Faster than Real-Time GPGPU Radiation Pressure Modeling Methods

by

P. W. Kenneally

B.Eng., Australian National University, 2010

B.A., Australian National University, 2010

M.S., University of Colorado, 2016

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Aerospace Engineering Sciences

2019

This thesis entitled:
Faster than Real-Time GPGPU Radiation Pressure Modeling Methods
written by P. W. Kenneally
has been approved for the Department of Aerospace Engineering Sciences

_____

Prof. Hanspeter Schaub

_____

Prof. Daniel Scheeres

_____

Prof. Jay McMahon

_____

Dr. Daniel Kubitschek

_____

Prof. Moriba K. Jah

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Kenneally, P. W. (Ph.D., Aerospace Engineering Sciences)

Faster than Real-Time GPGPU Radiation Pressure Modeling Methods

Thesis directed by Prof. Hanspeter Schaub

Solar radiation pressure (SRP) is a significant contributing dynamic force on spacecraft in all orbit regimes. Predicting, accommodating, and either leveraging or canceling its effect, is paramount to effective orbit determination, maneuver and mission design. As a result spacecraft numerical simulation requires computational models which provide the facility to model SRP with sufficient accuracy. However, typically the computationally intense nature of performing high-fidelity SRP evaluations has limited such evaluations to being an offline computation which generates lookup data. Precomputation limits the ability for a spacecraft dynamic simulation to accommodate the myriad time varying changes which occur to the spacecraft state during a mission.

In the past decade the computer graphics industry has driven the development of highly parallel graphics processing units (GPU) capable of performing many thousands of floating point operations per second. General purpose GPU programming (GPGPU) has been leveraged particularly in Engineering and the Sciences where the high computational power of parallel GPU hardware presents the opportunity for significant increases in the size and dimension of computational problems now manageable on personal computers.

This dissertation presents two modeling approaches which take advantage of the GPGPU aspect of commodity GPU hardware. The first contribution is a modeling approach which utilizes the vector graphics application programming interface (API) Open Graphics Library (OpenGL) and the GPGPU computing API Open Computing Language to develop a high geometric fidelity SRP modeling approach. The OpenGL-CL modeling approach computes SRP induced force and torque across a detailed spacecraft mesh model. The method utilizes the OpenGL-OpenCL shared context to facilitate modeling data between the two APIs. The OpenGL render pipeline is manipulated to render the sun-frame projected surface of the spacecraft into OpenGL Texture data objects. A

custom OpenCL parallel reduction kernel is developed which subsequently computes the SRP force and torque across the spacecraft rendered into the OpenGL Textures. The method presents faster than real time computation speeds while accommodating spacecraft meshes with many thousands of vertices, arbitrary articulated components and detailed spacecraft material optical parameters.

The second contribution is a GPU based parallel ray tracing modeling approach which exhibits faster than real time evaluation speeds. Techniques and algorithms from the computer graphics discipline are used to develop and implement a method which computes SRP force and torque across a detailed spacecraft triangulated mesh model. Efficient data structures such as bounding volume hierarchy (BVH) acceleration support a minimization of computational burden by reducing the ray-surface intersection search space. Accurate ray reflections are computed for complex materials by applying a Quasi-Monte Carlo integration method and importance sampling. Complex material bidirectional reflectance distribution functions (BRDF) are implemented with as both, ideal mirror-like specular and Lambertian diffuse, and as microfacet BRDF models. Arbitrary spacecraft articulation are accommodated at run time with no appreciable reduction in computational speed.

Both SRP models utilize the latent computing power of the GPU which is exists in the large majority of consumer grade personal computing systems. Further access to latent computing power is enabled by the development of a software simulation communication middleware called Black Lion (BL). The third contribution of this thesis is the description of a novel software architecture and the design principles applied to the development of the BL software. Black Lion enables the integration of multiple local or distributed heterogeneous applications never intended to run in a cooperative settings. It is shown that BL enables access to more powerful latent personal computing resources by creating a means to transparently facilitate distributed simulation across multiple simulation nodes and computers.

Finally, this dissertation demonstrates the utility of both modeling methods by their applications in two case studies. Firstly, the high-fidelity SRP effects are computed for an ongoing asteroid sample return mission. Agreement between the OpenGL-CL methods is demonstrated.Both SRP

modeling approaches make significant use of pre and post launch engineering data. The utility of direct access to a model's physical parameters is demonstrated in an analysis of contributors to possible error between modeled and estimated SRP accelerations. Secondly, capability of fast computational speed paired with high geometric resolution, of both OpenGL-CL and ray tracing methods, is demonstrated. Each method is employed in the simulation and long-term propagation of realistic multi-layer insulation (MLI) debris object mesh models and the effect of departing from the typical flat-plate MLI model is investigated.

Dedication

To family. Mum, Dad, Mat, Gen, Gran and Karen.

Acknowledgements

First I would like to thank my advisor Dr. Hanspeter Schaub for his guidance and support throughout this degree. As a Masters student, Dr. Schaub provided me with the opportunity to take up a Research Assistant (RA) position working on the EMM mission at the Laboratory of Atmospheric and Space Physics (LASP). The RA position gave rise to the first thoughts that continuing on to a PhD degree might be possible and achievable. Dr. Schaub is dedicated to the success of his students and supports this by providing students with many opportunities to learn and practice new skills both within and outside the academic context. Thank you to my committee members Dr. Jay McMahon, Dr. Daniel Kubitschek, Dr. Daniel Scheeres and Dr. Moriba K. Jah for their support of my work and generously giving their time and thought as members of my PhD committee.

I would like to thank Cody Allard, Thibaud Teil and John Alcorn. As the initial EMM team within the AVS Lab it was enjoyable to suffer and succeed with such capable and hard working individuals.

The opportunity to work on the EMM project at LASP was invaluable experience. What made it so was the time spent working under the leadership of Dan Kubitsheck and Scott Piggott. Dan Kubitschek demonstrates a brand of leadership which makes each team member feel valued and encouraged to produce their best work. Much of the technical work achieved in this dissertation would not have been possible without Scott Piggott's ability to provide guidance and insight to the myriad technical considerations demanded of each software project.

Colleagues in the Autonomous Vehicle Systems Laboratory have each played important roles

in shaping and supporting my time at the University of Colorado, and therefore the direction of this thesis. Thank you to Lee Jasper, JoAnna Fulton, Stephen O'Keefe, Joseph Hughes and Trevor Bennett. I would also like to thank friends here in Boulder and in Australia who have encouraged and supported my pursuit of this degree.

Finally, I would like to express deep gratitude to my family. When I would waiver, they supported me. When I had doubts, they offered reassurance. They have made me who I am. Thank you Mum, Dad, Mat, Gen, Gran and Karen.

Contents

Chapter

1 Introduction **1**

1.1 Motivation 1

1.2 Background 5

1.2.1 General Purpose GPU Programming 7

1.2.2 Distributed Spacecraft Simulation 9

1.3 Research Goals 12

2 SRP Theory **14**

3 OpenGL-OpenCL Solar Radiation Pressure **20**

3.1 The OpenGL Render Pipeline 21

3.2 Mesh Definition 23

3.3 Custom OpenGL Render Pipeline 25

3.4 OpenGL Algorithm Steps 27

3.4.1 Recursive Bounding Box Computation 28

3.4.2 Mesh Articulation 28

3.4.3 Vertex and Fragment Shader Stages 32

3.5 OpenCL Algorithm Steps 35

3.6 Model Validation 38

3.6.1 Impact Of Mesh Detail On Accuracy 40

3.6.2 Model Articulation and Detailed Material Properties 43
3.7 Computational Performance 47
3.8 Conclusions 49

4 OpenCL Ray Tracing **51**
4.1 GPGPU Parallel Algorithm Considerations 52
4.2 Algorithm Steps Overview 53
4.3 Radiation Pressure Particle Tracing Formulation 55
4.4 Force and Torque Evaluation 57
4.5 Intersection Testing 58
4.5.1 Bounding Volume Intersection 60
4.5.2 Triangle Facet Intersection 60
4.6 Bidirectional Reflection Distribution Functions 63
4.6.1 Ideal BRDF 66
4.6.2 Mircofacet Model BRDF 67
4.7 Evaluating Ray-Surface Interaction 70
4.7.1 Sampling Ideal BRDF 72
4.7.2 Sampling Microfacet BRDFs 72
4.7.3 Computing The Total BRDF 73
4.8 Model Validation 74
4.9 Multiple Ray Reflections 77
4.10 Model Articulation and Detailed Material Properties 85
4.11 BRDF Effect on Orbit Propagation 86
4.12 Computational Performance 89
4.13 Conclusions 93

5 Black Lion Distributed Simulation **95**
5.1 Distributed Spacecraft Simulation Architectures 96

5.2 Black Lion Architecture 97
5.2.1 Data Transport and Data Translation Layers 99
5.2.2 Black Lion Simulation Topology 102
5.2.3 Socket and Connection Definitions 102
5.3 Communication Between Nodes 105
5.4 Tick-Tock Synchronization 107
5.5 Black Lion Simulation Case Study 109
5.5.1 Basilisk Simulation Configuration 109
5.5.2 Simulation Results 112
5.6 Conclusions 114

6 Case Studies **116**

6.1 OSIRIS REx Case Study 117
6.1.1 ORex Case Study: SRP Modeling 118
6.1.2 ORex Case Study: Modeling Error and Bounding Analysis 122
6.1.3 ORex Case Study: Conclusions 126
6.2 Multi-Layer Insulation Case Study 127
6.2.1 MLI Case Study: Mesh Models 129
6.2.2 MLI Case Study: Ray Traced Force Comparison 129
6.2.3 MLI Case Study: Orbit Propagation 132
6.2.4 MLI Case Study: Conclusions 140

7 Conclusions and Future Work **141**

7.1 Conclusions 141
7.2 Recommendations for Future Work 143

Bibliography **145**

Appendix

A Simulation Architecture Basilisk **152**

A.1 Software Stack and Build 156

A.2 Modularity In Basilisk 157

A.2.1 Components 157

A.2.2 Message System 159

A.2.3 Dynamics Manager 162

A.3 Execution Control 163

A.4 Data Logging 166

A.5 Monte Carlo Capability 166

A.6 Development Approach - Open Source 167

A.7 Example of a Basilisk Simulation Configuration 168

A.8 Conclusion 174

Tables

Table

3.1 Spacecraft orbit parameters for sun-synchronous LEO orbit and GEO orbit. 44

3.2 Spacecraft sub-mesh material optical parameters. 44

4.1 SRP force for faceted evaluations. 77

4.2 Spacecraft orbit parameters for sun-synchronous LEO orbit and GEO orbit. 86

4.3 Spacecraft orbit parameters for sun-synchronous LEO orbit and GEO orbit. 88

5.1 Spacecraft sub-mesh material optical parameters. 113

5.2 Aqua spacecraft orbit parameters for polar LEO orbit 113

6.1 OSIRIS-REx material optical properties. 121

6.2 Computed SRP evaluations for sun-point ${}^{\mathcal{B}}\hat{\boldsymbol{s}} = [1, 0, 0]$. 123

6.3 Representative portions of acceleration error (relative to Hifi RT 3 bounce) [km/s^2]. 124

6.4 Computed SRP evaluations for sun-point ${}^{\mathcal{B}}\hat{\boldsymbol{s}} = (1, 0, 0)$. 124

Figures

Figure

1.1 Artist depiction of the MESSENGER spacecraft in orbit around Mercury.[1] 2

1.2 Artist depiction of the Hayabusa spacecraft approaching asteroid Itokawa.[2] 3

1.3 Ray traced scene of spheres with various material optical properties.[3] 4

1.4 Two iterations of spacecraft model fidelity for ICESat [73]. 6

1.5 Illustrative demonstration of the number of cores, memory proximity and stages. . . 8

2.1 Reflectance geometry 16

2.2 Reflection geometry for idealized specular and diffuse [72]. 18

3.1 The OpenGL API used to generate an orthographic projection of a multiple mesh models. 21

3.2 The default OpenGL pipeline. 22

3.3 Notional operations for each shader stage. 23

3.4 Aqua spacecraft .OBJ mesh model. 24

3.5 Overview of OpenGL render pipeline with required custom vertex shader and fragment shader stages outputting to the Textures held in a Framebuffer object. 26

3.6 Sub-meshes of the Aqua spacecraft mesh. 29

3.7 Illustration of two coordinate frames with different origins and orientations. 31

3.8 Loose sun frame AABB (dashed blue) and body frame AABB (solid green). Body frame axes $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{z}}$ are red, green and blue respectively and the sun heading $\hat{\boldsymbol{s}}$ as black. 33

3.9 Notional parallel reduction by summing sequenced addressing. 37

3.10 Notional parallel reduction striding over all allocated addresses to continue summing sequential blocks of memory in a the `While()` loop. 38

3.11 Error with respect to analytic cannonball evaluation for 1 m sphere mesh. 40

3.12 Mixed force validation. 40

3.13 OSIRIS REx spacecraft mesh models. 41

3.14 Force percentage difference between low fidelity models relative to the high-fidelity model with baseline value 5.73361×10^{-5} [N]. 43

3.15 Torque percentage difference between low-fidelity models relative to the high-fidelity model with baseline value 6.57817×10^{-5} [Nm]. 44

3.16 Force percentage difference between box and wing model relative to the hifidelity model with baseline value 5.73361×10^{-5} [N]. 45

3.17 Torque percentage difference between box and wing model relative to the hifidelity model with baseline value 6.57817×10^{-5} [Nm]. 45

3.18 Force percentage difference between HGA model relative to the high-fidelity model with baseline value 5.73361×10^{-5} [N]. 46

3.19 Torque percentage difference between HGA model relative to the high-fidelity model with baseline value 6.57817×10^{-5} [Nm]. 46

3.20 Body frame force components over two orbits. 47

3.21 Body frame torque components over two orbits. 48

3.22 Sequential rendered spacecraft in sun frame. 48

3.23 OpenGL-CL computation times for three different GPUs. 49

4.1 Increased GPU Work Group occupancy when tracing by ray rather than by pixel. . 52

4.2 Illustration of a set of five bounding boxes and a test ray. Intersections are recorded for the boxes with the dash outlines. 54
4.3 Ray generation for the MRO mesh model with a B frame oriented bounding box (dashed black) and an S frame bounding box (blue solid). 55
4.4 Illustration of a set of five bounding boxes and a test ray. Intersections are recorded for the boxes with the dash outlines. 59
4.5 Two BVH traversal structures. The left structure demonstrates a simple recursive BVH traversal. The right demonstrates the same BVH as shown on the left yet organized as a depth first search array with precomputed node skip pointers. 60
4.6 Example result of the parallel plane bounding box intersection algorithm. For the top left ray intersection, the algorithm returns t_max as greater than or equal to t_min. For the bottom right ray miss, the algorithm returns t_max as less than t_min. 62
4.7 Illustrations of Two Common BRDF Geometry Descriptions. 65
4.8 Conceptual illustration of microfacet with shadowing-masking geometry. 67
4.9 Isotropic Beckmann and GGX microfacet NDFs as a function of the angle between a ray direction and the the surface normal, θ_h, for roughness value $\alpha = 0.45$. 70
4.10 Test cube spacecraft model. Black and cyan vectors indicate body-frame sun headings evaluated. Red, green and blue vectors denote first, second and third body-frame axes respectively. 75
4.11 Error of the ray-traced force components relative to the faceted force norm for a specular material evaluation. 76
4.12 Error of the ray-traced force components relative to the faceted force norm for a diffuse material evaluation. 76
4.13 Error of the ray-traced force components relative to the faceted force norm for a mixed (diffuse and specular) material evaluation. 76

4.14 Test model with two surfaces which form a right angled face. The black vector indicates body-frame sun heading evaluated. Red, green and blue vectors denote first, second and third body-frame axes respectively. 78
4.15 Error of the ray-traced force components relative to the faceted force norm for multiple bounce evaluation. 79
4.16 Difference in force magnitude for resolving multiple bounces on hifidelity OSIRIS-REx. 80
4.17 Difference in torque magnitude for resolving multiple bounces on hifidelity OSIRIS-REx. 80
4.18 Force percentage difference between resolving second and first bounce relative to baseline value 5.62995×10^{-5} [N]. 81
4.19 Force percentage difference between resolving third and second bounce relative to baseline value 5.62995×10^{-5} [N]. 81
4.20 Torque percentage difference between resolving second and first bounce relative to baseline value 6.44875×10^{-5} [Nm]. 82
4.21 Torque percentage difference between resolving third and second bounce relative to baseline value 6.44875×10^{-5} [Nm]. 82
4.22 High percentage difference attitude for force, between one and two bounces. The sun heading latitude and longitude is (-90°, 0°) which is equivalent to ${}^{\mathcal{B}}\hat{\boldsymbol{s}} = [0.0, 0.0, -1.0]$. 83
4.23 High percentage difference attitude for torque, between one and two bounces . The sun heading latitude and longitude (-1.8°, 40.7°). 84
4.24 Resulting rendered image of the ray-traced CloudSat high resolution evaluation in false color. 85
4.25 Percentage error in the direction of the resultant force between each successive ray bounce relative to a high resolution evaluation. 86
4.26 Force on Aqua spacecraft mesh in polar LEO 87
4.27 Torque on Aqua spacecraft mesh in polar LEO. 87
4.28 Magnitude of the SRP acceleration in sun-synchronous LEO over one orbital period. 88

4.29 Magnitude of the SRP acceleration at GEO over two orbital periods. 89
4.30 Radial, intrack and crosstrack differences w.r.t the position of the simulated Idealized BRDF cube at LEO. The dashed line corresponds to the Beckmann and the solid line the GGX. 90
4.31 Radial, intrack and crosstrack differences w.r.t the position of the simulated Idealized BRDF cube at GEO. The dashed line corresponds to the Beckmann and the solid line the GGX. 91
4.32 Execution times for ray resolutions from 0.01 mm to 0.0016 mm, for one bounce. . . 92
4.33 Execution times for ray resolutions from 0.01 mm to 0.0016 mm, for a maximum of three bounces. 92

5.1 Virtualization of Spaceflight Components. 97
5.2 Communication Architecture: Central Controller, Delegate APIs and Router APIs. . 98
5.3 Relating the primary BL architectural layers to the OSI stack. 101
5.4 Socket Patterns between the Central Controller and Sample Nodes. 103
5.5 Socket Connections Types (Binding vs. Connecting) and Ports. 104
5.6 Node Actions between a "Tick-Tock": Publish, Subscribe, Step Simulation. 106
5.7 Nodes' Timely Nature: Synchronous, Asynchronous and Listener Behaviors. 108
5.8 Blacklion configured Baislisk simulation for an Earth orbiting scenario. 111
5.9 Aqua spacecraft .OBJ mesh model. 112
5.10 SRP Force on Aqua spacecraft mesh in polar LEO 114
5.11 SRP Torque on Aqua spacecraft mesh in polar LEO 115

6.1 Error of the ray-traced force components relative to the faceted force norm for multiple bounce evaluation [4]. 118
6.2 OSIRIS-REx box and wing models. 119
6.3 OSIRIS-REx spacecraft model sub-meshes. 120
6.4 Difference in ray-tracing SRP accelerations [km/s^2] with approximated 10-Plate model. 121

6.5 Difference in ray-tracing SRP accelerations [km/s^2] with approximated 9-Plate HGA model. 121
6.6 Error of the SH approximation relative to the generated ray-traced force vectors. . . 122
6.7 Change of acceleration magnitude for varied GBK diffuse and specular coefficients. . 125
6.8 Change of acceleration magnitude for varied SP diffuse and specular coefficients. . . 126
6.9 Hubble space telescope where the outer layers of the spacecraft are covered by MLI.[5] 127
6.10 Mars Reconnaissance Orbiter wrapped by MLI prior to launch.[6] 128
6.11 Mesh models for MLI sheet of equal surface area, 1.08 [m^2]. 130
6.12 Body frame force components for the flat plate MLI mesh model. 131
6.13 Body frame force components for the wrinkled MLI mesh model. 131
6.14 Force magnitude difference of the wrinkled model relative to the flat plate model. . . 132
6.15 Rendered MLI mesh model for one and two where the difference between the rendered images is shown. 132
6.16 Rendered MLI mesh model differences between successive bounces. 133
6.17 Ray tracing propagated angular rate evolution for MLI mesh model in GEO orbit. . 134
6.18 Ray tracing propagated attitude evolution for MLI mesh model in GEO orbit. 135
6.19 Ray tracing propagated Earth centered inertial frame force evolution for MLI mesh model in GEO orbit. 136
6.20 Ray tracing propagated body-frame torque evolution for MLI mesh model in GEO orbit. 137
6.21 OpenGL-CL propagated angular rate evolution for MLI mesh model in GEO orbit. . 137
6.22 OpenGL-CL propagated attitude evolution for MLI mesh model in GEO orbit. . . . 139
6.23 OpenGL-CL propagated Earth centered inertial frame force evolution for MLI mesh model in GEO orbit. 139
6.24 OpenGL-CL propagated body-frame torque evolution for MLI mesh model in GEO orbit. 139

A.1 An example layout of a complete Basilisk simulation where each element of the system has SWIG generated Python interfaces available in the Python environment. 158
A.2 Basilisk Task Group, Tasks and Module layout. 159
A.3 Basilisk messaging system memory layout and organization. 161
A.4 A notional messaging system publish and subscribe map for a message storage container of a single Task Group. 163
A.5 Basilisk high level flow of control for simulation execution. 165
A.6 Concept diagram of simple multi body gravity orbiter Basilisk simulation configuration. 169
A.7 Evolution of attitude error in each MRP component. 175
A.8 Evolution of computed reaction wheel torques (dashed) and the actual reaction wheel torques. 175
A.9 Evolution of reaction wheel speeds. 175

Chapter 1

Introduction

1.1 Motivation

Effective orbit determination, maneuver and mission design and mission numerical simulations require tools that enable accurate modeling of the spacecraft dynamical system. Of the contributing effects to a spacecraft's dynamics radiation pressure (RP) in its various forms plays a significant role [91, 72, 94]. Radiation pressure contributions arise from solar, earth albedo, planetary infra-red and spacecreaft thermal radiation sources. The contribution of each of these RP sources is dependent on the specific operational regime of the spacecraft. For example, solar radiation pressure (SRP) the momentum imparted to a body by impinging solar photons, becomes a dominant non-conservative force above the Low Earth Orbit (LEO) region[91]. Similarly, the impact of directed thermal radiation is shown to be a significant consideration in the determination of orbit determination and tracking efforts for interplanetary spacecraft [72, 8].

Effective modeling of the SRP induced perturbation of a spacecraft enables mission designers to consider SRP a valuable actuator rather than a disturbance. Such novel use of the SRP force in maneuver and mission design is exemplified by the MErcury Surface, Space ENvironment, GEochemistry and Ranging (MESSENGER) mission. The MESSENGER mission completed six planetary gravity assists during its journey to a Mercury orbit. The MESSENGER mission designers employed a solar sailing technique to perform each trajectory change maneuver (TCM) and accurately target each planetary flyby. Typical methods for performing TCM's use onboard thrusters to impart the required ΔV. However, using SRP as the TCM actuator allowed the

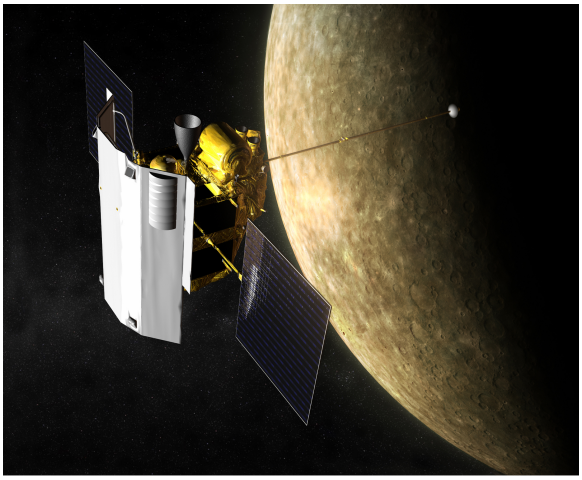


Figure 1.1: Artist depiction of the MESSENGER spacecraft in orbit around Mercury.[1]

MESSENGER team to perform TCM's with more accuracy and finer control due to the smaller magnitude of the SRP induced force. Additionally, the MESSENGER team was able to reduce fuel and related structural accommodations in the spacecraft design to reduce overall mission cost[67]. Additional opportunities exist in the design and simulation of atypical spacecraft maneuvers. For example, large deployable structures, such as the IKAROS solar sail, would gain the ability to iteratively evaluate with greater fidelity the time varying control actuation of the solar sail during and after deployment [32].

Maneuvers which utilize SRP as an actuator are further demonstrated by the design of the rescue maneuver for the Hayabusa spacecraft shown in Figure 1.2. Hayabusa, an astroid return mission, lost attitude control about a single axis due to a failure of the spacecraft's reaction control system. Upon returning to a power positive state ground teams regained attitude control via the electric propulsion system at the expense of valuable fuel reserves. As a result, a cruise maneuver was designed which incorporated SRP to balance the torque induced by the swirling electric thrusters, thus saving fuel for the return journey to Earth[55].

In both the MESSENGER and Hyabusa examples mission designers used a variety of online and offline techniques to simulate and verify the beneficial effect of SRP on the spacecraft's trajec-

[1] NASA / JHU/APL, Artist depiction of the MESSENGER spacecraft in orbit around Mercury, Accessed September 1, 2017 from: http://commons.wikimedia.org/

Figure 1.2: Artist depiction of the Hayabusa spacecraft approaching asteroid Itokawa.[2]

tory and attitude. However, to perform this modeling and verification at high-fidelity, and with the reduced uncertainty required by a flight mission consumes significant human resources. In the case of MESSENGER, TCM maneuver planing began at minimum five weeks prior to the event [67]. Additionally, a-priori SRP models used in these analyses are typically not adjustable or tunable. If modeling parameter inclusion and accuracy requirements change during the lifetime of the mission a costly model revision or redevelopment process is required [6]. Recent research is exploring how to create control formulations to exploit the SRP forces further to assist with attitude and orbital considerations [59, 50]. In this context a faster than real-time SRP evaluation method presents the potential to reduce costs to a mission, provide analyst with the ability to perform 'what-if' simulations for iterative spacecraft, mission and designs, pre-launch and post-launch.

Ray tracing methods are considered to offer the greatest modeling fidelity and are performed as an offline precomputation to provide input data to further modeling abstractions where they can be computed in an online (faster than realtime) simulation [3]. Computing SRP with increased fidelity in materials optical properties, spacecraft shape and spacecraft articulation is computationally expensive. The advent of high computational power Graphics Processing Unit (GPU) hardware, particularly the presence of a GPU in almost all modern consumer grade computers, presents an exciting opportunity to leverage the latent computing power they offer. The advent

[2] JAXA, Artist depiction of the Hayabusa spacecraft approaching asteroid Itokawa, Accessed October 1, 2018 from: http://global.jaxa.jp/projects/sas/muses_c/

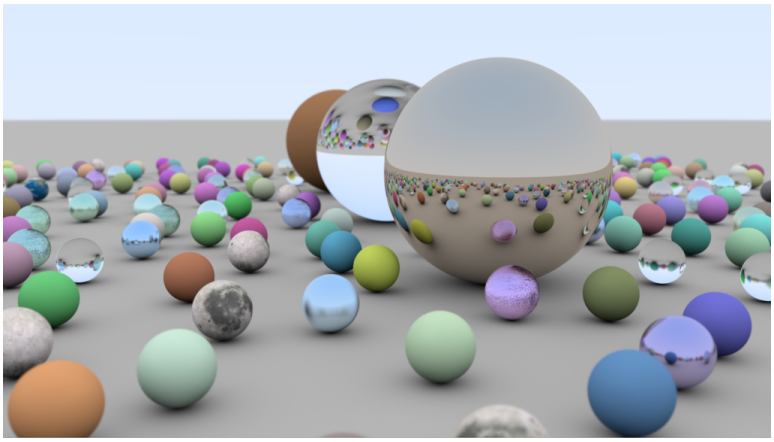


Figure 1.3: Ray traced scene of spheres with various material optical properties.[3]

of the category of computer processors referred to as systems-on-chips (SoC) or systems-on-modules (SoM) presents new opportunities to include computationally expensive algorithms as part of on-board flight software. These kinds of processors are most commonly seen in consumer mobile electronic devices. However, in recent years due to their low cost and useful computation power they are increasingly used as the processors aboard many nano sat and small sat spacecraft. The SoC devices often have as part of the chip a GPU, or parallel floating point arithmetic units which have the potential to facilitate the fast onboard execution of the dynamic models providing input to guidance and navigation systems.

To utilize this latent computing resource new algorithmic techniques are needed, which exploit the highly parallel execution environment of GPU hardware. Similar latent computing power is available in the form of remote computing or 'cloud' based computing resources. An example of a remote compute resource is a computationally powerful network connected single desktop computer, while an example of a cloud computing services is Amazon's Elastic Compute Cloud (EC2) [4] . Computation resources such as Amazon's EC2 provide on demand and scalable parallel and cluster computing resources, demonstrating cheap and ubiquitous latent computational resources available for 'everyday' analysis by an individual engineer[4]. Using these ubiquitous latent computational

[3] NVIDIA, Ray tracing demonstration, Accessed April 10, 2019 from: https://devblogs.nvidia.com/my-first-ray-tracing-demo/

[4] http://aws.amazon.com/ec2

resources presents an opportunity to overcome the computationally expensive aspects of hifidelity SRP modeling. Leveraging this latent computing power will offer SRP modeling approaches with the potential for a wide range of new 'everyday' analysis applications.

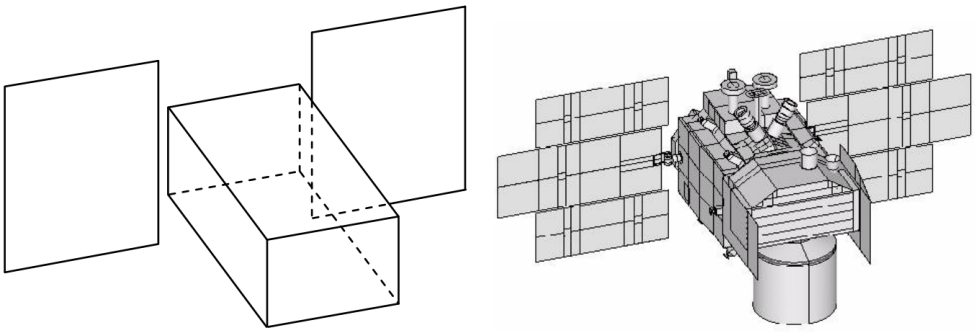
1.2 Background

A survey of the current landscape of RP research reveals a variety of approaches. The nature of the approaches can be characterized as analytic, semi-analytic or empirical. Whereas analytic models rely only on pre-launch engineering information, empirical models are constructed post-launch using flight data. Commonly, a semi-analytic model is used during a mission. These models are comprised of both analytic and empirical components with tunable parameters. Prior to flight, the tunable parameters are determined using an analytic model. Following launch, the parameters are incorporated into a parameter estimation process which tunes the model to more closely match flight data. Prominent examples of the three modeling approaches for SRP include the ROCK42 analytic model, the Bern semi-analytic model and the Jet Propulsion Lab (JPL) empirical model [28, 84, 6].

Continuing with SRP, the most basic analytic model employed is referred to as the cannonball model [56]. The cannonball model, given in Eq (1.1), is computed from the surface area upon which radiation is incident A, solar flux $\Phi_{\odot}$, the spacecraft mass M, speed of light c, heliocentric distance to the spacecraft r and the reflection, absorption and emission characteristics of the spacecraft surface which are grouped together within the coefficient of reflection C_r.

$$\boldsymbol{a}_{\odot} = -C_{\mathrm{r}} \frac{A\Phi_{\odot}}{Mc} \left(\frac{1AU}{r} \right)^2 \hat{\boldsymbol{s}} \tag{1.1}$$

Increased accuracy in analytic models is often achieved by representing the spacecraft as an approximation of various volumes or facets. A common approximation is to model the spacecraft bus and solar panels as a box and panels respectively. Increasing the fidelity of such box and wing models is often achieved by increasing the density on volumes and facets in an attempt to



(a) ICESat Micro Model (b) ICESat Micro Model

Figure 1.4: Two iterations of spacecraft model fidelity for ICESat [73].

approach the true shape of the spacecraft [57]. As shown in Figure 1.4, increasing spacecraft model accuracy was employed to great effect by the Ice, Cloud, and land Elevation Satellite (ICESat) mission. Additionally, the individual reflection, absorption and emission material characteristics are kept distinct for each surface and set based on known spacecraft material properties[61]. However, common among shape approximation methods is that they are augmented and become semi-analytic models where much of the modeling uncertainty is delegated to a parameter estimation process and the model is 'tuned' post-launch, to more accurately represent spacecraft tracking data.

Ray tracing approaches model, directly, the physics of light interaction with the spacecraft. For three decades ray tracing approaches, such as that developed by Antreasian and Rosborough in Reference [3], have been used to provide high geometric fidelity solar and thermal radiation pressure modeling for spacecraft. Similar modeling approaches are used for planetary albedo and Earth infrared radiation pressure (ERP). In these cases the same methods may be employed but rather than the sun, a model of the Earth albedo or ERP becomes the irradiating source to the spacecraft. Notably, Ziebart et al., developed an analytic modeling approach based on ray-tracing techniques for the assessment of SRP force analysis of spacecraft in the GLONASS constellation[96]. Ziebart's method precomputes the body forces over all 4π steradian attitude possibilities. Ziebart's approach is also capable of modeling self-shadowing and multiple ray reflection by ray-tracing a spacecraft model that comprises a set of volume primitives (boxes, cylinders etc.). Subsequent work extended this model to include albedo and ERP sources [95]. McMahon and Scheeres extend

Ziebart's approach to a semi-analytic model by aggregating the resultant SRP forces into a set of Fourier coefficients of a Fourier expansion[61]. The resulting Fourier expansion is available for both online and offline evaluation within a numerical integration process. Evaluation of the Fourier expansion in numerical simulation demonstrates successful prediction of the periodic and secular effects of SRP. Additionally, the Fourier coefficients may replace spacecraft material optical properties as parameters estimated during the orbit determination effort.

More recently, methods that make use of the parallel processing nature of GPUs have been developed. Tanygin and Beatty employ modern GPU parallel processing techniques to provide a significant reduction in time-to-solution of Ziebart's "pixel array" method[88] by generating dense lookup tables which are then queried via the GPU. With a similar goal, in Reference [51], the author demonstrates the use of the OpenGL vector graphics application programming interface (API) to dynamically evaluate the force of the incident solar radiation across a spacecraft structure approximated by many thousands of facets.

Further modeling approaches and associated studies have demonstrated that representing material optical properties, beyond basic mirror specular and lambertian diffuse, is a significant consideration when seeking high-fidelity SRP force resolution. Wetter et. al. demonstrate that accurate bidirectional reflection distribution functions (BRDF) representation is necessary in simulating the long duration propagation of spacecraft dynamics [94]. A material's BRDF governs the amount of impinging solar radiation absorbed and reflected and the directions in which the radiation is reflected. A typical BRDF description used for SRP modeling is comprised of diffuse lambertian and ideal, 'mirror like', specular reflection portions [40]. Absent in previous ray-tracing approaches is the physical modeling of diffuse ray reflections and by extension the ability to model arbitrary complex surface BDRFs.

1.2.1 General Purpose GPU Programming

The video game and animated video industries have driven the pursuit to create more vivid and realistic artificial worlds. This pursuit has resulted in highly optimized vector graphics software

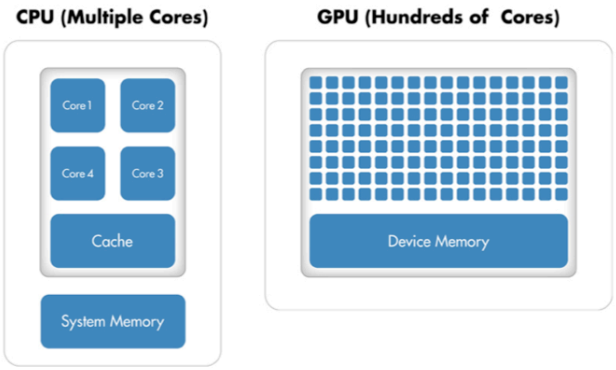


Figure 1.5: Illustrative demonstration of the number of cores, memory proximity and stages.

and GPU computer hardware capable of carrying out many thousands of floating point operations in parallel [68]. Use of the GPU computing power has been extended beyond the graphics realm to general-purpose processing (GPGPU) applications throughout the sciences.

As illustrated in Figure 1.2.1, current CPU hardware is characterized as a single instruction multiple thread (SIMT) device. A SIMT device accommodates instruction parallelism by executing multiple threads, potentially on multiple cores. Typically high data rates and bandwidth are facilitated by large cache mechanisms, high bandwidth and rapid adhoc memory access. This allows for a CPU to interleave and anticipate the execution of varied instructions from various processes and thus hide latency between processes. Conversely a GPU is characterized as a single instruction multiple data (SIMD) device. As a result GPUs achieve extreme computational efficiency with multicore designs that use both hardware multithreading and SIMD processing [27]. This device's architectural difference presents a key challenge of this thesis: to develop and implement algorithms which achieve the high level of data parallelism and minimal code branching needed to maintain the high throughput in the parallel execution environment of the GPU. These constraints require significant redesign of existing algorithms and data management strategies. Key considerations for redesign include efficient data packing, algorithm restructuring to reduce code branching, maximizing coherent memory reads and writes, and ensuring all executing GPU cores are doing so with full instruction occupancy [34]. If these constraints are accommodated the GPU hardware presents the option for significant reductions in time-to-solution for problems which would be considered

intractable in the real-time context.

The two GPU software APIs utilized in this work are Open Graphics Language (OpenGL) and Open Computing Language (OpenCL) . The OpenGL API is a language independent application programming interface (API) for rendering computer vector graphics [54]. OpenGL provides access to highly optimized code paths developed specifically for processing on the GPU. Each hardware vendor that produces a GPU device typically also releases accompanying software drivers for the hardware device. As a result the built in OpenGL API functionality used in this method, such as rasterization and fragmentation, exist across all OpenGL supporting GPUs and uses the highly optimized code paths with minimal additional code development. Similarly, the OpenCL API is a cross-platform standard for parallel programming across a range of processors including multi-core CPU devices and GPU devices [66]. Both APIs provide tools to send, retrieve and process data on an OpenGL and/or OpenCL compliant GPU/CPU.

1.2.2 Distributed Spacecraft Simulation

Software simulation is a necessary aspect of all aspects of space mission development. Various tools exist to simulate different subsets of components of the space system. These components include dynamics, kinematics and environment, ground system emulation and flight software emulation. In general, simulators tend to be sophisticated software products that are developed in parallel with the systems they are intended to test. This parallel development can impose constraints and limit the scope of application of the simulation tool. Consequently, the development of a general and flexible simulation software architecture presents opportunities for spacecraft simulation beyond a single application executing on a single machine. Further, a distributed simulation architecture allows for the integration of models which are developed to make use of the aforementioned latent and easily accessible computing power.

The architectural characteristics of software simulation dramatically influences its functionality and, therefore, applicability. With some generalization, simulation architectures are characterized by the degree to which system components are coupled. The coupling between simulation

components is manifested by the simulation structure, where the overall system may be either: integrated as a single system of required components; integrated as a modular system with optional components; or developed as a group of cooperative yet stand-alone components.

An example of a system which has increased coupling between components is Advanced Solutions Inc's (ASI) Spacecraft Object Library in STK (SOLIS). SOLIS is a commercial plug-in to the Analytical Graphics, Inc (AGI) Systems ToolKit (STK^{TM}) mission analysis software. The plug-in extends STK's orbit and space environment dynamics with the On-board Dynamic Simulation System (ODySSy^{TM}). ODySSy is an onboard spacecraft simulator providing additional models for rotational dynamics, sensors, actuators, power and thermal dynamics, and basic spacecraft control and guidance algorithms [23]. Using both SOLIS and ODySSy from ASI provides end-to-end spacecraft simulation functionality. However, it does so by requiring those tools specifically. There are minimal options to substitute one component with another which was not intended to operate with the SOLIS system.

A software suite which demonstrates increasing modularity in its architecture is the NASA Jet Propulsion Laboratory's Dshell system [9]. The Dshell system avoids tightly coupled components by establishing interconnections and communicating data between components via "connector signals". Connector signals allow each component to provide data to other components without requiring knowledge of the other components internals or availability [12]. The Dshell suite of components has grown since its initial development. Now, using the wide range of components developed for the Dshell system, engineers are able to support a wide variety of simulation configurations including both robotic and spacecraft simulation, software and hardware-in-the-loop testing and mission telemetry visualization [13].

The NASA Operational Simulator (NOS)[5] is a simulation system which exemplifies the characteristics of a loosely coupled system architecture. The NOS system is a generic software-only simulation architecture and was developed by NASA's Independent Verification and Validation (IV&V) Independent Test Capability (ITC). The NOS system achieves its flexible architecture by

[5] `https://www.nasa.gov/centers/ivv/jstar/jstar_simulation.html`

employing a message passing middleware application to connect various simulation components by a virtualized MIL-STD-1553 or SpaceWire messaging bus[80]. This middleware approach allows users to add or remove heterogeneous simulation components, unique to a particular spacecraft mission, without needing to rewrite or recompile model or application code [71].

