibre drawing furnace to numerically solve the equations

governing polymer behaviour: mass, momentum and energy conservation, together with an appropriate set of physical properties.

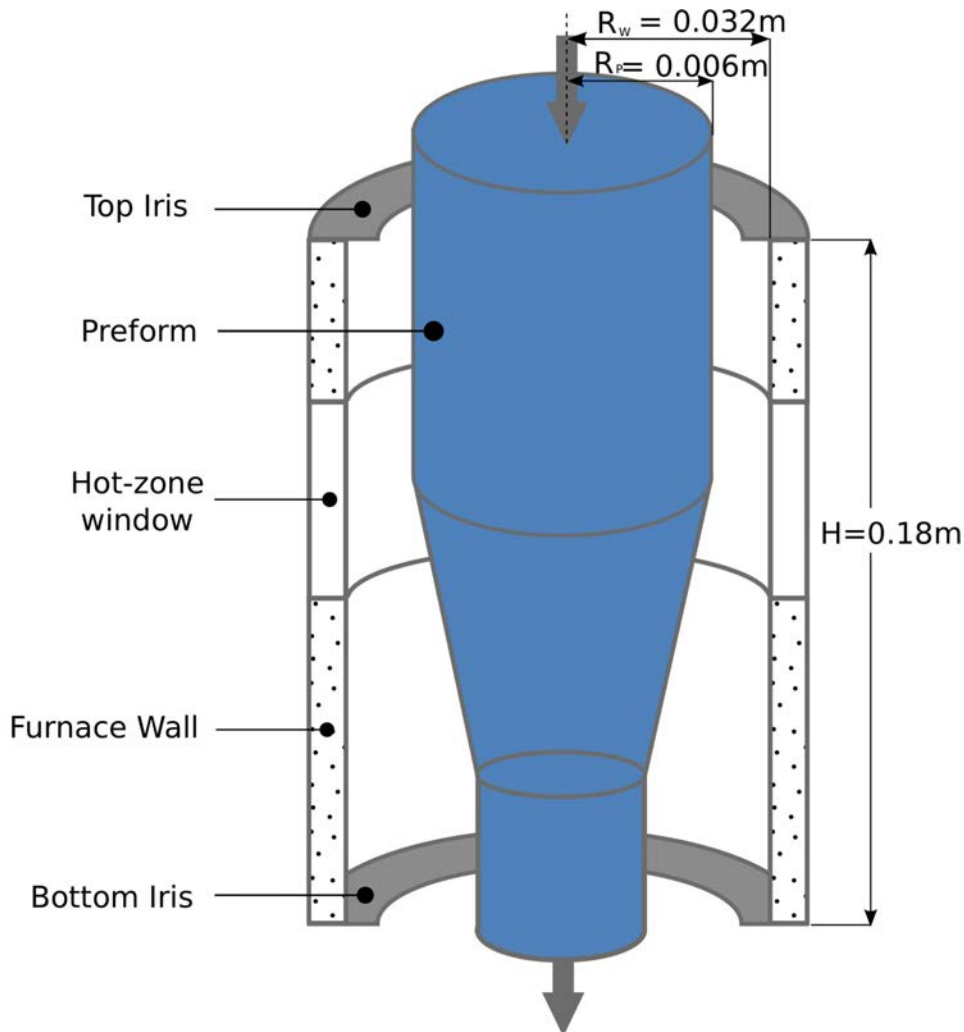


Figure 32: Schematic diagram of fibre drawing furnace

Although PolyFlow allows analysis of the various conductive and convective heat transfer modes within the drawing furnace, it does not have the functionality required for a comprehensive analysis of the radiative heat transfer components. Therefore, in order to create a realistic heat model, ‘cooperation’ between PolyFlow and RayFactor was required, and a methodology for interfacing the two programs was therefore developed (discussed later in section 7.1.3).

### 7.1.1 PolyFlow Modelling

Taking advantage of the geometry of the fibre-drawing furnace, a two-dimensional axi-symmetrical model was created in PolyFlow. After performing mesh refinement analysis to the convergent solutions, the computational domain was discretised into 5,760 quadrilateral elements to ensure results were mesh

size independent. This mesh was formed by 180 uniform cells along the length of the preform, 6 uniform cells across the preform radius and 26 uniform cells across the radial space between the preform and the furnace wall.

### ***Boundary Conditions on the Furnace Walls (Including the Irises)***

In order to correctly model the furnace, it is of critical importance to ensure that realistic boundary conditions are imposed. These boundary conditions are required at both the boundaries of the problem space (i.e. the furnace wall and the top and bottom irises), as well as at phase interfaces such as that between the solid preform and the surrounding gas.

In order to minimise heat lose, the operational furnace has several layers of insulating foam in addition to seals where the preform enters the furnace. As the top iris essentially provides an ‘air-tight’ seal<sup>13</sup>, a non-slip condition is applied at this point for the gas-phase ( $R_p \leq r \leq R_w$ ) as shown in Equation (105).

$$u_g = v_g = 0 \quad (105)$$

In addition to the conditions imposed on the radial and axial velocities at the iris surface,  $u$  and  $v$  respectively, continuous heat loss through the iris is accounted for using an overall heat loss coefficient  $U$  and a thermal boundary condition, as shown in Equation (106).

$$q_c = U(T - T_\infty) \quad (106)$$

Here, the ambient air temperature  $T_\infty$  was taken as 20°C (293 K), while the overall heat transfer coefficient  $U$  accounts for both conductive and convective thermal resistances at the top iris and was estimated to be approximately 10 W/m<sup>2</sup>.K, a figure obtained by matching model outputs and experimental results.

After travelling through the furnace, the preform (or drawn fibre) leaves through an adjustable metal vane system at the bottom iris. To prevent damage to the preform (or fibre) as it leaves the furnace, this vane is typically left slightly open

---

<sup>13</sup> The seals comprising the top iris were removed during temperature measurements to allow thermocouple leads to pass into the furnace. This creates a small (~1-2mm) air gap which is not characterised in the PolyFlow model.

(~1mm gap between the preform and vane) during normal operation. However, during these particular experimental measurements, this vane was left fully open for ease of operation, and therefore fully developed flow and temperature profiles are assumed at the bottom iris boundary, as shown in Equation (107).

$$\begin{aligned}\frac{\partial u}{\partial z}\bigg|_g &= \frac{\partial w}{\partial z}\bigg|_g = 0 \\ \frac{\partial T}{\partial z}\bigg|_g &= 0\end{aligned}\tag{107}$$

In terms of radiative heat transfer, the bottom iris is treated as a black surface at the ambient temperature  $T_\infty = 293K$ . In a similar manner to that employed at the top iris, a non-slip condition is imposed on the velocity at the furnace wall as shown in Equation (105), however, here the experimentally measured temperature profile along the wall  $T_w(z)$  is used as the thermal boundary condition, as shown in Equation (108).

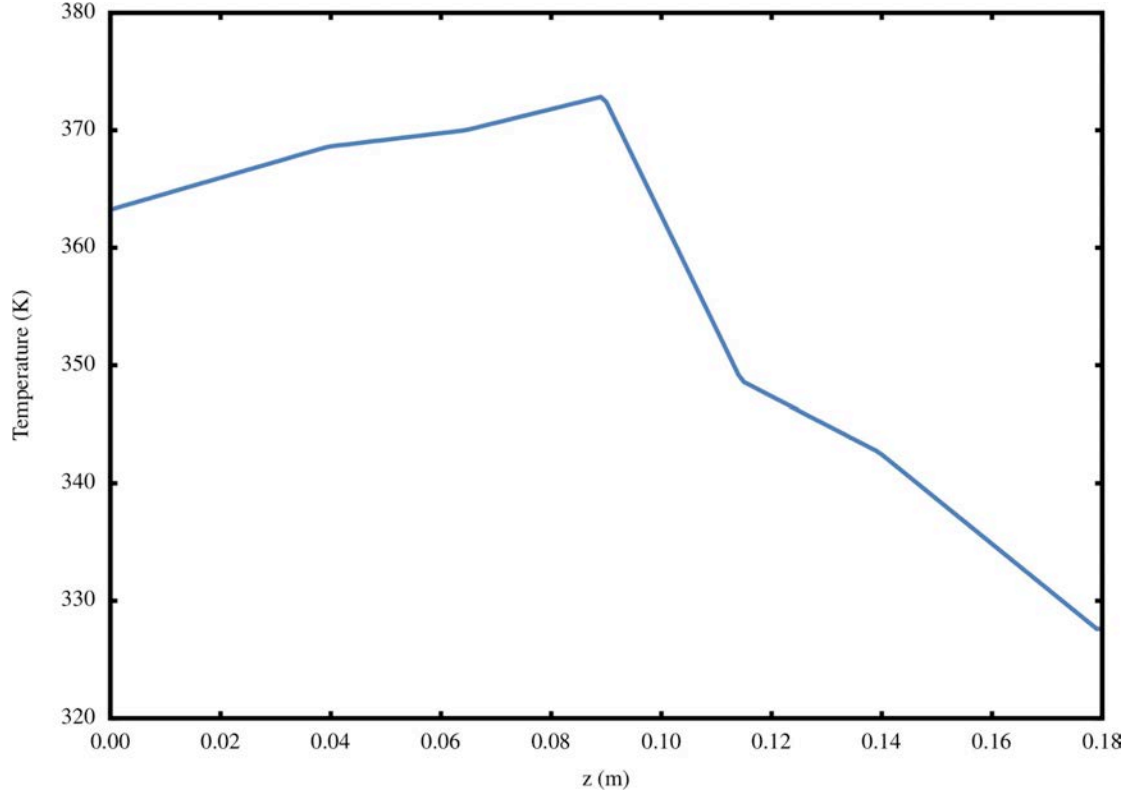
$$T = T_w(z)\tag{108}$$

The experimentally measured temperature profile of the furnace wall during non-deforming draw is shown in Figure 33. For the deforming cases this profile was again utilised, however, here a constant 70 K was added to ensure temperatures within the furnace for preform deformation to occur.

### ***Boundary Conditions at the Preform Interface***

For the interface between the gas phase and the preform ( $r = R, 0 \leq z \leq H$ ), a velocity continuity condition is applied, as shown in Equation (109).

$$\begin{aligned}u_g &= u_p \\ v_g &= v_p\end{aligned}\tag{109}$$



**Figure 33: Experimentally measured temperature profile along the furnace wall.**

From a heat transfer perspective, the gas phase is treated as a (radiatively) non-participating medium and net heat flux continuity is applied at the interface, as shown in Equation (110).

$$\kappa \frac{\partial T}{\partial n} \Big|_g + q_r = \kappa \frac{\partial T}{\partial n} \Big|_p \quad (110)$$

Here  $n$  is the local normal at the preform surface while  $q_r$  is the net radiative heat flux at the preform surface, as calculated using RayFactor (discussed in detail in Section 7.1.3).

In order to determine the free surface geometry of the preform as it deforms, kinematic and dynamic boundary conditions [114] are applied to perform a force balance on the preform surface, as shown in Equation (111).



$$\begin{aligned}
\mathbf{V}_g \cdot \mathbf{n} &= 0 \\
\boldsymbol{\sigma}_p : \mathbf{n}\mathbf{s} &= 0 \\
\boldsymbol{\sigma}_p : \mathbf{n}\mathbf{n} &= K\gamma
\end{aligned} \tag{111}$$

Where  $\mathbf{V} = u\mathbf{i} + v\mathbf{j} + w\mathbf{k}$  is the velocity vector,  $\mathbf{n}$  and  $\mathbf{s}$  denote the local normal and tangential vectors to the preform surface,  $\boldsymbol{\sigma}_p$  is the traction tensor acting on the preform surface,  $\gamma$  is the surface tension coefficient of the preform material (PMMA or polymethylmethacrylate) and  $K$  is the sum of the principal curvatures of the free surface.