A distributed simulation architecture enables analysts to incorporate models which are otherwise unable to run on a single machine. It is often the case that the personal computer on which analysis is being conducted has insufficient computational power to run computationally expensive models at the required speed or fidelity. Running such computationally expensive models is further complicated by the time consuming and expensive technical undertaking to re-architect entire simulation and analysis toolsets. Therefore, a means by which existing simulation toolsets can execute simulations in a distributed arrangement where most of the computation remains on the local machine while the computationally expensive module is executed on a remote more capable computing resource. Such examples may include simulation of a spacecraft's dynamics using a high fidelity ray traced radiation pressure model or a high degree spherical harmonics gravity model. On a single machine such a simulation would run much slower than realtime and Monte Carlo case studies would take prohibitively long to execute. However, running these expensive models on a computationally powerful remote GPU resource has the potential to reduce computation times to faster than realtime thus opening up increased options for analysis using existing simulation toolsets.

This dissertation presents the novel Blacklion (BL) simulation middleware architecture, which is inherently capable of distributed operation (multiple machine) and agnostic to the end point applications/models which participate in the simulation. The BL architecture facilitates the integration and execution of multiple software processes across heterogeneous computing platforms. For example, having a dedicated computer running a complex space environment model and another computer integrating spacecraft dynamics, both of them exchanging data dynamically through BL. Blacklion enables simulation use cases beyond that of the typical single application on a single machine.

1.3 Research Goals

In summary, the overarching novel contribution of this dissertation is to use the ubiquitous latent computational power available in consumer computing resources to enable the development, implementation and computation of SRP models with greater fidelity and computational speed than existing approaches. This work first aims to investigate how the unique parallel computing capability the GPU and distributed remote computing resources can be leveraged to produce SRP modeling approaches which provide computationally fast high geometric fidelity force resolution. Two primary SRP modeling approaches shall be investigated. The first approach resolves the per-facet SRP of a complex triangulated mesh model, while the second approach employs techniques in ray tracing to further capture optical behavior such as spacecraft self-reflection.

The developed SRP modeling approaches extend previous approaches documented in the literature by accommodating more complex material reflectance models, arbitrary changes to spacecraft articulation and doing so at faster than real-time computational speeds. The developed computational speed gives rise to the possibility for in the inclusion of high-fidelity SRP models in spacecraft ground software simulation, long-term dynamics propagation and Monte Carlo simulation. This work makes greater use of existing engineering data in the SRP modeling processes to reduce, where possible, uncertainty in the SRP force computation. Such pre-launch engineering data includes spacecraft geometry, material optical properties, and possible spacecraft time varying articulations.

Thirdly, this dissertation presents a modular implementation for each modeling method which allows for direct integration and online faster than real-time simulation within the Basilisk astrodynamics framework[6] . The pursuit of modularity culminates with the development and demonstration the Blacklion distributed simulation middleware which facilitates the execution of computationally demanding simulation models on arbitrary commodity computing resources.

This scope of this research is thus provided in the following list, grouped into two core research

[6] `http://hanspeterschaub.info/bskMain.html`

goals. They are:

(1) General Purpose GPU Solar Radiation Pressure Modeling Methods

- Introduce computational modeling methods which leverage the highly parallel execution environment of the GPU.
- Extend the modeling methods to accommodate arbitrary spacecraft articulation and complex material reflectance behaviors.
- Develop modular implementations of each SRP modeling method which facilitate integration into an astrodynamics simulator and demonstrates faster than realtime execution.

(2) Distributed Spacecraft Simulation Architecture

- Introduce a modular distributed simulation architecture which allows for the execution of simulation models in a mixed (local and/or remote) computing environment.
- Expand this architecture to demonstrate the execution of computationally intensive simulation models on remote computing resources.
- Provide examples of the ability for the simulation architecture and each SRP modeling method to provide improved resolution of the SRP force and torque upon a spacecraft simulated dynamics.

Chapter 2

SRP Theory

This section will present the fundamental theory describing the momentum imparted to a spacecraft by impinging solar photons. Two characterization of the resulting force are given. The first is a general description of the force due to an arbitrary surface bidirectional reflectance distribution function. The second characterization restates the commonly used expression which uses the idealized BRDF comprised of mirror-like specular and Lambertian diffuse reflection.

To begin, the momentum of a photon is defined. From Planck's law a photon with a frequency ν will transport an amount of energy E as

$$E = h\nu \tag{2.1}$$

where h is Planck's constant. Additionally, noting the mass -energy equivalence of special relativity yields the total energy of a moving body as

$$E^2 = m_0^2 c^4 + p^2 c^2 \tag{2.2}$$

where m_0 is the mass of the body when at rest, p the body's momentum and c the speed of light. Given that a photon has a zero mass at rest, its energy is given as the second term in Eq 2.2 and is

$$E = pc \tag{2.3}$$

The momentum of a single photon can be computed by equating Planck's law and its energy to give

$$p = \frac{h\nu}{c} \tag{2.4}$$

To compute the pressure exerted on a body by solar photons, first the momentum flux of photons must be determined. The energy flux ϕ at the spacecraft distance from the Sun, $|\boldsymbol{r}_{BS}|$ is

$$\phi = \phi_E \left(\frac{R_E}{|\boldsymbol{r}_{SB}|} \right)^2 \tag{2.5}$$

$$\phi_E = \frac{L_s}{4\pi R_E^2} \tag{2.6}$$

and ϕ_E the energy flux at the Earth's distance from the Sun. The pressure exerted is thus the change in momentum over a surface area with a normal direction vector aligned with the direction of the change in momentum.

$$P = \frac{1}{A} \left(\frac{\Delta p}{\Delta t} \right) \tag{2.7}$$

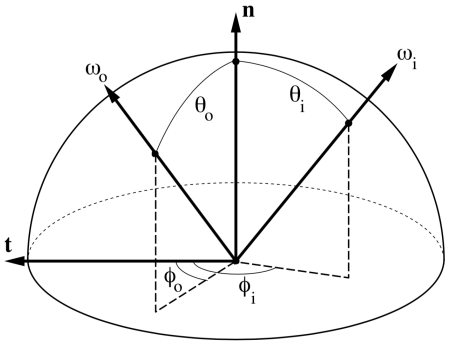
The final pressure can be simplified to

$$P = \frac{W}{c} \tag{2.8}$$

It is assumed that the incident light-surface interactions are occurring in an optical linear regime (an example of the non-linear regime are high-energy lasers). Under this linear regime it has been shown experimentally that there is a proportional relationship $dL_o(\omega_o) \propto dE(\omega_i)$. This allows for the development of the bidirectional scattering distribution function given in Eq. (4.18), where the proportionality relationship describes the observed radiance leaving a reflecting surface in the direction ω_o.

The geometry of the radiation pressure surface interaction is shown in Figure 2.1 where an observer is looking at a a ray intersection point on a surface $\boldsymbol{\omega}_o$ is the unit vector in the direction of the observer, $\hat{\boldsymbol{n}}_{\mathbf{x}}$ is the unit normal to the surface, and $\boldsymbol{\omega}_i$ is the unit vector in the direction of the incident radiation source. $\hat{\boldsymbol{h}}_{\mathbf{x}}$ is a normalized vector in the direction of the angular bisector of $\boldsymbol{w}_o$ and $\boldsymbol{w}_i$, and is defined by $\hat{\boldsymbol{h}}_{\boldsymbol{x}} = (\boldsymbol{\omega}_o + \boldsymbol{\omega}_i)/|\boldsymbol{\omega}_o + \boldsymbol{\omega}_i|$. From radiance the irradiance at a particular point $\mathbf{x}$ on a surface, defined as power per unit surface area, with units $[W \cdot m^{-2}]$ is given as

$$E(\mathbf{x}) = \frac{\mathrm{d}\phi(\mathbf{x})}{\mathrm{d}A(\mathbf{x})} \tag{2.9}$$



.

Figure 2.1: Reflectance geometry

A majority of expressions used to compute the SRP induced force hide the following subtlety; that *irradiance* of the space object due to radiation sources (e.g. albedo and solar), for each source is from a single direction and assumed to be a collimated beam. Each directional source can be handled individually and the aggregate of each source summed to obtain the total radiation force on the space object. This work includes the modeling of complex material bidirectional reflectance distribution functions which manifest the directional dependence of the *irradiance*. As a result the equations for radiation pressure on a space object will be presented in a general form which uses the definition of *radiance* rather than the immediate simplification to *irradiance*. The radiance is given in Eq.(2.10)

$$L(\mathbf{x},\omega) = \frac{\mathrm{d}^2\phi(\mathbf{x},\omega)}{\mathrm{d}A_x^{\perp}(\mathbf{x})\mathrm{d}\sigma(\omega)} \tag{2.10}$$

where the radiance is the flux arriving at $\mathrm{d}A_{\omega}^{\perp}(\mathbf{x})$, a surface element positioned at $\mathbf{x}$ perpendicular to the direction ω, contained within the solid angle element $\mathrm{d}\sigma(\omega)$.

The perpendicular surface area element $\mathrm{d}A_{\omega}^{\perp}(\mathbf{x})$ can be converted to its non-projected are element by $\mathrm{d}A_{\omega}^{\perp}(\mathbf{x}) = |\omega \cdot \hat{\boldsymbol{n}}_x|\mathrm{d}A_{\omega}(\mathbf{x})$ and the projection term $|\omega \cdot \hat{\boldsymbol{n}}_{\mathbf{x}}|$ associated with the $\mathrm{d}\sigma(\omega)$ term to give the relationship of irradiance to radiance in Eq.(2.11).

$$E(\omega_i) = L_i(\omega_i)\mathrm{d}\sigma^{\perp}(\omega_i) \tag{2.11}$$

It is assumed that the incident light-surface interactions are occurring in an optical linear

regime (an example of the non-linear regime are high-energy lasers). Under this linear regime it has been shown experimentally that there is a proportional relationship $\mathrm{d}L_o(\omega_o) \propto \mathrm{d}E(\omega_i)$. This allows for the development of the bidirectional scattering distribution function given in Eq. (4.18), where the proportionality relationship describes the observed radiance leaving a reflecting surface in the direction w_o.

$$f_s(\omega_i \to \omega_o) = \frac{\mathrm{d}L_o(\omega_o)}{\mathrm{d}E(\omega_i)} = \frac{\mathrm{d}L_o(\omega_o)}{L_i(\omega_i)\mathrm{d}\sigma^{\perp}(\omega_i)} \tag{2.12}$$

The relationship between the outgoing radiance and the incoming radiance for a particular optical surface is described at Eq.(4.19).

$$\mathrm{d}L_o(\omega_o) = \mathrm{d}L(\omega_i) f_s(\omega_i \to \omega_o)\mathrm{d}\sigma^{\perp}(\omega_i) \tag{2.13}$$

Integrating Eq.(4.19) yields the total radiance, over the hemisphere, leaving a surface area element. This relationship is often referred to as the *scattering function* [92].

$$L_o(\omega_o) = \int_{S^2} L(\omega_i) f_s(\omega_i \to \omega_o)\mathrm{d}\sigma^{\perp}(\omega_i) \tag{2.14}$$

To compute the force due to radiation pressure the directions of the force components must be defined. In general the radiation pressure force is composed of a force component in the direction of the incident radiance, a component determined by the reflected radiance and a component by the radiation of absorbed energy. The development of the force due to radiation pressure can then be defined generally in terms of radiance $L_i(\omega_i)$ for the force elements due to the incident radiation $\mathrm{d}\boldsymbol{F}_i$ and reflected radiation $\mathrm{d}\boldsymbol{F}_o$. Absent in this force balance is the component of absorbed radiation re-emitted $\mathrm{d}\boldsymbol{F}_e$. While important for a complete energy balance this presentation assumes that this component is accounted for in an appropriate thermal radiation model.

$$\mathrm{d}\boldsymbol{F}_i = \int_{S^2} \frac{L_i(\omega_i)|\boldsymbol{\omega}_i \cdot \hat{\boldsymbol{n}}_{\mathbf{x}}|}{c} \hat{\boldsymbol{\omega}}_i \mathrm{d}\sigma_{\mathbf{x}}(\omega_i)\mathrm{d}A_{\mathbf{x}} \tag{2.15a}$$

$$\mathrm{d}\boldsymbol{F}_o = \int_{S^2} \frac{L_i(\omega_i)|\boldsymbol{\omega}_i \cdot \hat{\boldsymbol{n}}_{\mathbf{x}}|}{c} f_s(\omega_i \to \omega_o)\mathrm{d}\sigma_{\mathbf{x}}(\omega_o)\mathrm{d}A_{\mathbf{x}} \tag{2.15b}$$

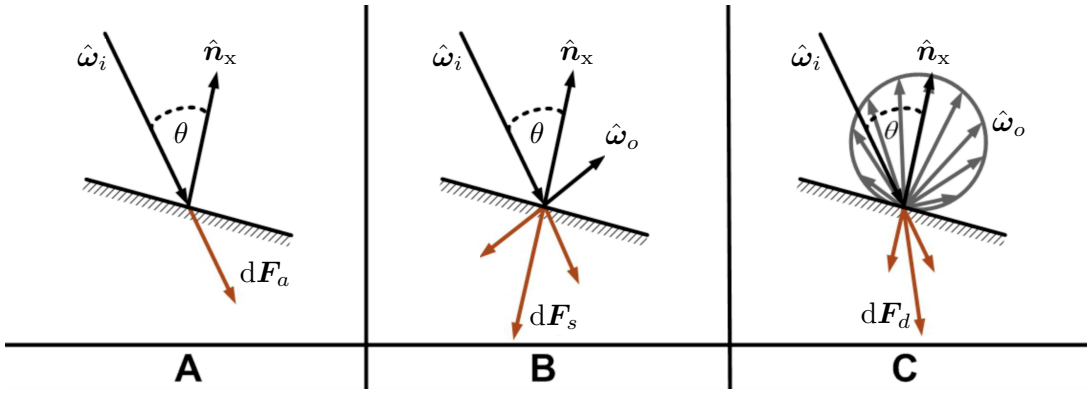


Figure 2.2: Reflection geometry for idealized specular and diffuse [72].

Of particular importance is Eq. (2.15b) where the direction of the resultant force is described by the distribution of scattered radiation over the hemisphere is described by the BSDF $f_s(\omega_i \rightarrow \omega_o)$. This work restricts scattering to the hemisphere normal in the direction of $\hat{\boldsymbol{n}}_{\mathbf{x}}$. As a result the $f_s(\omega_i \rightarrow \omega_o)$ notation is changed to $f_r(\omega_i \rightarrow \omega_o)$ and describes the bidirectional reflectance distribution function (BRDF).

Typical expressions for the radiation pressure assume a simplified BRDF comprised of a purely mirror-like specular component and a diffuse component (often Lambertian). The action of the idealized specular and diffuse reflection are shown in Figure 2. The incoming radiation is taken irradiance from an unpolarized plane wave front. These equations are given as

$$\mathrm{d}\boldsymbol{F}_a = P|\boldsymbol{\omega}_i \cdot \hat{\boldsymbol{n}}_{\mathbf{x}}|\mathrm{d}A_{\mathbf{x}}\hat{\boldsymbol{\omega}}_i \tag{2.16a}$$

$$\mathrm{d}\boldsymbol{F}_d = B_{\mathrm{f}}\gamma(1-s)P|\boldsymbol{\omega}_i \cdot \hat{\boldsymbol{n}}_{\mathbf{x}}|\mathrm{d}A_{\mathbf{x}}\hat{\boldsymbol{n}}_{\mathbf{x}} \tag{2.16b}$$

$$\mathrm{d}\boldsymbol{F}_s = -\gamma s P|\boldsymbol{\omega}_i \cdot \hat{\boldsymbol{n}}_{\mathbf{x}}|\mathrm{d}A_{\mathbf{x}}\hat{\boldsymbol{s}} \tag{2.16c}$$

where γ is the portion of the incoming radiation reflected by the surface material, s the portion of that which is reflected as specular reflection, B_{f} the diffuse reflection coefficient which is typically taken as 2/3 for Lambertian diffuse reflection. The mirror-like reflection direction, $\hat{\boldsymbol{s}}$, in Eq. (2.16) is given by

$$\hat{\boldsymbol{s}} = \hat{\boldsymbol{\omega}}_i - 2|\hat{\boldsymbol{\omega}}_i \cdot \hat{\boldsymbol{n}}_{\mathbf{x}}|\hat{\boldsymbol{n}}_{\mathbf{x}} \tag{2.17}$$

These equations will be used in various forms throughout this work and where appropriate additional formulations and arrangements will be detail at their point of use.

Chapter 3

OpenGL-OpenCL Solar Radiation Pressure

Modeling with high geometric fidelity, the SRP induced force and torque, on a spacecraft is challenging due to the often computationally expensive modeling requirements. Of these computationally expensive modeling requirements, three in particular present the greatest challenge. These requirements are to resolve arbitrary time varying articulated spacecraft shape models, spacecraft self shadowing, and varied arbitrary material optical properties. Typically spacecraft geometries are kept simple, ignoring important spacecraft detail which has a significant degradation of a model's ability to more closely evaluate the true SRP force and torque. Further, methods which do capture changes in spacecraft articulations and self shadowing do so as part of an offline evaluation which generate SRP force and torque lookup tables. Such offline evaluations are executed multiple times to accommodate the myriad different spacecraft configurations.

The OpenGL-CL modeling method presented in this chapter is able to to capture the three aforementioned modeling requirements as part of an implementation which has computational speed suitable for online execution. The approach builds upon the author's previous OpenGL faceted based approaches[52] and extends the application of OpenCL to allow for more flexible arbitrary computation. Additionally, the OpenGL-CL has many parallels to the work presented by Tanygin and Beatty in Reference [88] and incorporates certain algorithmic decisions made by that work. The goal of this section is to introduce the OpenGL and OpenCL APIs and the algorithmic steps of the OpenGL-CL modeling method. To begin, the OpenGL API is introduced in the context of the render pipeline. This is followed by a description of the important OpenGL-OpenCL shared

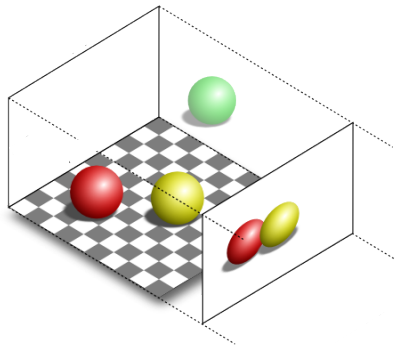


Figure 3.1: The OpenGL API used to generate an orthographic projection of a multiple mesh models.

memory context functionality. An optimized OpenCL kernel is developed to perform a parallel reductions across the rendered pixel space and thus the final force and torque vectors. Initial validation is provided and is followed by more complex spacecraft simulations which demonstrate the method's capability to capture the difference between spacecraft mesh models, while comfortably accommodating detailed meshes of many thousands of vertices.

3.1 The OpenGL Render Pipeline

The Open Graphics Library (OpenGL) is a language independent API for rendering computer vector graphics [81]. The API provides tools to send, process and retrieve data on OpenGL compliant GPUs. A rendered scene is generated by processing the vertices and primitives (triangle, polygon) of a mesh model within the OpenGL pipeline. The OpenGL pipeline allows for various stages to be programmable. A programmable stage is called a Shader Program or simply referred to as a shader. Each shader is a mini-program which serves to process vertices and primitives in a particular manner. Shaders are written using the OpenGL Shader Language (GLSL) and each shader stage has a defined set of data types as inputs and outputs, which are passed along the pipeline to subsequent shader stages. The default OpenGL render pipeline is shown in Figure. 3.2 identifying required and optional processing stages.

A shader stage operates on a single vertex, a set of vertices that define a shape primitive or a

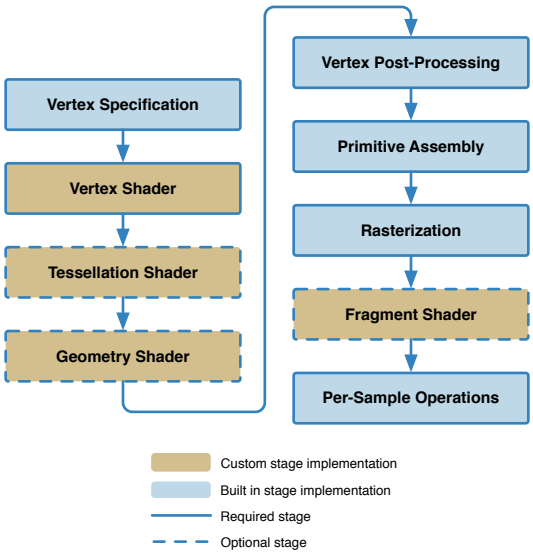


Figure 3.2: The default OpenGL pipeline.

fragment (rasterized pixel). Each of the vertices or primitives, are processed in parallel where thousands of shader instances are executed simultaneously for each stage in the pipeline. It is this highly parallel per vertex/primitive operation (many thousands of evaluations occurring simultaneously) for which GPU devices have been specifically designed.

A simplified representation of the default OpenGL render pipeline stages is shown in Figure. 3.3. A minimally valid OpenGL pipeline requires the implementation of the Vertex Shader and Fragment Shader stages. The addition of further Shader stages allows the software developer to create a custom render pipeline. The Vertex Shader processes the individual vertices of the model having vertex data as both input and output. An optional Tessellation Shader stage operates on patches of vertex data which are subdivided into smaller primitives (e.g. a large triangle into multiple smaller triangles). The optional Geometry Shader has as input a single primitive and may output one or more primitive definitions. Finally, the Fragment Shader computes per pixel operations following OpenGL's internal depth testing and rasterization processes. The two shader stages leveraged in this approach are the Vertex and Fragment Shader stages.

Vertex Shader (VS): processes the individual vertices of the model having vertex data as both input and output. The VS is used to perform setup for later shader stages by performing coordinate

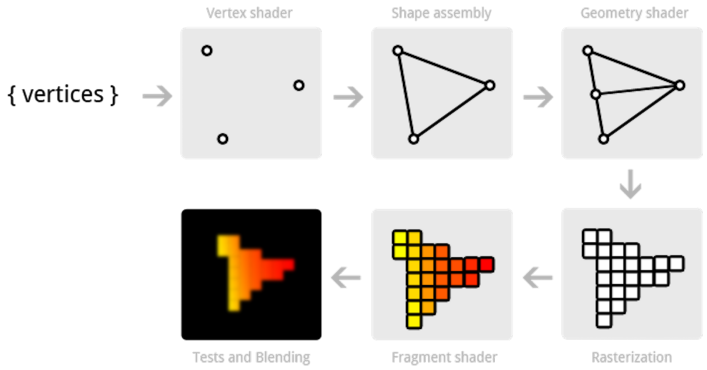


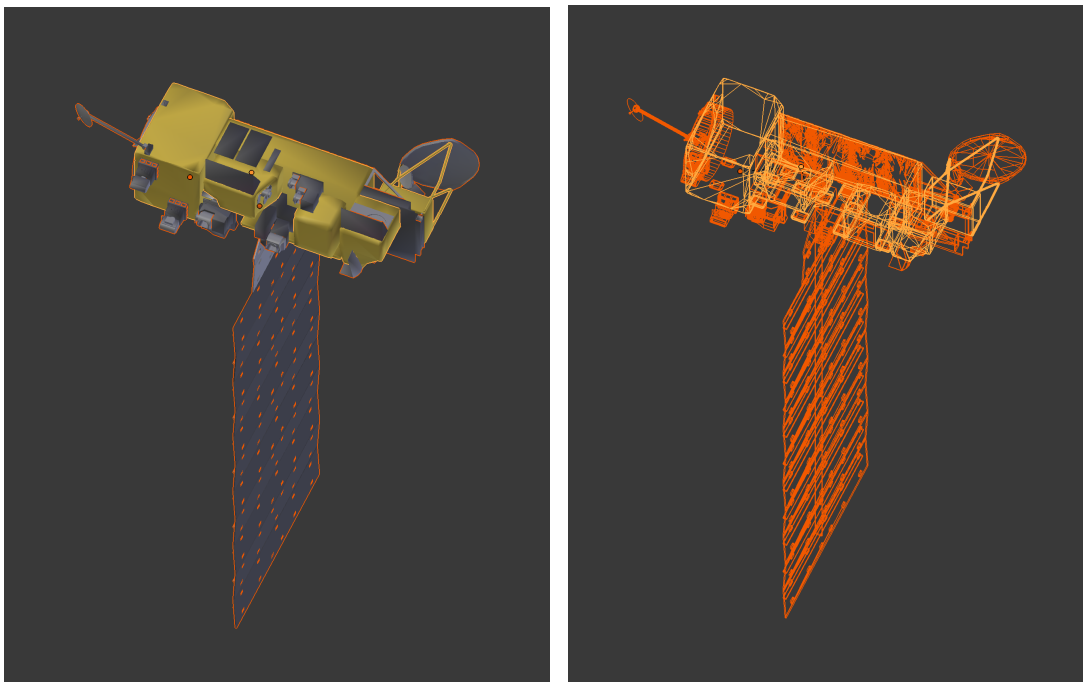
Figure 3.3: Notional operations for each shader stage.

frame transformations on vertex data by mapping vertices from the model body coordinate frame to the world, view and projection coordinate frames. As shown in Figure. 3.2 the vertex shader is a programmable and required stage in the pipeline.

Fragment Shader (FS): is executed after the pipeline has rasterized the projected scene. To rasterize the scene, the vertices of each primitive are mapped from $\mathbb{R}^3$ to $\mathbb{R}^2$ projection space samples. Each fragment/pixel can be manipulated within the FS and then written to one or many Texture objects attached to a Framebuffer. The Texture object data format is one of either a single or four (RGBA) 32-bit IEEE single precision floating point values per pixel. Typically an RGBA value is output to a texture for each pixel. For a typical render it is the color texture which is displayed to the screen.

3.2 Mesh Definition

Both modeling approaches presented in this work use a triangulated mesh model to approximate the spacecraft shape with high geometric accuracy. A triangulated mesh model provides a consistent input to the method and removes the need for code which handles a multitude of other primitive types. The model data format chosen is the Wavefront Object (.OBJ) [11]. The file format is user friendly due to its wide spread support by 3D modeling and animation tools, and it is simple to debug because it is human readable within a text editor (when encoded as the ASCII encoded file variant). Figures 3.4(a) and 3.4(b) show the .OBJ mesh model of the Aqua spacecraft



(a) Aqua spacecraft with materials. (b) Aqua spacecraft mesh.

Figure 3.4: Aqua spacecraft .OBJ mesh model.

with complete materials and only mesh structure, respectively.

The file format stores vertex positions, texture vertices and normal vectors in lists. Vertices are defined as x, y, z and w where w is an optional scaling component and defaults to 1.0. Texture coordinates are given in u, v and w coordinates, ranging between 0 and 1.0. Primitive normal vectors may be provided as x, y, z coordinates. If normal definitions are not present in the file the import code will generate consistent facet normal vectors using the counter clockwise ordered list of vertices defining the facet. The .OBJ file format may be accompanied by multiple Material Template Library (.MTL) files. The .MTL file defines common material properties associated with model shading or rendering. For the works of this dissertation the .MTL file is overloaded and a number of it's variables are taken to have a slightly different meaning than typical. Two key examples of this are the `Kd` and `Ks` parameters which indicate the RGB color mixture of the diffuse and specular optical phenomena for a material. Here, these variables are used as the diffuse and specular reflection coefficients commonly associated with faceted SRP computations. As such only

Listing 3.1: Wavefront .obj file format for a mesh model.

```
# Vertex coordinates
v1 0.123 0.234 0.345 1.0
v2 ...
# Texture coordinates
vt1 0.500 1 [0]
vt2 ...
# Face/vertex normal definitions
vn1 0.707 0.000 0.707
vn2 ...
# Face definitions
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3
```

the R channel of the RGB values is used for `Kd` $= \rho$ and `Ks` $= \gamma$. Overloading these variables allows for rapid manipulation of the spacecraft mesh model's material properties, through a 3D animation tool such as Blender, easy export from this tool and, consequently, import at run-time.

3.3 Custom OpenGL Render Pipeline

The custom OpenGL pipeline developed here builds upon the original presentation where Tanygin and Beatty employ the built in depth testing and rasterization stages of OpenGL to equivalently determine the first ray-surface interaction of a ray tracing approach. Similar to the earlier work this method implements custom VS and FS stages. An overview of this custom pipeline is shown in Figure. 3.5. The VS stage transforms mesh vertices to the projection frame (which shall be defined in Sec. 3.4) and outputs the normal, position and material parameters associated with the processed vertex. The pipeline depth testing and rasterization will determine which vertex values are sunlit, the outputs from the VS stage are provided as input for the corresponding pixel of the FS stage.

The FS executes for each pixel in the specified view port. The FS stage receives data for each corresponding pixel some of which will contain samples from the spacecraft mesh model that are visible from the sun-heading direction. The FS outputs values to Textures attached to

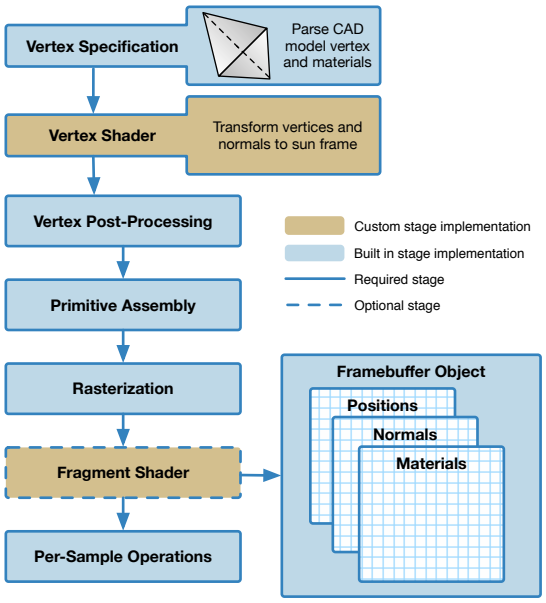


Figure 3.5: Overview of OpenGL render pipeline with required custom vertex shader and fragment shader stages outputting to the Textures held in a Framebuffer object.

a Framebuffer object. A Framebuffer object (FBO) allows a developer to define a non-default destination for render output. Results from the FS stage, shown in Figure 3.5, are written into the FBO's attached textures. Developers can manipulate the active FBO and the Texture storage attached to the FBO. This allows a developer to perform multiple render passes on a particular scene before performing a final compositing operation and displaying the Framebuffer on screen.

The FBO is a destination for specific data types generated during render. To store this data Texture objects are attached to the FBO. Textures are defined as a single array of pixels with a certain dimensionality (1D, 2D or 3D), and having a particular data format. A texture can be either an array of single values or 4-component vectors. Each value component of a single or vector value is specified as signed/unsigned integer, sign/unsigned normalized integer or 32-bit IEEE floating-point [54]. The FBO and associated Texture data structures are a key enabling portion of the OpenGL API which allows for the passing of OpenGL generated data to the subsequent OpenCL SRP computation. Specifically, 2D Textures, into which the spacecraft mesh model vertices, normal vectors and material optical proprieties are written are ultimately passed to the OpenCL SRP computation.

Listing 3.2: Creating a FBO, a texture and attaching to the FBO as a render target.

```
gl::GenFramebuffers(1, &m_fbo);
gl::BindFramebuffer(gl::FRAMEBUFFER, m_fbo);
gl::GenTextures(1, &m_rparams.pixelPosMap);
gl::BindTexture(gl::TEXTURE_2D, m_rparams.pixelPosMap);
gl::TexImage2D(gl::TEXTURE_2D, 0, gl::RGBA, fboWidth, fboHeight, 0, gl::RGBA, gl::FLOAT, NULL);
gl::FramebufferTexture2D(gl::FRAMEBUFFER, gl::COLOR_ATTACHMENT1, gl::TEXTURE_2D,
    m_rparams.pixelPosMap, 0);
```

The type of Texture object attached is determined by the data type being stored. OpenGL provides specific texture attachment points to which particular data products are written. For example, pixel depth values (distance from projection plane to mesh) are written to a texture at attachment point `DEPTH_ATTACHMENT`, while vector based data is written to any one of a number of `COLOR_ATTACHMENTi` attachment points, where `i` is the index of the attachment point. An example of this process is shown in Listing 3.2. In this listing the single output texture which will be used to store the position vector, with respect to the spacecraft center of mass, of a sampled point on the mesh model. This texture of position vectors is then used in the OpenCL computation of SRP torque.

3.4 OpenGL Algorithm Steps

The OpenGL portion of the algorithm moves through four phases. The first phase is the computation of the spacecraft mesh model projection into the sun frame, which shall produce the Projection, View, and Sun transformation matrices. Mesh articulation operations follow to configure the time varying kinematics of the spacecraft mesh as they vary during run-time. The third stage is carried out in the VS where the frame transformations of the Projection, View and Sun matrices are applied to each mesh vertex. Finally, the FS outputs to Textures the values to be used by the OpenCL kernel.

3.4.1 Recursive Bounding Box Computation

An axis aligned bounding box (AABB) computation simply loops through all mesh vertices and finds the furthest extents of each mesh vertex component, in each of the mesh body frame axes. From these furthest extents the vertices defining the corners of the bounding box can be computed. Such a computation is sufficient if the mesh vertices are not articulated and therefore computed only once at initialization. In the case where a model is comprised of multiple sub-meshes, which may be articulated, the naive AABB computation leads to repetitive computations and significantly increased computation time. A recursive AABB bounding box algorithm is implemented to reduce computation time. This algorithm relies on the constraint that while individual sub-meshes may be articulated and therefore move within the model body frame, the vertices within each sub-mesh are fixed. In other words, all sub-meshes in a model are rigid bodies. This constraint allows for the computation of an AABB for each sub-mesh once at the start of a simulation. As shown in the pseudo-code of Listing. 1, the AABB can now be computed recursively by computing the AABB of the vertices vertices which define the corners of each sub-mesh's AABB. This reduces the order of magnitude for the number of vertices to evaluate from potentially 10^5 to 10^1 or, in the very worst cases (models with greater than 10 sub-meshes) 10^2.

3.4.2 Mesh Articulation

The spacecraft mesh model defines the vertex vectors, normal vectors, indices and material optical properties of the spacecraft. Typically a spacecraft is made up of a number sub-meshes, each which defines separate components of the spacecraft. Segmenting a spacecraft model into multiple sub-meshes is required for two purposes. The first purpose is that for a large majority of 3D mesh model formats, only a single set of material optical properties can be assigned to a sub-mesh. To accommodate the variety of spacecraft materials of which a spacecraft model is comprised, all vertices that make up regions of the spacecraft with the same material properties, must be defined together in a sub-mesh. The second reason for utilizing sub-meshes is to facilitate arbitrary run-

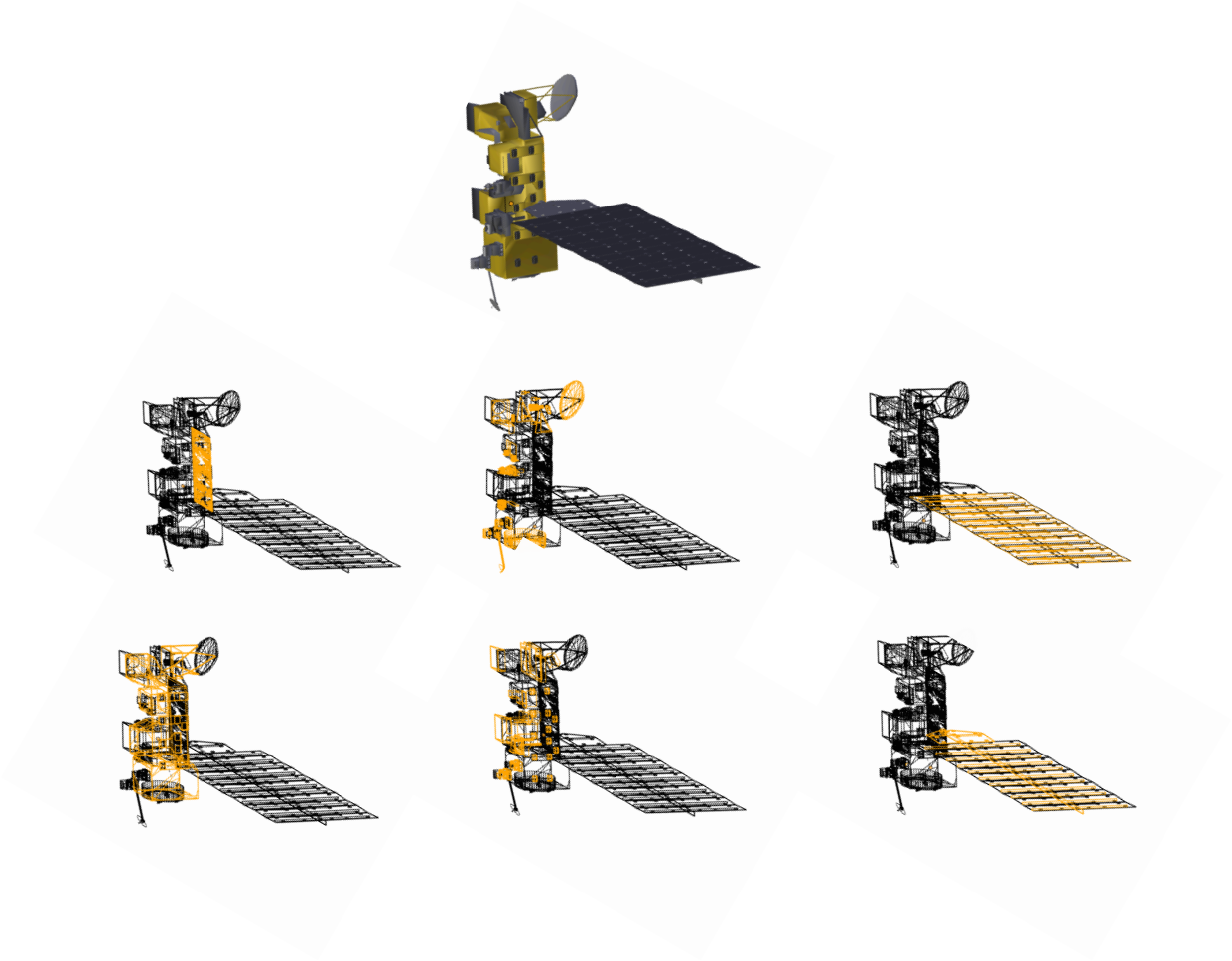


Figure 3.6: Sub-meshes of the Aqua spacecraft mesh.

```
   Data: node is the node in the mesh model tree structure
 1 if node is leaf then
 2 |   node.bbox ← computeBBox(node.vertices);
 3 |   node.bbox.transform(node.transformation);
 4 |   return
 5 end
   // node is not a leaf so we loop through all child nodes and call the
      function again
 6 bBoxUnion(i, j);
 7 for node in nodes(i) do
 8 |   computeNodeBBox(node);
   |   // For each node accumulate the child bounding boxes
 9 |   tmpBoxVertices(8, 3) ← node.bbox.getBBoxQuadMeshVertices ();
10 |   bBoxUnion(i, j)_{i=end} ← tmpBoxVertices(8, 3);
11 end
12 node.bbox ← computeBBox(bBoxUnion);
13 node.bbox.transform(node.transformation);
14 return
```

Algorithm 1: Recursive AABB algorithm.

time articulation of spacecraft components such as solar panel structures, antenna and instruments. Sub-meshes provide a convenient data structure whereby the vertices, indices, normals and material properties of a mesh are defined with an associated homogeneous transformation matrix. During simulation this transformation matrix is updated at each time step according to the time varying kinematics of the sub-mesh and the transformation is applied to the mesh thus performing the articulation.

All mesh vertex and normal data is defined with respect to the model frame body coordinate system $\mathcal{B} : \{\hat{\boldsymbol{b}}_1, \hat{\boldsymbol{b}}_2, \hat{\boldsymbol{b}}_3\}$. However, it is also convenient to allow the user to define sub-mesh articulations with respect to a coordinate frame, $\mathcal{C} : \{\hat{\boldsymbol{c}}_1, \hat{\boldsymbol{c}}_2, \hat{\boldsymbol{c}}_3\}$, with origin C and orientation different to the B coordinate frame. Such a frame may define the position and orientation of a solar panel with respect to its fixed hinge or gimbal point on the spacecraft bus.