### ***Boundary Conditions for the Preform***

Now turning attention to the preform itself, at the centre-line ( $r = 0, 0 \leq z \leq H$ ) an axi-symmetric boundary condition is applied, as shown in Equation (112).

$$\begin{aligned}
u_p &= 0 \\
\left. \frac{\partial v}{\partial r} \right|_p &= 0 \\
\left. \frac{\partial T}{\partial r} \right|_p &= 0
\end{aligned} \tag{112}$$

At the top of the preform ( $z = 0, 0 \leq r \leq R_p$ ), the velocity boundary conditions represent the preform being fed into the furnace with a constant axial velocity ( $V_i = 0.01 \text{ m/s}$ ), as shown in Equation (113).

$$\begin{aligned}
u_p &= 0 \\
v_p &= V_i
\end{aligned} \tag{113}$$

Using a heat transfer coefficient ( $h_p$ ) to characterise the heat transfer rate from the top of the preform ( $z = 0$ ) to the surrounding environment, a thermal boundary condition here may be used to account for heat loss due to conduction along the preform (and the free convection away from the cold preform before it enters the furnace), as shown in Equation (114).

$$\kappa \frac{\partial T}{\partial n} \Big|_p = -h_p(T - T_\infty) \quad (114)$$

Here the heat transfer coefficient  $h_p$  was determined using the correlation for heat transfer across the base of an infinitely long fin, as shown in Equation (115).

$$h_p = \sqrt{\frac{2h\kappa_p}{R_p}} \quad (115)$$

where  $h$  is the heat transfer coefficient for natural (or ‘free’) convection from a vertical cylinder which was estimated as 7 W/m<sup>2</sup>.K from the literature [115].

Finally, the boundary conditions at the bottom of the preform ( $z = H, 0 \leq r \leq R_p$ ) need to be considered. Here, the preform is being drawn down to a fibre and as such, experiences zero shear force and has an axial exit velocity  $V_o$  that is determined by the draw ratio<sup>14</sup>. For the simpler case of a non-deforming preform (examined in Section 7.2), the axial exit velocity is equal to the feeding speed  $V_i$ . A fully developed temperature field is assumed as the thermal boundary condition at the bottom of the preform, as shown in Equation (116).

$$\frac{\partial T}{\partial z} \Big|_p = 0 \quad (116)$$

In addition to the boundary conditions presented above, in order to accurately represent the behaviour of both the PMMA preform and the furnace air space, relevant material properties are required. These were entered directly into PolyFlow where possible, or else implemented as external User Defined Functions (UDFs) in the C Language Integrated Production System (CLIPS). The material properties used for the PMMA preform and the air within the furnace are provided in the following sections.

---

<sup>14</sup> Draw ratio is the ratio cross-sectional area of the preform to the cross-sectional area of the final fibre.

### **Physical Properties of PMMA**

The temperature variations likely to be experienced within the PMMA preform indicate that viscosity and heat capacity need to be considered as temperature dependent properties. The viscosity was determined by laboratory testing on the specific grade of PMMA used for optical fibre manufacture in the experimental drawing furnace, and may be determined as a function of temperature as shown in Equation (117).

$$\eta = 0.2661e^{2687.8/(T-273)} \quad (117)$$

The heat capacity of PMMA was modelled using the data reported by Bu et al [116] and can be determined using the glass transition temperature of PMMA ( $T_g = 393K$ ), as shown in Equation (118).

$$c_p = \begin{cases} 1420 + 4 \times (T - 298) & T < T_g - 20 \\ 1720 + 16.5 \times (T - 373) & T_g - 20 \leq T \leq T_g \\ 2050 + 5.5(T - T_g) & T > T_g \end{cases} \quad (118)$$

The density and thermal conductivity of the PMMA preform were each assumed to be constant, with values of  $\rho_p = 1170 \text{ kg/m}^3$  and  $\kappa_c = 0.17 \text{ W/m.K}$  respectively, as suggested in the literature [115].

Additionally, it is suggested that PMMA can be regarded as optically thick [117], allowing for only short-range transmission of thermal radiation. In this case, rather than analysing the radiative heat flux within the preform, the Rosseland approximation [118] was used to model radiative transfer as an effective conduction term, referred to as 'radiative conductivity',  $\kappa_r$  as shown in Equation (119).

$$\kappa_r = \frac{16n^2\sigma T^3}{3\alpha_m} \quad (119)$$

The infrared absorption spectrum for PMMA as presented by Xue et al [115] indicated that the mean absorption coefficient  $\alpha_m$  as a function of wavelength ( $\lambda$ ) may be calculated using a linear band model as shown in Equation (120).

$$\alpha_m = \begin{cases} 124\lambda - 224 \text{ cm}^{-1} & 1.9 \mu\text{m} \leq \lambda \leq 2.35 \mu\text{m} \\ 25.5\lambda + 8 \text{ cm}^{-1} & 2.35 \mu\text{m} \leq \lambda \leq 4 \mu\text{m} \\ 110 \text{ cm}^{-1} & 4 \mu\text{m} \leq \lambda \leq 7 \mu\text{m} \\ 138 - 4\lambda \text{ cm}^{-1} & 7 \mu\text{m} \leq \lambda \leq 13.3 \mu\text{m} \end{cases} \quad (120)$$

Once the radiative conductivity is calculated using the refractive index ( $n = 1.55$ ) and mean absorption coefficient ( $\alpha_m$ ), heat transfer within the preform can be described using a combined conductivity, as shown in Equation (121).

$$\kappa = \kappa_c + \kappa_r \quad (121)$$

It should be noted, however, that for the operating temperature range experienced in the drawing of PMMA optical fibres, radiative heat transfer is predicted to account for only 4% of the total heat transfer within the preform. This contribution is expected to increase when an upgraded version of this furnace is used for drawing 'metamaterials' fibres (comprising a structured metal core within a glass sheath) at temperatures in the range 1000-1100°C.

### ***Physical Properties of Air***

Temperature dependent properties of air have been well studied and are widely available in the literature [1]. Due to the modest temperatures changes within the furnace, the density of air was taken as constant with a value of  $\rho_a = 0.9980 \text{ kg/m}^3$ . The thermal expansion (buoyancy) of the air was approximated using the Boussinesq approximation, as shown in Equation (122) using a reference temperature  $T_0 = 350 \text{ K}$ .

$$\rho g = \rho_0 g [1 - \beta(T - T_0)] \quad (122)$$

where  $g$  is gravity ( $g = 9.81 \text{ m/s}^2$ ) and  $\beta$  is the thermal expansion coefficient. Temperature dependent functions for the viscosity, thermal conductivity and

heat capacity of the air were developed by low-order curve fitting to tabulated data available in the literature [1] and are listed in Equation (123).

$$\begin{aligned}\eta &= 2.075 \times 10^{-5} + 3.52 \times 10^{-8}(T - 350) \\ \kappa_c &= 0.03 + 7.24 \times 10^{-5}(T - 350) \\ c_p &= 1009 + 0.1(T - 350)\end{aligned}\tag{123}$$

For the purposes of modelling radiative heat transfer within the furnace (which will be discussed in the following section), the gas phase was considered to be a non-participating medium. This required the radiative analysis to only consider surface-surface exchanges, greatly reducing the complexity of the problem.

### 7.1.2 Modelling the Furnace in RayFactor

Radiative heat transfer within the fibre-drawing furnace was characterised using radiative view factors calculated by RayFactor.

Although not required by the RayFactor software, the preform and furnace walls were each discretised using 180 'slices' and represented using cylinder and frustum primitives, while the regions of the top and bottom iris each discretised into 26 annular elements so as to allow a direct comparison of the view factors with those obtained via numerical integration (discussed in Section 7.2) and a one-to-one correspondence to the mesh used within PolyFlow (which used this level of discretisation to ensure that the numerical results were grid independent).

### 7.1.3 Interfacing PolyFlow and RayFactor to Model Radiative Heat Transfer

Within the fibre drawing furnace, the preform surface may either gain or lose radiative energy to the surrounding surfaces depending on surface radiative properties, the relative temperature differences and how the surfaces 'view' each other.

Modelling of the radiative heat transfer to the preform surface was complicated by the fact that in PolyFlow, radiative heat transfer can only be introduced via the equivalent of an infinitely long heating surface. This limitation was managed by first calculating the net radiative flux  $q_r$  across the preform surface and subsequently deriving an equivalent heating temperature profile  $T_\sigma(z)$  for a

'dummy' infinitely long heating surface which can be used as the thermal boundary condition at the preform-gas and gas-preform interfaces.

In order to calculate this heating temperature profile  $T_o(z)$ , one must first determine the radiative flux  $q_r$  across the surface of the preform. This was completed by applying the net radiation method [117, 119] over the furnace enclosure. Here, the furnace surfaces are discretised into either ring elements (for the preform and furnace wall) or annular discs (for the top and bottom irises) small enough to consider the temperature as constant across each element. The net radiative heat flux for each element  $q_{r,k}$  is then calculated using the outgoing and incoming fluxes on the element  $q_{r,o,k}$  and  $q_{r,i,k}$  respectively, as shown in Equation (124).

$$q_{r,k} = q_{r,o,k} - q_{r,i,k} \quad (124)$$

By assuming that each element is a grey diffuse emitter with constant emissivity the outgoing radiative heat flux may be expressed as shown in Equation (125).

$$q_{r,o,k} = \varepsilon_k \sigma T_k^4 + (1 - \varepsilon_k) q_{r,i,k} \quad (125)$$

Here  $q_{r,i,k}$  is the sum of the radiative heat that is emitted by all other ( $N$ ) elements in the furnace and is incident on the surface of element  $k$  and may be described as shown in Equation (126).

$$q_{r,i,k} = \sum_{j=1}^N F_{j-k} q_{r,o,j} \quad (126)$$

By applying Equations (124) - (126) to each of the  $N$  elements within the furnace enclosure, a system of  $N$  linear equations may be obtained which if the temperature of the element is known (i.e. furnace walls, preform surface and bottom iris) has the form shown in Equation (127).

$$\sum_{j=1}^N [\delta_{kj} - (1 - \varepsilon_k) F_{j-k}] q_{r,o,j} = \varepsilon_k \sigma T_k^4 \quad (127)$$

where  $\delta_{kj}$  is the Kronecker delta. For elements where the net flux  $q_k$  is given, such as the insulated top iris, Equation (128) is used.

$$\sum_{j=1}^N [\delta_{kj} - F_{j-k}] q_{r,o,j} = q_k \quad (128)$$

Once the view factors are known, a system of linear equations can be constructed from Equations (127) and (128) and can be solved numerically using Gauss-Seidel iteration to determine the outgoing radiative heat flux of each element  $q_{r,o,k}$ . This may then in turn be used to calculate the incoming radiative heat flux with Equation (126) and the net radiative heat flux using Equation (124).