As illustrated in Figure 3.7, the transformation requires the definition of the origin and orientation of the sub-mesh articulation frame C with respect to the model frame B. Obtaining a vertex in B frame components from one defined in C frame components is achieved via the following

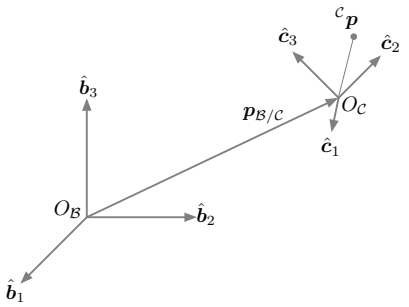


Figure 3.7: Illustration of two coordinate frames with different origins and orientations.

rotation and translation

$$^{\mathcal{B}}\boldsymbol{p} = {}^{\mathcal{B}}\boldsymbol{p}_{B/C} + [BC]^{\mathcal{C}}\boldsymbol{p}, \tag{3.1}$$

where $[BC]$ is the direction cosine matrix (DCM) defining the orthogonal transformation of a C frame vector to the B frame [76]. This translation and rotation operation can be concisely expressed as a 4×4 homogeneous transformation as

$$[\mathcal{BC}] = \begin{bmatrix} [BC] & {}^{\mathcal{B}}\boldsymbol{p}_{C/B} \\ 0_{1\times 3} & 1 \end{bmatrix}. \tag{3.2}$$

Assuming that all mesh vertices are initially defined in the body frame then the transformation described by Eq. (3.1) must be reversed. This is easily achieved by employing the inverse of the homogeneous transformation given as

$$[\mathcal{BC}]^{-1} = \begin{bmatrix} [BC]^T & -[BC]^T{}^{\mathcal{B}}\boldsymbol{p}_{C/B} \\ 0_{1\times 3} & 1 \end{bmatrix} \tag{3.3}$$

This transform yields a vertex defined in the C frame as[76]

$$^{\mathcal{C}}\boldsymbol{p} = [\mathcal{BC}]^{-1}{}^{\mathcal{B}}\boldsymbol{p} \tag{3.4}$$

Now that the vector $^{\mathcal{C}}\boldsymbol{p}$ is suitably defined in the C frame it can be transformed by the homogeneous transformation $[\mathcal{RC}]$ defining the articulation of the sub-mesh by

$$^{\mathcal{R}}\boldsymbol{p} = [\mathcal{RC}]^{\mathcal{C}}\boldsymbol{p} \tag{3.5}$$

The homogeneous transformation matrix obeys the same successive transformation property as the direction cosine matrix as exmplified in the following transformation[76].

$$^{\mathcal{N}}\boldsymbol{p} = [\mathcal{NA}][\mathcal{AB}]^{\mathcal{B}}\boldsymbol{p} = [\mathcal{NB}]^{\mathcal{B}}\boldsymbol{p} \tag{3.6}$$

A result of this successive transformation property is that sequential frame definitions and the subsequent mappings from one mesh articulation frame to a sub-mesh articulation frame can be carried out recursively. Such as demonstrated in the field of robotic manipulators, each mapping builds upon the previous. This facilitates an intuitive input for sub-mesh articulations where a parent sub-mesh holds references to further child sub-meshes and the sub-mesh transformation is defined relative to its parent mesh, rather than the body or inertial coordinate frames.

3.4.3 Vertex and Fragment Shader Stages

To facilitate OpenGL's depth testing and rasterization process, the model's vertices are to undergo three primary coordinate frame transformations;

(1) B frame to S frame, $[\mathcal{SB}]$

(2) S frame to view frame V, $[\mathcal{VS}]$

(3) V frame to projection frame P, $[\mathcal{PV}]$

The sun frame $\mathcal{S} : \{\hat{\boldsymbol{s}}_1, \hat{\boldsymbol{s}}_2, \hat{\boldsymbol{s}}_3\}$ is constructed where the sun heading in body frame components $^{\mathcal{B}}\hat{\boldsymbol{s}}$ is used as the first basis vector $\hat{\boldsymbol{s}}_1$. The remaining basis vectors, $\hat{\boldsymbol{s}}_2$ and $\hat{\boldsymbol{s}}_3$ are computed to provide an orthogonal frame. A sun frame loose AABB where the origin S is coincident with the centroid of the sun AABB

To facilitate the generation of the View and Projection coordinate frames a *loose* sun frame AABB is computed. This bounding box is referred to as loose because it is computed as the bounding box of the body frame bounding box vertices transformed into the sun frame. To compute a *tight* sun frame bounding box requires that all sub-mesh vertices be transformed into the sun frame and then the sun frame AABB computed from those sun frame vertices. Computing the

tight sun frame AABB would require operating on tens of thousands of vertices at each time steps; computing the loose sun frame AABB may require a few hundred at most (8 vertices for each sun-mesh bounding box).

The View frame is constructed with its origin at the centroid at the face of the sun frame bounding box which lies between the sun and the model. The projection frame is constructed as an orthographic projection of the mesh model into this same plane.

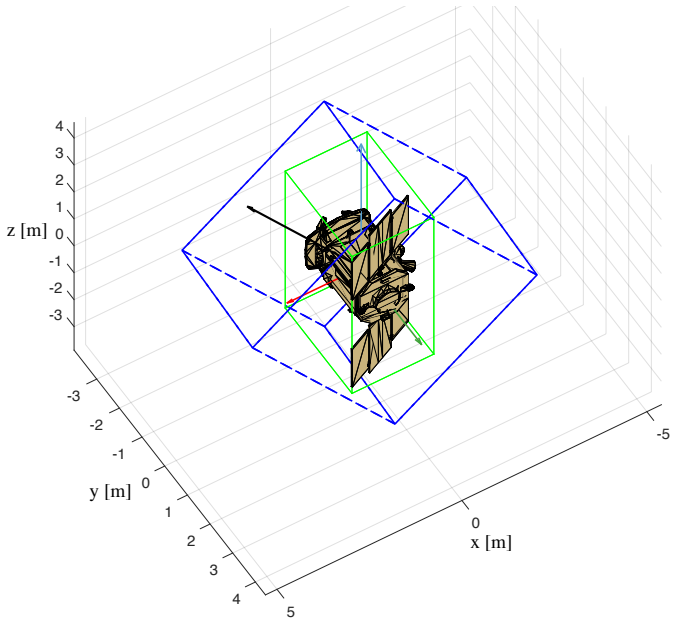


Figure 3.8: Loose sun frame AABB (dashed blue) and body frame AABB (solid green). Body frame axes $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{z}}$ are red, green and blue respectively and the sun heading $\hat{\boldsymbol{s}}$ as black.

Of the six sides of the loose sun frame bounding box, the centroid of the plane which has its normal as $-^{\mathcal{B}}\hat{\boldsymbol{s}}$ is set as the vector $^{\mathcal{S}}\boldsymbol{e}$ commonly referred to as the 'eye' location. This is the position of the notional camera. The center of the bounding box is set as $^{\mathcal{S}}\boldsymbol{c}$ and referred to as the camera 'target' vector. It is necessary to set the target vector at the center of the loose sun frame AABB rather than the model body frame because this ensures the model's extents are centered and captured within the view. The 'eye' and 'target' definitions allow a set of unit vectors to be

defined as

$$\hat{\boldsymbol{f}} = \frac{\boldsymbol{e} - \boldsymbol{c}}{|\boldsymbol{e} - \boldsymbol{c}|} \tag{3.7a}$$

$$\hat{\boldsymbol{u}} = [0, 1, 0]^T \tag{3.7b}$$

$$\hat{\boldsymbol{s}} = \hat{\boldsymbol{f}} \times \hat{\boldsymbol{u}} \tag{3.7c}$$

These unit vectors are used as the basis for the View matrix which is constructed as

$$V = \begin{bmatrix} \hat{\boldsymbol{s}}^T & 0 \\ \hat{\boldsymbol{u}}^T & 0 \\ -\hat{\boldsymbol{f}}^T & 0 \\ 0_{1\times 3} & 1 \end{bmatrix} \tag{3.8}$$

The output from the VS stage and therefore input to the FS stage requires that vertices be mapped from the view frame to the Normalized Device Coordinates (NDC) frame. The NDC space is defined as a cube with extents in all three axes of [-1, 1]. The orthographic projection matrix performs the mapping from View frame coordinates to NDC. The projection matrix P is constructed as

$$P = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$

where l is the left, r the right, b the lower extent, t the top extents, n the near plane of the volume and f the far plane of the volume.

The transformations are applied to each model vertex within the vertex shader. The body frame vertex is transformed to its sub-mesh articulation fame as

$${}^{\mathcal{R}}\boldsymbol{p} = [\mathcal{B}\mathcal{R}]^{-1}{}^{\mathcal{B}}\boldsymbol{p}. \tag{3.10}$$

The vertex is transformed from the original articulation frame R' to its updated articulated frame R' as

$${}^{\mathcal{R}'}\boldsymbol{p} = [\mathcal{R}'\mathcal{R}]^{\mathcal{R}}\boldsymbol{p}, \tag{3.11}$$

and then transformed back to the body frame

$$^{\mathcal{B}}\boldsymbol{p} = [\mathcal{B}\mathcal{R}']^{\mathcal{R}'}\boldsymbol{p}. \tag{3.12}$$

The vertex shader then transforms the body frame vertex to NDC by

$$^{\mathcal{P}}\boldsymbol{p} = [\mathcal{P}\mathcal{V}][\mathcal{V}\mathcal{S}][\mathcal{S}\mathcal{B}]^{\mathcal{B}}\boldsymbol{p}. \tag{3.13}$$

It is the vertex mapped to the NDC frame, which OpenGL will process for depth testing and then rasterization and ultimately use to determine which vertex values are sunlit and subsequently passed through to the FS stage.

As a final computation, the alpha component of each the RGBA value of the Texture containing the normal vectors, is written as the norm of the normal vector. The OpenGL `glClearColor`parameter controls the color used to reset the values in each pixel of color buffers when cleared between rendering frames. Here, the `glClearColor` parameter is set to an RGBA vector of (0.0, 0.0, 0.0, 0.0). Thus, if a pixel is unoccupied by the spacecraft mesh, the norm will be zero (due to the black color buffer value) and if occupied greater than zero. Setting the alpha component provides the OpenCL stage with a flag to avoid unnecessary computation given the following condition: if the value of the normal vector's fourth component is greater than zero, continue to compute SRP force and torque, otherwise return a zero vector for force and torque.

3.5 OpenCL Algorithm Steps

The OpenCL algorithm is contained within a single kernel program. This kernel program performs both the force and torque computation and a parallel reduction summation of each pixel contribution. The kernel program aims to reduce GPU memory read and write collisions, reduce code branching and remove unnecessary instruction overhead by unrolling loops. For each pixel the force computed as

$$\boldsymbol{F}_{\odot_k} = -P(|\boldsymbol{r}_{\odot}|)A_k \cos(\theta_k)\left\{(1-\rho_{s_k})\hat{\boldsymbol{s}} + \left[\frac{2}{3}\rho_{d_k} + 2\rho_{s_k}\cos(\theta_k)\right]\hat{\boldsymbol{n}}_k\right\} \tag{3.14}$$

where for each pixel k the coefficients of diffuse reflection ρ_{d_k}, specular reflection ρ_{s_k} are contained in one 2D Texture object, the surface normal $\hat{\boldsymbol{n}}_k$ contained in a different 2D Texture object. The torque is computed as

$$\boldsymbol{L}_{\odot_k} = \boldsymbol{r}_{P/C} \times \boldsymbol{F}_{\odot_k} \tag{3.15}$$

where the position vector $\boldsymbol{r}_{P/C}$, the point P of action of the force relative to the spacecraft center of mass at point C is contained in a third 2D Texture object.

At its simplest a parallel reduction algorithm aims to sum all the elements from a set by recruiting multiple threads or processors to each iteratively sum two elements until the final sum is obtained. A naive implementation may sum all elements in a binary tree operation sequence. While parallel, such an implementation does not account for the particular mechanisms by which GPUs provide parallel computation. These particulars included memory access patterns, instruction overhead and kernel launch time.

To reduce memory access collisions, sequential address striding is used to load the values to be summed from shared memory. Sequential addressing loads values from separate memory banks on the GPU thus avoiding contention from multiple threads attempting to load values from the same memory bank at the same time. An example of this sequential addressing procedure is shown in Figure. 3.9. In this example there are two OpenCL Work Groups (WG) which each contain two Work Items (WI). While terminology and low level chip design changes from GPU to GPU and vendor to vendor, conceptually a WG is a processor and a WI is a thread within that processor. Each WG is accessing a contiguous block of memory and each WI is sequentially addressing values to sum, thus avoiding access collisions.

To reduce kernel execution overhead each WI computes the force and torque and sums these values for two pixels. Rather than iterating the kernel to continue the reduction, the kernel then 'strides' each WG's starting index to a point in the pixel array where further pixels are yet to be processed. This striding continues until the strided index exceeds the number of pixels to be processed. The stride size is computed as $2 \times WGs \times WIs$ and this striding is demonstrated in

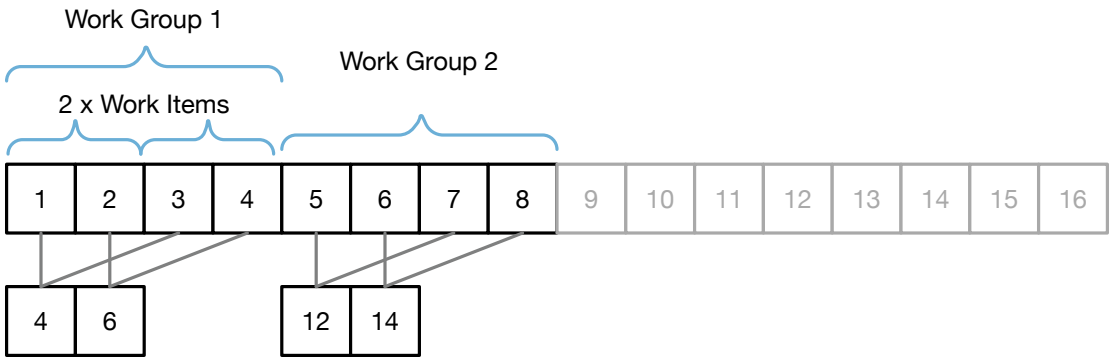


Figure 3.9: Notional parallel reduction by summing sequenced addressing.

Figure. 3.10. In this notional striding example once WG1's two WIs have finished processing their four values, the addressing index strides over all other WGs and continues to sum a new batch of four values. The `While()` loop is shown in an abbreviated version in Listing 3.3.

Listing 3.3: Abbreviated excerpt of the while loop in the OpenCL parallel reduction kernel.

```
1       unsigned int i = group_id * group_stride + local_id;
2       while(i < texture_size)
3       {
4               // compute x, y coords for lookup in square image map (texture)
5               int y = i / tex_width;
6               int x = i % tex_width;
7               float4 nHat_B = read_imagef(normalsMap, coords);
8               if (nHat_B[3] > 0) {
9                       // Perform positon and material read_imagef and
10                      // compute force and torque for pixel
11              }
12              // If the mesh size is smaller than the group_size then we have to stop
13              // trying to compute the second facet in the parallel reduction because there
14              // will be no more facets in the mesh.
15              unsigned int secondPixelIdx = i + group_size;
16              if (secondPixelIdx < textureSize) {
17                      float4 nHat_B_1 = read_imagef(normalsMap, coords_1);
18                      if (nHat_B_1[3] > 0) {
19                              // Perform positon and material read_imagef and
20                              // compute force and torque for pixel
21                      }
22              }
23              i += local_stride;
24      }
```

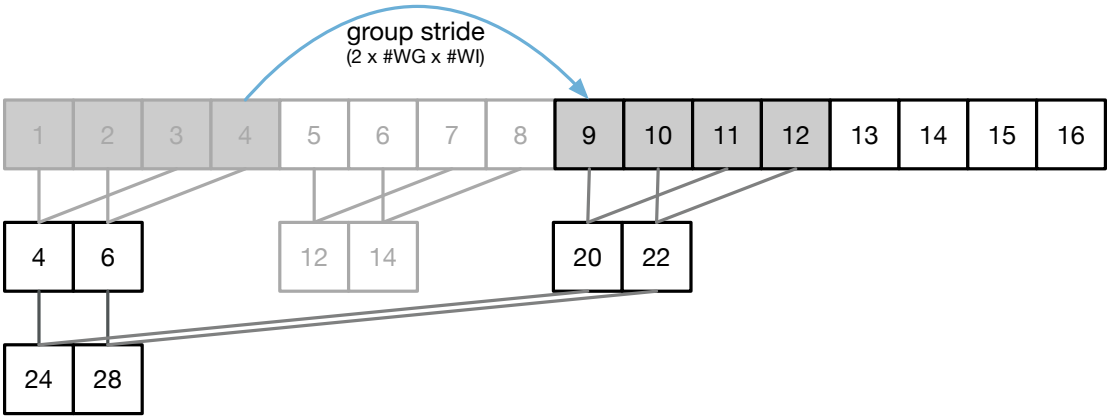


Figure 3.10: Notional parallel reduction striding over all allocated addresses to continue summing sequential blocks of memory in a the `While()` loop.

Additionally, time overhead is incurred by loop instructions. To avoid unnecessary instruction overhead, at the completion of the `While()` loop, the final parallel summations within a single Work Group are computed with unrolled loops. The purpose of the loop unroll optimization is to expose concurrency to the OpenCL compiler. Loop unrolling allows the OpenCL compiler to take advantage of the SIMD architecture by optimizing memory loads and scheduling of instructions given the fixed width of the unrolled loop iterations [5]. An abbreviated portion of the unrolled loops is shown in Listing. 3.4. The number of unrolled loops is controlled at kernel compile time by the the kernel macro `GROUP_SIZE`. At kernel compile time (application runtime) the `MAX_WORK_GROUP_SIZE` parameter is queried from the compute platform on which the code is executing. The queried value is used to set `GROUP_SIZE` and thus only expose the number of unrolled loop iterations which match the hardware's maximum Work Group size.

At the end of the first kernel execution the number of computed force and torque vectors is equal to the number of WGs $\times$ WIs. A second simple parallel reduction kernel is launched for all marshaled WIs to return the final resultant force and torque vectors.

3.6 Model Validation

In order to validate the approach two comparisons are performed. The first validation is to demonstrate that the force and torque on a spherical spacecraft model matches that computed by

Listing 3.4: Abbreviated excerpt of the unrolled loops in the OpenCL parallel reduction kernel.

```
        #if (GROUP_SIZE >= 512)
    if (local_id < 256) {
        ACCUM_LOCAL_F4(shared_force, local_id, local_id + 256);
        ACCUM_LOCAL_F4(shared_torque, local_id, local_id + 256);
    }
        #endif
        .
        // unrolled loops for GROUP_SIZEs 256, 128, 64, 32, 16, 8 and 4 omitted here.
        .
        #if (GROUP_SIZE >= 2)
    if (local_id <   1) {
        ACCUM_LOCAL_F4(shared_force, local_id, local_id +   1);
        ACCUM_LOCAL_F4(shared_torque, local_id, local_id +   1);
    }
        #endif

    if (get_local_id(0) == 0)
    {
        float4 v_force = LOAD_LOCAL_F4(shared_force, 0);
        float4 v_torque = LOAD_LOCAL_F4(shared_torque, 0);
        STORE_GLOBAL_F4(output_force, group_id, v_force);
        STORE_GLOBAL_F4(output_torque, group_id, v_torque);
    }
```

the analytic cannonball model. The spacecraft model used in the computation is a sphere of radius 1 meter made up of 5120 triangular facets. Evaluated at a distance of 1 AU the analytic cannonball model computes a force of $\boldsymbol{F}_{\odot}$ = [-1.42559×10^{-5}, 0.0, 0.0]. The percent error for increasing pixel resolutions of the mesh model force evaluation with respect to the analytic cannonball evaluation is shown in Figure 3.11. It can be seen that the error decreases rapidly as the resolution increases to 64×64 pixels. For resolutions of beyond 190×190 pixels the error remains less than 0.01% for the $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{z}}$ force components. The $\hat{\boldsymbol{x}}$ component maintains an offset in error of approximately 0.1% for all pixel resolutions. This is due to the fact that the the projected area of the mesh model is not precisely that of a circle, whereas for the analytic cannonball model the projected areas is exactly $A = \pi r^2$. This offset demonstrates the importance of a sufficiently accurate mesh model spacecraft representation. Increasing the sphere mesh to 20480 faces results in a lower offset of 0.034% and

for further for a 81920 facet sphere of 0.0076%. For the symmetric cannonball model the torque is expected as zero. Given the mesh models are not exact spheres, there is a small torque computed. This torque is of the order of 10^{-10} for the 5120 sphere, decreasing to 10^{-12} for the 20480 sphere, and 10^{-14} for the 81920 sphere.

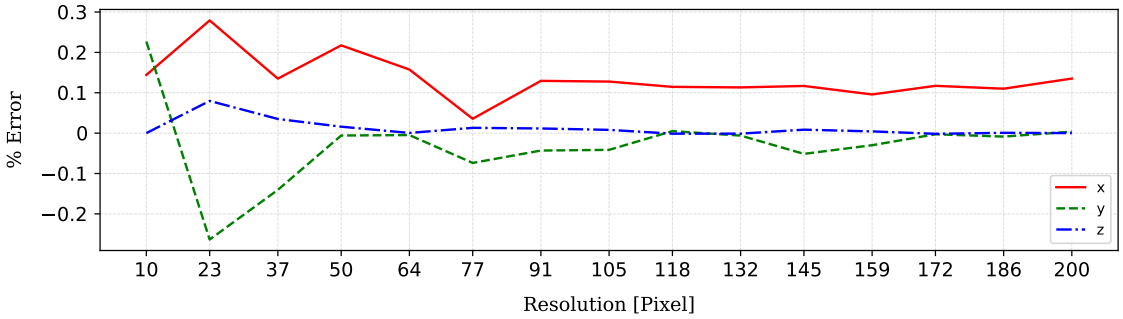


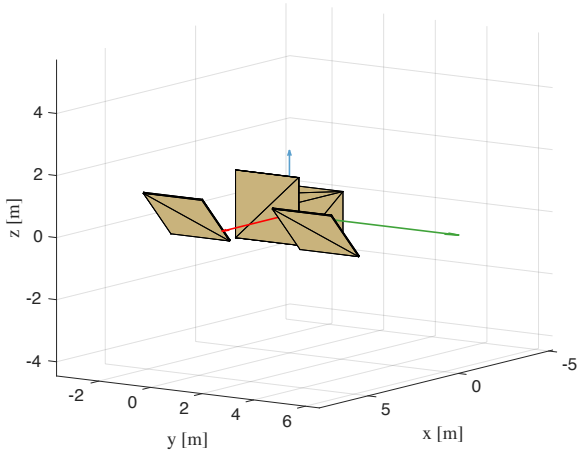
Figure 3.11: Error with respect to analytic cannonball evaluation for 1 m sphere mesh.

A second validation is carried out with a cube mesh with material coefficients of reflection for diffuse and specular properties of $\rho_d = 0.6$ and $\rho_s = 0.2$ respectively. The sun heading in the body frame is $\hat{\boldsymbol{s}} = [0.7071, 0.7071, 0]$. The resulting percentage force error with respect to a faceted evaluation of the same model is shown in Figure. 3.12. The error decreases with increased resolution where the error remains below 0.1% for resolutions above 1400×1400 pixels.

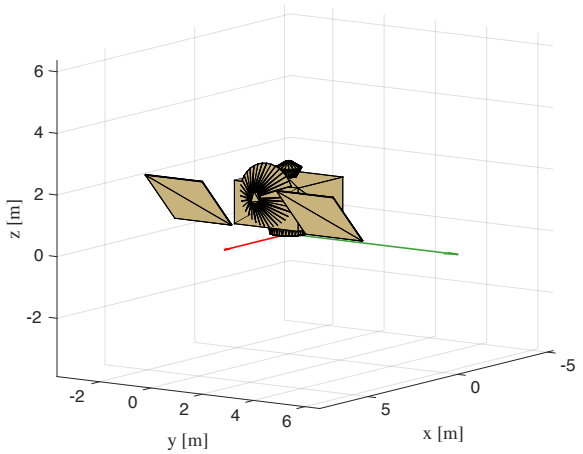
Figure 3.12: Mixed force validation.

3.6.1 Impact Of Mesh Detail On Accuracy

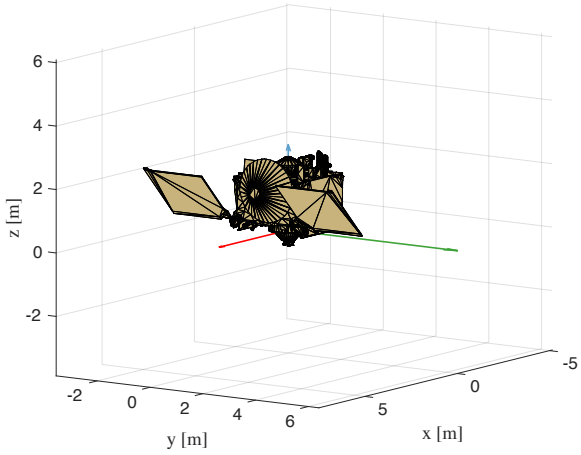
To demonstrate the method's ability to capture increased force resolution three model variants of the OSIRIS-REx spacecraft are evaluated over an evenly spaced sampling of spacecraft sun



(a) Box and wings.



(b) High gain antenna, thruster ring and sample return module.



(c) Hifidelity model.

Figure 3.13: OSIRIS REx spacecraft mesh models.

headings ${}^{\mathcal{B}}\hat{\boldsymbol{s}}$ over the 4π steradian sphere of possibilities. The three spacecraft models are shown in Figure 3.13. Each model represents an increase in the detail of the modeled spacecraft. The first model in Figure 3.13(a) is a simple box and wing model with a single large face oriented in the $\hat{\boldsymbol{x}}$ body frame direction in order to approximate the sun-pointing projected area of the spacecraft. The second spacecraft model in Figure 3.13(b) incorporates larger spacecraft components including the high gain antenna (HGA), thruster ring and sample return module. The final model in Figure 3.13(c) is the high fidelity spacecraft model exported from a CAD software package.

Evaluations of the high-fidelity model are treated as the baseline model evaluation. Force and

torque values, for all three models, are computed for all sampled sun headings. Material optical properties remain the same for each model to allow the comparison to better exemplify the effect on force resolution of increased mesh modeling fidelity. The materials are Germanium Kapton MLI (ρ_d=0.102 and ρ_s=0.408) and general solar panels (ρ_d=0.022 and ρ_s=0.088).

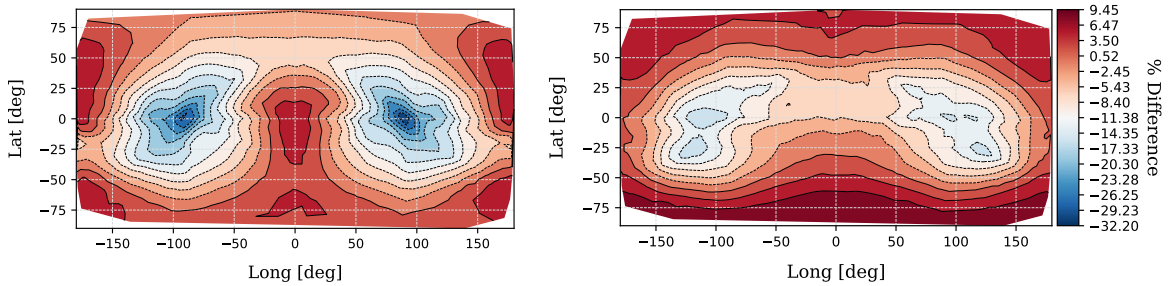
The percentage difference of each mesh evaluation relative to the high-fidelity mesh model evaluation is computed. To convey an intuitive sense of change in force and torque between evaluations of the different mesh models, the magnitude of the percentage difference is computed with respect to a baseline value as computed in Eq. (4.47). The baseline value is computed as the average of the magnitude of either the force or the torque computed over all sun headings of the high-fidelity OSIRIS-REx mesh. The percentage difference is thus computed as given in Eq. (4.48). This approach is used for plotting both the force and torque differences.

$$F_{\text{base}} = \frac{1}{N}\sum_{n=1}^{N} |\boldsymbol{F}_{\text{n}}| \tag{3.16}$$

$$\Delta F = \frac{F_{\text{model}} - F_{\text{hifi}}}{F_{\text{base}}} \times 100 \tag{3.17}$$

The force percentage differences of both low fidelity models relative to the high-fidelity model are shown in Figure 3.14. It is evident that the box and wing model over predicts the resultant force for sun headings in the $+\hat{\boldsymbol{x}}$ and $-\hat{\boldsymbol{x}}$ direction while significantly under predicting the force for headings in the $+\hat{\boldsymbol{y}}$ and $-\hat{\boldsymbol{y}}$. The torque percentage difference of both low fidelity models relative to the high-fidelity model are shown in Figure 3.15. It is clear that the absence of the HGA from the box and wing model results in an under prediction of torque for a large region of sun heading mid latitude and longitudes.

The force and torque percentage difference for the box and wing model relative to the high-fidelity model, in each of the body frame components, are shown in Figure 3.16 and Figure 3.17, respectively. The approximation of the box and wing model is most evident in the $\hat{\boldsymbol{x}}$ force component of Figure 3.16(a). In the region spanned by latitude range -40 deg to $+40$ deg and longitude -50 deg to $+50$ deg the box and wing model over predicts the force due to the absence of the high gain antenna. Additionally, at sun headings of longitude -90 deg and $+90$ deg the absence of the thruster



(a) Force box and wing model relative high-fidelity model

(b) Force HGA model relative high-fidelity model

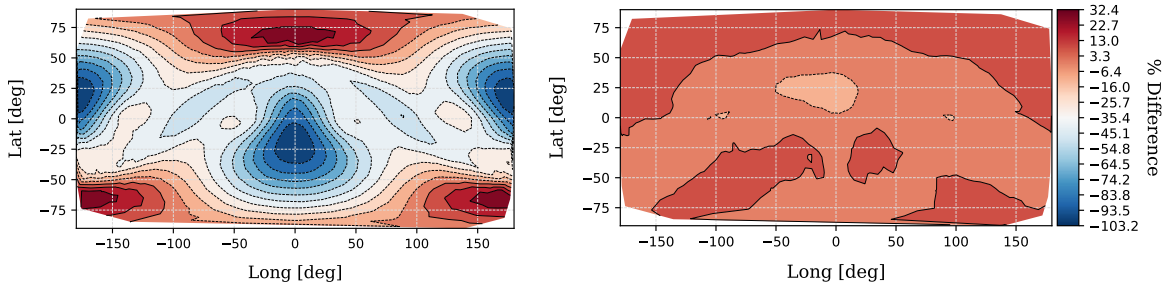
Figure 3.14: Force percentage difference between low fidelity models relative to the high-fidelity model with baseline value 5.73361×10^{-5} [N].

ring, sample return module and depth due to the HGA, result in over and under predictions of greater than 20%.

The force and torque percentage difference for the HGA model relative to the high-fidelity model, in each of the body frame components, are shown in Figure 3.18 and Figure 3.19, respectively. By comparison to the percentage error of the box and wing model shown Figure 3.16, the HGA model shows less error across mid-latitudes. The presence of the HGA, thruster ring and sample return module results in a clear improvement of torque resolution with a maximum over and under prediction of approximately 15%. However, the model does show a slight increase in difference when the sun heading predominates in the $+\hat{\boldsymbol{z}}$ and $-\hat{\boldsymbol{z}}$ body frame component.

3.6.2 Model Articulation and Detailed Material Properties

To demonstrate the articulation capability of this modeling method the Aqua spacecraft, shown in Figure 3.4(b), is simulated in Basilisk. The OpenGL-CL method is implemented as a Basilisk Dynamic Effector module which allows it to be integrated into the general propagation of a spacecraft rigid body hub. The simulation orbit is a 1000 km altitude polar orbit and the Keplerian orbital elements are listed in Table 4.3. The spacecraft's solar panel is controlled to articulate in a manner which causes the panel normal vector to track the inertial heading $[1, 0, 0]$. The spacecraft is assigned three reaction wheel control devices which serve to control the spacecraft



(a) Torque box and wing model relative high-fidelity model

(b) Torque box and wing model relative high-fidelity model

Figure 3.15: Torque percentage difference between low-fidelity models relative to the high-fidelity model with baseline value 6.57817×10^{-5} [Nm].

attitude to the computed reference attitude. The spacecraft maintains a nadir pointing attitude for it's instrument deck and the computed reference attitude is given by the orbital Hill frame. For reference, Appendix A contains a detailed technical description of the Basilisk framework and the development of a Basilisk Dynamic Effector module. The spacecraft is assigned material

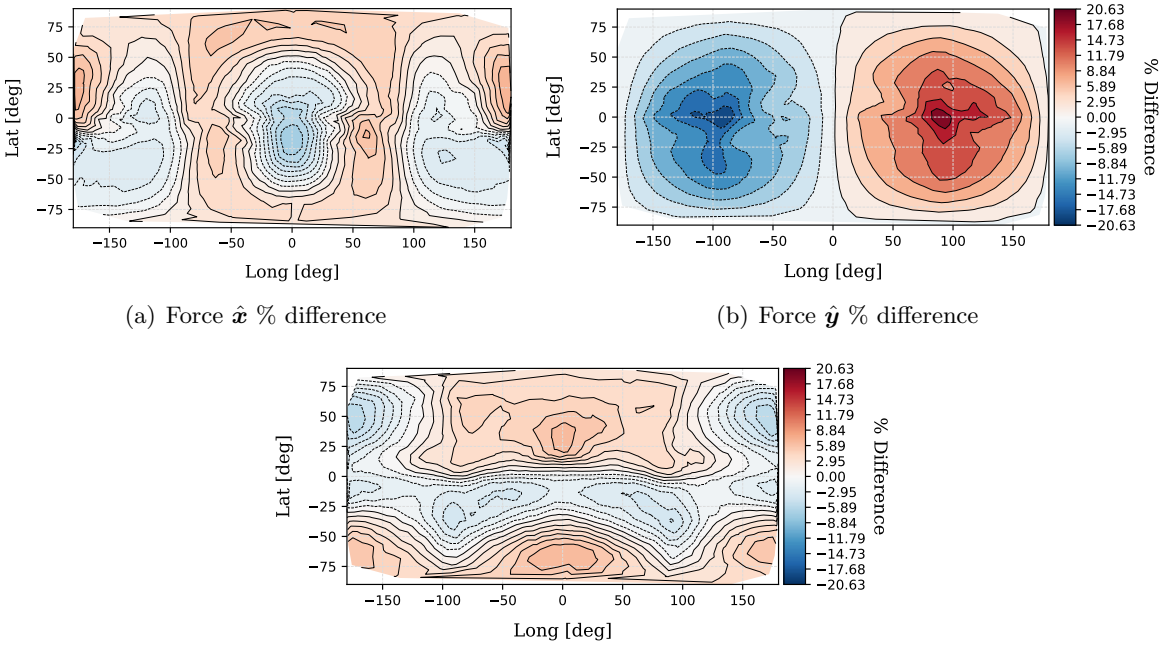
Table 3.1: Spacecraft orbit parameters for sun-synchronous LEO orbit and GEO orbit.

a km	7378
e	0
i, deg	90
M_0, deg	90
Ω, deg	0
ω, deg	0

parameters for each sub-mesh defined and shown previously in Figure 3.6. The material optical properties are given in Table 3.2. The material parameters are chosen loosely to provide variation amongst materials rather than to serve as an exact reference for the optical properties of each material.

Table 3.2: Spacecraft sub-mesh material optical parameters.

Material	Specular (ρ_s)	Diffuse (ρ_d)
Gold MLI	0.184	0.736
Silver MLI	0.66	0.16
Germanium MLI	0.3	0.3
Solar array rear	0.1	0.3
Solar array front	0.023	0.092
Solar array boom	0.3	0.3

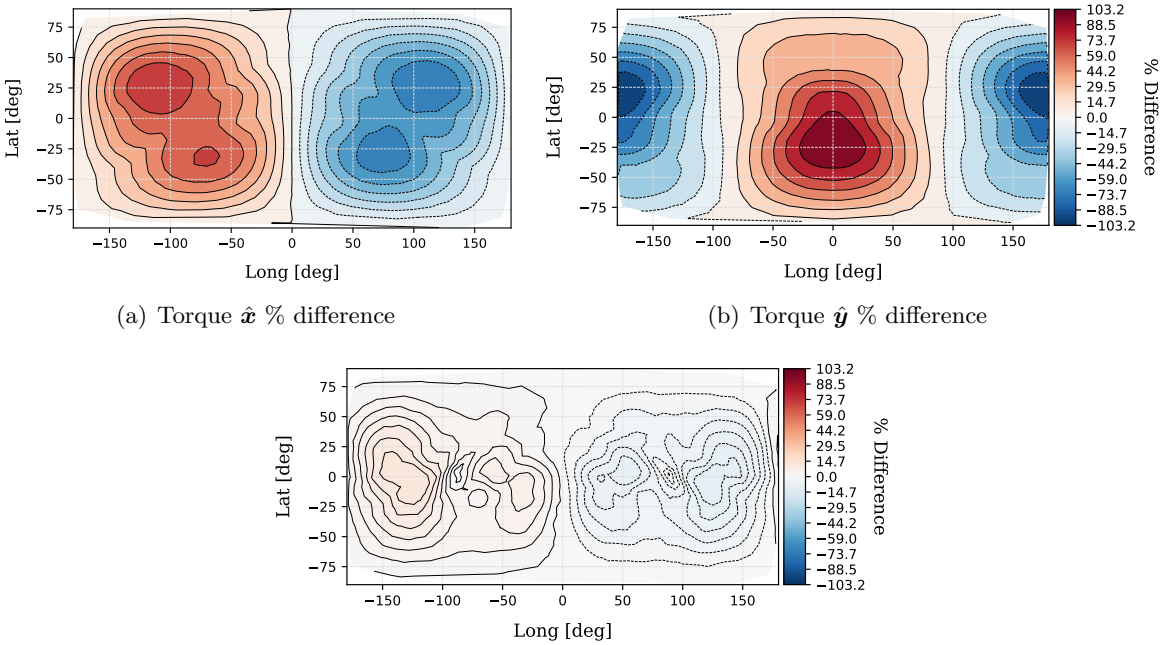


(a) Force $\hat{\boldsymbol{x}}$ % difference

(b) Force $\hat{\boldsymbol{y}}$ % difference

(c) Force $\hat{\boldsymbol{z}}$ % difference

Figure 3.16: Force percentage difference between box and wing model relative to the hifidelity model with baseline value 5.73361×10^{-5} [N].

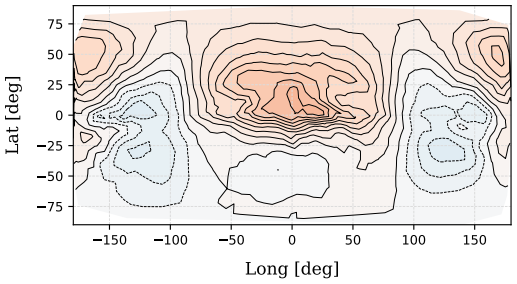


(a) Torque $\hat{\boldsymbol{x}}$ % difference

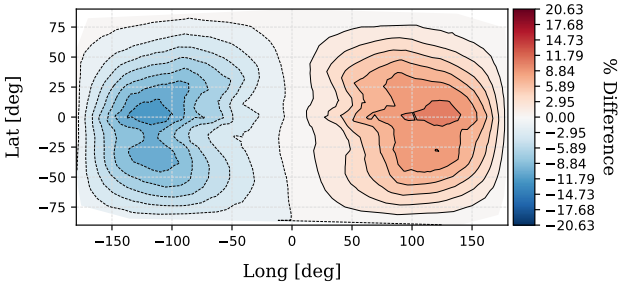
(b) Torque $\hat{\boldsymbol{y}}$ % difference

(c) Torque $\hat{\boldsymbol{z}}$ % difference

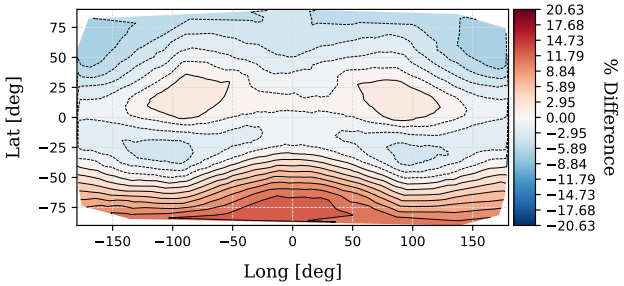
Figure 3.17: Torque percentage difference between box and wing model relative to the hifidelity model with baseline value 6.57817×10^{-5} [Nm].



(a) Force $\hat{\boldsymbol{x}}$ % difference

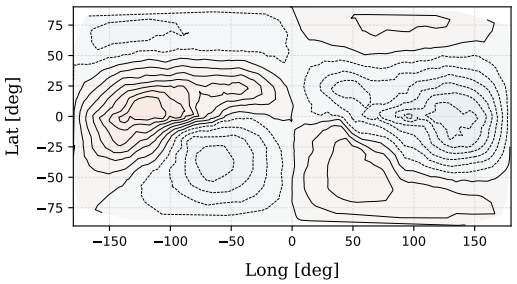


(b) Force $\hat{\boldsymbol{y}}$ % difference

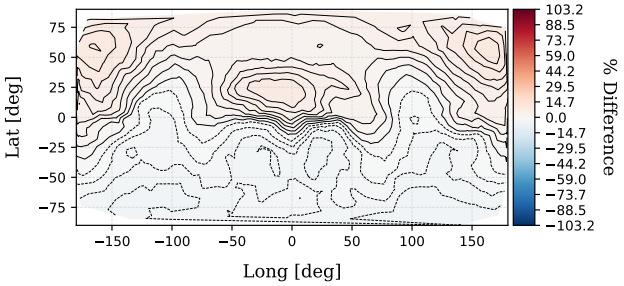


(c) Force $\hat{\boldsymbol{z}}$ % difference

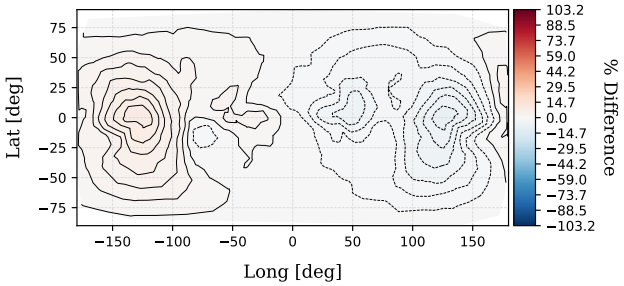
Figure 3.18: Force percentage difference between HGA model relative to the high-fidelity model with baseline value 5.73361×10^{-5} [N].



(a) Torque $\hat{\boldsymbol{x}}$ % difference



(b) Torque $\hat{\boldsymbol{y}}$ % difference



(c) Torque $\hat{\boldsymbol{z}}$ % difference

Figure 3.19: Torque percentage difference between HGA model relative to the high-fidelity model with baseline value 6.57817×10^{-5} [Nm].

The body frame force over two orbits is shown in Figure 4.26. The body torque over two orbits is shown in Figure. 4.27. The eclipse period is visible in both plots where the force and torque return zero.

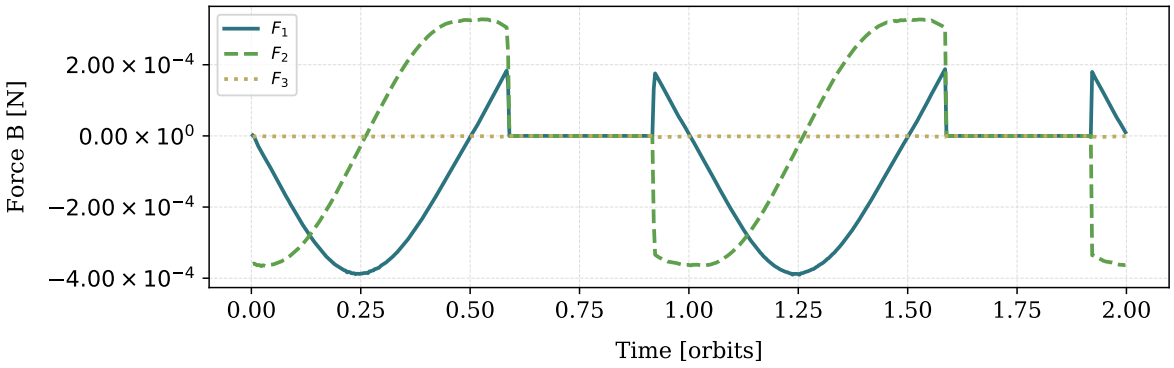


Figure 3.20: Body frame force components over two orbits.

The rendered output at three time steps is shown in Figure 3.22. The viewing orientation is looking in the $-^{\mathcal{B}}\hat{\boldsymbol{s}}$ direction in the sun frame bounding box. It is evident that the spacecraft's solar panel normal vector is directed along the sun heading in each frame while the spacecraft bus rotates to control to the attitude reference orbit Hill frame.

3.7 Computational Performance

To demonstrate the computational performance of this method a series of evaluations of the Aqua spacecraft model are carried out for increasing resolutions. Three different GPUs are used to exemplify three particular qualities. The first implementation consideration is the use of the OpenGL-OpenCL shared memory context. This feature, used to transparently share content data between the two APIs, offers signifiant performance benifits on GPU hardware which share a common direct random access memory (DRAM) space with the CPU[37]. The Intel HD Graphics 630 is employed to demonstrate the performance of an integrated GPU. The Advanced Micro Devices (AMD) Radeon Pro 560 is chosen to demonstrate the performance of a commodity low-to-mid range performance discrete GPU. The NVIDIA GTX 1070 is chosen to represent the high performance discrete GPU.

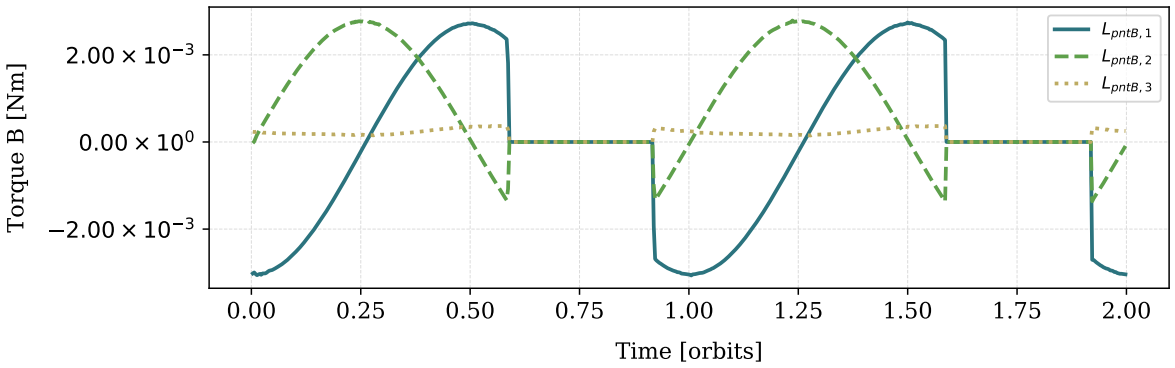
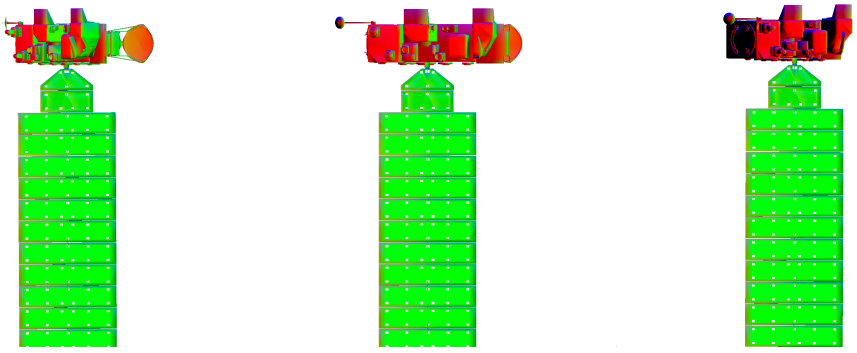


Figure 3.21: Body frame torque components over two orbits.



(a) Scene 1 (b) Scene 2 (c) Scene 3

Figure 3.22: Sequential rendered spacecraft in sun frame.

The computation time for resolutions from 10×10 pixels to 2048×2048 pixels are shown in Figure. 3.23. While the Intel HD Graphics 630 integrated GPU possesses less compute capability than the other GPUs, it is able to outperform the other GPUs up to resolutions of 1029×1029 pixels. The performance of the Intel integrated GPU is due to the DRAM shared with the CPU. This shared memory space facilitates zero-copy memory objects and sharing of pointers to data objects. While the discrete GPUs incur a data transfer latency, the integrated GPU does not need to copy and move data across device data buses. Only when the computational load passes a particular volume is the data transfer latency masked by computational latency. For resolutions 1029×1029 pixels and above the increased computational capability of the Radeon Pro 560 and NVIDIA GTX 1070 becomes clear. The greater computational capability of the NVIDIA GTX 1070 is evident as there is no appreciable increase in computation time for all resolutions tested.

Whereas for the Radeon Pro 560 the computation time steadily increases with increased resolution.

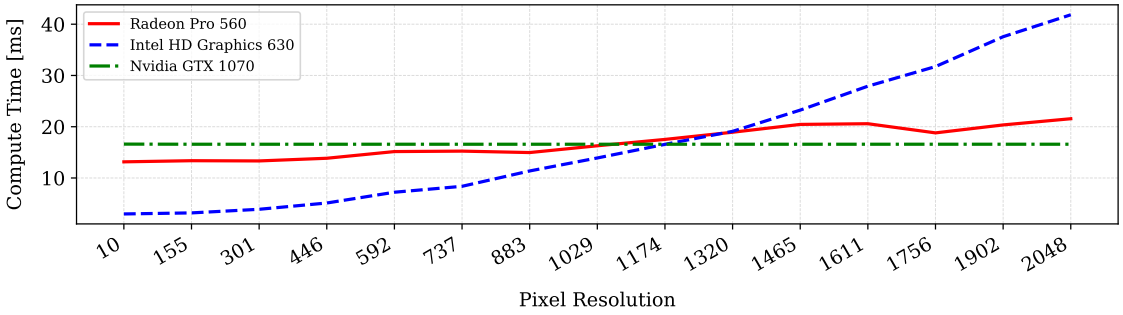


Figure 3.23: OpenGL-CL computation times for three different GPUs.

3.8 Conclusions

This chapter provides a detailed description of the OpenGL-CL SRP modeling approach. The approach provides a solution to resolving, in an online simulation context, arbitrary time varying articulated spacecraft shape models, spacecraft self shadowing, and varied arbitrary material optical properties. Arbitrary spacecraft mesh complexity and articulation are accommodated. The method quickly captures the difference between spacecraft mesh models and comfortably accommodates detailed meshes of many thousands of vertices.

Using the OpenGL-OpenCL shared context allows this modeling approach to easily be applied to other force and mesh modeling. For example, at orbit altitudes where the atmosphere is well modeled as free molecular flow, the same methodology as developed for SRP can be employed to model drag with only a few lines of change to the entire code base.

To achieve high computational performance, efficient algorithms are developed for both the CPU and GPU bound processes. For the CPU bound processes, algorithms for efficient bounding box computation and recursive mesh transformation are presented. Further, a general parallel reduction algorithm is described in which the computation of SRP is tightly integrated to seek best utilization of GPU computing resources through the OpenCL API. These enhancements provide a method for computing high geometric fidelity SRP in a computationally performant and config-

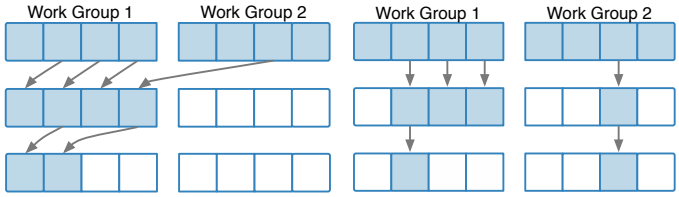
urable manner. The OpenGL-CL approach provides significant modeling capability and enables previously computationally prohibitive analysis on modest personal computing hardware.

Chapter 4

OpenCL Ray Tracing

The Faceted SRP Method using OpenGL/OpenCL is able to resolve the force and torque due to the primary irradiation of the spacecraft mesh and account for spacecraft self-shadowing. However, the method does not capture important self-reflections. Further, the resolving of the force for the first bounce for a wide range of more complex physically correct material bidirectional reflectance distribution functions (BRDF) representations is difficult without sampling rays or computing reflectance integrals. A proven method for accounting for the definicienies in capturing secondary reflections is to employ a ray tracing methodology [96]. Consequently, the ray-tracing approach presented is able to capture self-reflection behavior, model complex BRDFs and account for arbitrary spacecraft articulations and attitudes.

Typically, ray-tracing algorithms operate as a CPU based serial execution process. Additionally, serial CPU implementations are rarely capable of reaching computational throughput required for real-time or faster image generation rates. Achieving faster than real-time render speeds is enabled by a number of hardware and algorithmic developments. Early parallel ray-tracing approaches leverage dedicated parallel computers to achieve a parallel implementation [77, 70]. Algorithmic techniques which have introduced improved performance include functional decomposition techniques such as bounding volume hierarchy search structures [77], object space subdivision to compute the result of each region on a separate processor [16], and screen subdivision to faciliate dynamic loading of scene object meshes to be rendered[70]. Each of these three areas of development coupled with the advent of GPGPU computing enables the faster than real-time parallel



(a) SIMD by ray (b) SIMD by pixel

Figure 4.1: Increased GPU Work Group occupancy when tracing by ray rather than by pixel.

ray-tracing presented.

This chapter is presented in two primary sections. The first section proceeds by outlining the key algorithmic steps of GPGPU parallel ray-tracing and the algorithmic design decisions made for the presented implementation. Next, the importance sampling methodology employed to evaluate the ideal specular-lambertian and complex microfacet BRDFs is detailed. The second half of this chapter presents validation and verification of the modeling approach at first with simple cannonball and cube meshes and then later on the more complex OSIRIS-REx and Aqua spacecraft meshes.

4.1 GPGPU Parallel Algorithm Considerations

Typically, ray tracing has been executed as a serial algorithm where a single or set of ray reflections are computed using a recursive algorithm which does so at slower than real-time execution speeds. The parallel GPU computing environment provides the computing power and parallelism required to produce a faster than real-time ray tracing implementation. However, the GPU environment requires two key changes to the serial ray-tracing algorithm. The first change is required because recursive function execution is not available in GPU execution environments. As a result the recursive computation of ray reflections must be replaced by iterative function execution. Figure. 4.1 shows a notional arrangement of two fully marshaled OpenCL GPU work groups iteratively ray tracing. The second change is that rather than making the algorithm parallel by pixel as the serial implementation may suggest, the algorithm should be parallel by ray. This second change is necessitated by the SIMD architecture. The SIMD architecture is most efficient when code ensures that each compute unit on the GPU is actively working. In the case that the algorithm is parallel

by pixels, as the scene is traced the rays from certain pixels will terminate sooner than others. This leaves compute units inactive resulting in poor utilization of the computing resources GPU. Rather, as shown in Figure 4.1, by producing an algorithm which is parallel by rays cast, after each iteration terminated rays may be discarded and the reflected rays repacked for a second iteration to ensure all compute units marshaled are active.

4.2 Algorithm Steps Overview

An overview of the ray-tracing methodology is shown in Figure 4.2. As described in Section. 3.2 an articulated spacecraft mesh is taken as input to the algorithm. The ray tracing algorithm employs three key sub-algorithms to minimize the otherwise high computational load of naive ray-tracing. First, the ray-triangle intersection search space is reduced by generating an acceleration data structure called a bounding volume hierarchy (BVH). This data structure reduces the search space by spatially grouping triangular facets and allowing the intersection testing to initially test facet groups rather than each individual facet in the model. Additionally, two algorithms particularly suited to GPU implementation due to the minimal code branching and few memory accesses are implemented. The first algorithm is the bounding box intersection testing algorithm using clipping planes originally presented by Kay and Kajiya [49]. The second is the memory efficient and computational fast Möller-Trumbore ray and triangle-ray intersection testing algorithm[64]. At each time update, a new wave of ray vectors is generated. The spacecraft-to-sun unit direction vector $\hat{\boldsymbol{s}}_B$ is computed and used as the first axis in an orthogonal Sun S frame. The direction cosine matrix $[SB]$ which defines the rotation from the body frame B to the S frame is constructed and used to generate an S frame axis-aligned bounding box of the mesh model [76]. As shown in Figure 4.3(a), the side of the bounding box nearest the sun is used as a finite plane from which the origins of all ray vectors is defined. The wave of rays are computed in parallel using a dedicated OpenCL ray generation kernel.

The ray plane is divided into unit areas determined by the resolution unit chosen by the user. For example, a 2 m x 1 m plane can be divided into areas of 1 mm x 1 mm giving a plane

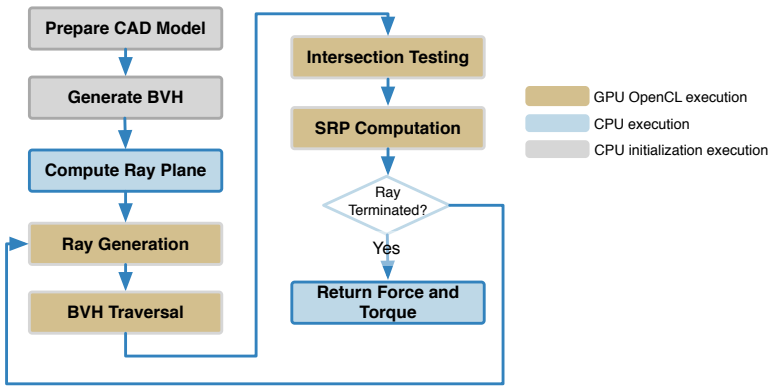
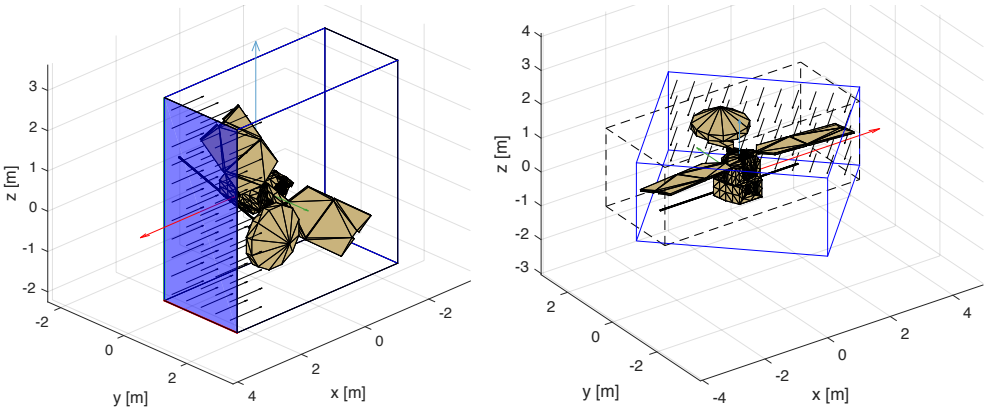


Figure 4.2: Illustration of a set of five bounding boxes and a test ray. Intersections are recorded for the boxes with the dash outlines.

of 2000 x 1000 squares, producing 2 000 000 rays, where each ray has an area of 1 mm^2. Figures 4.3(a) and 4.3(b) provide an example of the Mars Reconnaissance Orbiter mesh model surrounded by an S frame bounding box (blue solid) and a B frame oriented bounding box (dashed black), respectively. The red, green and blue vectors are the respective frame axes and the black vectors indicate rays originating from the blue ray-plane. The origin for a ray is taken as the corresponding center of a unit area and the direction for all rays is taken as $-\hat{\boldsymbol{s}}_B$. The ray intersection testing must occur in the same coordinate frame in which the spacecraft vertices are defined. As a result, the ray vectors are mapped from sun-frame S to the body-frame B using the $[BS]$ rotation matrix. The discretization of the incident radiation wave front into individual rays has the potential to introduce errors into the computation. Ziebart shows in a study of this discretization error that for representative test geometries, the error, for a maximum ray cross section of 10 mm^2, is 2% and decreases to less than 1% for ray cross sections of less than 5 mm [96].

Each compute unit on the GPU launches an instance of the OpenCL kernel program. Each kernel instance accesses the ray wave front data in the GPU global memory space copying a specific ray and the BVH traversal array to local memory in the compute unit. For each ray, the BVH is traversed to test for bounding volume intersections. A positive result for a bounding volume intersection yields a ray-surface intersection with a specific mesh facet.

The ray-surface interaction is computed with the facet's particular material BRDF as the



(a) S frame generated rays. (b) S frame rays mapped to the B frame.

Figure 4.3: Ray generation for the MRO mesh model with a B frame oriented bounding box (dashed black) and an S frame bounding box (blue solid).

primary input. The BRDF evaluation produces an outgoing ray direction from which the force and torque contribution of that single ray-surface interaction is computed. The ray termination condition is evaluated and a new ray vector origin and direction of travel generated for a continued ray path. The parallel ray-tracing algorithm iterates through BVH traversal, intersection testing and SRP computation until all rays have exited the scene or reached the nominated termination condition. The total force and torque contribution of all ray-surface interactions is summed and returned to the CPU bound process where the values can be integrated into the dynamics propagation of a numerical simulation.

4.3 Radiation Pressure Particle Tracing Formulation

In this section a formalism is introduced which describes the process of generating and evaluating a set of weighted sample rays. This formalism initially introduced by Veech in Refernece[92], rigorously describes the ray tracing algorithm employed in this work. Veech presents a general description of how an estimate of some quantity can be computed with respect to some measure, by generating a set of weighted sample rays. Here, the quantity will be the solar radiation pressure force due to radiance incident on the spacecraft. Two key assumptions will simplify the resulting description. The first assumption is that ray samples are taken uniformly from a plane wave,

discretized into smaller areas, of collimated radiation. A unit of the discretized area is referred to as a 'pixel' and the pixel's ray resolution represents the length of one side of the pixel area. The second assumption is that the mechanism controlling ray continuation is the maximum number of ray-surface interactions rather than a probabilistic measure such as the Russian Roulette approach[92].

This path tracing algorithm produces an unbiased estimate of the integral of force over the spacecraft mesh surface. In the case of radiation pressure the sample weight α_i is the radiation throughput of each sample ray. The unbiased estimate is constructed by generating a set of weighted sample rays

$$(\alpha_i, \boldsymbol{r}_i), \tag{4.1}$$

where $\boldsymbol{r}_i$ is a ray and α_i the corresponding weight/throughput. The goal is to produce samples that give an unbiased representation of the equilibrium force F over the spacecraft that holds for any importance function W_e. This estimator is given in Eq. 4.2 where N is the number of rays. Again the first assumption dictates that all samples are given equal importance, therefore W_e is trivially set as 1.

$$E\left[\frac{1}{N}\sum_{n=i}^{N}\alpha_i W_e(\boldsymbol{r}_i)\right] = \langle W_e, F\rangle \tag{4.2}$$

To begin the particle tracing process, an initial ray is defined as $\boldsymbol{r}_0 = (\mathbf{x}_0, \omega_0)$ where $\mathbf{x}_0$ is an origin vertex and ω_0 a direction. Each sample ray has a corresponding weight α and each ray has an initial state of $(\alpha_0, \boldsymbol{r}_0)$. For radiation transport the initial weighting is the incident radiance throughput

$$\alpha_0 = \frac{L(\boldsymbol{r}_0)}{p_0(\boldsymbol{r}_0)} \tag{4.3}$$

where $p_0(\boldsymbol{r}_0)$ is the probability density from which $\boldsymbol{r}_0$ is sampled. Here, the first assumption, of uniformly distributed rays, simplifies the throughput to simply

$$\alpha_0 = L(\boldsymbol{r}_0) \tag{4.4}$$

Given the current state of a ray $(\alpha_i, \boldsymbol{r}_i)$, if $i < k$, with k being the maximum number of ray-surface interactions, the ray is continued. If an intersection occurs, $\mathbf{x}_{i+1}$ is the intersection point of

the ray $\boldsymbol{r}_i = (\mathbf{x}_i, \omega_i)$. A random scattering direction ω_{i+1} is chosen from the BRDF, $f_r(\mathbf{x}_{i+1}, \omega_{i+1} \rightarrow -\omega_i)$, according to a probability density function approximating the BRDF $p_{i+1}(w_{i+1})$. Employing importance sampling, the next ray throughput (weight) is computed as

$$\alpha_{i+1} = \alpha_i \frac{f_r(\mathbf{x}_{i+1}, \omega_{i+1} \rightarrow -\omega_i)|\hat{\boldsymbol{\omega}}_{i+1} \cdot \hat{\boldsymbol{n}}(\mathbf{x}_{i+1})|}{p_{i+1}(\omega_{i+1})} \tag{4.5}$$

From the recursive relationship in Eq.(4.5), the ray continuation step is repeated until the maximum number of ray-surface interactions is reached and results in a set of sample rays with weights given as

$$\alpha_i = L(\mathbf{x}_0, \omega_0) \prod_{j=0}^{i-1} \frac{f_r(\mathbf{x}_{j+1}, \omega_{j+1} \rightarrow -\omega_j)|\hat{\boldsymbol{\omega}}_{j+1} \cdot \hat{\boldsymbol{n}}(\mathbf{x}_{j+1})|}{p_{j+1}(\omega_{j+1})} \tag{4.6}$$

The process computes a set of sample rays $\boldsymbol{r}_0, ...\boldsymbol{r}_k$, each with with weight α_i.

4.4 Force and Torque Evaluation

Where an intersection is found, the force on the spacecraft due to the incident throughput of the ray is evaluated in the spacecraft body frame as

$$\boldsymbol{F}_i = \alpha_i \boldsymbol{\omega}_i + \alpha_i \frac{f_r(\mathbf{x}_{i+1}, \omega_{i+1} \rightarrow -\omega_i)|\boldsymbol{\omega}_{i+1} \cdot \hat{\boldsymbol{n}}(\mathbf{x}_{i+1})|}{p_{i+1}(\omega_{i+1})} \tag{4.7}$$

The flux of all ray-surface interactions of a ray originating from a pixel is given as

$$\boldsymbol{F}_k = \sum_{i=1}^{K} A\boldsymbol{F}_i \tag{4.8}$$

where K is the maximum number of permitted ray-surface interactions and A is the cross sectional area of the pixel.

The total force is computed by summing the flux components for each pixel and finally multiplying the flux by the solar irradiance and scaling by the spacecraft distance to the sun as shown in Eq. (4.9).

$$\boldsymbol{F} = \frac{\Phi \mathrm{AU}^2}{cr^2} \sum_{k=1}^{N} \boldsymbol{F}_k \tag{4.9}$$

Here N denotes the total number of pixels, Φ is the radiation flux (solar radiation flux at 1 AU for SRP), AU is one astronomical unit, c the speed of light and r the sun-spacecraft distance. Following

the force computation, the torque $\boldsymbol{L}_k$ contribution of a single intersection is given as

$$\boldsymbol{L}_i = \mathbf{x}_{i+1} \times \boldsymbol{F}_i \tag{4.10}$$

4.5 Intersection Testing

A Bounding Volume Heirachy data structure is used to reduce the intersection testing search space and therefore reduce computational load. A range of acceleration data structures are presented in ray and path tracing literature, each offering advantages and disadvantages. These advantages and disadavantages are dependent on the characteristics of the scene to be rendered [83]. The characteristic of primary importance to this application is the articulation of any sub-meshes within the model. In the case of a naive BVH implementation where the BVH is a monolithic data structure, were a sub-mesh to undergo some transformation, then the BVH structure would need to be rebuilt entirely. This is a computationally wasteful process. As such, this method employs a BVH structure which is composed of two levels. The bounding volumes of each sub-mesh occupy the lower level while the upper level groups all of the outer most bounding volumes in the lower level. If a sub-mesh is to be transformed then that homogeneous transformation matrix is stored with the sub-mesh's bounding volume and inversely applied to each ray being tested for intersections. While this two-level BVH provides computationally efficient handling of articulated sub-meshes, it prohibits recursive application of sub-mesh transformations. Whereas mesh transforms for the OpenGL-CL method were developed and implemented to operate on each other recursively the two level BVH requires that transformations be defined relative to the spacecraft mesh body frame only.

To build the bounding volume hierarchy a bounding volume is computed for each triangular facet in the spacecraft mesh model. In this implementation the bounding volume is computed as a bounding box aligned to the spacecraft model body frame. To begin, the list of bounding volumes is sorted along the first spacecraft body-frame axis. The sorted list is then divided in half and a new bounding volume is computed around each half of the list. This process is carried out recursively

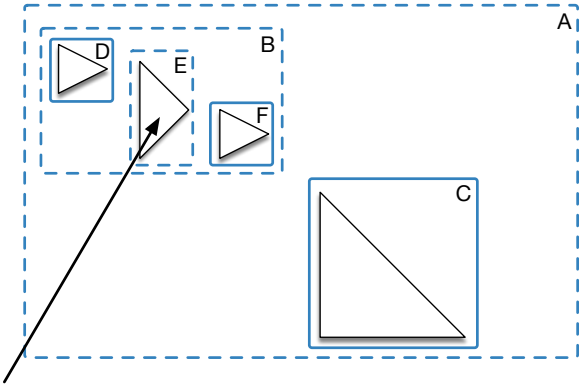


Figure 4.4: Illustration of a set of five bounding boxes and a test ray. Intersections are recorded for the boxes with the dash outlines.

while, at each new split, sequentially selecting the sort axis as the next axis in the body-frame triad. This results in a bounding volume hierarchy that groups successive bounding volumes as containing facets spatially near to each other.

An efficient method of traversing the bounding volume hierarchy is a key aspect in the development of real-time SRP ray-tracing[83]. This implementation uses as the BVH traversal method a depth first search array as described by Smits in Reference [83]. An example BVH hierarchy comprising six nodes is shown in Figure 4.5; first as a recursive depth first search tree and second as a depth first search array with precomputed skip pointers. In the recursive tree structure, if bounding volume node A is intersected, the search recursively descends to test for an intersection against node B. If no intersection is found at node B the recursion meets a termination condition and the search moves back up the tree and proceeds down the next search branch to test node C. For the array traversal structure, if bounding volume A is intersected, the next node to try is the next node in the array which is node B. If the bounding volume at node B is not intersected, the next node is found by following the precomputed skip pointer to the next sibling in the array, which for node B is node C. The array traversal algorithm is shown as pseudo code in Listing 2.

The depth first array search structure avoids the function call overhead inherent in a recursive search tree traversal and takes advantage of the fact that the next node in the search tree can be precomputed and stored with the left most sibling as a skip pointer to the next node. An

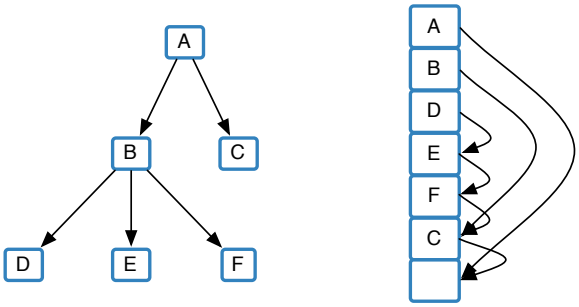


Figure 4.5: Two BVH traversal structures. The left structure demonstrates a simple recursive BVH traversal. The right demonstrates the same BVH as shown on the left yet organized as a depth first search array with precomputed node skip pointers.

additional benefit to the array BVH traversal structure is that the structure results in greater memory coherency for large meshes and therefore more efficient contiguous memory accesses on the GPU given its sequential nature [83].

4.5.1 Bounding Volume Intersection

Bounding volume intersection uses the algorithm originally presented by Kay and Kajiya [49]. The algorithm models the bounding box as 3 sets of parallel planes. The algorithm employs each set of parallel planes as clipping planes. As demonstrated in Figure 4.6, once the ray is clipped by each set of planes, any remaining portion of ray inside the bounding volume indicates an intersection. The algorithm is particularly suited to implementation in the GPU environment because it does not require code branching (the execution of divergent code paths based on conditional code statements). This parallel plane algorithm employs the non-branching `min()` and `max()` functions and results in an intersection test with no code branching or division operations.

4.5.2 Triangle Facet Intersection

To compute a ray to triangle intersection, the Möller-Trumbore algorithm is used. This algorithm is a fast and memory-efficient ray to triangle intersection algorithm making it ideal for use in the memory constrained GPU computation environment. The basis of the algorithm is the knowledge that the point of intersection of a line through a triangle in barycentric coordinates

```
Data: idx is the index of the current node in the BVH depth first sorted traversal array
while idx is in range do
    node = fetch next node at idx;
    if intersectBbox then
        if node is leaf then
            intersectTriangle;
        else
            idx = idx + 1 (move to first child node);
            continue;
        end
    end
    idx = skip pointer at node (follow skip pointer to next node)
end
```

Algorithm 2: Algorithm to traverse the depth first sorted BVH array using skip pointers.

(u, v) must lie within coordinate bounds which are easily testable as boolean values. The bounds defined by the barycentric coordinate system require $u \geq 0$, $v \geq 0$ and $u + v \leq 1$ [64].

To begin, a point, $T(u, v)$, on a triangle described by vertices $\boldsymbol{V}_0, \boldsymbol{V}_1$ and $\boldsymbol{V}_2$ and mapped to barycentric coordinates is described as given in Eq. (4.11).

$$\boldsymbol{T}(u, v) = (1 - u - v)\boldsymbol{V}_0 + u\boldsymbol{V}_1 + v\boldsymbol{V}_2 \tag{4.11}$$

The ray equation is given in Eq. (4.12) where $\boldsymbol{O}$ is the ray origin, t the distance from the ray origin to the intersection point and $\boldsymbol{D}$ the ray direction.

$$\boldsymbol{R}(t) = \boldsymbol{O} + t\boldsymbol{D} \tag{4.12}$$

It is then evident that for a ray to intersect the barycentric description of the triangle, the ray equation must be equal to a point on the triangle, $\boldsymbol{R}(t) = \boldsymbol{T}(u, v)$, and results in the expression at Eq. (4.13).

$$\boldsymbol{O} + t\boldsymbol{D} = (1 - u - v)\boldsymbol{V}_0 + u\boldsymbol{V}_1 + v\boldsymbol{V}_2 \tag{4.13}$$

Rearranging the equation into a matrix form yields Eq. (4.14) where it is evident that the terms $\boldsymbol{V}_1 - \boldsymbol{V}_0$ and $\boldsymbol{V}_2 - \boldsymbol{V}_0$ are the edges of the triangle and are substituted for $\boldsymbol{E}_1 = \boldsymbol{V}_1 - \boldsymbol{V}_0$ and $\boldsymbol{E}_2 = \boldsymbol{V}_2 - \boldsymbol{V}_0$. Additionally, the substitution $\boldsymbol{T} = \boldsymbol{O} - \boldsymbol{V}_0$ can be made and is interpreted as a

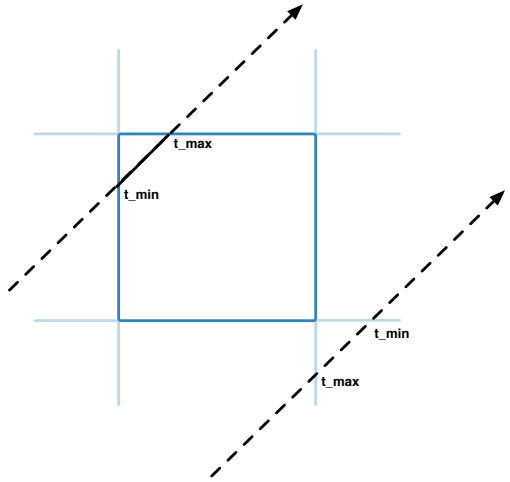


Figure 4.6: Example result of the parallel plane bounding box intersection algorithm. For the top left ray intersection, the algorithm returns t_max as greater than or equal to t_min. For the bottom right ray miss, the algorithm returns t_max as less than t_min.

translation of the ray origin to the barycentric coordinate frame origin.

$$[-\boldsymbol{D}, \boldsymbol{V}_1 - \boldsymbol{V}_0, \boldsymbol{V}_2 - \boldsymbol{V}_0] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \boldsymbol{O} - \boldsymbol{V}_0 \tag{4.14}$$

Using Cramer's rule, a solution can be found for u, v and t as shown in Eq. (4.15). The solution for u, v and t will provide the values with which to test against the barycentric coordinate bounds conditions.

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-\boldsymbol{D}, \boldsymbol{E}_1, \boldsymbol{E}_2|} \begin{bmatrix} |\boldsymbol{T}, \boldsymbol{E}_1, \boldsymbol{E}_2| \\ |-\boldsymbol{D}, \boldsymbol{T}, \boldsymbol{E}_2| \\ |-\boldsymbol{D}, \boldsymbol{E}_1, \boldsymbol{T}| \end{bmatrix} \tag{4.15}$$

A final solution is computed with slightly more efficiency by recognizing that each determinant of the form $|\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}| = -(\boldsymbol{A} \times \boldsymbol{B}) \cdot \boldsymbol{C}$, is shown in Eq. (4.16). This is further simplified by computing

Input : The ray data structure containing an origin, direction and inverse direction vectors and the bounding box extents.
Output: True for intersection, False otherwise

```
1 tx1 ← (box.min.x - r.o.x)*r.dirinv.x;
2 tx2 ← (box.max.x - r.o.x)*r.dirinv.x;
3 t_min ← Min (tx1, tx2);
4 t_max ← Max (tx1, tx2);
5 ty1 ← (box.min.y - r.o.y)*r.dirinv.y;
6 ty2 ← (box.max.y - r.o.y)*r.dirinv.y;
7 t_min ← Max (t_min, Min (ty1, ty2));
8 t_max ← Min (t_max, Max (ty1, ty2));
9 return t_max ≥ t_min;
```

Algorithm 3: Example of fast bounding box intersection computation for a box in a plane.

the cross products $\boldsymbol{P} = (\boldsymbol{D} \times \boldsymbol{E}_2)$ and $\boldsymbol{Q} = (\boldsymbol{T} \times \boldsymbol{E}_1)$ once and then substituting, yielding Eq. (4.17).

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{(\boldsymbol{D} \times \boldsymbol{E}_2) \cdot \boldsymbol{E}_1} \begin{bmatrix} (\boldsymbol{T} \times \boldsymbol{E}_1) \cdot \boldsymbol{E}_2 \\ (\boldsymbol{D} \times \boldsymbol{E}_2) \cdot \boldsymbol{T} \\ (\boldsymbol{T} \times \boldsymbol{E}_1) \cdot \boldsymbol{D} \end{bmatrix} \tag{4.16}$$

$$= \frac{1}{\boldsymbol{P} \cdot \boldsymbol{E}_1} \begin{bmatrix} (\boldsymbol{Q} \cdot \boldsymbol{E}_2) \\ (\boldsymbol{P} \cdot \boldsymbol{T}) \\ (\boldsymbol{Q} \cdot \boldsymbol{D}) \end{bmatrix} \tag{4.17}$$

As shown in Listing 4 the mapped ray and triangle can be tested against the barycentric coordinate frame's bounds. These tests occur at lines 10 and 15 in Listing 4. An additional test is performed early in the execution at line 5 to determine if the facet is facing the ray (facet normal vector opposite in direction to the ray unit direction vector). If the facet is not facing the ray, the algorithm returns early as no intersection is possible.

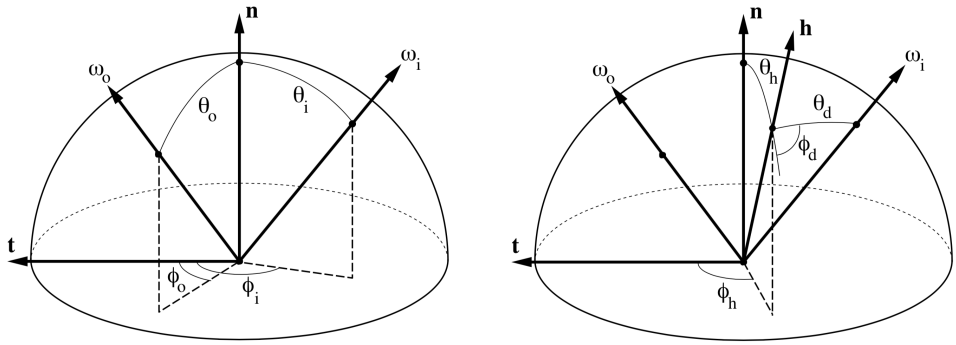
4.6 Bidirectional Reflection Distribution Functions

The resultant force vector due to an impinging light ray is coupled to the nature of the ray's interaction with the spacecraft surface materials. The total spatial distribution of a reflected light ray is described by the reflecting material's BRDF. The BRDF is defined as the ratio of reflected

Data: $\boldsymbol{o}$ is the origin of the ray
Data: $\hat{\boldsymbol{d}}$ is the direction of the ray
Data: $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, $\boldsymbol{v}_3$ triangle vertices
Result: Return triangle intersection

$\boldsymbol{e}_1 = \boldsymbol{v}_2 - \boldsymbol{v}_1$;
$\boldsymbol{e}_2 = \boldsymbol{v}_3 - \boldsymbol{v}_1$;
$\boldsymbol{p} = \hat{\boldsymbol{d}} \times \boldsymbol{e}_2$;
$det = \boldsymbol{e}_1 \cdot \boldsymbol{p}$;
if det $<$ **eps then**
  return false;
end
$\boldsymbol{t} = \boldsymbol{o} - \boldsymbol{v}_1$;
$u = (\boldsymbol{t} \cdot \boldsymbol{p}) det^{-1}$;
if $u < 0$ **or** $u > 1$ **then**
  return false;
end
$\boldsymbol{q} = \boldsymbol{t} \times \boldsymbol{e}_1$;
$v = (\hat{\boldsymbol{d}} \cdot \boldsymbol{q}) det^{-1}$;
if $v < 0$ **or** $u + v > 1$ **then**
  return false;
end
$t = (\boldsymbol{e}_2 \cdot \boldsymbol{q}) det^{-1}$;

Algorithm 4: Möller-Trumbore algorithm used for fast and memory efficient triangle intersection testing.