As PolyFlow requires the radiative heat transfer to the preform to be expressed in the same form as for two infinitely long concentric cylinders, as shown in Equation (129), the radiative heat flux needs to be converted to an ‘equivalent’ radiative heating temperature  $T_{\sigma,k}$ .

$$q_{r,k} = \varepsilon_k \sigma (T_k^4 - T_{\sigma,k}^4) \quad (129)$$

This equivalent radiative heating temperature may be determined for each element of the preform through manipulation of Equation (125) to obtain the relation shown in Equation (130).

$$T_{\sigma,k} = \begin{cases} \sqrt[4]{\frac{q_{r,i,k}}{\sigma}} & \text{for } q_{r,i,k} \geq 0 \\ -\sqrt[4]{\left| \frac{q_{r,i,k}}{\sigma} \right|} & \text{for } q_{r,i,k} < 0 \end{cases} \quad (130)$$

Using the methodology presenting in this section to develop and incorporate a suitable radiative heat transfer model into PolyFlow, two scenarios of furnace operation were examined. In the first case, a non-deforming draw was examined,

where the PMMA preform is heated to a temperature below its glass transition temperature, so that the diameter of the preform does not change as it passes through the furnace. Here the results of the model were compared to experimental measurements taken from the furnace detailed in Section 7.2. In the second case, the heating of the preform above its glass transition temperature was considered with the preform being (computationally) drawn under tension to produce a PMMA fibre. These two cases are discussed in the following sections.

#### 7.1.4 Emissivity

From Equation (127), it is apparent that radiative heat transfer within the furnace is dependent on the emissivity of the participating surfaces and therefore this material property must be considered.

First consider the preform itself. The linear band model for the absorptivity coefficient presented in Equation (120) indicates that PMMA is highly absorptive around the wavelength of maximum emission ( $\lambda_{peak} \approx 6.5 \mu m$ , as calculated by the Wien displacement law) for the anticipated furnace operating temperature region (298-470K). Therefore a constant hemispherical emissivity ( $\varepsilon_p = 0.96$ ) was used in this modelling as suggested in the literature [113, 117].

Next consider the glass wall of the furnace. Sayles and Caswell [120] presented the following relationship for the estimation of the hemispherical spectral emissivity of a glass cylinder:

$$\varepsilon_\lambda = 0.885(1 - e^{-\delta_\lambda}) \quad (131)$$

where  $\delta_\lambda = 2\alpha_\lambda R$  is the optical thickness of a glass cylinder of radius  $R$ . However, for cylinders with a radius greater than 0.25 cm, a precise value of the absorption coefficient is not required [121] as the asymptotic value ( $\varepsilon_\lambda = 0.885$ ) can be used. This asymptote is reached for wavelengths greater than about  $3 \mu m$ , which from Wien's displacement law is applicable at temperatures experienced in the fibre drawing furnace.

Finally, the bottom iris must be considered. As this adjustable vane was fully open during the experimental measurements used in this study, the bottom iris may be considered as essentially a blackbody with a hemispherical emissivity of  $\varepsilon = 1$ . Given these emissivities one must now determine the furnace views



factors in order to satisfy Equations (127) and (128) and therefore calculate the net radiative fluxes within the furnace.

### 7.1.1 Calculating Radiative View Factors Within the Furnace

Prior to inclusion in the PolyFlow furnace model, the sensitivity of the view factor results to ray density was examined by calculating the view factor matrix for ray densities in the range  $10^3 \leq \rho_{ray} \leq 10^6$  and comparing the results to those obtained through numerical integration (discussed later in Section 7.2). The relative run-times for the calculation of the complete drawing furnace radiative view factor matrix is shown in Table 18.

**Table 18: Relative run-times for RayFactor (a CPU based environment using 2 x 2.26 GHz Intel E5520 and RayFactorCL (a GPU-based environment using a NVidia GTX580 system)**

| Ray Density<br>(rays/unit <sup>2</sup> ) | RayFactor<br>(2 x Intel E5520) | RayFactorCL<br>(NVidia GTX580) |
|--|--------------------------------|--------------------------------|
| $10^3$                                   | 0.11                           | 0.01                           |
| $10^4$                                   | 1.00                           | 0.11                           |
| $10^5$                                   | 9.87                           | 1.07                           |
| $10^6$                                   | 98.61                          | 10.74                          |

Here the run-time for RayFactor with a ray density of  $10^4$  was approximately 3.36 minutes, with the run-time for a ray density of  $10^5$ , which was ultimately selected for modelling being 33.2 minutes<sup>15</sup>. Although this furnace modelling work was largely completed prior to the development of the GPU based RayFactorCL software, it is worth noting that RayFactorCL had a computational time of only 3.6 minutes using a ray density of  $10^5$ . This increase in computational speed will undoubtedly prove extremely beneficial in future research into high temperature drawing of sub-micron ‘meta-material’ fibres, particularly if a large number of iterations is required to obtain heat transfer model convergence (discussed later in Section 7.3).

## 7.2 Non-deforming Preform Case

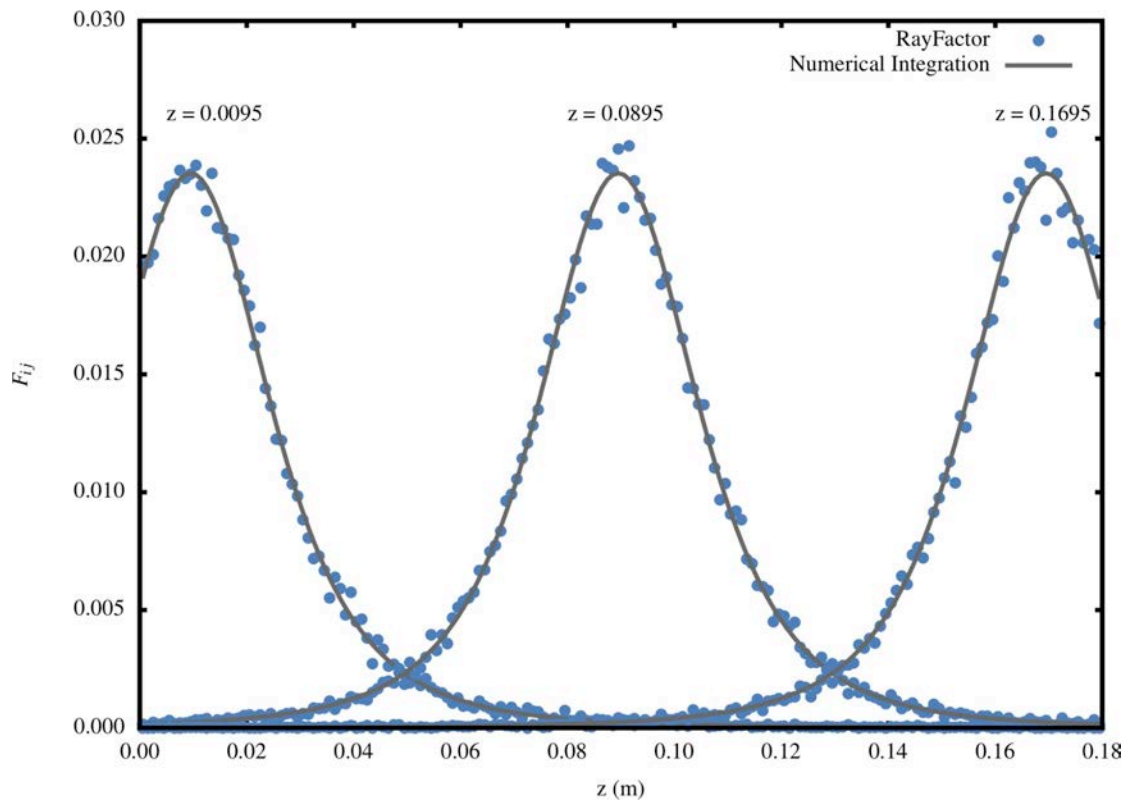
In the non-deforming case, the temperature of the preform does not exceed its glass transition temperature and therefore retains its dimensions as it travels

---

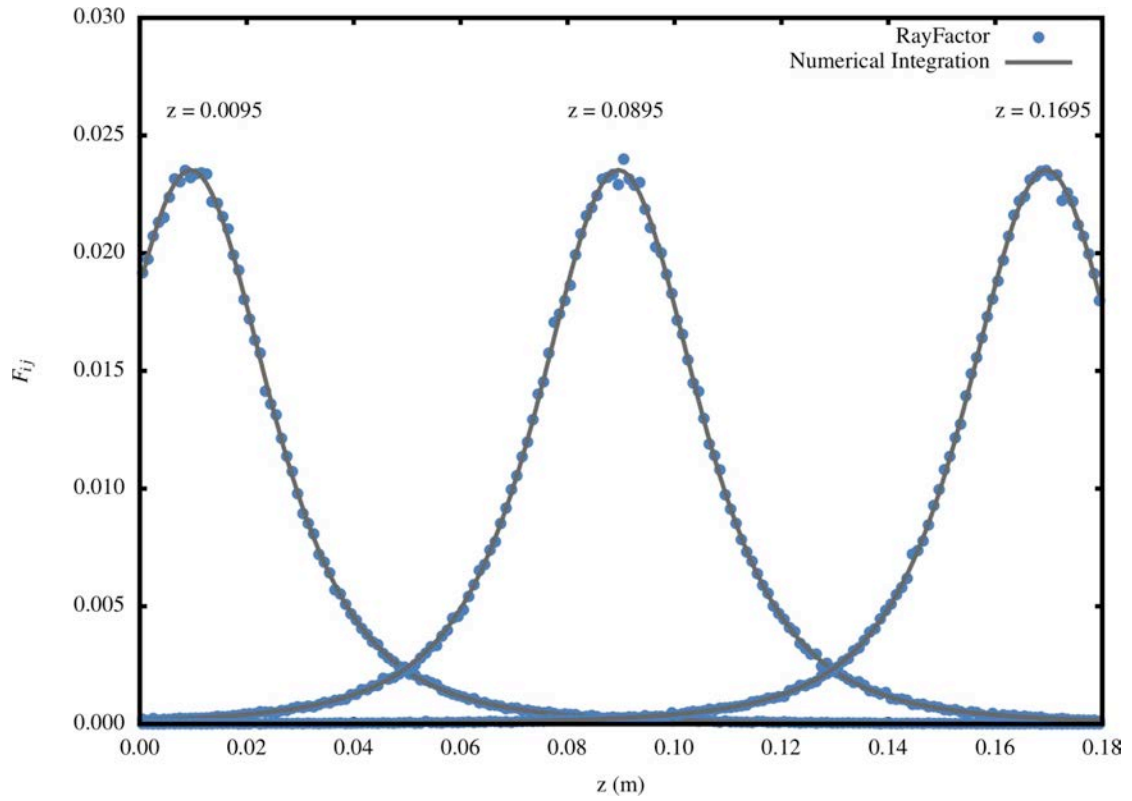
<sup>15</sup> Early versions of RayFactor (prior to the optimisations discussed in Chapter 4) required a computational time in excess of 6 hours at ray densities of  $10^5$ .

through the fibre-drawing furnace. In this case, the preform may be modelled within RayFactor using only cylindrical primitives.

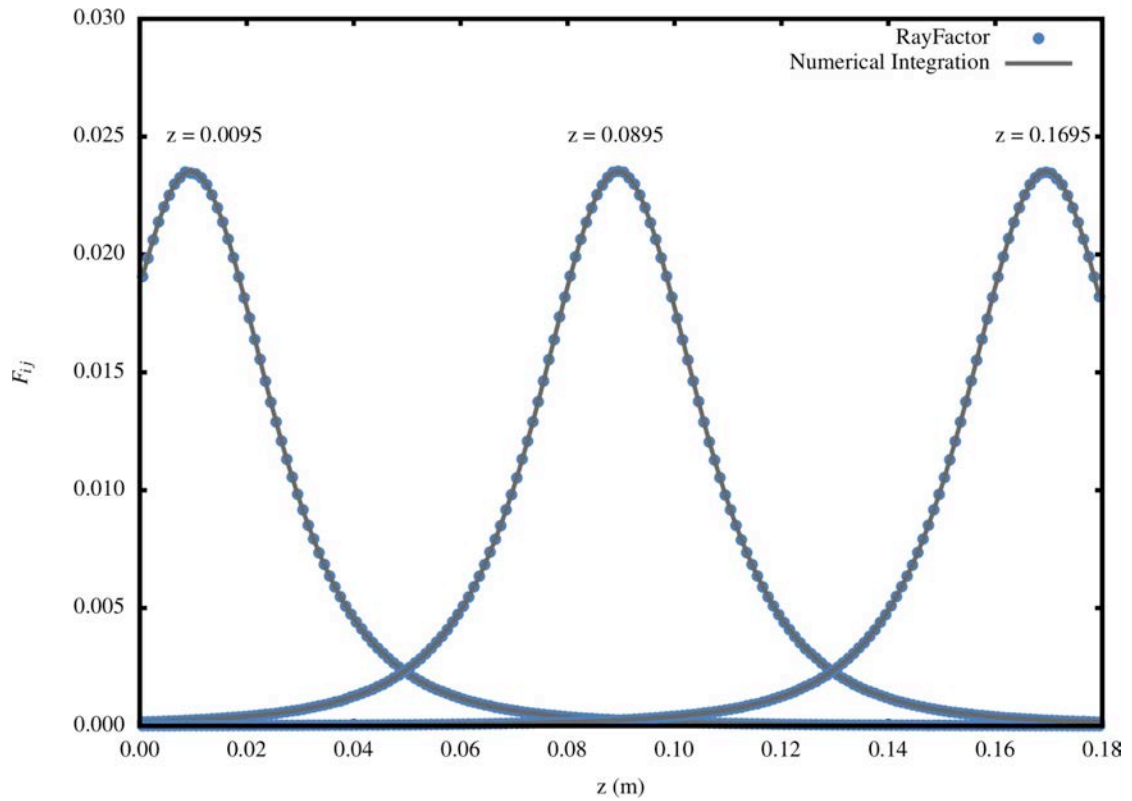
RayFactor results using ray densities of  $10^3$ ,  $10^4$  and  $10^5$  are given in Figure 34 through to Figure 36, where in each case the view factors calculated by RayFactor (shown as discrete symbols) are compared to those obtained by numerical integration (shown as solid lines). Although all possible view factors for this furnace enclosure were determined, values are only given here between the preform surface and the furnace wall (observed from three different axial positions  $z = 0.0095$ ,  $0.0895$ , and  $0.1695$  of one surface to the entire other surface or to itself) so as to provide a direct comparison between the two methods. Here  $F_{p-f}$  refers to the view factor from a position on the preform surface to the furnace wall, while  $F_{f-f}$  refers to the view factor from a position on the furnace wall to itself. The  $z$  values chosen show the essential geometric symmetry of a system that comprises a constant diameter cylinder located along the axis of a cylindrical enclosure. Under conditions where the preform undergoes neck-down to form a smaller diameter fibre, however, this symmetry is broken, as discussed later in Section 7.3.



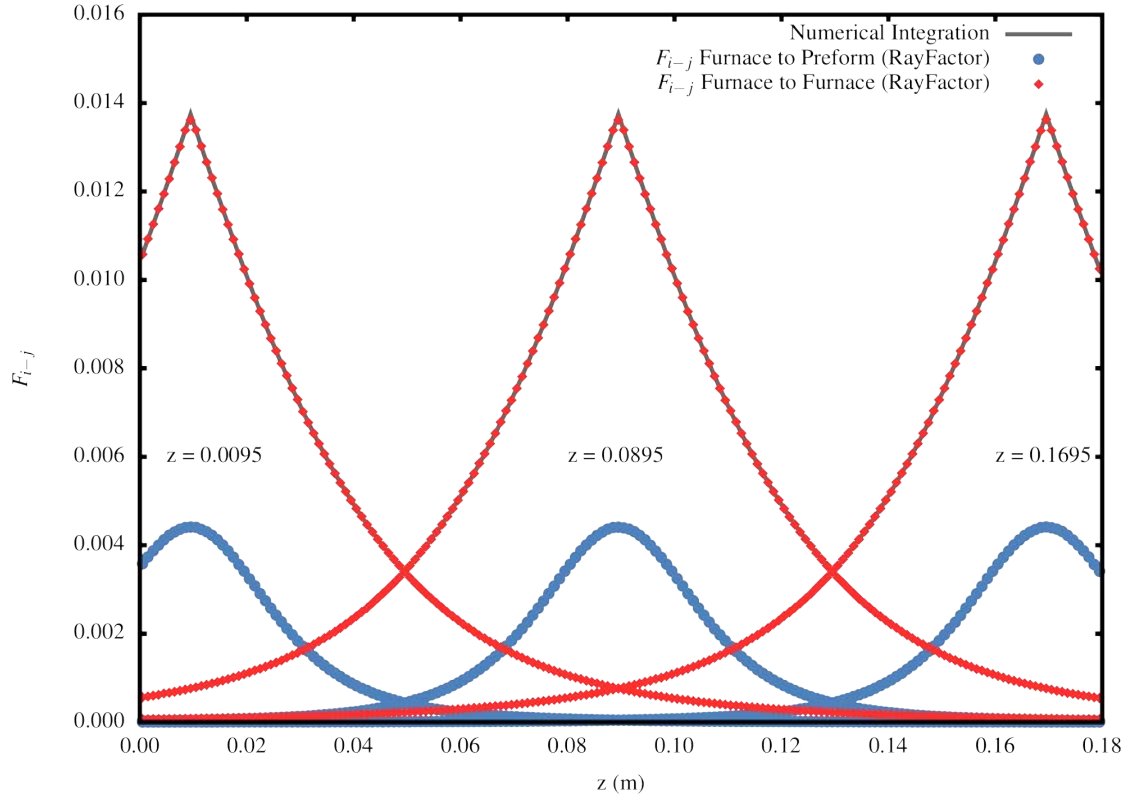
**Figure 34: View factors from three positions ( $z_p = 0.0095$ ,  $0.0895$ ,  $0.1695$ ) on the preform to the furnace wall with ray density  $\rho = 10^3$  rays per unit area.**



**Figure 35:** View factors from three positions ( $z_p = 0.0095, 0.0895, 0.1695$ ) on the preform to the furnace wall with a ray density  $\rho = 10^4$  rays per unit area.



**Figure 36:** View factors from three positions ( $z_p = 0.0095, 0.0895, 0.1695$ ) on the preform to the furnace wall with a ray density  $\rho = 10^5$  rays per unit area.

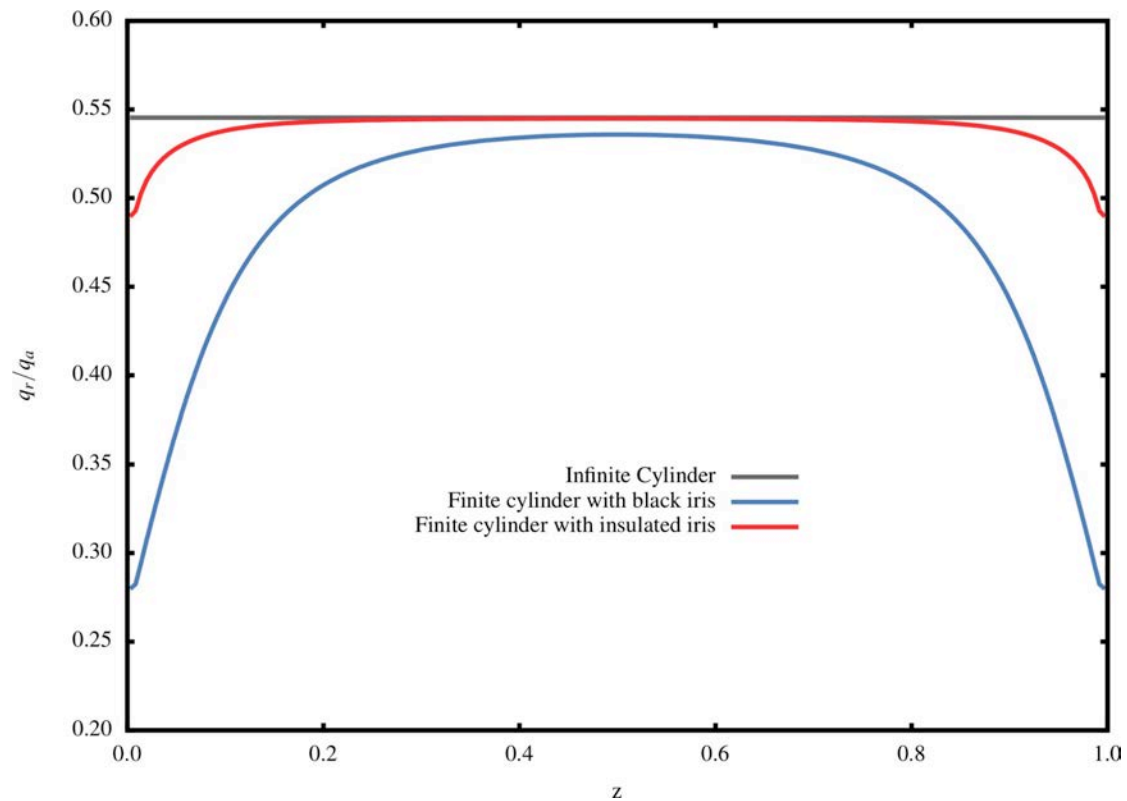


**Figure 37: View factors from three positions ( $z_f = 0.0095, 0.0895, 0.1695$ ) on the furnace wall to the preform surface and to itself; ( $\rho = 10^5$  per unit area; cylindrical preform).**

The accuracy of the view factor calculations is implied by both the close agreement between all  $F_{i-j}$  profiles obtained from the numerical integration and ray-tracing methods, and the ‘summation rule’ where the total of the view factors for each surface within the enclosure is essentially unity. Near perfect agreement between view factors calculated by RayFactor and those obtained through numerical integration was obtained at a ray density of  $10^5$  and therefore this ray density was utilised for all subsequent view factor calculations.