(a) BRDF is parameterized by the intuitive (θ_i, ϕ_i) and (θ_o, ϕ_o)

(b) BRDF is parameterized as function of the half angle (θ_h, ϕ_h) and a difference angle (θ_d, ϕ_d)

Figure 4.7: Illustrations of Two Common BRDF Geometry Descriptions.

radiance $\mathrm{d}L_r$ to the incident radiance $\mathrm{d}E_i$ [65].

The geometry of the reflection interaction is shown in Figure 4.7 where $\mathbf{x}$ is the ray intersection point on a surface and ω_o is the direction of the outgoing ray, $\hat{\boldsymbol{n}}_{\mathbf{x}}$ is the unit normal to the surface at $\mathbf{x}$, and ω_i is the direction of the incident radiation. The normalized vector, $\hat{\boldsymbol{h}}_{\mathbf{x}}$ is in the direction of the angular bisector of $\boldsymbol{\omega}_o$ and $\boldsymbol{\omega}_i$, and is defined by $\hat{\boldsymbol{h}}_{\mathbf{x}} = (\boldsymbol{\omega}_o + \boldsymbol{\omega}_i)/|\boldsymbol{\omega}_o + \boldsymbol{\omega}_i|$. In this dissertation a particular notation convention from the field of computer graphics is used for the various directional quantities denoted by ω. For instances where the quantity is solely directional the quantity is denoted as ω. Where it is meaningful to be used as a vector the quantity is written as $\boldsymbol{\omega}$ and is assumed to be a unit vector.

It is assumed that the incident light-surface interactions are occurring in the optical linear regime. Under this linear regime it has been shown experimentally that there is a proportional relationship between exitant radiance and irradiance, $\mathrm{d}L_o(\omega_o) \propto \mathrm{d}E(\omega_i)$. This allows for the development of the bidirectional reflectance distribution function, $f_r(\omega_i \to \omega_o)$, given in Eq. (4.18), where the proportionality relationship describes the observed radiance leaving a reflecting surface in the direction ω_o and the projected solid angle defined as $\mathrm{d}\sigma^{\perp}(\omega) = |\omega \cdot \hat{\boldsymbol{n}}|\mathrm{d}\sigma(\omega)$.

$$f_r(\omega_i \to \omega_o) = \frac{\mathrm{d}L_o(\omega_o)}{\mathrm{d}E(\omega_i)} = \frac{\mathrm{d}L_o(\omega_o)}{L_i(\omega_i)\mathrm{d}\sigma^{\perp}(\omega_i)} \tag{4.18}$$

The relationship between the outgoing radiance and the incoming radiance for a particular optical

surface is described as

$$\mathrm{d}L_o(\omega_o) = \mathrm{d}L(\omega_i) f_r(\omega_i \rightarrow \omega_o) \mathrm{d}\sigma^{\perp}(\omega_i) \tag{4.19}$$

Integrating Eq. (4.19) yields the total radiance, over the hemisphere, leaving a surface area element as [92]

$$L_o(\omega_o) = \int_{S^2} L(\omega_i) f_r(\omega_i \rightarrow \omega_o) \mathrm{d}\sigma^{\perp}(\omega_i). \tag{4.20}$$

This work employs physically plausible BRDFs. A physically plausible BRDF adheres to the symmetry expression given at Eq. (4.21) and energy conservation condition given by Eq. (4.22).

$$f_r(\omega_i \rightarrow \omega_o) = f_r(\omega_o \rightarrow \omega_i) \quad \text{for all} \quad \omega_i, \omega_o \tag{4.21}$$

$$\int_{\mathcal{S}_o^2} f_r(\omega_i \rightarrow \omega_o) \mathrm{d}\sigma^{\perp}(\omega_o) \leq 1 \quad \text{for all} \quad \omega_i \in \mathcal{S}_i^2 \tag{4.22}$$

For a large majority of materials a BRDF can be described as the combination of a specular component and a diffuse component. Whereas specular reflection is due to surface reflection, diffuse reflection is due to subsurface scattering and surface microgeometry. While subsurface scattering contributes to the generation of diffuse reflections this work does not model internal material refraction nor transmission between transparent layers. As a result the contribution of subsurface scattering is represented by adding a diffuse term to the specular term giving the complete BRDF description

$$f_r(\omega_i \rightarrow \omega_o) = \rho R_d + s R_s \tag{4.23}$$

where ρ and s are the proportions of the surface behaving as a diffuse and a specular respectively and $\rho + s = 1$.

4.6.1 Ideal BRDF

A commonly used first order BRDF is the combination of diffuse lambertian and ideal specular mirror-like reflection. This BRDF expression is given in Eq. (4.24), where F_0 is the Fresnel reflection coefficient, ρ the diffuse scaling constant of the material and $\hat{\boldsymbol{\omega}}_o$ outgoing mirror reflected direction given by $\hat{\boldsymbol{\omega}}_o = 2(\hat{\boldsymbol{\omega}}_i \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}} - \hat{\boldsymbol{\omega}}_i$.

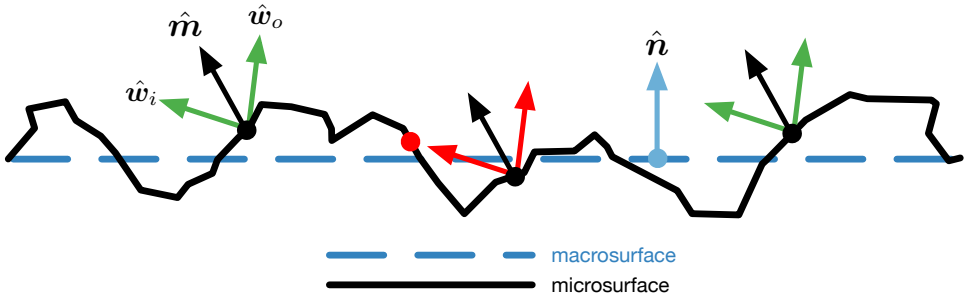


Figure 4.8: Conceptual illustration of microfacet with shadowing-masking geometry.

$$f_r(w_i \rightarrow \omega_o) = d\left(\frac{\rho}{\pi}\right) + s\left[\frac{F_0\delta(\hat{\boldsymbol{\omega}}_i - \hat{\boldsymbol{\omega}}_o)}{\cos\theta_i}\right] \tag{4.24}$$

4.6.2 Mircofacet Model BRDF

The microfacet model is an approach to describe the specular term of a BRDF by modelling the material surface as a collection of optically flat statistically distributed facets. The facets are assumed to be at a scale too small for shading considerations and significantly larger than the wavelength of the incident radiation [69]. Figure 4.8 presents a conceptual explanation of the microfacet geometries. Three points have the same macrosurface normal $\hat{\boldsymbol{n}}$ and microsurface normal $\hat{\boldsymbol{m}}$. The green reflection pairs are visible in both the $\hat{\boldsymbol{w}}_o$ and $\hat{\boldsymbol{w}}_i$ directions, while the red is blocked (in $\hat{\boldsymbol{w}}_i$ in this case) [93]. Given these assumptions of the surface microgeometry, the distribution of the facets and the resulting specular BRDF term is given in Eq. (4.25).

$$R_s = \frac{D(\omega_h)G(\omega_o, \omega_i)F(\omega_o)}{4(\hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{\omega}}_o)(\hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{\omega}}_i)} \tag{4.25}$$

In Eq. (4.25) the function $D(\omega_h)$ is the microgeometry normal distribution function (NDF) which describes the distribution of the macro surface area, as microfacets, which are oriented with respect to the incoming radiation direction, such that the incoming radiation could reflect in a given direction on the hemisphere. The geometry function, $G(\omega_o, \omega_i)$, governs the fraction of area of facets which are not shadowed by other facets. The function $F(\omega_o)$ is the Fresnel reflection factor of the active microfacet areas and controls how much of the radiation is reflected from the facets

given the radiation angle of incidence. Finally, the denominator $4(\hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{w}}_o)(\hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{w}}_i)$ is a correction factor, which resolves quantities being mapped between the local microgeometry coordinate space and that of the macrosurface coordinate space.

The Fresnel equations are the solution to Maxwell's equations at smooth surfaces and describe a material's reflectance for two polarization states of the incident illumination. The reflectance depends on the materials index of refraction and the angle between the incident radiation and the surface normal vector. Assuming a radiation source of mixed polarization, the rational approximation as developed by Schlick is used to compute the Fresnel factor with a significant reduction in computational complexity [79]. The Schlick approximation is given in Eq. 4.26, where the angle considered $(\boldsymbol{\omega}_i \cdot \hat{\boldsymbol{h}})$ is the angle between the incoming radiation source and the microfacet normal.

$$F(\boldsymbol{\omega}_i \cdot \hat{\boldsymbol{h}}) = f_0 + (1 - f_0)(1 - (\boldsymbol{\omega}_i \cdot \hat{\boldsymbol{h}}))^5 \tag{4.26}$$

Selection of the terms $D(\omega_h)$ and $G(\omega_o, \omega_i)$ determine the resulting specular distribution of reflected radiation. The NDF function $D(\omega_h)$ controls the size, brightness, and overall shape of the BRDF's specular highlight[69]. Many different NDFs have been presented in the computer graphics literature. Often the NDF function is Gaussian-like and includes a parameter which describes the "roughness" or variance of microfacet normals [21]. The geometry function $G(\omega_o, \omega_i)$ is a probability that surface points, with a given microgeometry normal $\hat{\boldsymbol{n}}$, will be visible from both the light direction $\hat{\boldsymbol{w}}_i$ and the view direction $\hat{\boldsymbol{w}}_o$.

This work implements two microfacet BRDF formulations. A microfact BRDF is defined by the combination of a particular NDF and masking/shadowing function ($G(\omega_o, \omega_i)$). The specific NDFs emplyed are the Beckmann distribution and Trowbridge-Reitz (GGX) distributions. The distributions are anisotropic and the size of the specular lobe in each distribution is controlled by the roughness parameters α_x and α_y and the distribution NDF for each model.

The first microfacet BRDF uses the NDF by Beckmann and Spizzichino and is shown in Eq. 4.27, where θ_h is the angle $\omega_i \cdot \hat{\boldsymbol{h}}$ [7].

$$D(\omega_h) = \frac{e^{-\tan^2(\theta_h)/\alpha^2}}{\pi\alpha^2 \cos^4(\theta_h)} \tag{4.27}$$

The shadowing function used with the Beckmann distribution is one originally developed by Smith [82] and given as

$$G(\omega_o, \omega_i) = \frac{1}{1 + \Lambda(\omega_o)} \frac{1}{1 + \Lambda(\omega_i)} \tag{4.28}$$

where $\Lambda()$ is given by

$$\Lambda(\omega) = \frac{1}{2} \left(\text{erf}(a) - 1 + \frac{e^{-a^2}}{a\sqrt{\pi}} \right) \tag{4.29}$$

For the Smith masking function to be an exact solution for the stochastic microfacet surface an assumption is made that microfacet normal vectors and microfacet masking/shadowing are statistically indpendent [41]. Additionally, rather than evaluate the computationally expensive $\text{erf}(x)$ and e^x functions in Eq. 4.29, Walter et al. present the rational polynomial approximation shown in Eq. 4.30 where $a = 1/(\alpha + |\tan^{-1}(\omega_i \cdot \hat{\boldsymbol{h}})|)$. This approximation evaluates to a relative error of less than 0.35% [93].

$$\Lambda(\omega) = \frac{3.535a + 2.181a^2)}{(1.f + 2.276f * a + 2.577f * a2} \tag{4.30}$$

The GGX distribution, originally introduced by Walter et al. [93] and later generalized to a anisotropic distribution by Heitz [41] is given in Eq. 4.31.

$$D(\omega_h) = \frac{1}{\pi \alpha_x \alpha_y \cos^4(\theta_h) \left(1 + \tan^2(\theta_h)(\cos^2 \phi_h / \alpha_x^2 + \sin^2 \phi_h / \alpha_y^2)\right)^2} \tag{4.31}$$

The masking function used with the GGX distribution function deviates from the Smith developed function in Eq. 4.28. Whereas the Smith masking fucntion assumes that visible microfacets are uncorrelated. However, for many surfaces the probability of a microfacet being visible from both the incoming and outgoing directions is greater the closer together the microfacets. as such the masking function used for the GGX distribution is

$$G(\omega_o, \omega_i) = \frac{1}{1 + \Lambda(\omega_o) + \Lambda(\omega_i)} \tag{4.32}$$

where $\Lambda(\omega)$ is

$$\Lambda(\omega) = \frac{-1 + \sqrt{1 + \alpha^2 \tan^2(\theta)}}{2} \tag{4.33}$$

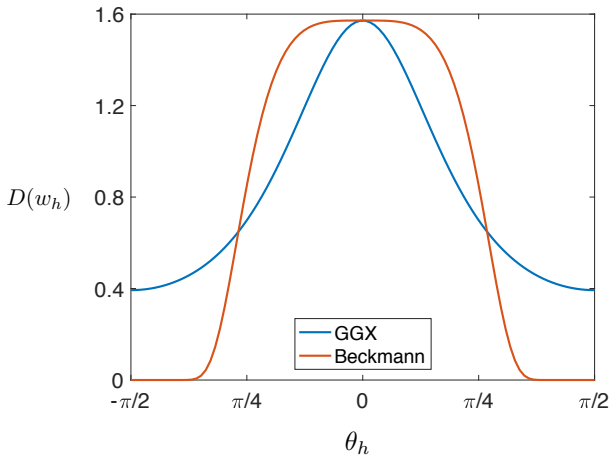


Figure 4.9: Isotropic Beckmann and GGX microfacet NDFs as a function of the angle between a ray direction and the the surface normal, θ_h, for roughness value $\alpha = 0.45$.

As a demonstration of the difference between the two NDFs, Figure 4.9 shows the magnitude of the differential area of the distribution of lit microfacets for grazing angles $-\pi/2$ to $\pi/2$. Figure. 4.9 is generated with a roughness value α =0.45. The value α =0.45 is chosen to be instructive as it will be used in the following section's orbit propagation results. The α parameter controls the spread of the lobe of the specular reflection. The GGX distribution has greater magnitude at the extremities and falls off to zero more slowly, than the Beckmann, for directions far from the surface normal.

4.7 Evaluating Ray-Surface Interaction

Evaluating the force and reflected ray direction requires computing the integral given by in Eq. 4.20. This integral contains the potentially complex analytic expression for the BRDF $f_r(\omega_i \rightarrow \omega_o)$. Often these BRDF functions have no developed analytic solution or simply contain discontinuities (such as the delta function describing mirror specular reflection) which make such general solutions unattainable. As a result numerical methods are used for evaluating these complex integrals. Numerical quadrature is a common computational method for computing an integral, however, its computational load increases as the dimensionality of the integral increases[REF]. For the BRDFs used in this dissertation the integral dimension is at three and possibly five $(\omega_o, \omega_i, \hat{\boldsymbol{h}})$

for anisotropic materials. To overcome this dimensionality constraint a quasi-Monte Carlo method is used to generate an estimate of the scattering integral. The basic Monte Carlo estimator is shown in Eq. 4.34, where N is the number of samples taken and X_i a sample point at which to evaluate the integral which is chosen from the probability density function (pdf) p.

$$\langle F^N \rangle = \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(X_i)}{p(X_i)} \tag{4.34}$$

Monte Carlo importance sampling is used to evaluate the integral in Eq. (4.20). In the context of graphics rendering, Monte Carlo ray tracing casts many rays from a single pixel to estimate the radiance received at that pixel. The number of rays cast for a single pixel is given by the quantity N in Eq. (4.2). With a sufficiently large value for N a good estimate of the scattering equation can be attained and, in turn, its effect on the resultant SRP force direction and magnitude determined. However, this typically requires casting many rays per pixel in multiple ray waves. Each ray wave incurs the communication and data overhead of launching the various OpenCL ray generation and tracing kernels. This communication overhead is a significant source of latency in GPGPU programming. To overcome this latency and thus maintain faster-than-realtime evaluations, it is assumed that features of the spacecraft mesh model surface area are much larger (multiple cm^2) than the ray cross sectional area (mm^2) and that each feature on the spacecraft mesh model possess a common BRDF. As a result, rather than casting $N = 100$ waves of rays at a particular ray resolution e.g. 1 cm^2, a single densely packed wave of rays at ray resolution 1 mm^2 is cast. This densely packed wave of 1 mm^2 rays is equal to the 100 waves of 1 cm^2 rays.

At each ray-surface interaction the ray throughput α_i, its directional force $\boldsymbol{F}_i$ and reflected direction ω_o, must be computed. Due to the importance sampling approach, the throughput of the ray continuation is a function of the BRDF and probability density function from which the outgoing ray direction ω_o is sampled. This importance sampling is accommodated in Eq. (4.35).

$$\alpha_i = L(\mathbf{x}_0, \omega_0) \prod_{j=0}^{i-1} \frac{f_r(\mathbf{x}_{j+1}, \omega_{j+1} \rightarrow -\omega_j) |\hat{\boldsymbol{\omega}}_{j+1} \cdot \hat{\boldsymbol{n}}(\mathbf{x}_{j+1})|}{p_{j+1}(\omega_{j+1})}. \tag{4.35}$$

Therefore, the process of generating outgoing ray direction ω_o and computing the weighting of the ray according to the BRDFs PDF, is implemented in a manner that is specific for each BRDF type.

4.7.1 Sampling Ideal BRDF

Computing ω_{j+1} and $p_{j+1}(\omega_{j+1})$ for a specular reflection is trivially the reflected direction computed by

$$\hat{\boldsymbol{\omega}}_{o_{j+1}} = 2(\hat{\boldsymbol{\omega}}_i \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}} - \hat{\boldsymbol{\omega}}_i \tag{4.36}$$

The PDF is evaluated as

$$p_{j+1}(\omega_{o_{j+1}}) = 1 \tag{4.37}$$

The reflected ray direction for a lambertian diffuse interaction is computed by uniformly sampling the hemisphere above the intersection point. The vector components of $\boldsymbol{\omega}_{o_{j+1}}$ for a diffuse ray are given by Eq. (4.38) where ϵ_1 and ϵ_2 are random samples chosen from a uniform distribution [69].

$$x = \cos(2\pi\epsilon_2)\sqrt{1-\epsilon_1^2} \tag{4.38a}$$

$$y = \sin(2\pi\epsilon_2)\sqrt{1-\epsilon_1^2} \tag{4.38b}$$

$$z = \epsilon_1 \tag{4.38c}$$

The probability of selecting this direction is given by Eq. (4.39), where θ_o corresponds to the angle marked in Figure 4.7.

$$p_{j+1}(\omega_{o_{j+1}}) = \frac{\cos(\theta_o)}{\pi} \tag{4.39}$$

4.7.2 Sampling Microfacet BRDFs

Given that microfacet BRDFs model the surface as a distribution of micro specular facets one must choose a micro facet normal direction vector and compute the reflected ray about that normal according to Eq. 4.36. The probability of this outgoing ray is equal to the probability of selecting the particular micro facet normal vector. To reduce the variance of the Monte Carlo estimator it is wise to choose samples from a PDF which is as close to the actual BRDF integral function as possible. Of course, if one could choose a pdf which is exactly proportional to the integral then this would obviate the need to perform Monte Carlo integration in the first place. Therefore, a common technique in image rendering is to select micro facet normal vector samples directly from the NDF

in the BRDF. The NDF primarily controls the shape of a microfacet BRDF and therefore leads to a significant reduction in estimate variance[43]. However, at small grazing angles the microfacets visible from a given outgoing direction is very different from the overall distribution. Therefore, as shown by Heitz in Reference [42], microfacet normal vector samples will be drawn from the distribution of visible normals as given by

$$D_\omega(\omega_h) = \frac{D(\omega_h)G(\omega, \omega_h)\max(0, \boldsymbol{\omega} \cdot \boldsymbol{\omega_h})}{\cos(\theta)} \tag{4.40}$$

Because the normal vector sampled from $D_\omega(\omega_h)$ is defined with respect to the half angle vector $\boldsymbol{h}$, the PDF $D_\omega(\omega_h)$ is transformed to a sample in the outgoing ray direction $D_\omega(\omega_o)$.

$$p_{j+1}(\omega_{o_{j+1}}) = D_\omega(\omega_o) = \frac{D_\omega(\omega_h)}{4(\boldsymbol{\omega}_o \cdot \boldsymbol{\omega}_h)} \tag{4.41}$$

4.7.3 Computing The Total BRDF

When computing the outgoing ray direction a single ray direction is chosen. However, the multiple contributing models to the BRDF (eg. specular and diffuse) when sampled result in different reflection behaviors. Therefore, the BRDF component, from which the outgoing ray direction is computed, is selected in a probabilistic manner. Either the diffuse or specular BRDF is chosen with a probability proportional to the BRDF's contribution to the final weighted BRDF as shown in Eq. 4.23. The sub-mesh material is defined as a weighted combination of the diffuse and specular BRDFs. At run time the two contributing BRDFs are read in from the mesh model file and allocated to either the `1st` BRDF slot or the `2nd` BRDF slot. The previously presented general combined weighted BRDF expression is shown again at Eq. 4.42

$$f_r(\omega_i \rightarrow \omega_o) = \rho R_d + s R_s \tag{4.42}$$

The weighting for this material's BRDF combination is computed as

$$\text{weight} = \frac{\rho_{i_{1\text{st}}}}{\rho_{i_{1\text{st}}} + \rho_{i_{2\text{nd}}}} \tag{4.43}$$

```
sample = uniformSample();
if sample < weight then
    idx = mat.firstBrdfIdx;
    brdf = scene.materials [idx ];
else
    idx = mat.secondBrdfIdx;
    brdf = scene.materials [idx ];
end
```

Algorithm 5: Selecting material BRDF.

where the $\rho_{i_{1\text{st}}}$ and $\rho_{i_{2\text{nd}}}$ variables correspond to either the ρ or s from Eq. 4.42 according to the order (1st or 2nd) in which the BRDFs are loaded. Finally, a uniformly distributed sample in the range $[0, 1]$ is used to toggle the choice between BRDFs as shown in Algorithm. 5.

4.8 Model Validation

Model validation is performed by computing the percentage error of the force vector for both a surface evaluated with the ray-tracing approach relative to a surface evaluated with the faceted approach. The magnitude percentage relative error is computed as

$$\frac{|F_{\text{ray}} - F_{\text{facet}}|}{|F_{\text{facet}}|} \tag{4.44}$$

and similarly for each force vector component. Additionally, to demonstrate the ray-tracing method's ability to capture the diffusely reflected rays, the relative error is computed for a completely specular surface, completely diffuse surface and a mixed specular and diffuse surface. For the completely specular and completely diffuse surface the cube mesh shown in Figure 4.10 is evaluated at the sun heading $\hat{\boldsymbol{s}}_B = (1.0, 0.0, 0.0)$. To demonstrate the dependence of force direction and magnitude on the ray resolution, the mixed case evaluation is conducted with $\hat{\boldsymbol{s}}_B = (0.7071, 0.7071, 0.0)$. The spacecraft model is a simple cube of side length one meter shown in Figure 4.10. Material characteristics are controlled by the coefficients for absorption ρ_a, diffuse ρ_d and specular ρ_s reflections.

The percentage error of the force direction vector components for a specular cube mesh model is shown in Figure 4.11. The error in the $\hat{\boldsymbol{x}}_B$ component is consistently $2.6{\times}10^{-6}$ %. This small

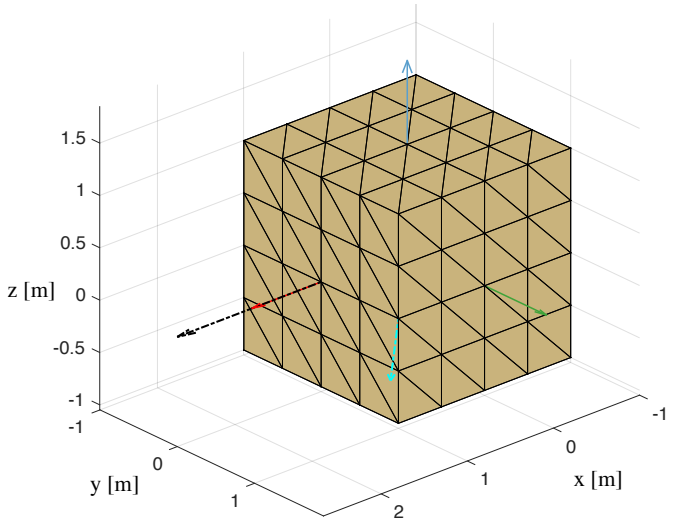


Figure 4.10: Test cube spacecraft model. Black and cyan vectors indicate body-frame sun headings evaluated. Red, green and blue vectors denote first, second and third body-frame axes respectively.

relative error is attributed to floating point precision errors in the representation of the model's facet normal vectors. These small errors manifest the same reflection geometry for each evaluation at different ray resolutions and therefore a small consistent force direction error. The percentage force error $\hat{\boldsymbol{y}}_B$ and $\hat{\boldsymbol{z}}_B$ directions are of order 10^{-14} and less.

For a lambertian diffuse cube mesh evaluation the error of ray-traced force components relative to the faceted force norm are shown in Figure 4.12. It is again evident that as the ray resolution decreases the relative error to the faceted model also decreases to less than half a percent. As the ray resolution decreases, the number of rays being cast to approximate the integral of the diffuse BRDF increases. For ray resolutions less than approximately 5 mm the error remains below one percent. The increased number of rays produces an improved estimate to the integral. This simple test demonstrates that this ray-tracing method is accurately capturing the diffuse ray reflections and their impact on the resultant force. The error of the ray-traced force components relative to the faceted force norm for a mixed (diffuse and specular) material evaluation are shown in Figure 4.13. The error relative to the faceted evaluation decreases with increased ray density and remains below one percent for ray resolutions approximately less than 3 mm. It is evident that for mixed material case a smaller small ray size is required to achieve a less than one percent error.

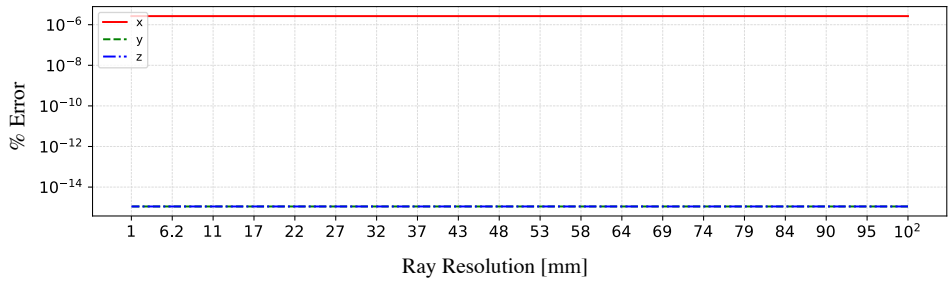


Figure 4.11: Error of the ray-traced force components relative to the faceted force norm for a specular material evaluation.

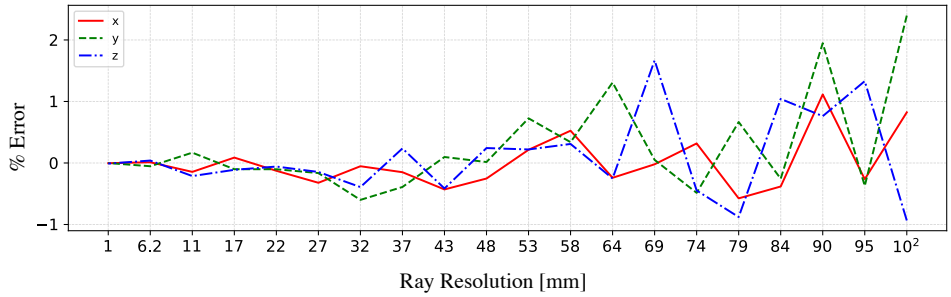


Figure 4.12: Error of the ray-traced force components relative to the faceted force norm for a diffuse material evaluation.

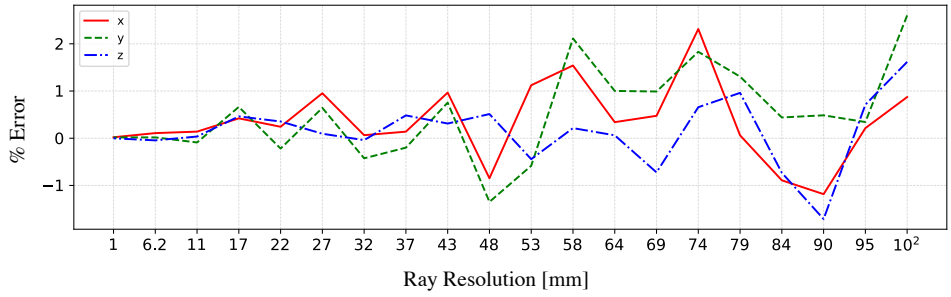


Figure 4.13: Error of the ray-traced force components relative to the faceted force norm for a mixed (diffuse and specular) material evaluation.

Table 4.1: SRP force for faceted evaluations.

Method (ρ_a, ρ_d, ρ_s)	Force [N] $\times 10^{-5}$
Diffuse (0.2, 0.8, 0.0)	-2.783188, 0, 0
Specular (0.2, 0.0, 0.8)	-3.267220, 0, 0
Mix (0.2, 0.4, 0.4)	-2.157385, -2.157385, 0

For the mixed material the ray resolution required is 3 mm, whereas in the solely diffuse case a ray resolution of 5 mm is sufficient. This is due to the number of rays being probabilisticaly selected as either diffuse or specular reflection. For example, given a 1 mm ray size, 100 rays will intersect an are of 1 cm^2. If the contributions of of diffuse and specular phenomena are equal then it is likely that 50 rays will reflect as specular and 50 as diffuse. This reduces by half the number of diffuse rays approximating the diffuse scattering function. A smaller ray size therefore increases the number of rays intersecting the 1 cm^2 area and provides an improved estimate of the scattering integral of Eq. (4.20).

4.9 Multiple Ray Reflections

To prove multiple reflections are being effectively captured, the model and sun-spacecraft heading, shown in Figure 4.14 are evaluated. This model demonstrates the change in resultant direction of the force due to multiple surface ray bounces. A manual computation of the faceted method is carried out to compute the force direction for the two bounces which will occur for the incoming radiation. The model material is completely specular with $\rho_s = 0.8$. The resulting faceted force is $\boldsymbol{F}_{\text{facet}} = (0.0, -4.25016, 0.0) \times 10^{-5}$ [N]. Computing the ray-traced evaluation with multiple bounces yields the percent error in force relative to the faceted method shown in Figure 4.15. For all ray resolution the force components are near or less than one percent error. In particular the magnitude of the $\hat{\boldsymbol{x}}_B$ and $\hat{\boldsymbol{z}}_B$ components are on the order of 10^{-15} and below.

To investigate the importance of resolving multiple ray continuations an evaluation of the high-fidelity OSIRIS-REx spacecraft model is carried out for a uniform distribution of spacecraft sun-headings ${}^{\mathcal{B}}\hat{\boldsymbol{s}}$ in the 4π str attitude space. The ray resolution is 2.5 mm and the solar intensity is taken as at distance from the sun 1 AU. To more conveniently convey an intuitive sense of the

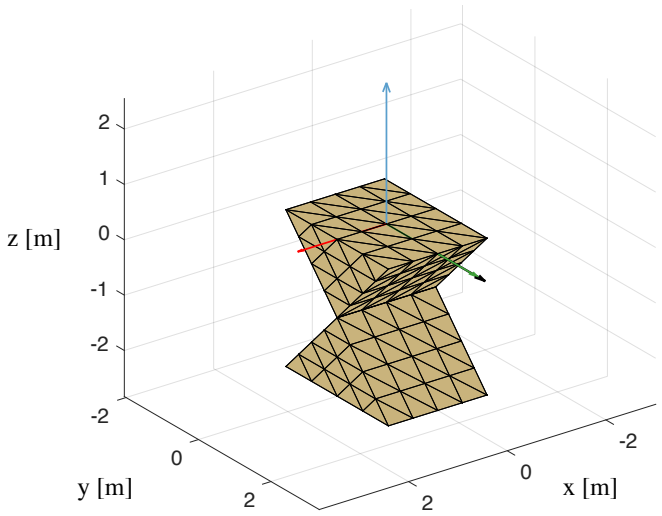


Figure 4.14: Test model with two surfaces which form a right angled face. The black vector indicates body-frame sun heading evaluated. Red, green and blue vectors denote first, second and third body-frame axes respectively.

magnitude of the percentage difference is computed with respect to a baseline value as shown in Eq. (4.47). The baseline value is computed using the force and torque computed from a single ray bounce. In both plots the percentage difference is computed according to Eq. (4.46), where i indicates the number of bounces used to compute the force $\boldsymbol{F}$.

$$F_{\text{base}} = \frac{1}{N}\sum_{n=1}^{N} |\boldsymbol{F}_{\text{n}}| \tag{4.45}$$

$$F_{\text{i+1, i}} = \frac{|\boldsymbol{F}_{\text{i+1}}| - |\boldsymbol{F}_{\text{i}}|}{F_{\text{base}}} \times 100 \tag{4.46}$$

The force magnitude percentage difference between the first and second ray bounce is shown in Figure 4.16(a), and the difference between second and third shown in Figure 4.16(b). It is evident that the majority of the force difference from scattered radiation is captured in tracing rays beyond the first intersection. Figure 4.16(a) demonstrates that if one is concerned only with a sun point attitude, which is equivalent to latitude and longitude of approximately (0°, 0°), then the resultant error for only computing the first surface interaction is a small over prediction of approximately 2%. However, for most other attitudes computing only the first intersection produces an under prediction of the force of at least 3% up to almost 8%.

The torque magnitude difference between the first and second bounce is shown in Fig-

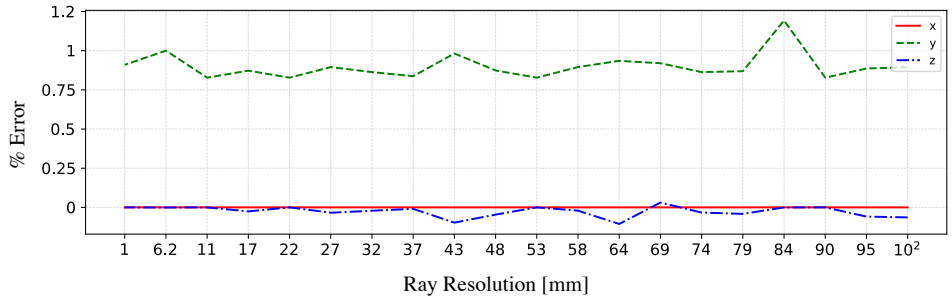


Figure 4.15: Error of the ray-traced force components relative to the faceted force norm for multiple bounce evaluation.

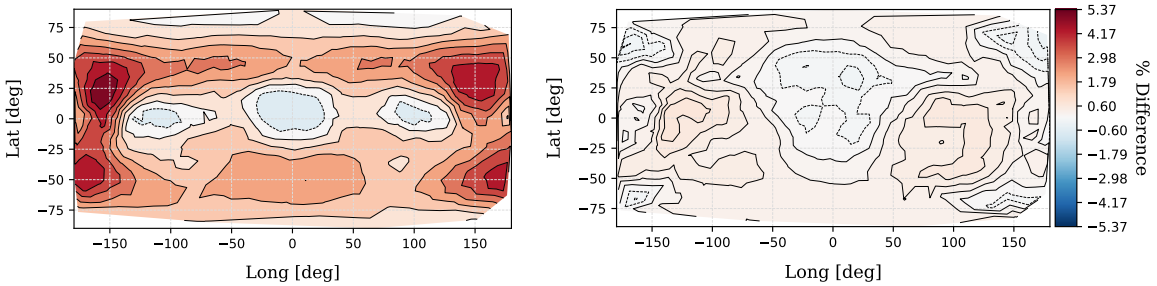
ure 4.17(a), and the difference between second and third shown in Figure 4.17(b). The same relative difference measure as used in Eq. (4.46) is used here, yet for torque values rather than force. Figure 4.17(a) shows that computing torque for only one bounce results in torque magnitude under prediction of approximately 10% for almost all sun-headings. As with the force magnitude results computing at least two bounces results in a significant reduction in torque magnitude error.

The force percentage difference, in each of the body-frame components, between the first and second bounce, is shown in the figure set 4.18. Similarly, the difference between resolving the second and third bounce is shown in Figure. 4.19. The percentage difference is thus computed as given in Eq. (4.48), where $\Delta F_{(\mathrm{i+1,\ i})_k}$ is the percentage difference between bounce iterations i and $i+1$ for force vector component k. This approach is used for both plotting of the force and torque.

$$F_{\mathrm{base}} = \frac{1}{N}\sum_{n=1}^{N} |\boldsymbol{F}_{\mathrm{n}}| \tag{4.47}$$

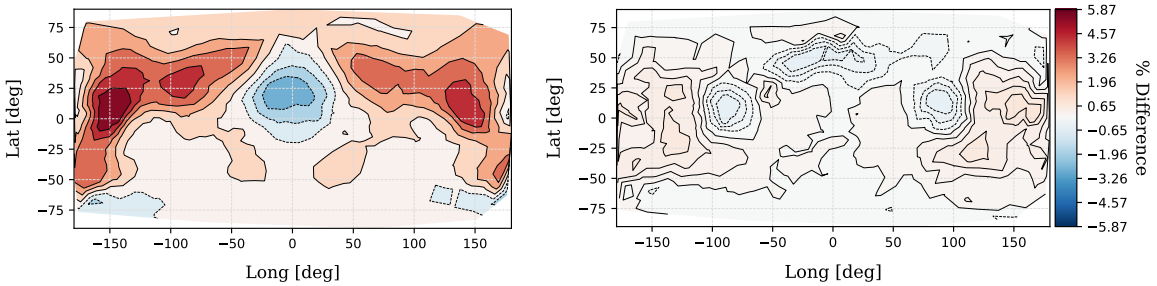
$$\Delta F_{(\mathrm{i+1,\ i})_k} = \frac{F_{(\mathrm{i+1})_k} - F_{\mathrm{i}_k}}{F_{\mathrm{base}}} \times 100 \tag{4.48}$$

The force percentage difference between resolving the first and second bounce is shown for each component in the body-frame, in Figure 4.20. The difference between the second and third bounce is shown in Figure 4.21. For the OSIRIS-REx spacecraft mesh, inspection reveals that in general resolving only the first bounce results in an under prediction of force in the $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$ body frame components and an over prediction in the $\hat{\boldsymbol{z}}$ component. The torque percentage difference between resolving the first and second bounce, for each torque component in the body-frame, is shown in



(a) Force intersection 2 - 1 (b) Force intersection 3 - 2

Figure 4.16: Difference in force magnitude for resolving multiple bounces on hifidelity OSIRIS-REx.

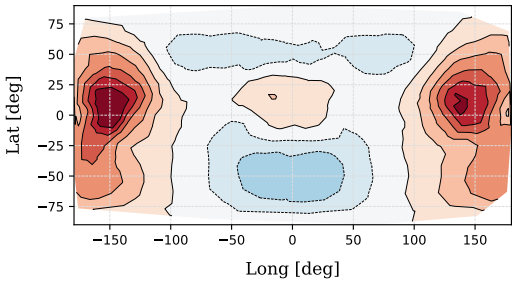


(a) Torque intersection 2 - 1 (b) Torque intersection 3 - 2

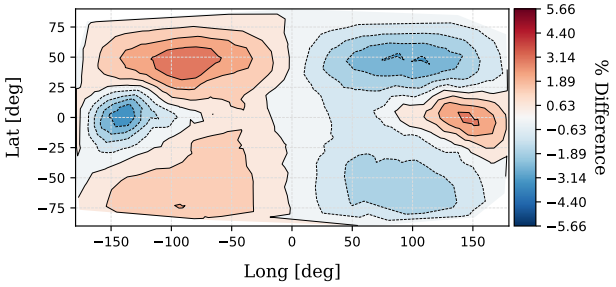
Figure 4.17: Difference in torque magnitude for resolving multiple bounces on hifidelity OSIRIS-REx.

Figure 4.20 . The difference between resolving the second and third bounce is shown in Figure 4.21. Individual assessment of the torque body-frame components reveals that for a large portion of sun-headings the torque difference between one and two bounces remains below 10%. The significance of the impact of such differences is tied to the spacecraft's planned guidance and navigation concept of operations (CONOPS). The degree to which the nominal CONOPS for the spacecraft operates in any of these high torque error attitudes will determine whether one will choose to model one, two or more bounces. However, given the computational speed of this method, the computational penalty for computing two bounces is negligible.