Using these view factors, as a first check on the radiative heat transfer methodology to be employed for the drawing furnace, this mode of heat transfer was calculated using the net radiation method between a stationary, finite length, solid preform, and the furnace surfaces where the furnace heating wall was at a uniform temperature and the top and bottom irises are either insulated or treated as black surfaces. For this simplified case, if the aspect ratio of the cylindrical preform is large enough, then the net radiative heat flux profile over the central section of the preform surface should approximate that for the limiting case of two infinitely long concentric cylinders. Figure 38 shows the results when a stationary PMMA preform ( $R = 6$  mm; constant hemispherical total emissivity  $\varepsilon_p$  of 0.96 [113, 117]; initially at a uniform temperature of 293 K) is heated inside a furnace (with a constant emissivity  $\varepsilon = 0.885$  [113]; height  $H =$

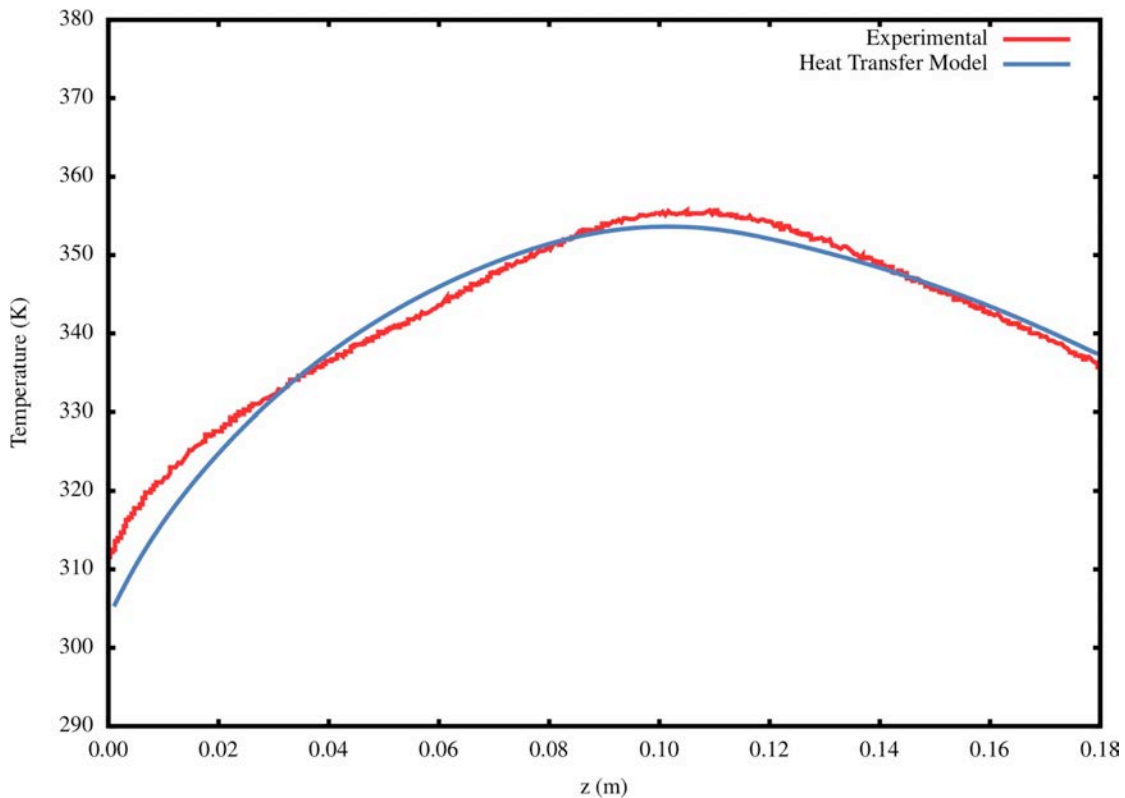
180 mm; radius  $R_w = 32$  mm; uniform wall temperature  $T_w$  of 373 K). Note that in Figure 38, the net heat flux for the preform has been scaled by the emitted flux  $q_b$  from a black surface with the same uniform wall temperature, that is by  $q_b = \sigma_0 T_w^4$ , while the axial length has been scaled by  $H$ . With an aspect ratio for the cylindrical preform of 30, when both irises are treated as insulated surfaces, the net radiative heat flux over the middle portion of the preform surface is consistent with the analytical result for two infinitely long concentric cylinders. However, if both irises are treated as black surfaces, then due to heat loss through the irises, the overall net radiative heat flux to the preform surface is reduced along the length of the preform, with the reduction becoming more marked when approaching the ends of the preform. This difference indicates the importance of correctly describing the boundary conditions in such a modelling exercise.



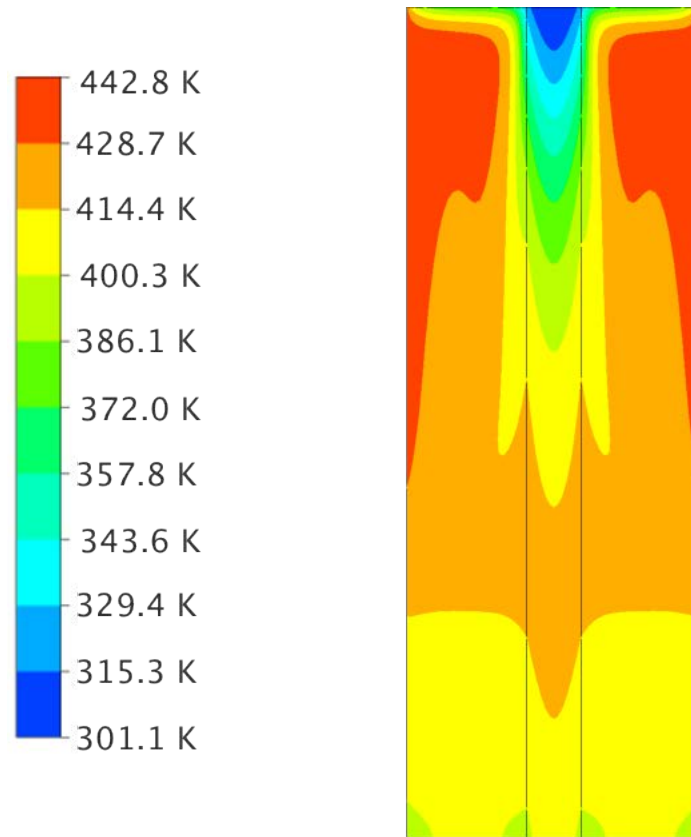
**Figure 38: Net radiative heat flux profiles along a stationary cylindrical preform within a furnace with a uniform heating wall temperature; two different thermal boundary conditions are used for the irises.**

Once testing of the view factor and radiative heat transfer calculations was complete, results from the fully conjugate furnace heat transfer model were compared with experimental temperature measurements taken (via imbedded

thermocouples) at the preform surface and centreline. For this moving preform case, Figure 39 demonstrates that a close match between experimental data and model predictions was obtained. The residual error between the experimental and modelling results are felt to be the result of the unmodelled ( $\sim 1\text{-}2\text{ mm}$ ) air gap around the preform as it passed through the iris at the furnace entry. This level of model/experimental discrepancy was deemed acceptable in terms of the intended model use for furnace redesign so as to maximise polymer fibre-drawing flexibility, and subsequent upgrading to metamaterial fibre manufacture. The resulting temperature profile of a vertical cross-section of the fibre-drawing furnace is shown in Figure 40, showing temperatures within both the solid preform and the surrounding gas phase.



**Figure 39: Comparison of experimental and furnace model results for preform surface temperature.**



**Figure 40: Temperature profile of a vertical cross-section of the furnace for non-deforming draw.**

### 7.3 Deforming Preform Case

In the deforming case, using the validated furnace model, the preform is heated above its glass transition temperature, which when combined with the draw tension applied, results in the preform being drawn down into a fibre. The calculation of radiative view factors by direct numerical integration becomes challenging for the case of a deforming preform due to the variation of the preform diameter along the length of the furnace. It is here that the real value of the MC-RT approach becomes evident. The validated heat transfer furnace model was used over a range of preform draw ratios ( $D_r = 4, 10$  and  $50$ ).

An iterative approach was required, however, as at each solution step, the preform shape had changed, meaning that the view factor profiles and thus the net radiative heat flux profile to the preform had changed. The computational flow sheet for iteratively converging the furnace heat transfer model for a deforming preform case is shown in Figure 41.

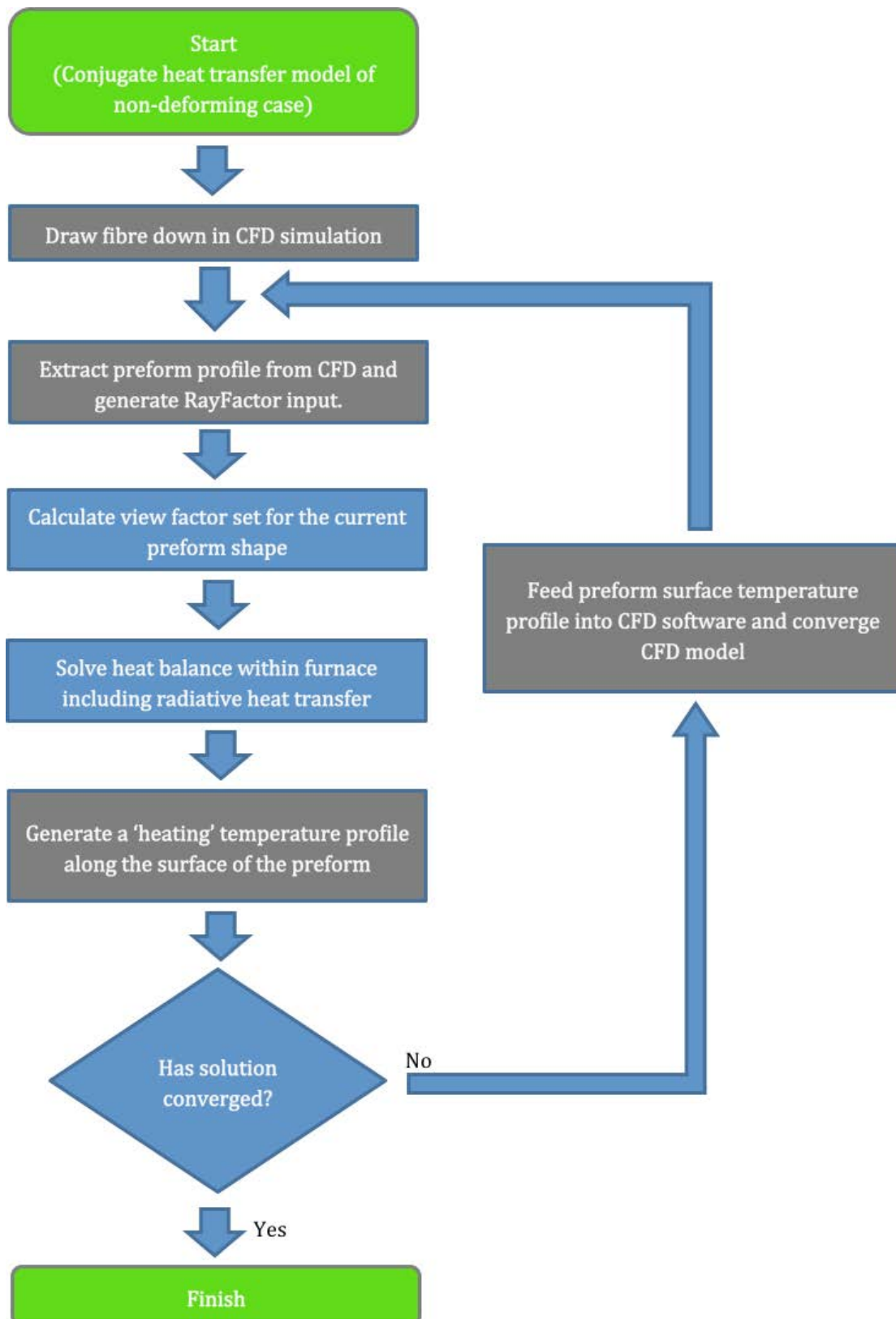
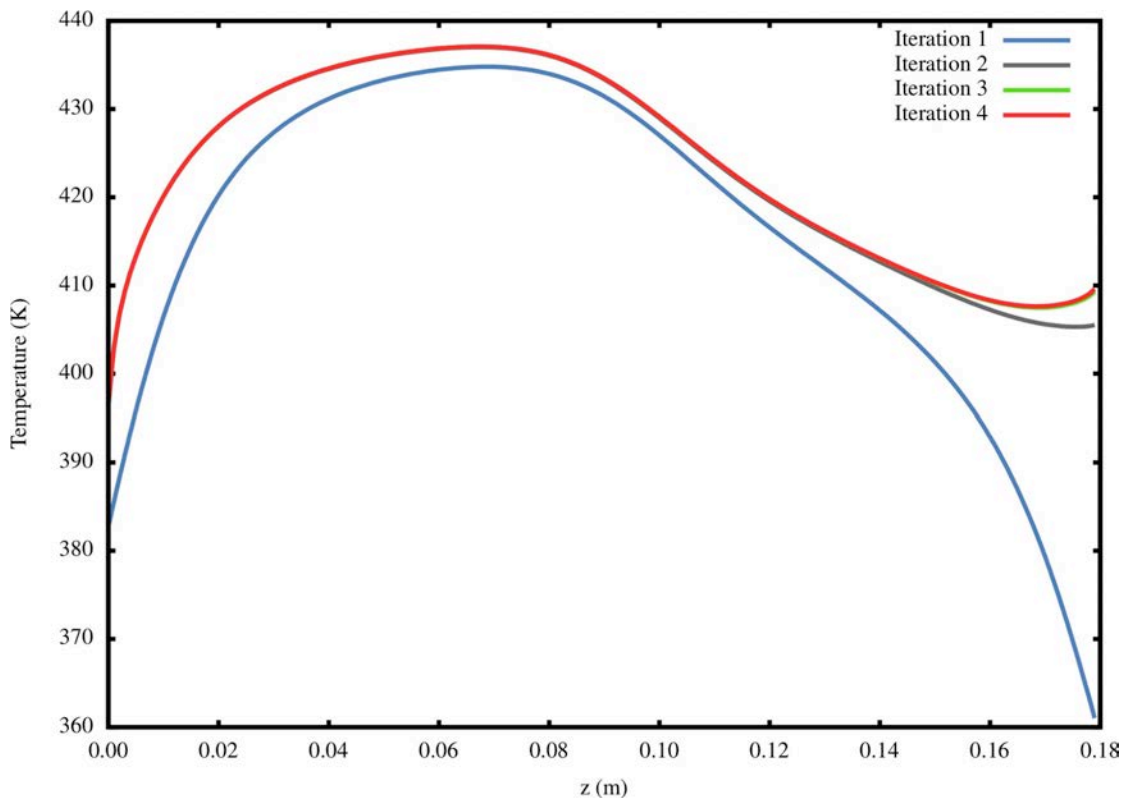