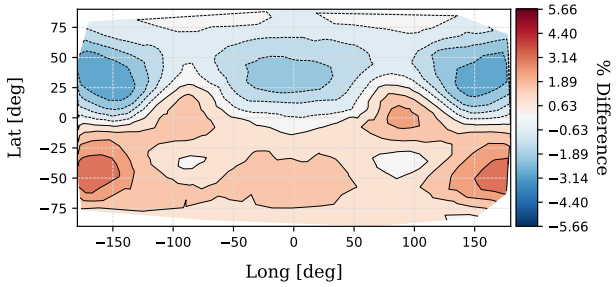
The speed and easy configuration of the parallel ray-tracing method enables quick visual analysis of spacecraft attitudes of interests. To demonstrate the visual analysis, the over prediction shown in the one ray bounce relative to two ray bounces for the $\hat{\boldsymbol{y}}$ component percent difference shown in Figure 4.18(b), is assessed in Figure 4.22. Each figure shows the rendered model in



(a) Force x % difference intersection 2 - 1

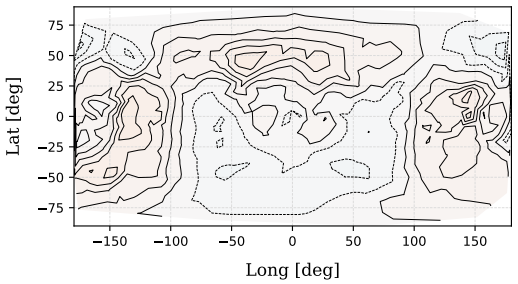


(b) Force y % difference intersection 2 - 1

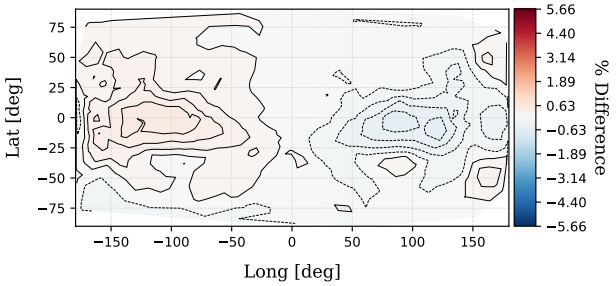


(c) Force z % difference intersection 2 - 1

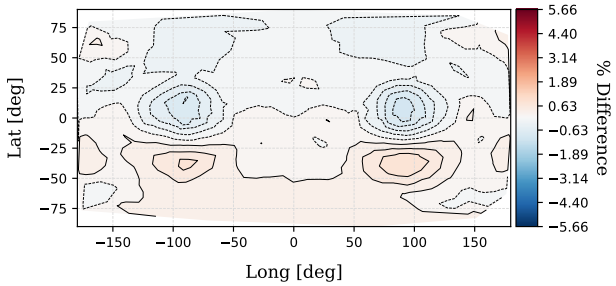
Figure 4.18: Force percentage difference between resolving second and first bounce relative to baseline value 5.62995×10^{-5} [N].



(a) Force x % difference intersection 3 - 2

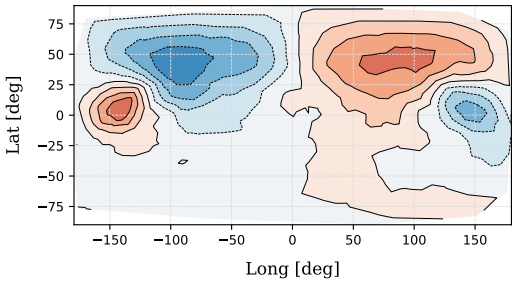


(b) Force y % difference intersection 3 - 2

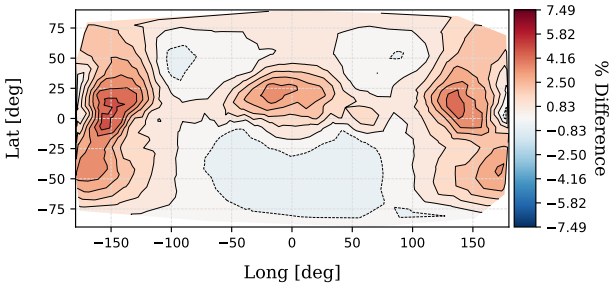


(c) Force z % difference intersection 3 - 2

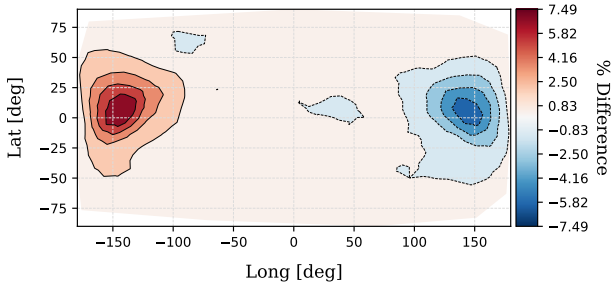
Figure 4.19: Force percentage difference between resolving third and second bounce relative to baseline value 5.62995×10^{-5} [N].



(a) Torque x % difference intersection 2 - 1

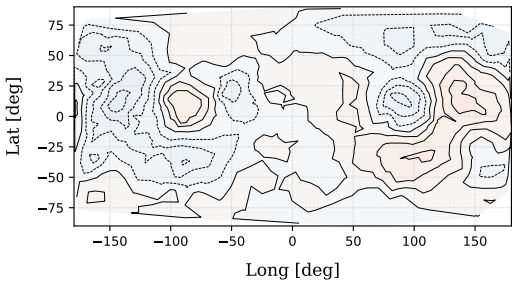


(b) Torque y % difference intersection 2 - 1

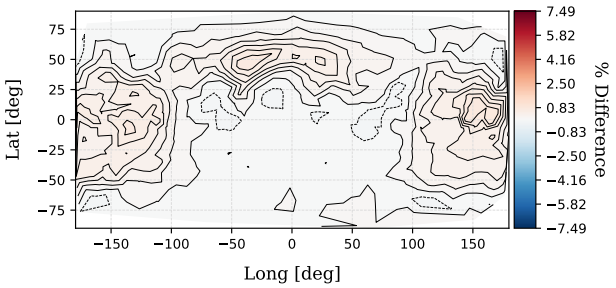


(c) Torque z % difference intersection 2 - 1

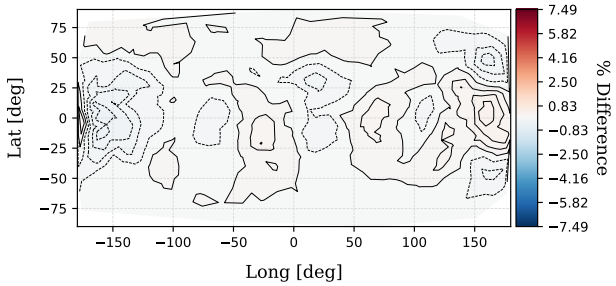
Figure 4.20: Torque percentage difference between resolving second and first bounce relative to baseline value 6.44875×10^{-5} [Nm].



(a) Torque x % difference intersection 3 - 2

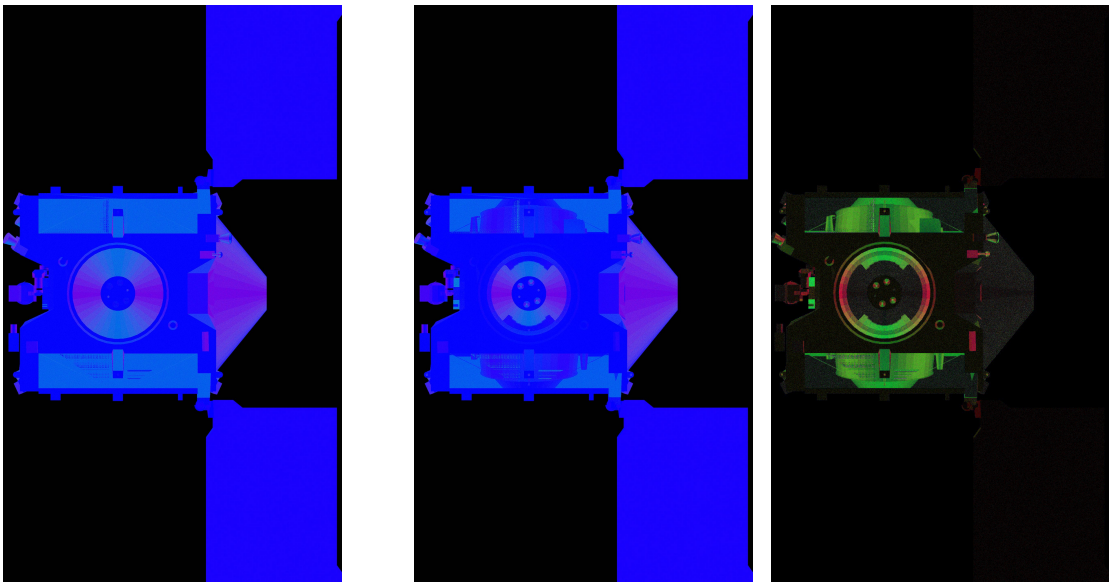


(b) Torque y % difference intersection 3 - 2



(c) Torque z % difference intersection 3 - 2

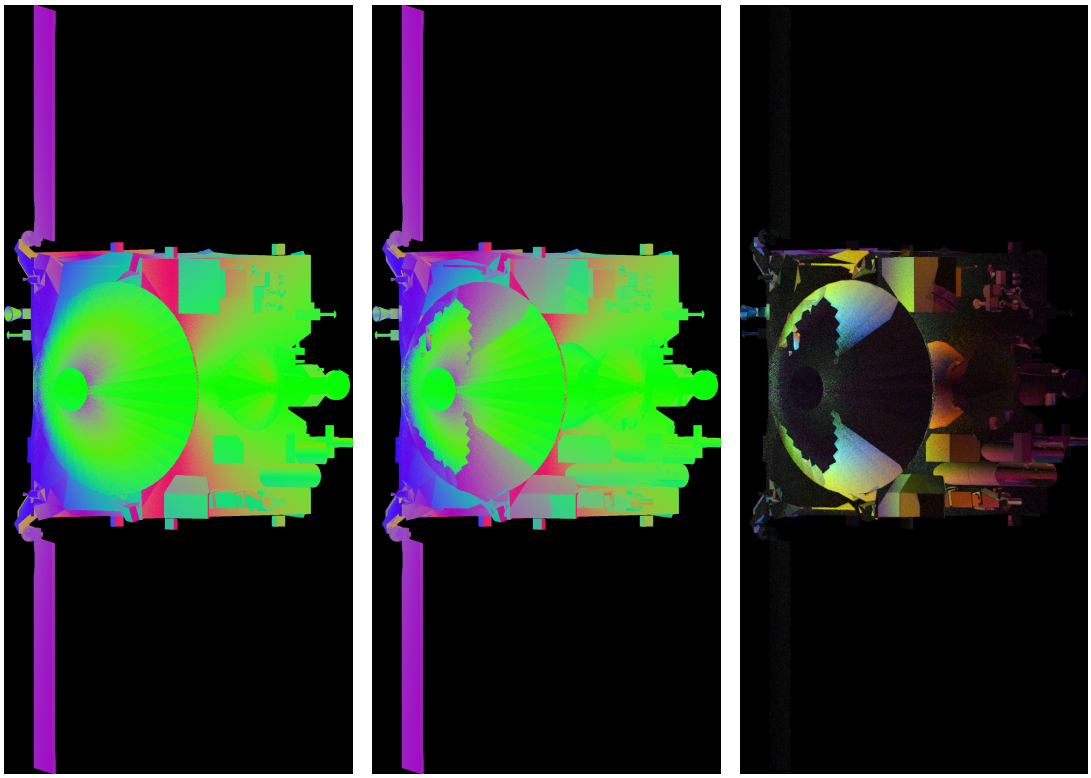
Figure 4.21: Torque percentage difference between resolving third and second bounce relative to baseline value 6.44875×10^{-5} [Nm].



(a) One bounce. (b) Two bounces. (c) Image difference.

Figure 4.22: High percentage difference attitude for force, between one and two bounces. The sun heading latitude and longitude is (-90°, 0°) which is equivalent to ${}^{\mathcal{B}}\hat{\boldsymbol{s}} = [0.0, 0.0, -1.0]$.

the sun frame where ${}^{\mathcal{B}}\hat{\boldsymbol{s}}$ points out of the page. The image's false RGB color is generated by mapping the $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{z}}$ force components to the R, G and B channels, respectively. Inspection of Figure 4.22(a) shows the spacecraft rendered with one bounce while Figure 4.22(b) shows the spacecraft rendered after two bounces. To highlight the difference in force resolution between Figure 4.22(a) and Figure 4.22(b), the difference of the pixels in these two figures is computed and presented in Figure 4.22(c). It is evident that the large angled surfaces of the thruster ring and the thermal vents redirect a significant portion of force at this attitude. Additionally, the large percentage error in the $\hat{\boldsymbol{y}}$ component is directly visible in the green coloring of the image where the green channel corresponds to the $\hat{\boldsymbol{y}}$ force component. A similar demonstration carried out for a high torque percentage difference sun heading. The sun heading latitude and longitude (-1.8°, 40.7°). Again, each figure shows the rendered model in the sun frame where ${}^{\mathcal{B}}\hat{\boldsymbol{s}}$ points out of the page. The image's false RGB color is generated by mapping the $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{z}}$ torque components to the R, G and B channels, respectively. Inspection of Figure 4.23(a) shows the spacecraft rendered with one bounce while Figure 4.23(b) shows the spacecraft rendered after two bounces. The difference



(a) One bounce. (b) Two bounces. (c) Image difference.

Figure 4.23: High percentage difference attitude for torque, between one and two bounces . The sun heading latitude and longitude (-1.8°, 40.7°).

between the first and second bounce is presented in Figure 4.23(c). It is evident that the high gain antenna and spacecraft instrument deck redirect a significant portion of incident radiation.

To demonstrate the importance of resolving multiple ray continuations an evaluation of the CloudSat spacecraft is computed for up to six ray continuations [85]. A high precision 'truth' ray-tracing evaluation is generated as a baseline value for comparison. This evaluation uses a ray resolution of two millimeters, a ray termination condition of maximum six ray continuations and 100 waves of rays resulting in the casting of over 350 million primary rays. A debug image of the ray-traced spacecraft is shown in Figure 4.24. The same evaluation is performed for a single wave of two millimeter rays at successively more ray continuations from one to six. The resulting percentage error relative to the high resolution evaluation for each of the force vector components is shown in Figure. 4.25. For the evaluated spacecraft attitude significant error exists when only accounting

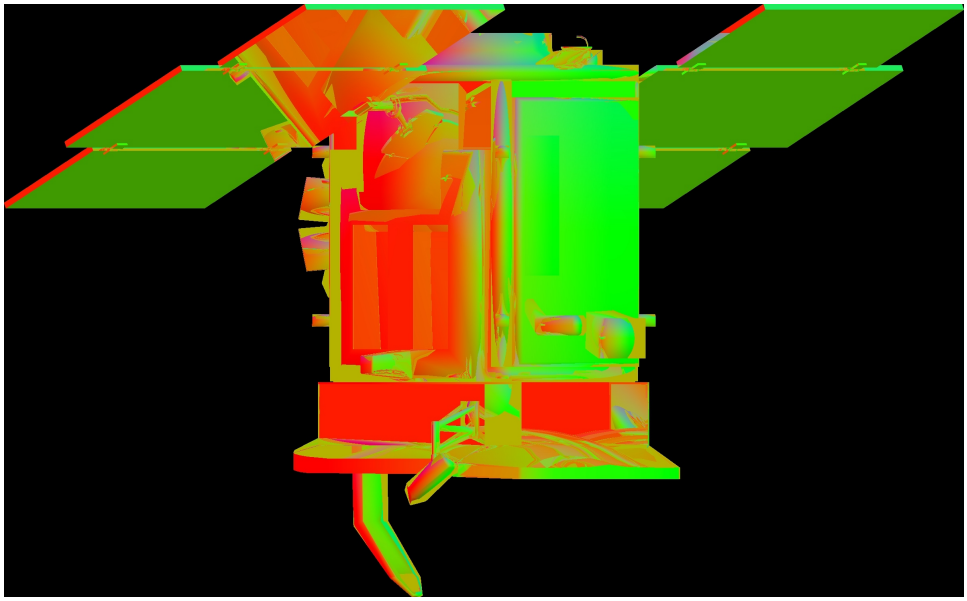


Figure 4.24: Resulting rendered image of the ray-traced CloudSat high resolution evaluation in false color.

for the primary ray intersection. Accounting for spacecraft self-reflection significantly reduces the error in the SRP evaluation with the error remaining at or under 1% for ray continuations of two or more. This result again validates previous work where it is seen that three ray continuations are sufficient to reduce the relative force to acceptable levels [53]. Generally, the number of spacecraft ray self-reflections will vary because it is dependent on the spacecraft material optical properties and the sun-spacecraft heading. However, given that a majority of spacecraft resemble an architypcal box and wing structure, this demonstration is instructive showing that for particular spacecraft attitudes there is significant error in the SRP evaluation when spacecraft self-reflections are not modeled.

4.10 Model Articulation and Detailed Material Properties

The ray-tracing SRP model is integrated into the Basilisk astrodynamics simulation software as a Dynamic Effector module as detailed in Appendix A. To demonstrate the capturing of mesh articulation during simulation the Aqua spacecraft mesh is simulated in an Earth orbit with Basilisk. Without again describing the Aqua spacecraft simulation presented in Section 3.6.2, the simulation

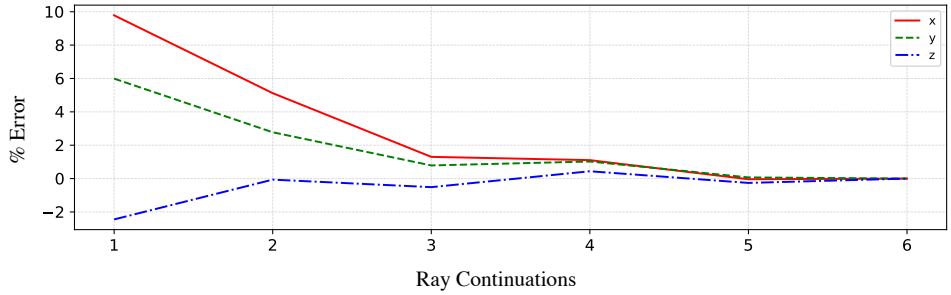


Figure 4.25: Percentage error in the direction of the resultant force between each successive ray bounce relative to a high resolution evaluation.

orbital parameters are provided in Table 4.2 as a reminder.

Table 4.2: Spacecraft orbit parameters for sun-synchronous LEO orbit and GEO orbit.

a km	7378
e	0
i, deg	90
M_0, deg	90
Ω, deg	0
ω, deg	0

The evolution of the SRP force and SRP torque are shown in Figure 4.26 and Figure 4.27, respectively. The eclipse periods are visible where the body frame force components are zero.

4.11 BRDF Effect on Orbit Propagation

Accommodation of complex BRDFs is an advancement of the OpenCL ray tracing method when compared to other SRP modeling approaches. Many SRP models assume ideal specular and diffuse reflection models. However, the total reflection behavior of spacecraft materials can deviate significantly from this ideal. To demonstrate the effect of employing complex BRDFs in the SRP evaluation, the orbit of a cube of side length one meter and mass 1 kg is propagated under the influence of SRP. Two simulations scenarios are developed where the first simulation is a 1000 km altitude sun-synchronous LEO orbit and the second a geosynchronous equatorial orbit (GEO). The simulation orbit parameters are listed in Table 4.3.

For each of the LEO and GEO scenarios, three simulations are executed, where the cube object is assigned a different BRDF model. The three BRDFs are the idealized model given

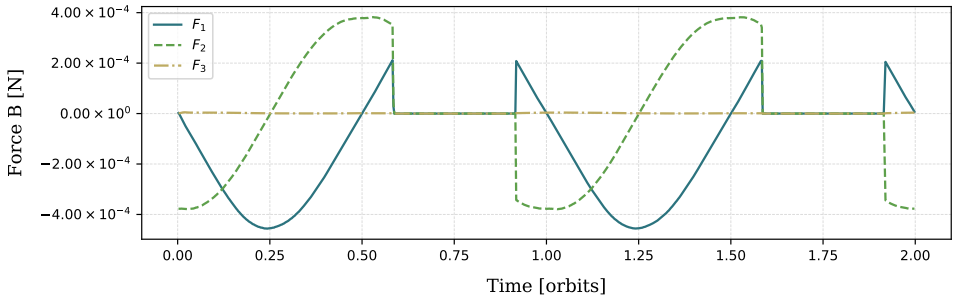


Figure 4.26: Force on Aqua spacecraft mesh in polar LEO

in Eq. (4.24), the Beckmann microfacet model and the GGX microfacet model. Each model is constructed using the same weights for the diffuse and specular contributions with $\rho = s = 0.5$. For the two microfacet BRDFs their roughness parameter (parameter controlling the spread of the specular lobe) is set as $\alpha = 0.45$. The cube in each scenario is given a slow initial body rate to demonstrate the variation in force due to changing incident angle as previously demonstrated by Figure 4.9. The initial rate for the LEO scenario is $w_{B/N} = [0.06, 0, 0]$ deg/s and for the GEO scenario $w_{B/N} = [0.006, 0, 0]$ deg/s.

The magnitude of the acceleration due to SRP for each BRDF model in the LEO and GEO scenarios is shown in Figures 4.28 and 4.29, respectively. In both orbital scenarios the cube body rate can be seen in the variation of the SRP acceleration. The simulation in which the BRDF is set as the Beckmann microfacet consistently has the greatest magnitude. This agrees with the distribution shown in Figure 4.9 where the Beckmann distribution models a greater area of specular microfacets 'seeing' the sun than that of the GGX distribution. Similarly, for incident light angles of greater than approximately $\pi/4$ the magnitude of Beckmann distribution drops sharply and

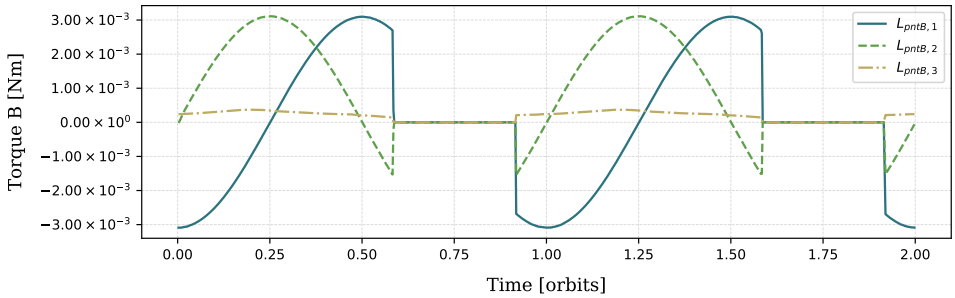


Figure 4.27: Torque on Aqua spacecraft mesh in polar LEO.

Table 4.3: Spacecraft orbit parameters for sun-synchronous LEO orbit and GEO orbit.

	LEO	GEO
A km	7378	42164
e	0	0
i, deg	98	0
M_0, deg	90	90
Ω, deg	0	0
ω, deg	0	0

is almost equal to the GGX distribution magnitude. This is evidenced by the sharp decrease in acceleration in the Beckmann simulation as the cube rotates to briefly orient a cube edge towards the sun bringing the incident light angle on the sides of the cube near 45 deg.

The difference in position, with respect to the simulated orbit of the cube with the idealized BRDF, is plotted for the Beckmann and GGX models. The position differences are given in the radial R, intrack S and cross track W, $[RSW]$ relative position frame. The relative positions for the LEO simulation are displayed in Figure 4.30 and for the GEO simulation in Figure 4.31. The differences in acceleration magnitude of the Beckmann BRDF and GGX BRDF are easily seen in all plots due to the growth in differences in each axis. In the LEO scenario, after a single orbit, radial position differences of a meter are demonstrated. In the GEO scenario radial differences of over a kilometer are shown after less than two orbits. These differences in relative position demonstrate and reinforce the importance of modeling a spacecraft's various BRDFs with appropriate BRDF models.

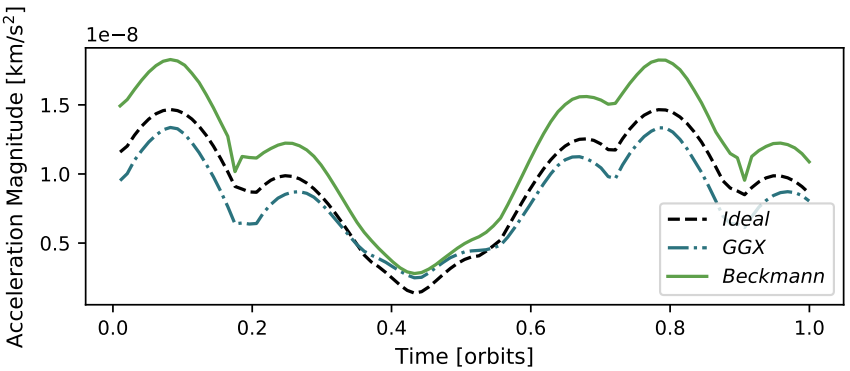


Figure 4.28: Magnitude of the SRP acceleration in sun-synchronous LEO over one orbital period.

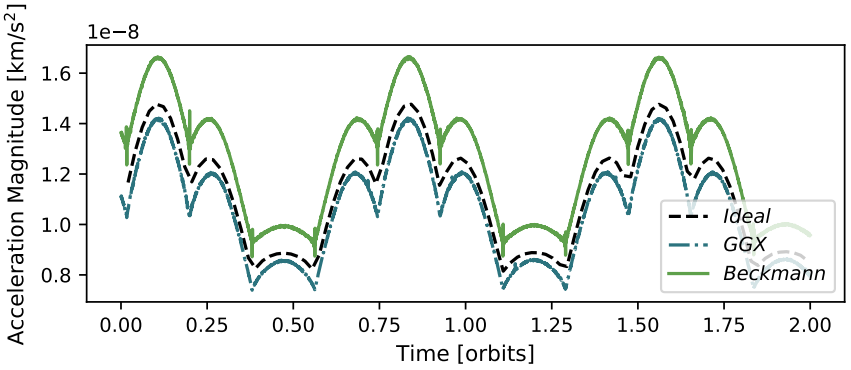
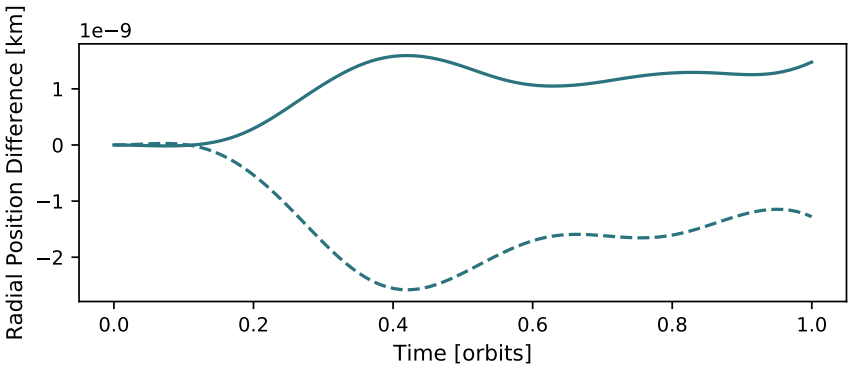


Figure 4.29: Magnitude of the SRP acceleration at GEO over two orbital periods.

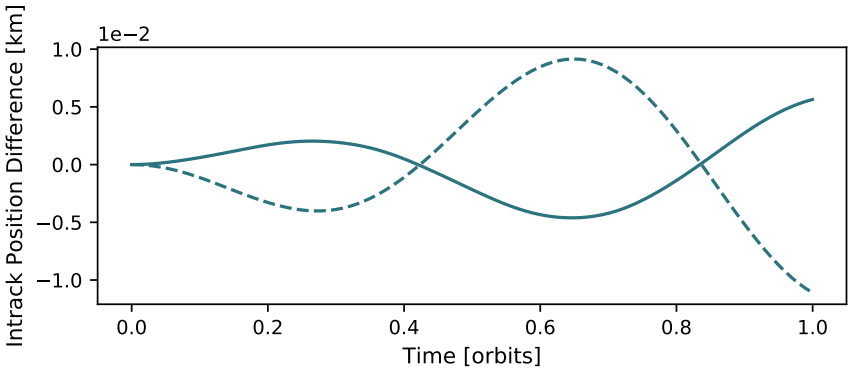
4.12 Computational Performance

An analysis is performed to characterize the execution time of the OpenCl ray tracing approach. The execution time is computed as the time point when ray generation begins to the time point when the CPU-bound process receives the final force value. The computation times for ray resolutions from 1.5 mm to 10 mm, with a maximum of three ray continuations are recorded for an evaluation of the CloudSat model. The computation is executed on three consumer grade GPUs. The first GPU is the AMD Radeon Pro 560 4096 Mb, the second GPU is the integrated Intel HD Graphics 630 1536 Mb and the third is the NVIDIA GTX 1070 8 Gb. The computation times for all hardware options are shown in Figure 4.32 for one ray bounce and Figure 4.33 for three ray bounces. The evaluation using the most capable GPU (NVIDIA) shows execution durations for both one and three bounce case almost unchanged. For ray resolutions of 5mm and above computation times remain below 10 ms. of less than 30 ms for ray resolutions greater than 6 mm (323,000 initial rays). The number of rays to be computed is dependent on the ray resolution, the size of the spacecraft and therefore the area of the sun ray plane to be discretized. Additionally, the number of rays that a GPU can accommodate is also dependent on the GPU's maximum memory buffer allocation.

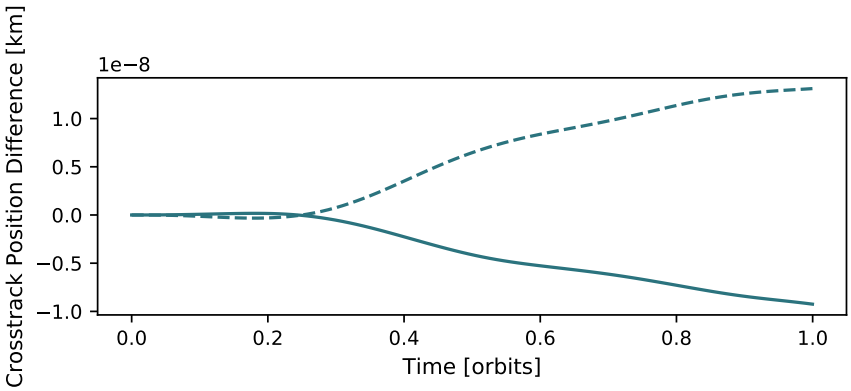
It is instructive to compare the computational performance results of the OpenGL-CL method, presented in Section 3.7, with the performance of the GPU ray-tracing approach. A comparison for both the integrated Intel GPU and discrete AMD GPU show that the OpenGL-CL method is not



(a) Radial position difference from Ideal BRDF.

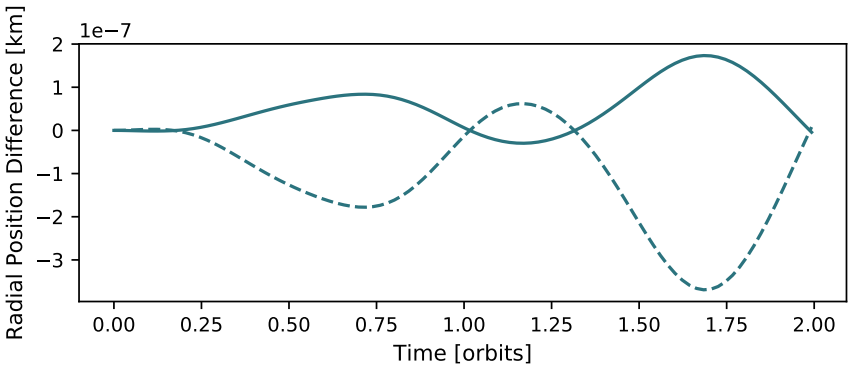


(b) Intrack position difference from Ideal BRDF.

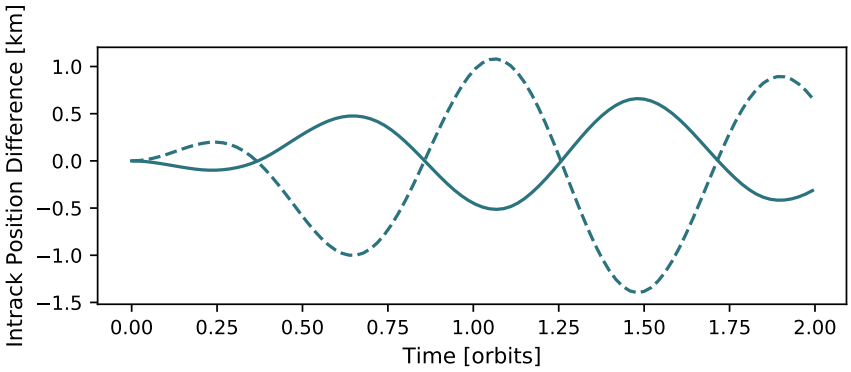


(c) Crosstrack position difference from Ideal BRDF.

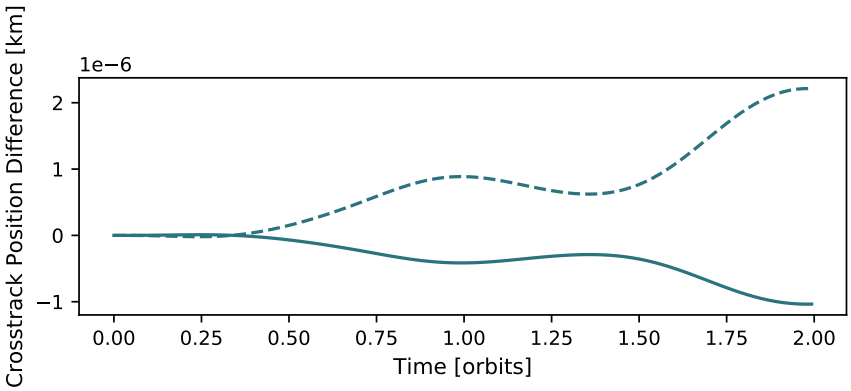
Figure 4.30: Radial, intrack and crosstrack differences w.r.t the position of the simulated Idealized BRDF cube at LEO. The dashed line corresponds to the Beckmann and the solid line the GGX.



(a) Radial position difference from Ideal BRDF.



(b) Intrack position difference from Ideal BRDF.



(c) Crosstrack position difference from Ideal BRDF.

Figure 4.31: Radial, intrack and crosstrack differences w.r.t the position of the simulated Idealized BRDF cube at GEO. The dashed line corresponds to the Beckmann and the solid line the GGX.

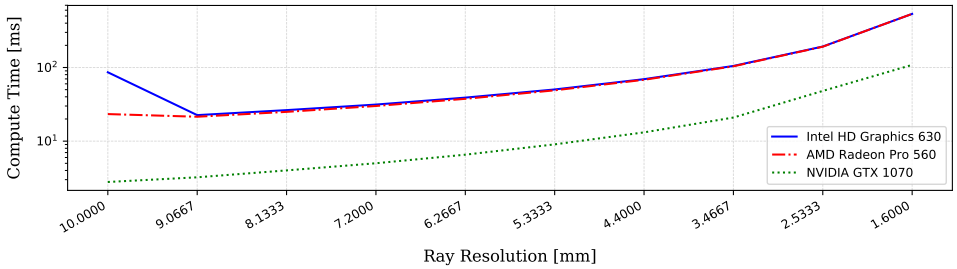


Figure 4.32: Execution times for ray resolutions from 0.01 mm to 0.0016 mm, for one bounce.

computationally bound but rather CPU to GPU communication bound. Moreover the integrated GPU shows much faster total computation times due to its DRAM. However, for the ray tracing approach the integrated GPU and the discrete AMD GPU trace the same performance curve for ray resolutions of less than 6 mm. This demonstrates that the ray-tracing approach is computationally bound rather than communication bound. The time savings introduced by the reduced CPU-GPU latency of the integrated GPU are matched by the increased computational load. Therefore it is clear that the reduced communication overhead of the integrated GPU is matched for by increased computational power of the discrete AMD GPU, resulting in the consistently close computation times.

At the time of this analysis the GPU hardware used is considered to be modest by a computational performance standard. Graphic processing units are commonly available that will significantly outperform the three GPUs used. Given these execution times and the combination of more capable GPU hardware it is simple to see that further reductions in time to solution are cheaply available by consumer grade GPUs.

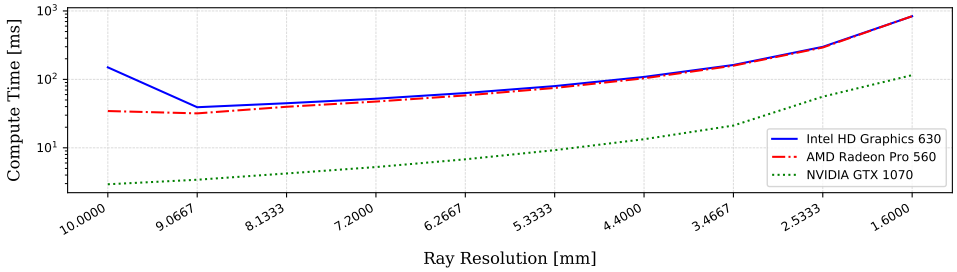


Figure 4.33: Execution times for ray resolutions from 0.01 mm to 0.0016 mm, for a maximum of three bounces.

4.13 Conclusions

This OpenCL ray tracing method demonstrates how SRP forces and torques can be resolved for complex spacecraft structures more accurately and at faster-than-real-time computational speeds. The presented approach leverages significant advances in ray tracing tehcniques and algorithms from the computer graphics discipline. However, this approach is not burdened by the requirement for visual accuracy which is levied upon those in computer graphics. As a result the OpenCl method is able to select appropriate shortcuts to extract further computational time savings.

The GPU imposes constraints on execution order and memory access. The OpenCL GPU-focused ray-tracing methodology accommodates these constraints by converting the serial recursive tracing algorithm to a parallel iterative algorithm. Further, the implementation appropriately uses GPU resources by seeking to maximizes Work Group utilization by relating ray continuation to ray paths rather than a pixel in the ray plane.

Previous ray tracing SRP modeling techniques only approximated diffuse ray reflections. Importance sampling, a technique from the computer graphics discipline, provides efficient numerical evaluation of the scattering equation for complex spacecraft surface material BRDFs. The importance sampling technique is critical to enabling this ray-tracing implementation to maintain faster than realtime computation speeds. Capturing the effect of both diffuse and spectacularly reflected rays is validated by comparison to the same evaluation computed with the faceted SRP evaluation technique. Additionally, it is shown that with increasing ray density the resultant force vector converges towards a 'truth' evaluation.

Multiple ray continuations are validated for both simple cube mesh models and the complex OSIRIS-REx and CloudSat mesh models. The importance of capturing spacecraft radiation self-reflections is demonstrated by the evaluation of the OSIRIS-REx model throughout its full sphere of attitude possibilities. For a range of typical spacecraft geometries, it is shown that computing up to three ray-surface interactions significantly reduces the force and torque error present when

not computing spacecraft self-reflections.

Chapter 5

Black Lion Distributed Simulation

The second major contribution presented in this dissertation is the design and implementation of the simulation communication middleware called Black Lion (BL). The BL architecture facilitates the integration and execution of multiple software processes across heterogeneous computing platforms. Black Lion enables simulation use cases beyond that of the typical single application on a single machine. For example, having a dedicated computer running a complex space environment model and another computer integrating spacecraft dynamics, both of them exchanging data dynamically through BL.

Black Lion is a communication architecture designed to enable a distributed software-simulation (SW-sim) of a spacecraft system. The BL middleware was originally motivated by a need for a SW-sim to support flat-sat testing for an ongoing interplanetary mission in which the Laboratory for Atmospheric and Space Physics (LASP) and the Autonomous Vehicle Systems (AVS) laboratory at the University of Colorado Boulder are collaborating. The goal of a SW-sim is to provide a comprehensive simulation test-bed that is purely software-based and therefore quickly deployable and scalable to a large number of users and concurrent spacecraft test activities.

This chapter describes the software architecture which enables the novel BL distributed simulation software. A discussion of the design consideration for spacecraft distributed simulation architectures is given. The key development, of separating the data and transport layers is defined and it's impact on the run-time operations described. Finally, an example distributed simulation, using the ray tracing model, is presented.

5.1 Distributed Spacecraft Simulation Architectures

The BL architecture facilitates the integration of multiple software processes across heterogeneous computing platforms, thus enabling simulation use cases beyond a single machine and tightly integrated sets of models. For example BL allows for a multi machine simulation configuration where a dedicated computer runs a complex space environment model, another computer integrates the spacecraft dynamics, while both of machines exchange data dynamically through BL. Further, in the context of a mission, the BL architecture can be applied to bridge the gap between multiple legacy software components that were never designed to work together. The primary demonstrated use of BL in this work is as a distributed simulation middleware enabling the evaluation of computationally expensive models on latent available commodity compute resources (e.g. a separate machine with significant GPU resource). Specifically, it is shown how the BL system facilitates the execution of a high-fidelity SRP ray-tracing evaluation on a remote computer resource while the primary dynamics simulation is executed on another computer resource.

Considering a hardware flat-sat testing scenario, there are multiple mission components interacting with each other: the ground system (GS), the flight computer (FC) and its flight software (FSW) algorithms, as well as spacecraft models to simulate the dynamics, kinematics and space environment (DKE). Software simulation testing uses virtual models or emulators in place of physical assets. Figure 5.1 depicts the idea behind the virtualization of the GS, the FC and the spacecraft's DKE. As illustrated, the GS emulator ingests the same mission scripts as the real ground system and contains the same command/telemetry databases, while the FC emulator runs the actual mission FSW.