Figure 41: Flowchart of method employed for converging deforming preform heat transfer model.

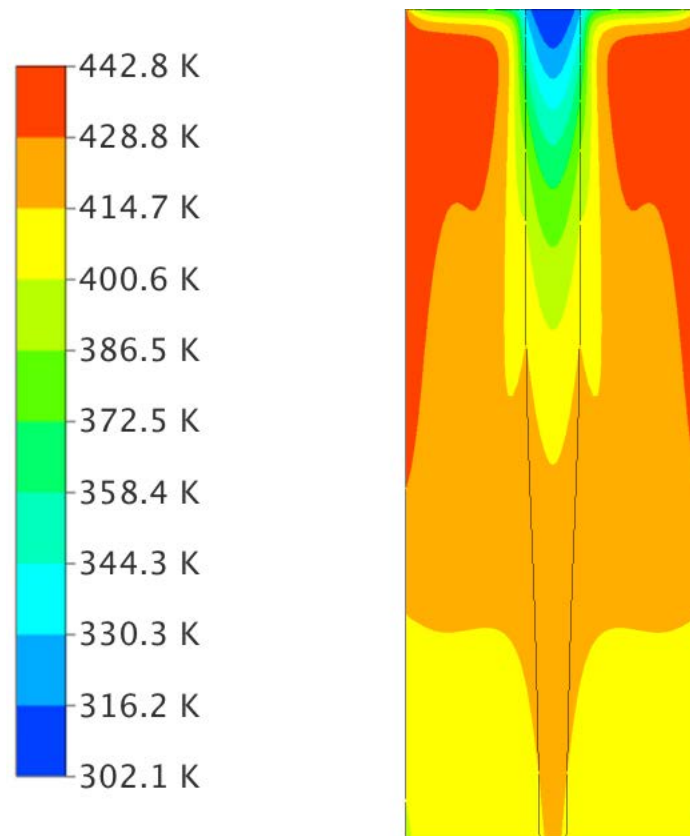


The deforming preform furnace model converged to a final solution in just 3-4 iterative steps over the complete range of draw ratios considered. Figure 42 shows this convergence via the calculated  $T_o$  profile at each iteration for the most extreme case studies where  $D_r = 50$ . Rapid convergence was no doubt aided by the fact that due to the relatively low furnace temperatures, convection (rather than radiation) is the major mode of heat transfer ( $\sim 70\%$ ) to the preform as it moves through the furnace. However, the need for an iterative solution in problems such as this one was a key driver for reducing the run-time required for calculating 3D view factors via MC-RT. In this case, MC-RT required approximately 30 minutes per iteration (on the CPU), comprising around 50% of the total run-time required for each iteration. If the GPU-based RayFactorCL version (which was not developed until much of the furnace modelling work was complete) was utilised, view factor calculation would account for less than 9% of the total run-time for each iteration).

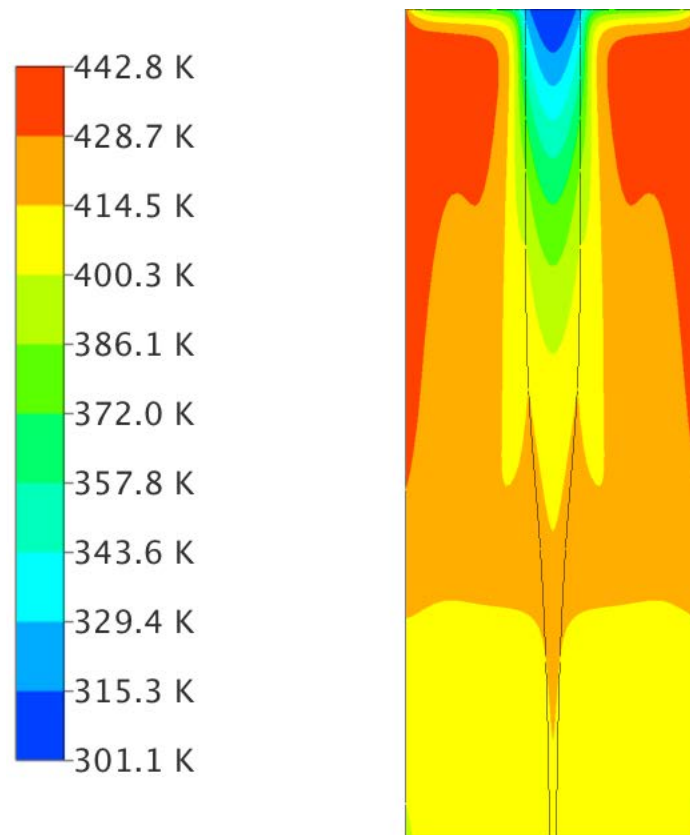


**Figure 42: Convergence of  $T_o$  temperature profile over four successive iterations.**

The temperature profiles for a vertical cross-section of the fibre-drawing furnace are shown in Figure 43 and Figure 44 for draw ratios of 4 and 50, respectively. Here, it can be seen that as the fibre gets smaller (i.e. as the draw ratio increases), it becomes so thin that it quickly equilibrates with the surrounding air (aided by the increased external heat transfer coefficient afforded by higher fibre exit speeds).



**Figure 43: Temperature profile for a vertical cross-section of the fibre drawing furnace for a draw ratio  $Dr = 4$ .**



**Figure 44: Temperature profile for a vertical cross-section of the fibre drawing furnace for a draw ratio  $Dr = 50$ .**

## 7.4 Conclusions

It has been demonstrated that radiative heat transfer within a fibre drawing furnace can be successfully characterised using radiative view factors calculated by RayFactor and the methodology described in Section 7.1. Although, the time taken to compute the furnace view factor matrix was relatively high at the time the original modelling work took place (approximately 50% of total solution time), it was substantially reduced through the development of RayFactorCL which took advantage of the speedups possible using a GPU based computing environment. Indeed, it was the relative slowness of RayFactor when used within a furnace heat transfer model that required iteration between PolyFlow and RayFactor that drove the research to consider GPU based computational tools.

Having established the benefits of primitive based MC-RT on a GPU based platform this research is well positioned to benefit upcoming research into high temperature drawing of metamaterial fibres (as noted, sub-micron fibres with a metal core surrounded by a glass sheath).

## Conclusions and Recommendations

---

View factors play an important role in the calculation of radiative heat transfer. However, for complex three-dimensional (3D) geometries, their determination can be a computational challenge. At the time this research began, finite element methods were the preferred option for such 3D situations, while the idea of combining Monte Carlo based ray-tracing methods with geometric ‘primitives’ as a means of building complex radiative heat transfer scenarios was a largely untested approach. In the early stages of this work, the computational demands of this new approach to numerically determining radiative view factors were considerable indeed. Thus, much of this thesis then became concerned with examining computationally efficient methods for using Monte Carlo Ray Tracing (MC-RT) on modern computer hardware, and in particular in researching how best to ‘marry’ the numerical methods to a computer’s internal architecture. As expected, this research followed paths that were never imagined at the start, with the following key conclusions emerging:

- 1) MC-RT is indeed a robust tool for the determination of radiative view factors in complex three-dimensional environments.
- 2) Object representation via geometric primitives can provide substantial benefits in simulation accuracy and/or run-time when compared to the use of finite elements methods (FEM), particularly in situations where the geometry under consideration involves curved surfaces.
- 3) The use of geometric primitives will never be the preferred option in all cases. However, even in situations where they may be unsuitable as a stand-alone option (such as combined radiative-conductive models), geometric primitives can still be used to advantage in a bounding volume configuration to enhance the performance of an FEM based approach.
- 4) It is critically important that the pseudorandom number generator (PSRNG) employed in a MC-RT solution be both high quality in terms of the sequences it generates, and well matched to the underlying computer hardware and its internal data management structure.

- 5) The ‘embarrassingly parallel’ nature of MC-RT readily lends itself to the exploitation of vectorised computer hardware such as the latest generation of Central Processing Units (CPUs) and emerging General Purpose Graphics Processing Units (GPGPU). However, the significant differences between these two computer architectures call for quite different approaches to code optimisation.
- 6) Highly impressive performance improvements can be achieved by exploiting a GPGPU-based computational platform. The GPGPU version of the MC-RT software developed for this thesis (which required careful coding in Open CL) exhibited up to 32 times faster performance than was possible using optimised code on a CPU-based platform.
- 7) The MC-RT software (RayFactor) developed as part of this thesis can be readily interfaced with commercial flow packages (such as PolyFlow) to allow radiative heat transfer to be included in complex 3D geometries.