A spacecraft flat-sat SW-sim configuration provides the capability to initialize and run the operational command and telemetry databases, as well as the unmodified FSW executable. Independent software models are used for the GS and the FC. While for other required hardware components like sensors, actuators and avionics high-fidelity DKE models are integrated in order to test the flight software system in realistic closed-loop simulations.

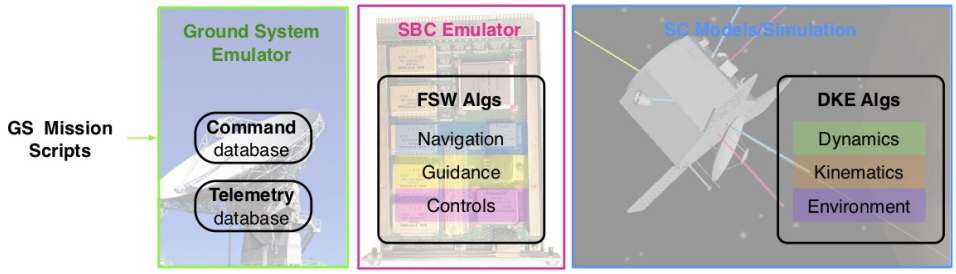


Figure 5.1: Virtualization of Spaceflight Components.

The virtual models used in a SW-sim are usually pre-existent resources for the particular mission being assessed. These virtual components, referred to as 'nodes' from this point onward, are stand-alone heterogeneous applications. The applications are heterogeneous due to the fact that they are written in different programming languages, maintain different internal data representations and potentially have a variety of execution run-loop strategies (execution speeds, fixed time, multi-threaded etc.). Therefore, a common communication layer is required to synchronize and facilitate the exchange of data between the multiple nodes.

The BL communication architecture is designed to connect all the nodes of a distributed simulation while being as transparent as possible to the internals of these nodes, such that different mission users can plug-and-play virtual models. Examples of further BL applications include the integration of large clusters of spacecraft, complex simulation components running on supercomputers or cloud servers, as well as distributed simulation of both spacecraft sensor and actuation systems.

5.2 Black Lion Architecture

The purpose of BL is to achieve the described communication goals while being completely transparent to the internals of each node. Recall that the nodes in the example SW-sim are stand-alone applications that are initially unaware of any other nodes. The heterogeneity between the multiple components drives the need for a dedicated communication middleware. The term communication, involves multiple goals:

(1) **Transport** of binary data between nodes.

(2) **Serialization** of binary data because each node must know how to convert the received bytes into structures that can then be managed internally.

(3) **Synchronization** of nodes to keep all the nodes in lock-step during simulation execution.

(4) **Dynamicity** in the connections map to allow a more flexible simulation environment that is minimally dependent on static components (the less strictly required (static) components in a network the greater the flexibility and resistance to faulting) [44].

A communication layer that is transparent and abstracted from the nodes allows simulation users to plug-and-play their models of choice, while having the flexibility to add and remove components at will. With this purpose in mind, the BL architecture is notionally depicted in Figure 5.2 showing the central controller and three nodes. The BL system's implementation is comprised of a single central controller and two APIs that are attached to each node. Each of the components depicted in Figure. 5.2 will be described in subsequent sections of this chapter.

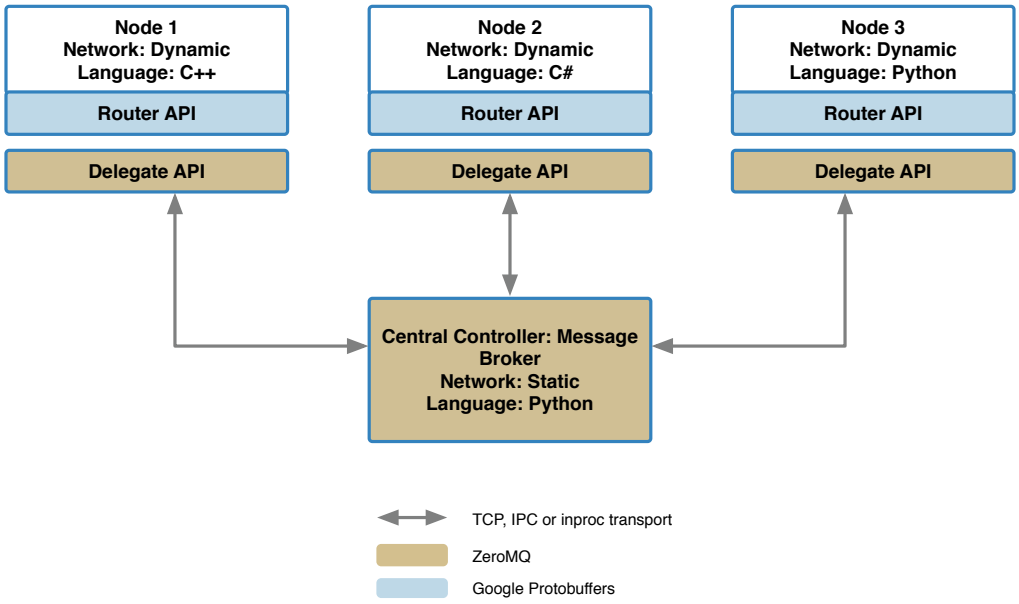


Figure 5.2: Communication Architecture: Central Controller, Delegate APIs and Router APIs.

In Figure 5.2 the coloration of BL components indicates the integration of two third-party software libraries: ZeroMQ[1] : ZeroMQ is a high-performance asynchronous messaging library. The

[1] `http://zeromq.org`

library facilitates fast, reliable and protocol independent inter-process messaging for distributed or concurrent applications. ZeroMQ provides message queuing, however, it aims for *decentralized intelligence*, where application components incorporating ZeroMQ are inherently aware of any network intermediation. Here, intermediation means those networking components between nodes such as proxies, queues, forwarders or brokers, depending on the context [44]. The ZeroMQ library is available in a wide range of programming languages, which can perfectly interact with each other.

ZeroMQ provides a range of communication topology abstractions. These abstractions define endpoints and connection points within a network and are referred to as sockets. While referred to and utilized with similar processes of connect, bind, send and receive, the underlying communication protocol is transparent to the layers above and can be one of inter-thread communication, Inter-process communication (IPC), Transmission Control Protocol (TCP), Transparent Inter Process Communication (TIPC) or Pragmatic General Multicast over IP (PGM).

Google Protobuffer[2] : Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. It is important to clarify that, while ZeroMQ is used for all transport of binary data through BL, Google Protobuffers are not used for un/serialization of binary data in all simulation nodes. For instance, Protobuffers prove extremely useful for un/serializing binary data that is shared between the BSK Modules. In contrast, for FSW or the GS models, the protobuffer libraries are not used because it does not replicate how these components operate in flight-like configuration.

5.2.1 Data Transport and Data Translation Layers

A key aspect of the BL architecture is to employ a separation of the data transport layer from the data translation layer. This technique can be formalized by referring to the Open Systems Interconnection (OSI) model. The OSI model is a conceptual model, developed by 'International Organization of Standardization' (ISO) [89]. The model characterizes and standardizes the communication functions of a computing system without reference to the system's internal structure

[2] `https://developers.google.com/protocol-buffers/`

and technology. The seven layers in the model are shown in Figure 5.3. Each layer characterize the specific functions to support the layers above and the services offered to the layers below [20]. The four lowest layers focus on passing traffic through a network to an end system. The top three layers act as translation layers by interfacing data input and output from the lower four layers. Black Lion groups the operations of the top three OSI layers into a set of functionality which is defined by the Router API. The Router API performs the functions characterized by layers one to three of the OSI model by packaging, addressing and routing data that each simulation node ingests and publishes. Its purpose is to route data in and out of the internals of a simulation node. For instance, when routing out, the Router API gathers the node internal data, translates the data into a standardized BL system format, and then passes the data to the node's Delegate API. The Router API is implemented as a generic class with node-specific callbacks. The node-specific callbacks allow simulation developers to write the minimal custom code required to package and route the node specific data structures, ready to be sent to other nodes throughout a simulation.

The Delegate API groups the operations of the lower four layers of the OSI model to manage each node's communication connections with the central controller. The Delegate API utilizes the ZeroMQ library to provide a 'neutral' transport. This means that ZeroMQ transports data without regard to the underlying data structure. ZeroMQ transports data between nodes as messages. The ZeroMQ defined message data structure (`msg_t`) takes any node data as a byte stream as a variable length payload and transmits this data through the BL system. The `Delegate` API is implemented once for each desired programming language and then compiled or executed with the node application. The Delegate class is currently implemented for Python nodes, C++ nodes and C# nodes.

Both Figures 5.2 and 5.3 intentionally use the same coloration to show the separation of data translation and data transport. The ZeroMQ library is used to support data transport while Google ProtoBuffers are used to support data translation. The separation of translation and transport is the key development which leads to the inherent flexibility of BL's multi-process and multi-machine capability.

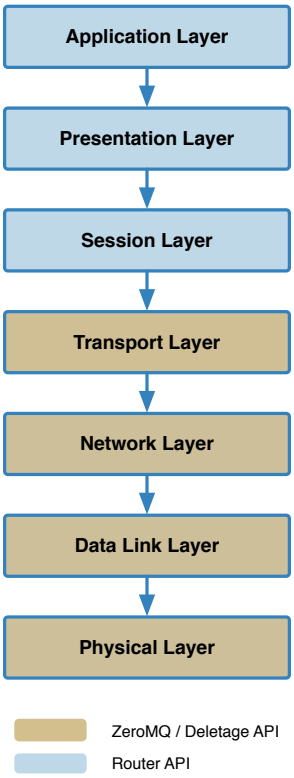


Figure 5.3: Relating the primary BL architectural layers to the OSI stack.

5.2.2 Black Lion Simulation Topology

A network topology in which there are fewer static pieces is more robust and flexible [44]. It is more robust because the networked system does not explicitly rely on particular nodes being operational to initialize and run. Further, if a data dependency is not satisfied from one node to another then it is expected that the node suffering from insufficient input accommodates this faulted state and continues to operate accordingly. Not only does this design choice support the development of a robust flexible system, it also more accurately mimics the simulated reality. An example of this mimicked reality is a spacecraft not receiving an updated schedule due to missing a ground station communication pass. The spacecraft therefore waits in a sun-point mode until the next communication pass opportunity. Similarly, consider a simulation configured with a Ground Station node that simulates the transmission of spacecraft activity schedules to the spacecraft. In a distributed simulation, if the Ground Station node faults and ends execution, the simulated spacecraft flight software shall take the same action as it would on orbit and switch the spacecraft to a sun-point mode and wait until a schedule or commands are received.

To achieve this flexible network topology the BL system instantiates the BL central controller as the only static piece in the network (i.e. it has a static IP address and is required for a simulation to execute). Notionally, depicted in Figure 5.2, the central controller acts as a 'Broker' which synchronizes the execution of nodes and brokers exchange of data between nodes.

5.2.3 Socket and Connection Definitions

In order to understand how the central controller operates as a broker, it is necessary to first explain the ZeroMQ socket and connection types used in Black Lion's implementation. Two types of ZeroMQ socket patterns are used to transport data: the request-reply pattern and the publish-subscribe pattern. The publish-subscribe pattern is applied in two different configurations.

Request (REQ) - Reply (REP): the central controller has a REQ socket for each node instantiated in the simulation. The REQ socket is used to make requests of each simulation node.

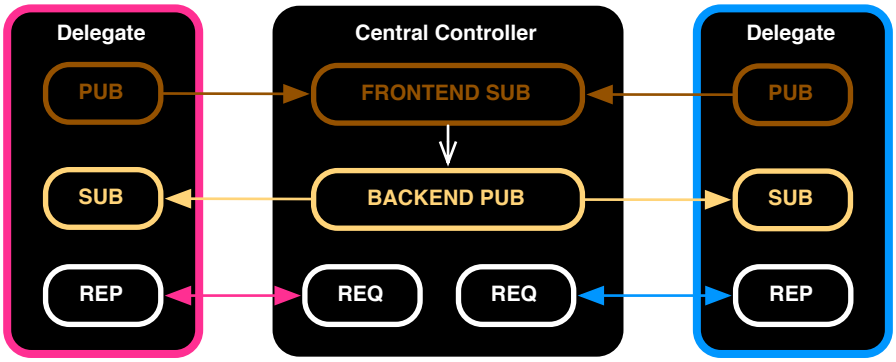


Figure 5.4: Socket Patterns between the Central Controller and Sample Nodes.

In turn, each node has a REP socket that receives and parses the request, performs the commanded task, and replies back to the central controller indicating accomplishment.

Publish (PUB) - Subscribe (FRONTEND SUB): Every node has a PUB socket through which it shares its own internal data by publishing. In turn, the central controller has a SUB-frontend with a SUB socket that subscribes to the publications from all nodes.

Publish (BACKEND PUB) - Subscribe (SUB): Additionally, the central controller has a PUB-backend. The messages received at the SUB-frontend are routed through the controller to the PUB-backend socket, which then re-publishes the data. In turn, each node has a SUB socket that subscribes to the messages of interest coming from the controller's PUB-backend publisher socket.

The relationship between sockets just described is exemplified in Figure 5.4. The figure depicts the central controller in the middle and two sample nodes highlighted in magenta and blue. As shown in Figure 5.4, the sockets are encapsulated by the Delegate API.

Now that the socket types are defined, the connections of these sockets to a given IP address and port is discussed. All the socket connections in the system fall into either one of these categories: static connection (i.e. **binding** type in ZeroMQ terms) or dynamic type (i.e. **connecting** type in ZeroMQ terms). The static connections are all associated to sockets in the central controller, while the dynamic connections are associated to the sockets in each of the nodes' `Delegate` API.

Central Controller: it is the only static component in the network due to the frontend-backend

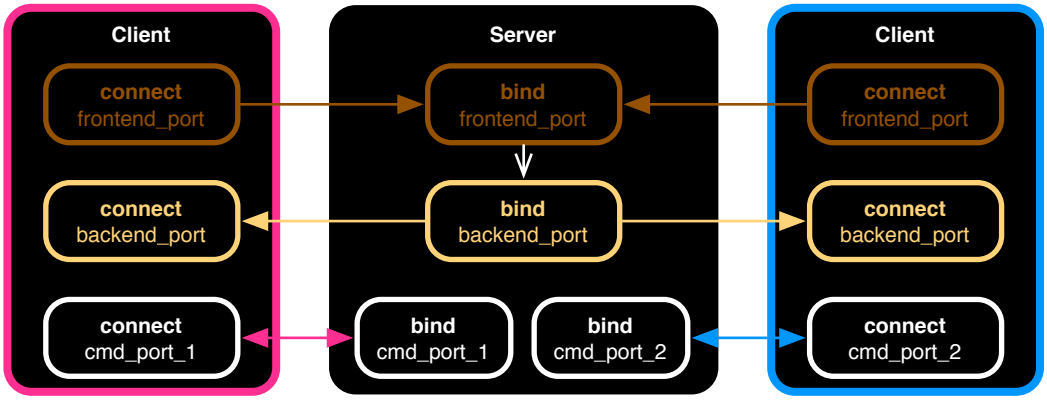


Figure 5.5: Socket Connections Types (Binding vs. Connecting) and Ports.

(broker) approach. The controller acts as a server in the sense that it **binds** to a static IP address for the duration of a simulation instance. Binding to an address creates a fixed point in the network to which nodes connect. With the same address, it uses a total of $(2+N)$ ports, where N is the number of nodes instantiated: One port for the SUB-frontend, one port for the PUB-backend, and a command port for each of the node-request sockets.

Nodes' Delegate API: through the `Delegate` API attached to each one of the nodes, the nodes become dynamic such that they can join and leave without ongoing operation of the rest of the simulation system. This dynamic nature is reflected in the fact that nodes **connect** only to an address and port, rather than bind. As nodes **connect** to the controller, ZeroMQ creates a message queues for each node connected to the bound sockets.

Through the described strategy, the central controller is always required and the nodes are optional independent entities that do not intrinsically rely on each other. The use of ZeroMQ also allows all the connections to be protocol independent (TCP, IPC, inproc, etc.). The idea of socket binding (static nature) versus socket connecting (dynamic nature) is illustrated in the topology showcased in Figure 5.5. The figure also reflects the fact that there is only one static IP address in the entire BL system and within this address multiple ports are associated with that IP address. As before, the figure displays the central controller router in the middle and two sample nodes (clients) highlighted in magenta and blue colors.

5.3 Communication Between Nodes

Requests and commands from the controller are handled by the Delegate API built in to each node. These requests are not spacecraft or simulation commands, but rather they are communication and synchronization commands exclusive to the BL middleware. The BL controller issues five different requests, some of which come in the form of multi-part messages. Figure 5.6 shows the complete BL runtime, for a controller and two nodes, as a Unified Model Language (UML) Sequence diagram [10]. This diagram shows synchronous commands as arrows with complete arrow heads, asynchronous commands as half arrow heads. The vertical blocks represent an executing process while the dotted line for each process (controller, node 1 and node n) indicates the lifetime of the process.

To begin BL simulation initialization, an **"Initialize" request** is issued to each node. The **"Initialize" request** is a multi-part message comprised of the "Initialize" signal, the controller's frontend address and port and the controller's backend address and port. The actions taken by the requested node are: self initialization, connect its pub-socket to the controller's frontend and connect its sub-socket to the controller's backend.

The **"pollSubscribedMessages" request**, is then sent, which instructs each node to report all the message names to which the node wishes to subscribe. Next, the **"pollPublishedMessages" request** is sent which is a multi-part message containing the "Match" signal and a list of all the message names for which the other nodes have asked. As a response each node returns a reduced list with only the message names for which the node shall publish. With initialization now complete, the main network run-loop begins. Each loop begins with a **"Tick" request**, which is used at every time-step of a simulation to synchronize the start of each node's message exchanging. This request contains the time duration of the next time-step (i.e. Δt). Once the requested node has accomplished all the tasks that must happen after a "Tick", it sends back a "Tock" reply. Eventually, there is the **"Finish" request**, which is a signal for the node to close the sockets, clean up and shut down.

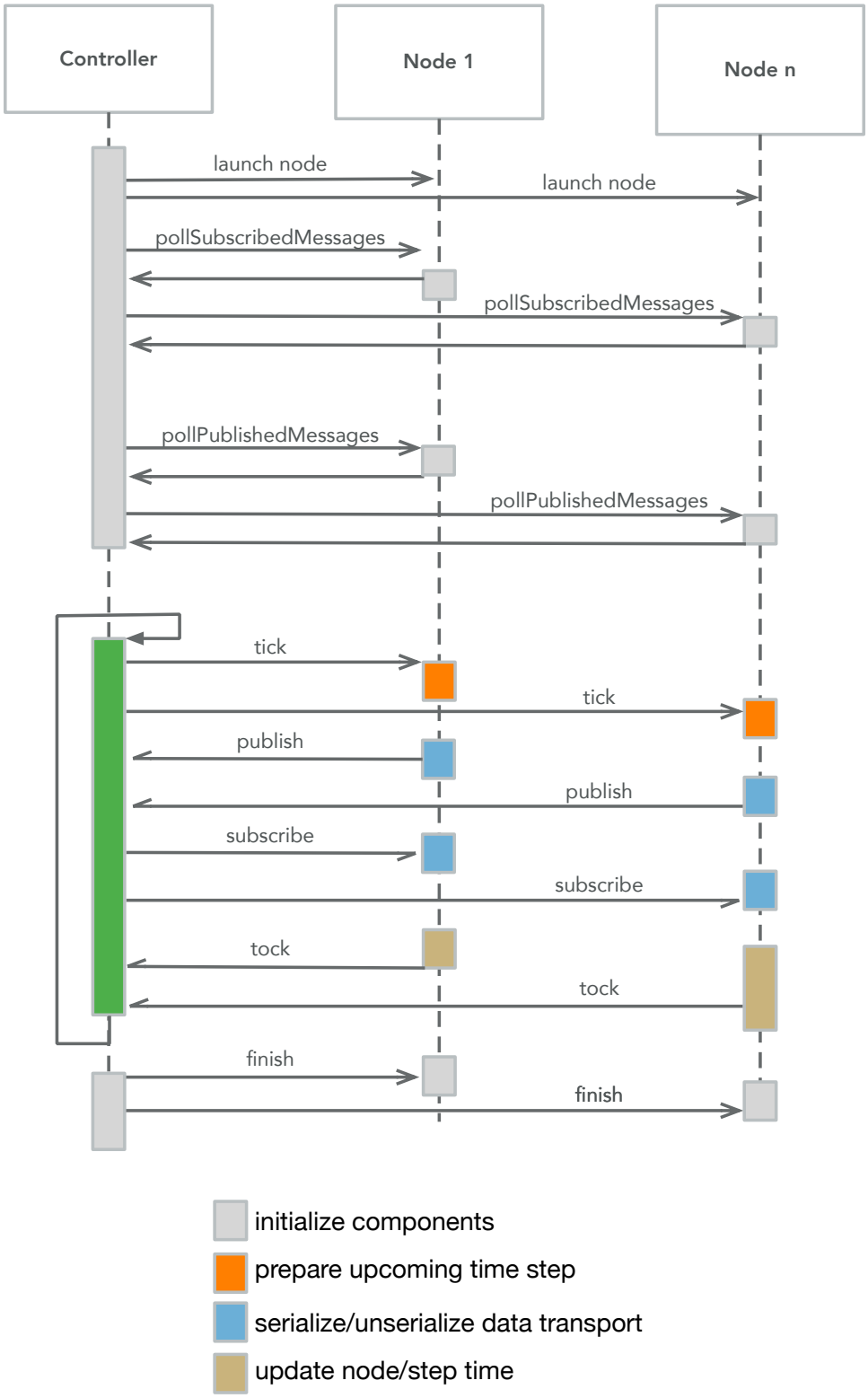


Figure 5.6: Node Actions between a "Tick-Tock": Publish, Subscribe, Step Simulation.

5.4 Tick-Tock Synchronization

Three actions occur in sequence inside each node between the parsing of a "Tick"-request and the sending of a "Tock"-reply: **publish**, **subscribe** and **step simulation** forward. Note that these three actions are node internal calls triggered by the "Tick" request. Figure 5.6 depicts the sequence of interactions and actions happening between the central controller and two sample nodes.

Parse Tick Request: Each node is alerted to prepare for a new simulation single execution step.

Publish: In the *publish* internal call, the node's Router collects the application internal data and makes it available to the node's Delegate for publication to the controller's frontend.

Subscribe: In the *subscribe* internal call, the node's Delegate receives external data coming from the controller's backend and passes the data to the node's Router. The Router is responsible for writing these messages down into the internals of the node application.

Step Simulation: In general terms, the *step simulation* internal call implies executing the node's application during Δt in order to generate new data, where Δt is a message part of the "Tick" request sent by the controller. However, there are nuances in the precise meaning of *step simulation* for nodes that are synchronous (i.e. run in cycles, like FSW or the spacecraft's DKE simulation) and for nodes that are asynchronous (i.e. are event-based, like the GS).

Because each node is an independent process that runs at a different speed, the "Tick-Tock" signal ensures that all of them are kept in lock-step. In the previously introduced example SW-sim the four nodes integrated into a cooperative BL simulation are: the FC emulator, the spacecraft DKE simulation, the GS emulator and the visualization GUI. Figure 5.7 depicts the synchronous nature of the FC and DKE simulation nodes, the asynchronous nature of the GS emulator, and the listener nature of the visualization node.

Both the FC node and the SC simulation node are synchronous in nature as they run in

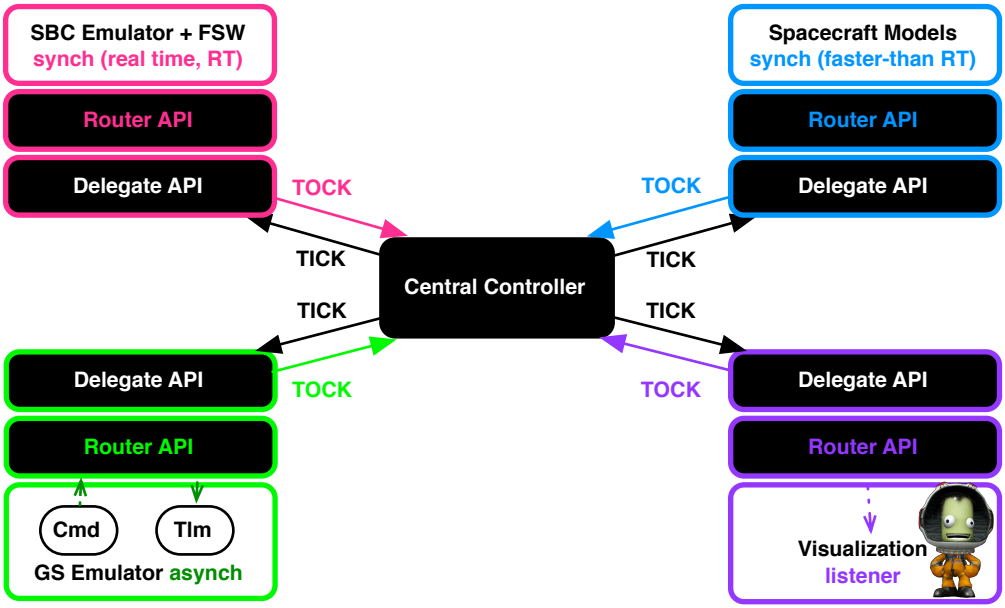


Figure 5.7: Nodes' Timely Nature: Synchronous, Asynchronous and Listener Behaviors.

cycles or at predefined rates. Because the FSW executable is running inside a FC emulator using a real-time operating system, the speed of the FSW execution is real time. In contrast, the DKE simulation runs faster than real time but contains the notion of a time step, as is required for numerical integration. For the synchronous nodes, the *step simulation* call implies running as many cycles as there are within Δt before exchanging data again with the rest of the system. If one node finishes simulating Δt earlier, it sends the "Tock" reply indicating its completion and awaits a new "Tick" request from the controller. Because the controller will not proceed until it has received all the "Tock" signals from all the nodes, the SW-sim speed is naturally driven by the slowest component.

In contrast, the GS node is asynchronous: the sending of spacecraft commands and the receiving of telemetry are discrete-time events. The GS also receives a "Tick" command because the exchange of data (i.e. the publishing and subscribing) must still happen in parallel among all the nodes. The asynchronous nature of a node like the GS demands a special treatment of the Delegate and Router APIs: the communication through the APIs must execute in a different thread from which the main application is running. The Visualization node is another case on its

own: it can be simply regarded as a "listener" governed by the DKE simulation. Therefore, it only subscribes to the DKE simulation messages and, within the *step simulation* call, it displays the spacecraft time evolution according to the received set of telemetry messages.

5.5 Black Lion Simulation Case Study

The BL system facilitates the distribution of a simulation across multiple compute resources. A BL simulation can be comprised of multiple nodes executing on a single machine or multiple nodes executing across a range of compute resources. Compute resources may be a single computer with multiple high performance GPUs or remote GPU or CPU hardware administered as 'cloud' based compute service.

This section presents a Basilisk simulation of an Earth orbiting spacecraft carried out as a BL orchestrated simulation. The BL configuration includes two nodes which are executed on two separate computers. The first computer is a consumer grade laptop while the second computer has installed a higher performance GPU (NVIDIA GTX 1070). This GPU hardware on the second computer will be used to support the ray traced SRP force and torque evaluation of the spacecraft mesh. The simulation configuration is described and the closed loop result of the spacecraft's simulated dynamic response is presented.

5.5.1 Basilisk Simulation Configuration

As shown in Figure 5.8, two Basilisk simulation nodes are configured. The first simulation node, `node1`, is a Basilisk simulation which is comprised of ten modules. Each module can be categorized as either a dynamics, environment or flight software module. The `SpacecraftPlus` module models the spacecraft rigid body hub (bus) and through the `DynamicsManager` facilitates the propagtion of dynamic contributions from the `ReactionWheels` state effector module and the `Gravity` dynamics effector module. The Sun and Earth are the two gravity bodies configured within the `Gravity` module. The `SPICE` module is inlcuded to provide ephemeris information and it is configured to output the states for the Sun and Earth bodies. The simulation SPICE coordinate

system is configured to have an Earth centered zero base (reference frame zero point). The `Eclipse` module is included to provide eclipse information based on the spacecraft's position. The `Eclipse` module computes a shadowing scaling factor based on apparent disc sizes of the Sun and eclipsing body for transition through umbra and penumbra[38]. The `SimpleNav` module is employed to generate actual navigation system data. It provides the ability to configure and generate realistic navigation transnational and rotational states. The module, applies provides specified navigation errors to the spacecraft true state so that the functionality of the guidance and control subsystem can be verified as acceptable in the presence of navigation errors. For this simulation the navigation states are not perturbed. A set of four FSW modules provide the cascading of algorithms required to produce the guidance and control subsystem. The modules are configured in a cascaded manner where the output messages from one module are the input messages to the next module. The first FSW module is the `HillPoint` module. The `HillPoint` module compute the spacecraft reference attitude as the current orbital Hill reference frame [75]. Next, is the `AttitudeError` module which computes the spacecraft attitude error from the reference attitude. The `AttitudeError` module takes as input the perturbed states generated by the `SimpleNav` module and the reference attitude from the `HillPoint` module. Subsequently, the `MRPControl` module ingests the attitude error message output by the `AttitudeError` moudel and computes required control torques. The module implements a MRP feedback algorithm as detailed in Example 8.16 of Reference[76]. This nonlinear attitude tracking control includes an integral measure of the attitude error. Further, the algorithm seeks to avoid quadratic angular rate terms thus reducing the likelihood of control saturation during a detumbling phase. Finally, the `RWMotorTorque` module maps the computed control torque onto the wheel spin axes of a set of reaction wheels and generates torque commands for each reaction wheel.

The second simulation, `node2`, is simply the Basilisk OpenCL ray tracing SRP module. The module transparently receives all required input data via the BL system. To compute the SRP force and torque the module requires information regarding the spacecraft state, the eclipse scale factor and the Sun's ephemeris data. For this particular simulation configuration the BL system

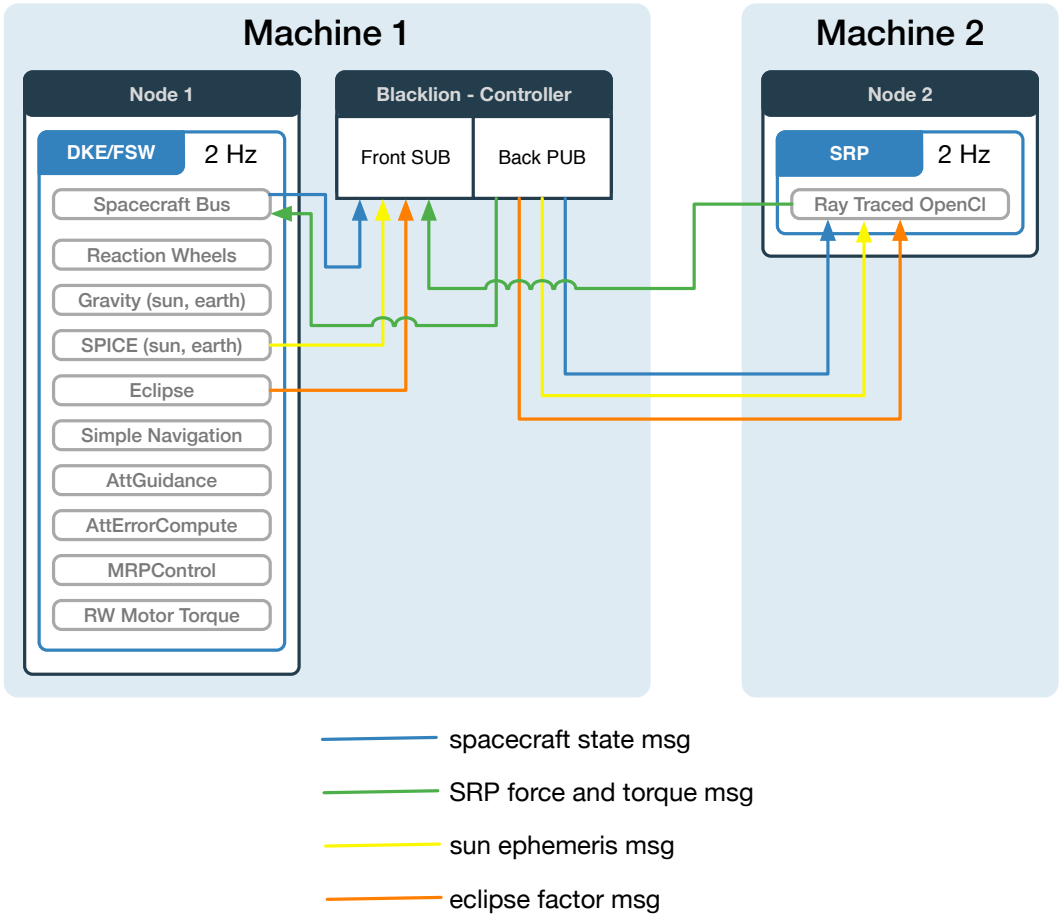
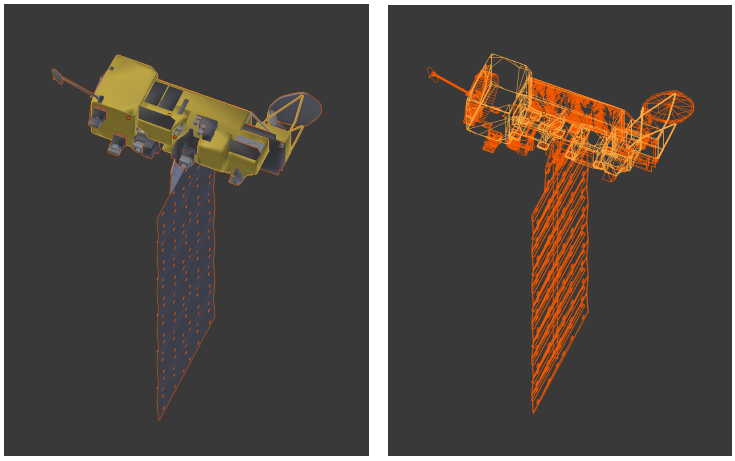


Figure 5.8: Blacklion configured Baislisk simulation for an Earth orbiting scenario.



(a) Aqua spacecraft with materials.

(b) Aqua spacecraft mesh.

Figure 5.9: Aqua spacecraft .OBJ mesh model.

transports the four messages shown in Figure 5.8. The fourth message, published by the ray traced OpenCL module in `node2`, is contains the computed SRP force and torque vectors.

The simulated spacecraft is the Aqua spacecraft mesh shown in Figure 5.9. The mesh material properties are given in Table 5.1. The simulation orbital parameters are shown in Table 5.2.

5.5.2 Simulation Results

The force vector in body frame components is shown in Figure 5.10. The eclipse periods are evident where all force and torque are components zero. To confirm the BL simulation's closed loop response is correct one can compare the BL simulation force and torque with that in Section 4.10. Comparing Figure 5.10 with Figure 4.26 for force and Figure 5.11 with Figure 4.27 for torque demonstrates that the BL simulation across multiple machines produces the same result as the same simulation run on only a single machine.

The benefit of the BL middleware is evident when comparing the the simulation duration when executed with BL configuration and without. On a single machine the simulation duration approximately 15 minutes. The single machine simulation uses the laptop's onboard discrete GPU (AMD Radeon Pro 560 4096 Mb). The ray tracing SRP model evaluates the large Aqua mesh at a ray resolution of 5 mm for two ray bounces. The simulation time step is 5 seconds. A single

Table 5.1: Spacecraft sub-mesh material optical parameters.

Material	Specular (ρ_s)	Diffuse (ρ_d)
Gold MLI	0.184	0.736
Silver MLI	0.66	0.16
Germanium MLI	0.3	0.3
Solar array rear	0.1	0.3
Solar array front	0.023	0.092
Solar array boom	0.3	0.3

time step SRP evaluation of the Aqua spacecraft mesh can take upto 150 ms. The BL simulation performs notably better with a simulation duration of approximately 4 minutes. The more capable GPU computes a single SRP time step within 30 ms. The round trip network communication latency of the BL system is on average 0.3 ms on a local Ethernet network. It is clear that the benefit of utilizing latent GPU computing power on an idle laboratory computer vastly outweighs any time penalty incurred by network data transport latency.

This case study demonstrates how BL enables multiple machine simulation without significant changes to the applications running. The time evolution of SRP force and torque from the closed loop BL simulation demonstrates agreement with the same simulation run on a single machine without BL. As a result the BL software facilitates significant reductions in time to solution by allowing one to easily segment a Basilisk simulation into two Basilisk simulation scenarios. This separation of simulation nodes can be applied generally whether the nodes are basilisk simulations or the heterogeneous applications of a SW-sim.

Table 5.2: Aqua spacecraft orbit parameters for polar LEO orbit

a	7378 [km]
e	0
i	90 [deg]
M_0	90 [deg]
Ω	0 [deg]
ω	0 [deg]

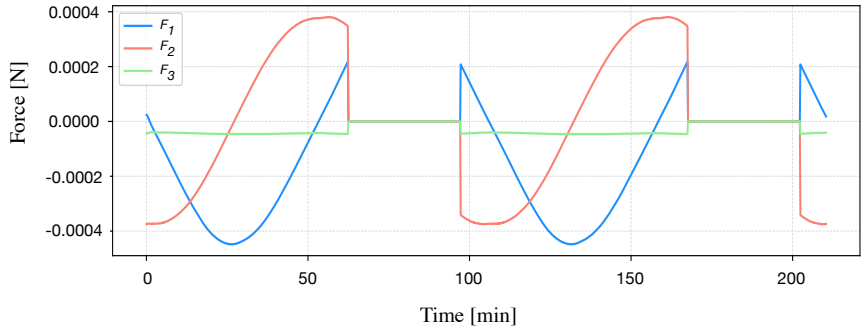


Figure 5.10: SRP Force on Aqua spacecraft mesh in polar LEO

5.6 Conclusions

Black Lion's distributed nature makes it suitable to a wide range of simulation configurations. Black Lion is currently supporting flat-sat testing in a purely software environment for an ongoing interplanetary mission. Yet its inherently distributed architecture facilitates application to spacecraft numerical simulation where particularly computationally intensive simulation models can be run on more capable compute resources.

The key architectural design decision is the separation of the data translation from the data transport layers. This separation enables the development of two generic APIs, the Router API and the Delegate API. Each API abstracts the operations required for a node to participate in a BL simulation. Additionally, by providing two separate APIs for multiple programming languages (currently C, C++ and Python) the amount of work required to retrofit an application such that it may participate as a BL node is significantly reduced.

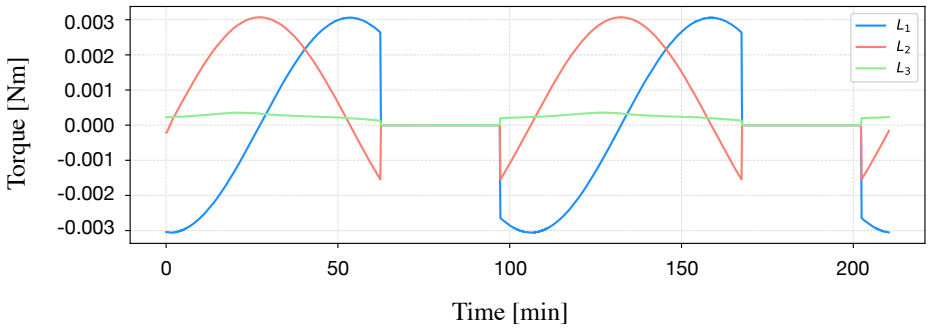


Figure 5.11: SRP Torque on Aqua spacecraft mesh in polar LEO

Chapter 6

Case Studies

Each of the developed SRP modeling approaches enables new analysis opportunities through online faster than real time simulation. With minimal configuration the ray tracing model makes use of existing pre-launch engineering data such as detailed spacecraft shape models, spacecraft articulation and complex BRDF material properties. Direct manipulation of actual spacecraft physical parameters provides engineers with greater insight when seeking to investigate differences between a computed force model, state estimation and tracking data. Further, the fast evaluation capability of both methods offers new opportunities for long term orbit propagation of objects comprised of a realistic shape model and material optical properties.

This section presents two case studies, each of which demonstrates an enabling quality of the developed SRP modeling approaches. The first case study shows how both the ray tracing and OpenGL-CL SRP models are integrated as complimentary force models for the ongoing Origins Spectral Interpretation Resource Identification Security Regolith Explorer (OSIRIS-REx) asteroid sample retrieval mission. Both modeling approaches are used to provide high-fidelity SRP force modeling data as input to the spacecraft Orbit Determination (OD) campaigns.

The second case study demonstrates the ray tracing model's ability to capture arbitrary mesh detail therefore improving force and torque resolution. This case study investigates the efficacy of modeling a high area-to-mass (HAMR) multi-layer insulation (MLI) object as a flat plate or as a sheet with realistic wrinkles and folds. It is shown, through a Basilisk simulation, that with improved mesh resolution the propagation of an uncontrolled HAMR object can be carried out

in sufficiently short compute time and provide increased insight to the evolution of the object's dynamics.