The overarching aim of any simulation exercise is to achieve maximum accuracy in the minimum computational time. In this context, this research has clearly shown that using geometric primitives to describe 3D scenarios within a GPGPU-based computing environment is a powerful combination for radiative view factor estimation. However, to fully exploit this combination will require on-going research in the following key areas:

- (i) A more systematic integration of FEM-based and primitives-based object representational methods to exploit the computational advantages of each. Part of this tighter integration would be the development of more efficient GPGPU-based spatial sub-division algorithms, noting that the version implemented in this thesis was far from optimal.
- (ii) The development of robust ‘load balancing’ algorithms for efficient MC-RT using heterogeneous computing. This work has amply demonstrated that simply moving from a CPU-based environment to a GPGPU-based environment is only part of the solution; matching the characteristics of the numerical problem, the computational hardware, and the features of the coding language (noting that Open CL is still very much in the early development stage) is still a core requirement of any ‘good’ solution.

As a final note, it should perhaps be reiterated that this project emerged from an urgent need to calculate the radiative heat transfer within an operational fibre drawing furnace. Hence the focus on the numerical estimation of 3D radiative view factors. The latter part of the work, however, evolved into an exploration of how best to match Monte Carlo based numerical techniques to the challenges and opportunities presented by modern computing hardware. This was, indeed, an interesting journey and one that has still some way to go.

## References

- 
1. Incropera, F.P. and D.P. DeWitt, *Fundamentals of heat and mass transfer*. 5th ed 2002, New York: John Wiley & Sons.
  2. Howell, J.R. *A Catalog of Radiation Heat Transfer Configuration Factors*. 2011 [cited 2011 13/10/2011]; Available from: <http://www.me.utexas.edu/~Howell/index.html>.
  3. Siegel, R. and J.R. Howell, *Thermal Radiation Heat Transfer* 2001: Taylor & Francis. 896.
  4. Ambirajan, A. and S.P. Venkateshan, *Accurate determination of diffuse view factors between planar surfaces*. International Journal of Heat and Mass Transfer, 1993. **36**(8): p. 2203-2208.
  5. Feng, Y.T. and K. Han, *An accurate evaluation of geometric view factors for modelling radiative heat transfer in randomly packed beds of equally sized spheres*. International Journal of Heat and Mass Transfer, 2012. **55**(23–24): p. 6374-6383.
  6. Rammohan Rao, V. and V.M.K. Sastri, *Efficient evaluation of diffuse view factors for radiation*. International Journal of Heat and Mass Transfer, 1996. **39**(6): p. 1281-1286.
  7. Howell, J.R., *Calculation of Radiant Heat Exchange by the Monte Carlo Method*, in *NASA Technical Memorandum* 1965.
  8. Howell, J.R., *The Monte Carlo method in radiative heat transfer*. Journal of Heat Transfer, 1998. **120**(3): p. 13.
  9. Li, B.X., et al., *Analysis of directional radiative behavior and heating efficiency for a gas-fired radiant burner*. Journal of Quantitative Spectroscopy and Radiative Transfer, 2005. **92**(1): p. 51-59.
  10. Maxwell, G.M., M.J. Bailey, and V.W. Goldschmidt, *Calculations of the radiation configuration factor using ray casting*. Computer-Aided Design, 1986. **18**(7): p. 371-379.
  11. Pellegrini, M., *Monte carlo approximation of form factors with error bounded a priori*. Discrete & Computational Geometry, 1997. **17**(3): p. 319-337.
  12. Vueghs, P., et al., *Use of geometry in finite element thermal radiation combined with ray tracing*. Journal of Computational and Applied Mathematics, 2010. **234**(7): p. 2319-2326.
  13. Vujičić, M.R., N.P. Lavery, and S.G.R. Brown, *Numerical Sensitivity and View Factor Calculation Using the Monte Carlo Method*. Proceedings of the
-

- Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science, 2006. **220**(5): p. 697-702.
14. Walker, T., S.-C. Xue, and G.W. Barton, *Numerical Determination of Radiative View Factors Using Ray Tracing*. Journal of Heat Transfer, 2010. **132**(7): p. 6.
  15. Wang, L., et al., *Application of the full-spectrum -distribution method to photon Monte Carlo solvers*. Journal of Quantitative Spectroscopy and Radiative Transfer, 2007. **104**(2): p. 7.
  16. Howell, J.R., *Application of Monte Carlo to heat transfer problems*. Advances in Heat Transfer, 1968. **5**: p. 54.
  17. Farmer, J.T. and J.R. Howell, *Monte Carlo Strategies for Radiative Transfer in Participating Media*. Advances in Heat Transfer, 1998. **31**: p. 333-428.
  18. Burns, P.J., J.D. Maltby, and M.A. Christon, *Large-scale surface to surface transport for photons and electrons via Monte Carlo*. Computing Systems in Engineering, 1990. **1**(1): p. 75-99.
  19. Appel, A., *Some techniques for shading machine renderings of solids*, in *Proceedings of the April 30--May 2, 1968, spring joint computer conference* 1968, ACM: Atlantic City, New Jersey. p. 37-45.
  20. Green, S.A. and D.J. Paddon, *A highly flexible multiprocessor solution for ray tracing*. The Visual Computer, 1990. **6**(62): p. 11.
  21. Purcell, T.J., *Ray Tracing on a stream processor*, 2004, Stanford University.
  22. Singh, J.M. and P.J. Narayanan, *Real-Time Ray Tracing of Implicit Surfaces on the GPU*. Visualization and Computer Graphics, IEEE Transactions on, 2010. **16**(2): p. 261-272.
  23. Wald, I., et al., *State of the Art in Ray Tracing Animated Scenes*. Computer Graphics Forum, 2009. **28**(6): p. 1691-1722.
  24. Bentley, J.L., *Multidimensional binary search trees used for associative searching*. Commun. ACM, 1975. **18**(9): p. 509-517.
  25. Garanzha, K. and C. Loop, *Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing*. Computer Graphics Forum, 2010. **29**(2): p. 289-298.
  26. Glassner, A.S., *Space subdivision for fast ray tracing*. Computer Graphics and Applications, IEEE, 1984. **4**(10): p. 15-24.
  27. Haverkort, H.J., *Results on Geometric Networks and Data Structures*, in *Faculty of Mathematics and Information* 2004, Utrecht University. p. 174.
  28. Qiming, H., et al., *Memory-Scalable GPU Spatial Hierarchy Construction*. Visualization and Computer Graphics, IEEE Transactions on, 2011. **17**(4): p. 466-474.
  29. Wächter, C. and A. Keller. *Instant Ray Tracing: The Bounding Interval Hierarchy*. in *In rendering techniques 2006 - Proceedings of the 17th Eurographics symposium on rendering*. 2006. Nicosia, Cyprus.
  30. Wald, I., S. Boulos, and P. Shirley, *Ray tracing deformable scenes using dynamic bounding volume hierarchies*. ACM Trans. Graph., 2007. **26**(1): p. 6.
  31. Abdel-Ghany, A.M. and T. Kozai, *Radiation exchange factors between specular inner surfaces of a rectangular enclosure such as transplant production unit*. Energy conversion and management, 2006. **47**(13): p. 1988-1998.

32. Maltby, J.D. and P.J. Burns, *Performance, accuracy, and convergence in a three-dimensional Monte Carlo radiative heat transfer simulation*. Numerical Heat Transfer, Part B Fundamentals, 1991. **19**(2): p. 191-209.
33. Mirhosseini, M. and A. Saboonchi, *View factor calculation using the Monte Carlo method for a 3D strip element to circular cylinder*. International Communications in Heat and Mass Transfer, 2011. **38**(6): p. 821-826.
34. Singh, B.P. and M. Kaviany, *Independent theory versus direct simulation of radiation heat transfer in packed beds*. International Journal of Heat and Mass Transfer, 1991. **34**(11): p. 2869-2882.
35. Walker, T., S.-C. Xue , and G.W. Barton, *A robust monte carlo based ray-tracing approach for the calculation of view factors in arbitrary three-dimensional geometries*. Computational Thermal Sciences, 2012. **4**(5): p. 425-442.
36. Zavargo, Z., M. Djuric, and M. Novakovic, *Radiation heat exchange between non-diffuse gray surfaces separated by isothermal absorbing-emitting gas*. International Journal of Heat and Mass Transfer, 1991. **34**(4-5): p. 1003-1008.
37. Baek, S.W., D.Y. Byun, and S.J. Kang, *The combined Monte-Carlo and finite-volume method for radiation in a two-dimensional irregular geometry*. International Journal of Heat and Mass Transfer, 2000. **43**(13): p. 2337-2344.
38. Steward, F. and P. Cannon, *The calculation of radiative heat flux in a cylindrical furnace using the Monte Carlo method*. International Journal of Heat and Mass Transfer, 1971. **14**(2): p. 245-262.
39. Wong, B.T. and M.P. Mengüç, *Comparison of Monte Carlo techniques to predict the propagation of a collimated beam in participating media*. Numerical Heat Transfer: Part B: Fundamentals, 2002. **42**(2): p. 119-140.
40. Yi, H.-L., et al., *Radiative heat transfer in a participating medium with specular-diffuse surfaces*. International Journal of Heat and Mass Transfer, 2009. **52**(19): p. 4229-4235.
41. Maruyama, S., *Radiation Heat Transfer Between Arbitrary Three-Dimensional Bodies with Specular and Diffuse Surfaces*. Numerical Heat Transfer, Part A: Applications: An International Journal of Computation and Methodology, 1993. **24**(2): p. 16.
42. Mezrhab, A. and M. Bouzidi, *Computation of view factors for surfaces of complex shape including screening effects and using a boundary element approximation*. Engineering Computations: Int J for Computer-Aided Engineering, 2005. **22**(2): p. 132-148.
43. Vercammen, H.A.J. and G.F. Froment, *An Improved Zone Method Using Monte Carlo Techniques for the Simulation of Radiation in Industrial Furnaces*. International Journal of Heat and Mass Transfer, 1980. **23**: p. 9.
44. Coddington, P., *Random Number Generation for Parallel Computers*. The NHSE Review, 1997. **2**: p. 26.
45. Marsaglia, G., *Random Number Generators*. Journal of Modern Applied Statistical Methods, 2003. **2**(1): p. 11.
46. Marsaglia, G. and A. Zaman, *A New Class of Random Number Generators*. The Annals of Applied Probability, 1991. **1**(3): p. 19.
47. Marsaglia, G., A. Zaman, and W.W. Tsang, *Toward a universal random number generator*. Statistics and Probability Letters, 1990. **9**(1): p. 5.



48. Matsumoto, M. and T. Nishimura, *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator*. ACM Trans. Model. Comput. Simul., 1998. **8**(1): p. 3-30.
49. Saito, M. and M. Matsumoto, *SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator*, in *Monte Carlo and Quasi-Monte Carlo Methods 2006*, A. Keller, S. Heinrich, and H. Niederreiter, Editors. 2008, Springer Berlin Heidelberg. p. 607-622.
50. Salmon, J.K., et al., *Parallel random numbers: as easy as 1, 2, 3*, in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis 2011*, ACM: Seattle, Washington. p. 1-12.
51. Coddington, P., *Analysis of random number generators using Monte Carlo simulation*, 1994, Northeast Parallel Architecture Center.
52. James, F., *A review of pseudorandom number generators*. Computer Physics Communications, 1990. **60**(3): p. 329-344.
53. L'Ecuyer, P., *Random numbers for simulation*. Commun. ACM, 1990. **33**(10): p. 85-97.
54. Marsaglia, G. *A current view of random number generators*. in *Computer Science and Statistics, Sixteenth Symposium on the Interface*. Elsevier Science Publishers, North-Holland, Amsterdam. 1985.
55. Park, S.K. and K.W. Miller, *Random number generators: good ones are hard to find*. Communications of the ACM, 1988. **31**(10): p. 1192-1201.
56. Yalta, A.T. and S. Schreiber, *Random Number Generation in gretl*. Journal of Statistical Software, Code Snippets, 2012. **50**(1): p. 1-13.
57. Vanleersum, J., *A Method for Determining a Consistent Set of Radiation View Factors From a Set Generated by a Nonexact Method*. International Journal of Heat and Fluid Flow, 1989. **10**(1): p. 3.
58. Lawson, D.A., *An Improved Method for Smoothing Approximate Exchange Areas*. International Journal of Heat and Mass Transfer, 1995. **38**(16): p. 3.
59. Taylor, R.P. and R. Luck, *Comparison of Reciprocity and Closure Enforcement Methods for Radiation View Factors*. Journal Of Thermophysics and Heat Transfer, 1995. **9**(4): p. 7.
60. Loehrke, R.I., J.S. Dolaghan, and P.J. Burns, *Smoothing Monte Carlo Exchange Factors*. Journal of Heat Transfer, 1995. **117**(2): p. 3.
61. Larsen, M.E. and J.R. Howell, *Least-Squares Smoothing of Direct-Exchange Areas in Zonal Analysis*. Journal of Heat Transfer, 1986. **108**(1): p. 4.
62. Daun, K.J., D.P. Morton, and J.R. Howell, *Smoothing Monte Carlo Exchange Factors Through Constrained Maximum Likelihood Estimation*. Journal of Heat Transfer, 2005. **127**(10): p. 5.
63. Campbell, P.M., *Monte carlo method for radiative transfer*. International Journal of Heat and Mass Transfer, 1967. **10**(4): p. 8.
64. Hill, F.S.J., *Computer Graphics Using Open GL 2001*, Upper Saddle River, New Jersey: Prentice Hall.
65. Maillot, P.-G., *Using quaternions for coding 3D transformations*, in *Graphics gems*, S.G. Andrew, Editor 1990, Academic Press Professional, Inc.: San Diego, CA, USA. p. 498-515.
66. Gray, J., *Olinde Rodrigues' paper of 1840 on transformation groups*. Archive for History of Exact Sciences, 1980. **21**(4): p. 375-385.

67. Marsaglia, G., *Choosing a Point from the Surface of a Sphere*. The Annals of Mathematical Statistics, 1972. **43**(2): p. 2.
68. Leiserson, C.E., T.B. Schardl, and J. Sukha, *Deterministic parallel random-number generation for dynamic-multithreading platforms*, in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* 2012, ACM: New Orleans, Louisiana, USA. p. 193-204.
69. Panneton, F., P. L'Ecuyer, and M. Matsumoto, *Improved long-period generators based on linear recurrences modulo 2*. ACM Trans. Math. Softw., 2006. **32**(1): p. 1-16.
70. Gustafson, J.L., *Reevaluating Amdahl's law*. Commun. ACM, 1988. **31**(5): p. 532-533.
71. Howell, J.R., *The Monte Carlo Method in Radiative Heat Transfer*. ASME J. Heat Transfer, 1998. **120**: p. 13.
72. Goldberg, D., *What Every Computer Scientist Should Know About Floating Point Arithmetic*. ACM Computing Surveys, 1991. **23**(1): p. 48.
73. Cornejo Díaz, N., A. Vergara Gil, and M. Jurado Vargas, *Assessment of the suitability of different random number generators for Monte Carlo simulations in gamma-ray spectrometry*. Appl Radiat Isot, 2010. **68**(3): p. 5-5.
74. L'Ecuyer, P. and R. Simard, *TestU01 : A C Library for Empirical Testing of Random Number Generators*. Vol. 33. 2007, New York, NY, ETATS-UNIS: Association for Computing Machinery.
75. Marsaglia, G., *The Marsaglia Random Number CDRom, with The Diehard Battery of Tests of Randomness*, 1995, Florida State University, Tallahassee: Department of Statistics.
76. McCullough, B.D., *A review of TESTU01*. Journal of Applied Econometrics, 2006. **21**(5): p. 677-682.
77. Ramanathan, R.M. *Intel multi-core processors: Making the move to quadcore and beyond*. Intel White Paper, 2006. 8.
78. Fang, Q. and D.A. Boas, *Monte Carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units*. Opt. Express, 2009. **17**(22): p. 20178-20190.
79. Gao, W., et al., *Real-time 2D parallel windowed Fourier transform for fringe pattern analysis using Graphics Processing Unit*. Opt. Express, 2009. **17**(25): p. 23147-23152.
80. Lu, P., et al., *Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the International Space Station*. Journal of Real-Time Image Processing, 2010. **5**(3): p. 179-193.
81. Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, in *Intel® 64 and IA-32 Architectures Software Developer Manuals*, I. Corporation, Editor 2012, Intel Corporation.
82. Green, R., *Faster Math Functions*, S.C.E. America, Editor 2003, Sony Computer Entertainment America.
83. Moshie, S.L. *Cephes Mathematical Library*. 2001 14/05/2001 [cited 25/9/2011; Available from: <http://www.netlib.org/cephes/>].
84. Shibata, N., *Efficient evaluation methods of elementary functions suitable for SIMD computation*. Computer Science - Research and Development, 2010. **25**(1-2): p. 25-32.

85. Fraser, W. and J.F. Hart, *On the computation of rational approximations to continuous functions*. Commun. ACM, 1962. **5**(7): p. 401-403.
86. Fuchs, H., Z.M. Kedem, and B.F. Naylor, *On visible surface generation by a priori tree structures*. SIGGRAPH Comput. Graph., 1980. **14**(3): p. 124-133.
87. Meagher, D., *Geometric modeling using octree encoding*. Computer Graphics and Image Processing, 1982. **19**(2): p. 129-147.
88. Guttman, A., *R-trees: a dynamic index structure for spatial searching*, in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data* 1984, ACM: Boston, Massachusetts. p. 47-57.
89. Wald, I., *Realtime Ray Tracing and Interactive Global Illumination*, PhD in Computer Graphics Group 2004, Saarland University.
90. Zeeb, C.N., J.S. Dolaghan, and P.J. Burns, *An Efficient Monte Carlo Particle Tracing Algorithm for Large, Arbitrary Geometries*. Numerical Heat Transfer, Part B: Fundamentals, 2001. **39**(4): p. 325-344.
91. Mazumder, S., *Methods to Accelerate Ray Tracing in the Monte Carlo Method for Surface-to-Surface Radiation Transport*. Journal of Heat Transfer, 2006. **128**: p. 7.
92. Dan, W., et al. *Implementation of Parallel Game Tree Search on a SIMD System*. in *Information Engineering (ICIE), 2010 WASE International Conference on*. 2010.
93. Eisemann, M., et al., *Geometry Presorting for Implicit Object Space Partitioning*. Computer graphics forum (Eurographics), 2012. **31**(4): p. 11.
94. Yoon, S.-E. and D. Manocha, *Cache-Efficient Layouts of Bounding Volume Hierarchies*. Computer graphics forum (Eurographics), 2006. **25**(3): p. 9.
95. Buck, I., et al., *Brook for GPUs: stream computing on graphics hardware*. ACM Trans. Graph., 2004. **23**(3): p. 777-786.
96. Group, K.O.W., *The OpenCL Specification*, 2009, The Khronos Group Inc.
97. NVidia, *OpenCL programming guide for the CUDA architecture*, 2012, NVidia Corporation.
98. Gjermundsen, J., *CPU and GPU Co-processing for Sound*, in *Department of Computer and Information Science* 2010, Norwegian University of Science and Technology.
99. Luong, T.E., N. Melab, and E.-G. Talbi, *GPU-based island model for evolutionary algorithms*, in *Proceedings of the 12th annual conference on Genetic and evolutionary computation* 2010, ACM: Portland, Oregon, USA. p. 1089-1096.
100. Pospíchal, P., J. Schwarz, and J. Jaros, *Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu*. 16th International Conference on Soft Computing MENDEL, 2010: p. 64-70.
101. Shah, R., P. Narayanan, and K. Kothapalli. *GPU-Accelerated Genetic Algorithms*. in *Workshop on Parallel Architectures for Bio-inspired Algorithms (PACT Workshop)*. 2010.
102. Frigo, M., *A fast Fourier transform compiler*. SIGPLAN Not., 1999. **34**(5): p. 169-180.
103. Havel, J. and A. Herout, *Yet Faster Ray-Triangle Intersection (Using SSE4)*. IEEE Transactions on Visualization and Computer Graphics, 2010. **16**(3): p. 434-438.
104. Ludvigsen, H. and A.C. Elster. *Real-Time Ray Tracing Using Nvidia OptiX*. in *Eurographics Short Papers*. 2010.

105. Moller, T. and B. Trumbore, *Fast, minimum storage ray/triangle intersection*, in *ACM SIGGRAPH 2005 Courses* 2005, ACM: Los Angeles, California. p. 7.
106. Shevtsov, M., A. Soupikov, and E. Kapustin, *Ray-Triangle Intersection Algorithm for Modern CPU Architectures*, in *Proceedings of GraphiCon 2007* 2007. p. 33-39.
107. Fujimoto, A., T. Tanaka, and K. Iwata, *ARTS: Accelerated Ray-Tracing System*. Computer Graphics and Applications, IEEE, 1986. **6**(4): p. 16-26.
108. Foley, T. and J. Sugerman, *KD-tree acceleration structures for a GPU raytracer*, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* 2005, ACM: Los Angeles, California. p. 15-22.
109. Nakasato, N., *Implementation of a parallel tree method on a GPU*. Journal of Computational Science, 2012. **3**(3): p. 132-141.
110. Thrane, N. and L.O. Simonsen, *A comparison of acceleration structures for GPU assisted ray tracing*, 2005.
111. Weghorst, H., G. Hooper, and D.P. Greenberg, *Improved Computational Methods for Ray Tracing*. ACM Trans. Graph., 1984. **3**(1): p. 52-69.
112. Large, M.C.J., et al., *Microstructured Polymer Optical Fibre* 2007: Springer.
113. Xue, S., et al., *The role of material properties and drawing conditions in the fabrication of microstructured optical fibres*. Journal of Lightwave Technology, 2006. **24**(2): p. 853-860.
114. Xue, S., et al., *Fabrication of microstructured optical fibres – Part I: Problem formulation and numerical modelling of transient draw process*. Journal of Lightwave Technology, 2005. **23**(7): p. 9.
115. Xue, S., et al., *Transient heating of PMMA preforms for microstructured optical fibres*. J. Lightwave Technology, 2007. **25**: p. 1177-1183.
116. Bu, H.S., W. Aycock, and B. Wunderlich, *Heat capacities of solid, branched macromolecules*. Polymer, 1987. **28**(7): p. 1165-1176.
117. Xue, S.-C., et al., *Radiative heat transfer in preforms for microstructured optical fibres*. International Journal of Heat and Mass Transfer, 2007. **50**(7-8): p. 7.
118. Bardos, C., F. Golse, and B. Perthame, *The rosseland approximation for the radiative transfer equations*. Communications on Pure and Applied Mathematics, 1987. **40**(6): p. 691-721.
119. Siegel, R. and J.R. Howell, *Thermal Radiation Heat Transfer* 2001, New York: Taylor & Francis.
120. Sayles, R. and B. Caswell, *A finite element analysis of the upper jet region of a fiber drawing flow field of a temperature-sensitive material*. International Journal of Heat and Mass Transfer, 1984. **27**(1): p. 57-67.
121. Myers, M.R., *A model for unsteady analysis of preform drawing*. AIChE Journal, 1989. **35**(4): p. 592-602.