6.1 OSIRIS REx Case Study

The Origins Spectral Interpretation Resource Identification Security Regolith Explorer (OSIRIS-REx) is an asteroid sample return mission to asteroid 1999-RQ36 (Bennu) [33]. Once at the asteroid Bennu the mission's primary goals are to:

- Return and analyze an asteroid sample
- Provide ground truth or direct observations for telescopic data of the entire asteroid population
- Map the chemistry and mineralogy of a primitive carbon rich asteroid
- Measure the effect of the Yarkovsky effect on asteroid's orbit
- Document the regolith at the sampling site at scales down to the sub-centimeter

The spacecraft arrived at asteroid Bennu on December 3rd 2018 completing the asteroid Approach Phase. During the following Preliminary Survey phase, initial estimates of Bennu's gravitational constant $\mu = Gm$ were made. These estimates are necessary input parameters which guide the Flight Dynamics (FD) team in their navigation about the body. On December 31st 2018 the OSIRIS-REx spacecraft went into orbit about Bennu. Once entering orbit the spacecraft began taking direct optical navigation measurements which now allow the FD team to conduct Bennu-centric navigation [35].

The KinetiX[1] OD team have sought to develop force models grounded in physics rather than fitting models to empirical flight data. To this end, the OSIRIS-REx dynamical models undergo a continuous integration process where each model is revised and improved in order to navigate the spacecraft more accurately and obtain Bennu physical parameter estimates with increasing

[1] KinetiX, Inc., Space Navigation and Flight Dynamics (SNAFD), Simi Valley CA, 93065.

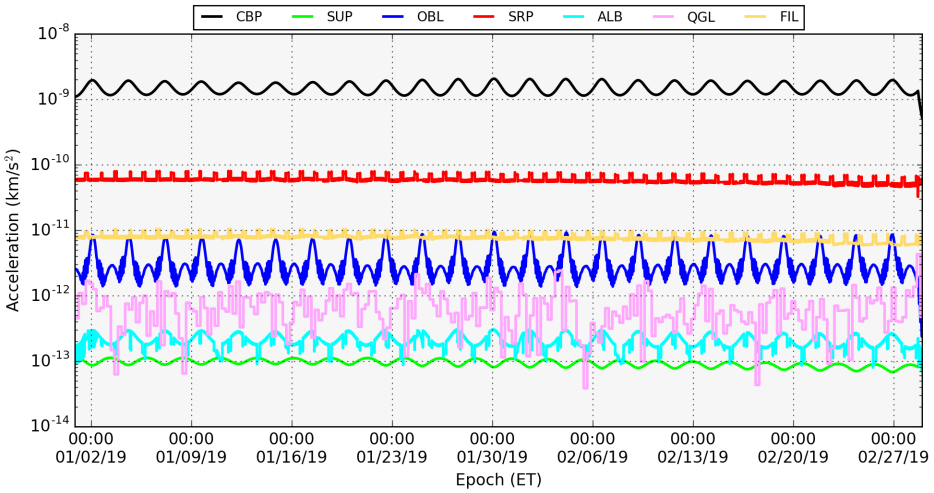


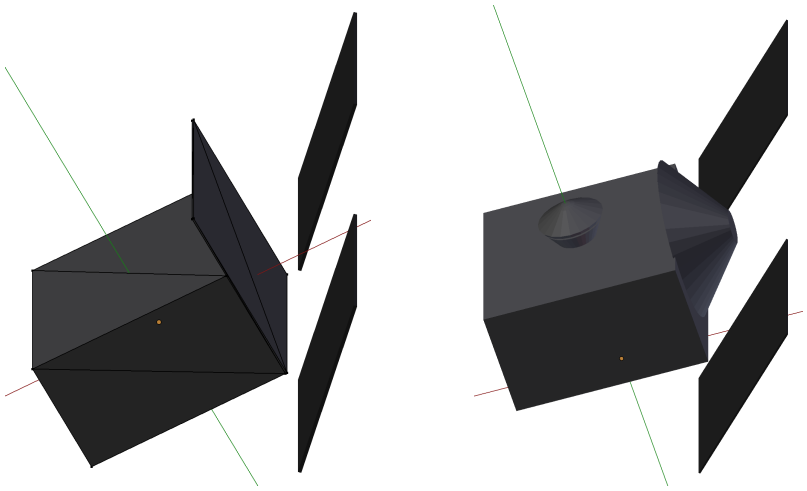
Figure 6.1: Error of the ray-traced force components relative to the faceted force norm for multiple bounce evaluation [3].

levels of confidence. Now in orbit about the asteroid Bennu, the dominant forces on the OSIRIS-REx spacecraft are SRP and thermal re-radiation/antenna pressure. Figure 6.1 illustrates the magnitudes of the various forces impacting the OSIRIS-REx trajectory during the Orbital-A Phase. The largest force after the Bennu GM (CBP) is the Solar Radiation Pressure (SRP), followed by the thermal re-radiation and antenna pressure combined (FIL). Other forces such as the obliquity (OBL), Bennu albedo (ALB), and Sun 3rd body effects (SUP) are orders of magnitude smaller. This case study presents a contribution to the SRP force modeling improvements in the OSIRIS-REx OD effort. Both the OpenGL-CL and ray tracing SRP modeling approaches are used to compute the SRP force and torque on the OSIRIS-REx spacecraft.

6.1.1 ORex Case Study: SRP Modeling

The need to model the SRP induced accelerations for the OSIRIS-REx OD process has evolved as the mission has progressed. A variety of SRP models have been developed by the KinetiX team with each model being applied at different points during the mission. This is consistent with KinetiX's approach of continuous integration and improvement of OD force models. Initially, a 10-plate faceted model was developed which included averaged specular and diffuse coefficients

[3] Reproduced with the permission of KinetiX, Inc.



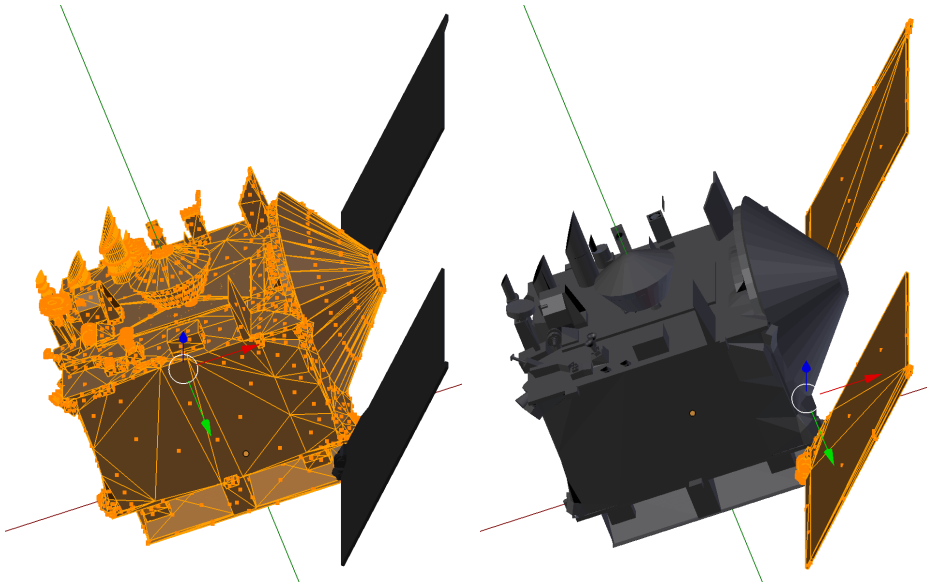
(a) Box and wing (b) Box and HGA

Figure 6.2: OSIRIS-REx box and wing models.

for each plate. The diffuse and specular material coefficients for the 10-plates approximating the bus and panels are computed by weighting all the spacecraft's components specular and diffuse contributions by the area as seen along each of the six body frame axis directions. The weighted specular and diffuse values are applied to each plate as the optical properties for SRP evaluation.

The second SRP model was developed as a response to an increase in observed stochastic accelerations. The increased accelerations are particularly significant during Earth-point mode attitudes (HGA directed toward Earth). While the 10-plate model is sufficient for Sun-point attitudes where the HGA is directed towards the Sun, at Earth-point attitudes the HGA's true conic shape significantly deviates from the flat plate approximation. Consequently, an updated model was developed which includes nine plates and a truncated spherical cone to approximate the HGA. Examples of the approximations made by these two models are demonstrated by the models shown in Figure 6.2.

To further develop the SRP modeling and evaluate the utility of the various box and wing models the OpenGL-CL and ray tracing methods have been applied to a high fidelity CAD generated mesh model of the OSIRIS-REx spacecraft. The mesh model used for the SRP evaluation is shown in Figure 6.3. The model is made up of two sub-meshes to accommodate the articulation of the



(a) Bus mesh (b) Solar panels mesh

Figure 6.3: OSIRIS-REx spacecraft model sub-meshes.

solar panels and uniquely set the optical material properties for each mesh. The spacecraft material optical properties are listed in Table 6.1. The Lockheed Martin thermal subsytem design team has indicated that almost the entirety of the spacecraft bus is covered with Germanium Black Kapton (GBK) Multi Layered Insulation (MLI) [15]. The solar panels are triple junction Gallium-Arsenide by SolAero [86].

Using the ray tracing approach, a table of SRP forces normalized to a solar flux at 1 AU is generated for 6000 sun headings over the 4π steradian attitude possibilities. The model is evaluated assuming ideal lambertian and specular reflection, incorporating the optical properties as given in Table 6.1. Rays are evaluated up to the second surface intersection. The difference of the 10-plate model and nine-plate HGA models with respect to the ray tracing approach are shown in Figure 6.4 and Figure 6.5, respectively. In both plots the spacecraft Sun-point attitude, where the $+{}^{\mathcal{B}}\hat{\boldsymbol{x}}$ body vector (equivalently the HGA bore sight) is directed toward the Sun, corresponds to a sun heading longitude and latitude of (0°, 0°). Given this reference attitude, it is evident that the difference between the 10-plate model and the ray tracing model is reduced in all vector components of the acceleration by the introduction of the 9-plate HGA model.

Table 6.1: OSIRIS-REx material optical properties.

Material	α	γ	ρ_d	ρ_s
MLI with GBK	0.49	0.51	0.102	0.408
Solar Panels (SP) Front	0.885	0.116	0.092	0.023
Solar Panels (SP) Rear	0.95	0.05	0.05	0.0

To utilize the ray tracing generated force vectors in the OD process, the table of force vector is approximated using a 10×10 spherical harmonics (SH) model. The difference between the 10×10 SH and the original SRP force values are shown in Figure 6.6. Again, in Figure 6.6, the spacecraft Sun-point attitude corresponds to a sun heading longitude and latitude of (0°, 0°). The approximation error of the SH model at this attitude is less than 2×10^{-13} km/s^2. While the Sun-point attitude is significant as it is the attitude which the spacecraft primarily holds, it is clear from Figure 6.6 that for a large majority of attitudes the error between the SH and the ray traced data remains on the order of 10^{-13} or less.

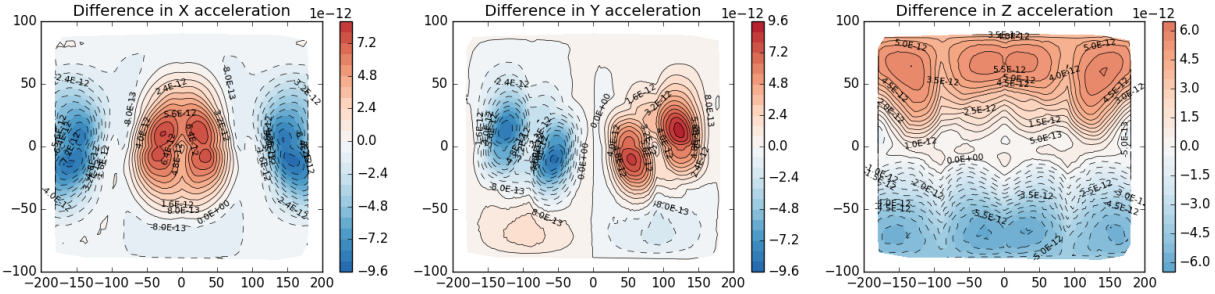


Figure 6.4: Difference in ray-tracing SRP accelerations [km/s^2] with approximated 10-Plate model.

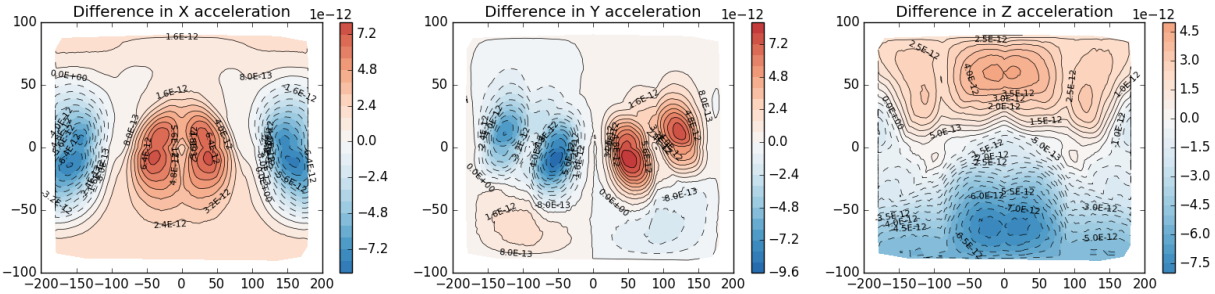


Figure 6.5: Difference in ray-tracing SRP accelerations [km/s^2] with approximated 9-Plate HGA model.

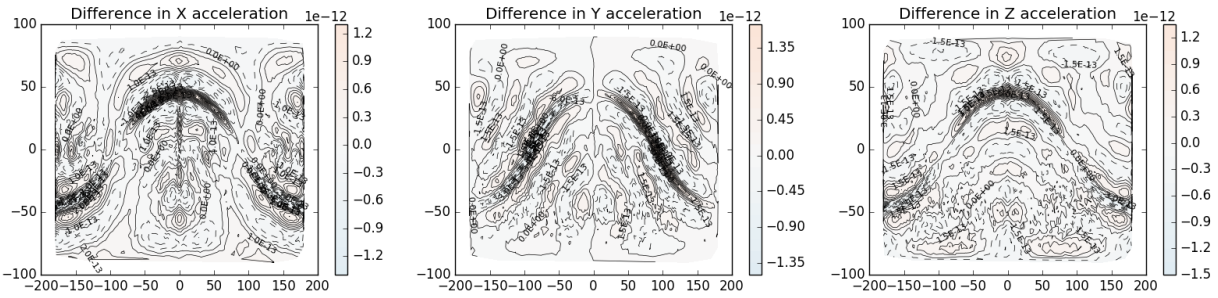


Figure 6.6: Error of the SH approximation relative to the generated ray-traced force vectors.

6.1.2 ORex Case Study: Modeling Error and Bounding Analysis

Following the initial OSIRIS-REx SRP model development there remained an error between the SRP acceleration estimated by the KinetiX OD team and the accelerations computed by all SRP models. A goal of the ray tracing and OpenGL-CL methods is to make appropriate use of existing pre-launch engineering data directly within the model. This is a particularly useful aspect of both approaches where direct access is available to meaningful spacecraft physical parameters such as surface material properties, spacecraft shape model and articulation. Direct access to these parameters allows for physically meaningful analysis to be performed and related back to flight telemetry and tracking data.

In order to assess possible contributions to the observed difference between the SRP force model and the estimated accelerations, four aspects of the spacecraft configuration are examined. The four possible sources of modeling error include attitude determination error, mesh model accuracy error, error in solar panel orientation telemetry, and material properties. The Sun-point attitude is used as a baseline attitude upon which to compare estimated accelerations with the model computed accelerations. The Sun-point accelerations for the various modeling efforts are shown in Table 6.2. These values correspond to the spacecraft at a distance of 0.90018 AU. The material optical properties for each model evaluation are those quoted in Table 6.1. The solar panels are oriented at 45° as measured from the $\hat{\boldsymbol{z}}$ body axis. The KinetiX OD effort has found that a scale increase of 7.5% is required in order for the OD included SRP model to fit the estimated accelerations. This equates to an increase in acceleration of approximately 4.5×10^{-12} [km/s^2] above

Table 6.2: Computed SRP evaluations for sun-point ${}^{\mathcal{B}}\hat{\boldsymbol{s}} = [1, 0, 0]$.

Model	Acceleration [km/s^2]
KinetiX Box And Wing	[-5.93729$\times10^{-11}$, -3.50682$\times10^{-17}$, 2.82388$\times10^{-12}$]
Hifi RT 1 bounce	[-5.70989$\times10^{-11}$, 9.41156$\times10^{-14}$, 3.18398$\times10^{-12}$]
Hifi OpenGL-CL	[-5.71870$\times10^{-11}$, 9.88224$\times10^{-15}$, 3.18490$\times10^{-12}$]
Hifi RT two bounce	[-5.66393$\times10^{-11}$, 4.41768$\times10^{-14}$, 3.01813$\times10^{-12}$]
Hifi RT three bounce	[-5.64594$\times10^{-11}$, 4.43321$\times10^{-14}$, 2.75911$\times10^{-12}$]

the Hifi RT two bounce acceleration level shown in Table 6.2.

Initial ray traced modeling yields an acceleration at Sun-point for the two bounce of $\boldsymbol{a}$ =[-5.63602 $\times10^{-11}$, 3.92259$\times10^{-14}$, 2.75019$\times10^{-12}$] (and similarly for the three bounce). This initial evaluation used only the GBK material for the bus mesh and SP Front for both the front and back of the solar panels. It is noticed that, compared to the single bounce evaluation, the two and three bounce evaluations demonstrate a smaller acceleration magnitude in the dominant $\hat{\boldsymbol{x}}$ component (compare -5.71$\times10^{-11}$ [km/s^2] to -5.63$\times10^{-11}$ [km/s^2]). It is evident that a portion of the radiation reflected by the HGA is intersecting the rear face of the solar panels. As such the solar panel sub-mesh was assigned a second set of material optical properties, SP Back as given in Table 6.1. The inclusion of a unique solar panel back material results in a small increase in the acceleration $\hat{\boldsymbol{x}}$ component to -5.66$\times10^{-11}$ [km/s^2].

For the bounding case analysis to follow, the three bounce ray traced evaluation will be used as the baseline modeling approach. The approximate error magnitudes of the three bounce ray trace acceleration relative to the estimated acceleration is given in Table 6.3. While the observed error is being attributed to SRP modeling, it is expected that a portion of this error is due to aliasing in the estimation process and is thus attributable to the spacecraft thermal model. The thermal model is a constant dynamic model within the estimation process. Given that no parameters of the thermal model are being estimated, it is unclear what portion of the modeling error is attributable to the thermal model. If it is assumed that the attributable portions of this error are commensurate with the order of magnitude of accelerations due to the SRP and thermal models, then the thermal model acceleration is approximately 15% of the total error and also given in Table 6.3.

Table 6.3: Representative portions of acceleration error (relative to Hifi RT 3 bounce) [km/s^2].

Model	$\hat{\boldsymbol{x}}$	$\hat{\boldsymbol{y}}$	$\hat{\boldsymbol{z}}$
SRP	-3.83$\times10^{-12}$	2.81$\times10^{-15}$	1.92$\times10^{-13}$
Thermal	-0.68$\times10^{-12}$	0.49$\times10^{-15}$	0.34$\times10^{-13}$

Considering attitude determination error, it is possible that the spacecraft does not hold exactly the commanded Sun-point attitude and therefore does not maintain exactly a sun heading of ${}^{\mathcal{B}}\hat{\boldsymbol{s}} = (1, 0, 0)$. It is this exact sun heading at which the ray traced model is evaluated. To establish a bounds on the possible error contribution the accelerations are computed for four spacecraft attitudes which correspond to a 0.45° offset from the sun-point heading in the $+\hat{\boldsymbol{y}}$, $-\hat{\boldsymbol{y}}$, $+\hat{\boldsymbol{z}}$ and $-\hat{\boldsymbol{z}}$. The spacecraft attitudes represented as Modified Rodriguez Parameters (MRP) and the accelerations are shown in Table 6.1.2. It is evident that excursions from the exact Sun-point heading in the $+\hat{\boldsymbol{y}}$ direction result in an increase in the acceleration magnitude for the $\hat{\boldsymbol{x}}$ component. However, the $\hat{\boldsymbol{z}}$ acceleration component moves further from the estimated value. From this it can be assumed that attitude errors are not the dominant source of this modeling error.

The second possible source of modeling error is solar panel orientation misalignment. It is assumed that the solar panels are oriented at exactly 45° with respect to the $\hat{\boldsymbol{z}}$ body frame axis. The area of each solar panel is 4.930 [m^2] giving a combined area of 9.860 [m^2] and a sun heading projected area of 6.93 [m^2]. Considering the occasion where the solar panel angle is in error such that the solar panel sun projected area would increase, at 44° the sun heading projected area is 7.048 [m^2]. This yields an increased sun project area of 0.118 [m^2] and an increase in acceleration of [-5.732179$\times10^{-11}$, 4.878957$\times10^{-14}$, 3.094433$\times10^{-12}$] which corresponds to an approximately 23% reduction of the observed acceleration error.

Table 6.4: Computed SRP evaluations for sun-point ${}^{\mathcal{B}}\hat{\boldsymbol{s}} = (1, 0, 0)$.

Offset Direction (attitude σ)	Acceleration [km/s^2]
$+\hat{\boldsymbol{y}}$ (0.0, -0.002, 0.0)	[-5.67644$\times0^{-11}$, 4.69400$\times10^{-14}$, 3.19395$\times10^{-12}$]
$-\hat{\boldsymbol{y}}$ (0.0, 0.002, 0.0)	[-5.61663$\times10^{-11}$, 4.55766$\times10^{-14}$, 2.32282$\times10^{-12}$]
$-\hat{\boldsymbol{z}}$ (0.0, 0.0, -0.002)	[-5.65510$\times10^{-11}$, -3.44146$\times10^{-13}$, 2.75717$\times10^{-12}$]
$+\hat{\boldsymbol{z}}$ (0.0, 0.0, 0.002)	[-5.65060$\times10^{-11}$, 4.28389$\times10^{-13}$, 2.75756$\times10^{-12}$]

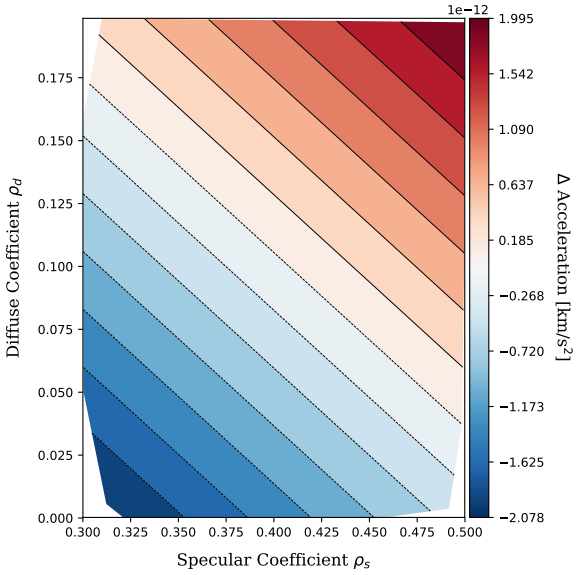


Figure 6.7: Change of acceleration magnitude for varied GBK diffuse and specular coefficients.

A further source of possible modeling error is in the allocation of material optical properties or a change in those properties while on orbit. The material optical properties used correspond to the beginning of life (BOL) values. Consequently, it is expected that the SRP acceleration resulting from the BOL optical properties will result in the largest sun pointing acceleration, only to decrease as the surface optical properties degrade over the mission duration. The diffuse and specular material optical properties are varied about their nominal values and the difference of the acceleration magnitude and a baseline acceleration magnitude is computed as $\Delta a_{\mathrm{i}} = |\boldsymbol{a}_{\mathrm{i}}| - a_{\mathrm{base}}$, where $a_{\mathrm{base}} = 5.73 \times 10^{-11}$ [km/s^2]. While material degradation has been deemed an unlikely source of modeling error the effect of mismodeling material optical properties is shown in Figure 6.7 for the spacecraft bus GBK material and in Figure 6.8 for the solar panels. As expected an increase in specularity for both materials results in an increase in acceleration. The potential acceleration increase for the GBK is approximately an order of magnitude greater than that for the solar panels.

The fourth and most rudimentary possible error source is an error in the spacecraft mesh model. The spacecraft mesh may be missing small yet important details, in particular mesh details which result in variations to the sun directed projected area. Further, increases in spacecraft bus

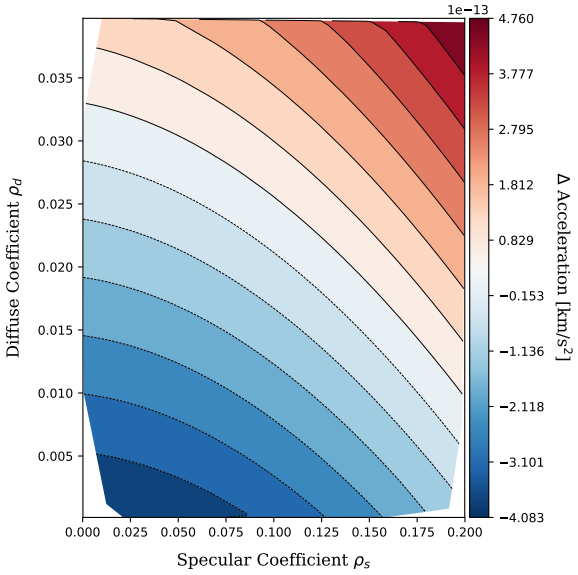


Figure 6.8: Change of acceleration magnitude for varied SP diffuse and specular coefficients.

projected area may compound with favorable errors in the GBK specular reflectance coefficient. To demonstrate the requisite acceleration increase required, a simple scale factor of 1.06 is applied to the mesh model. This results in an increased acceleration [-6.37199×10^{-11} 5.29214×10^{-14} 3.40106×10^{-12}] which accounts for 100% of the observed error.

6.1.3 ORex Case Study: Conclusions

Determination of the sources of error in the modeled and estimated radiation pressure forces is an ongoing effort. A conservative allocation of the errors gives a 85% / 15% split between the SRP and thermal radiation sources. The unknown error in the thermal model parameters makes it difficult at this stage to provide further determination of what aliasing is occurring in the estimation process.

When focusing solely on SRP modeling, significant portions of the error can be accounted for an increase in sun projected areas. This increase may be compounded by an increase in both reflectivity and importantly, the specular coefficient value of the GBK bus materials. While the majority of the spacecraft material optical properties are made up of SP and GBK the ray tracing model makes it easy to add more fidelity to material properties across the mesh. Increasing the

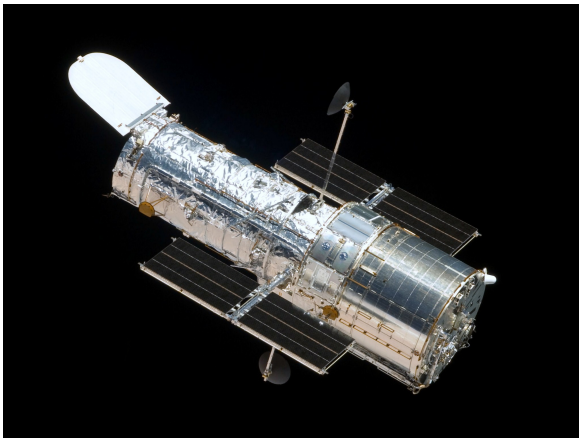


Figure 6.9: Hubble space telescope where the outer layers of the spacecraft are covered by MLI.[4]

allocation of unique material properties will help characterize and reduce uncertainty regarding the contribution of material properties to the error.

Finally, as a book-keeping exercise the spacecraft mesh model should be compared with detailed technical drawings and CAD models. Additionally, solar panel orientations should be confirmed based on spacecraft telemetry and pre-launch verification of the solar panel actuator's accuracy.

6.2 Multi-Layer Insulation Case Study

Multi-layer insulation (MLI) is a common passive thermal control material used on many satellites. As indicated by its name, multiple layers of thermally reflective and insulating material is layered to create a thermal radiation barrier. This barrier aims to reflect external solar and planetary radiation while also maintaining the internal temperature of a spacecraft within specified operational temperature range [48]. Layer thicknesses range from 0.25 mm to 1 mm with the outer layers often being thicker. The thin nature of these materials leads to mean area-to-mass ratios of between 6 m^2/kg to 110 m^2/kg [31]. As demonstrated by MLI samples returned from the Hubble Space Telescope, MLI degrades in the harsh space environment, layers peel, become brittle and in some cases become dislodged from the satellite [25]. These detached MLI sheets are categorized

[4] NASA, About the Hubble Space Telescope, Accessed April 11, 2019 from: https://www.nasa.gov/mission_pages/hubble/story/index.html

Figure 6.10: Mars Reconnaissance Orbiter wrapped by MLI prior to launch.[5]

as high area-to-mass ratio (HAMR) objects and pose a significant challenge to the identification, tracking, and cataloging of the various objects in debris population at the Geostationary orbit (GEO) regime. Significant efforts have been made to estimate and characterize the contribution of radiation forces to the orbital and attitude motion of the space object. Two primary directions of enquiry exist for determining space debris object's state given observations. The first direction seeks to extract the object's dyamical state through observation and estimation/filtering. Such approaches rely on light curve analysis and a subsequent dynamics estimation process [62, 17, 46]. The second and complimentary approach seeks to model the object's dynamics long term dynamics and determine bounds on the possible evolved dynamics states. Further, Monte Carlo simulation approaches can provide bounding predictions on long term orbit propagation. These approaches benefit from utilizing radiation pressure models which accurately resolve the objects shape, material properties and changes in illumination. Efforts to capture the wrinkles in these objects using various

[5] NASA, Mars Reconnaissance Orbiter fully assembled prior to launch , Accessed April 12, 2019 from http://mediaarchive.ksc.nasa.gov/detail.cfm?mediaid=26685

methods requiring computation are demonstrated in References [30] and [14].

Typically ray tracing has been too computationally intensive to include in an online simulation and propagation of HAMR object dynamics. However, the OpenCL ray tracing approach presents an opportunity to compute the SRP force and torque on a HAMR object of arbitrary and complex shape. This case study presents a demonstration of how the ray tracing approach to SRP computation, integrated as a Basilisk module, can provide analysts with the ability to propagate uncontrolled space objects with increased shape modeling fidelity.

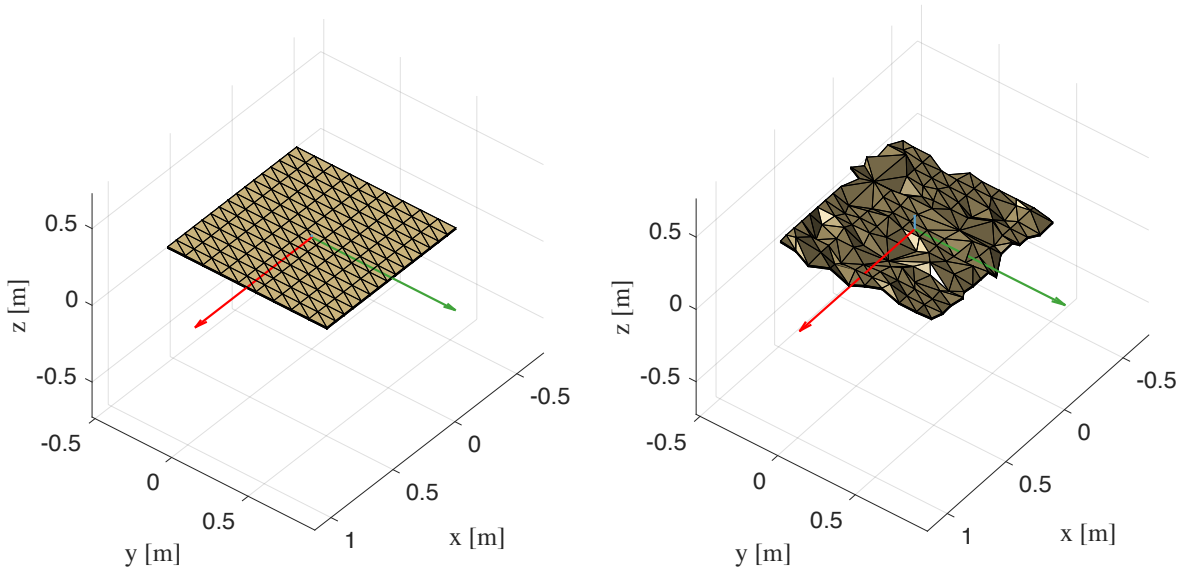
6.2.1 MLI Case Study: Mesh Models

A sheet of MLI is modeled as a flat plate as shown in Figure 6.11(a) and with increased geometric fidelity as a wrinkled sheet shown in Figure 6.11(b). The surface area of each mesh model is 1.08 [m^2]. The surface optical properties for each mesh are configured as though the sheet has peeled off a spacecraft bus and one side of the sheet is the a reflective coated (metal deposition) layer while the other side is a weakly reflective diffuse uncoated face. The optical properties for coated MLI material are ρ_s =0.60, ρ_d=0.26 and ρ_a=0.14 and ρ_s =0.00, ρ_d=0.10 and ρ_a=0.90 for the uncoated material [31].

Each model has a constant thickness of 10 mm as an MLI lay-up contains small air gaps between each sheet. The edge facets of the mesh are assigned the uncoated material optical properties. The reasoning for assigning the edge facets the uncoated optical properties lies with the assumption that the edge of the MLI sheet is a mixture of exposed coated and uncoated layers with little space between. Any rays which intersect such a region are likely to undergo many surface reflections between the layers and be absorbed within the materials.

6.2.2 MLI Case Study: Ray Traced Force Comparison

The SRP induced force is evaluated for 1000 equally spaced sun headings over the 4π [str] attitude sphere. The evaluation is carried out for each mesh model at 1 AU distance from the sun. The force magnitude for each of the $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{z}}$ components is shown in Figure 6.12 for the flat



(a) Flat plate MLI

(b) Wrinkled MLI.

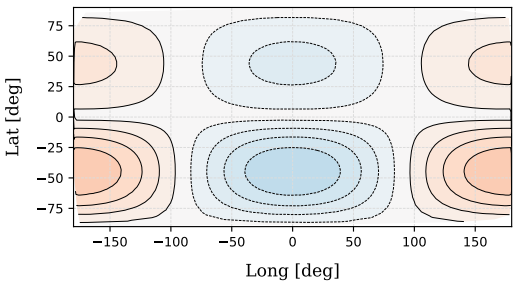
Figure 6.11: Mesh models for MLI sheet of equal surface area, 1.08 [m^2].

plate model and in Figure 6.13 for the wrinkled model. The flat geometry of both plates is visible where the force magnitude is near zero sun heading latitudes of 0°. Further, the lower $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$ force magnitude and increased $\hat{\boldsymbol{z}}$ force magnitude show the effect of the more specular coated front side. Conversely, the lower hemisphere demonstrates the more diffuse and absorptive uncoated back side.

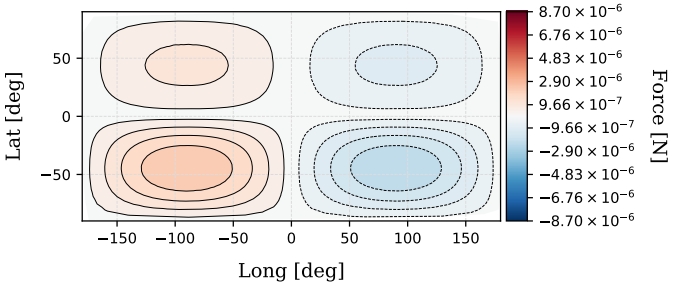
The idealized flat plate model shows a larger $\hat{\boldsymbol{z}}$ component magnitude than the magnitude of the $\hat{\boldsymbol{z}}$ component for the wrinkled MLI. Similarly, correlations exist in the decreased lateral force magnitudes for the flat plate. The percentage difference between the force magnitudes of the two models, computed as,

$$\Delta F_{\%} = \left(\frac{|\boldsymbol{F}_{\text{wrinkle}}| - |\boldsymbol{F}_{\text{flat}}|}{|\boldsymbol{F}_{\text{flat}}|} \right) \times 100 \tag{6.1}$$

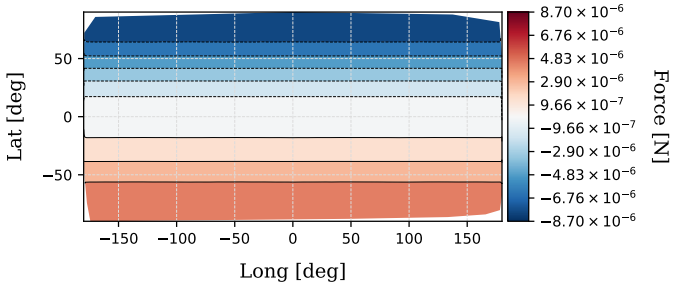
and is shown for all attitudes in Figure 6.14. As expected the force magnitude of the wrinkled object is less than the flat plate object for the majority of attitudes with the differences ranging from -5.0% to at most -13.23%. In this figure the maximum error has been capped at 50% so that discernible relief is maintained within the plot. The percentage error can be as large as 427% for sun-headings in the $\hat{\boldsymbol{x}}$-$\hat{\boldsymbol{y}}$ plane. For these attitudes the flat plate cross section becomes extremely small compared to the wrinkle plate cross section. The difference between the multiple bounces for



(a) Force x component flat plate model

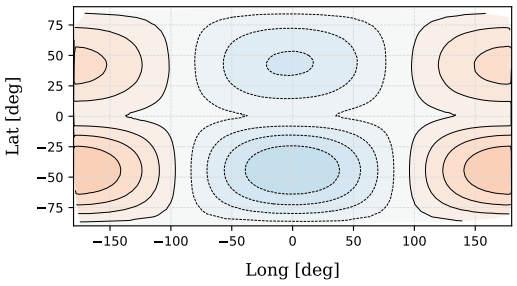


(b) Force y component flat plate model

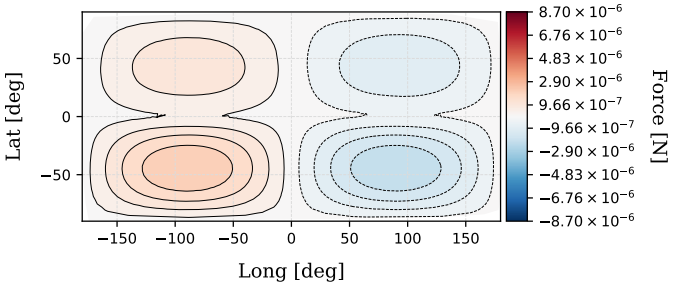


(c) Force z component flat plate model

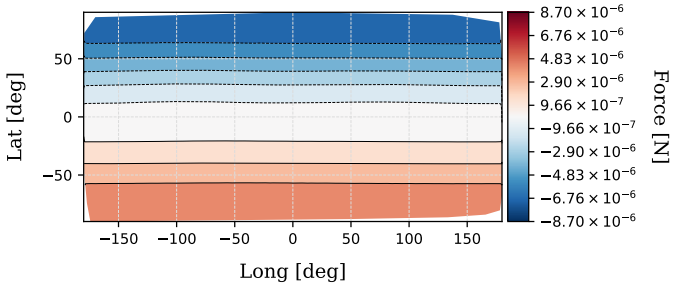
Figure 6.12: Body frame force components for the flat plate MLI mesh model.



(a) Force x component wrinkled model



(b) Force y component wrinkled model



(c) Force z component wrinkled model

Figure 6.13: Body frame force components for the wrinkled MLI mesh model.

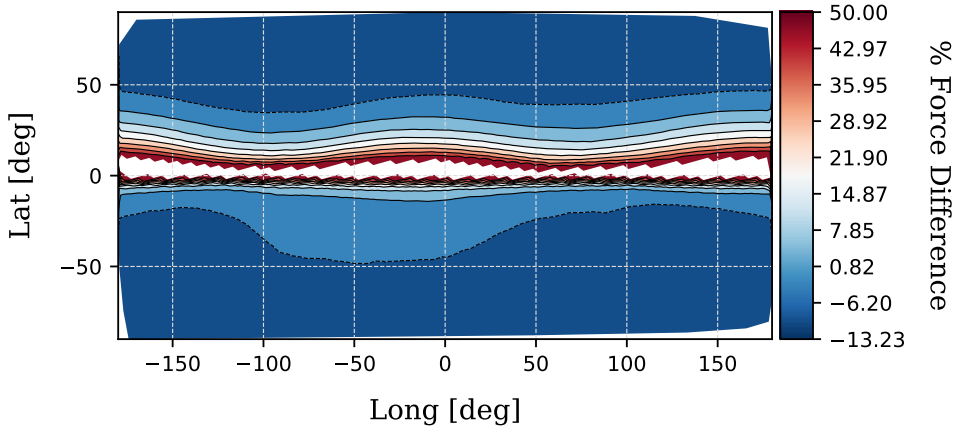
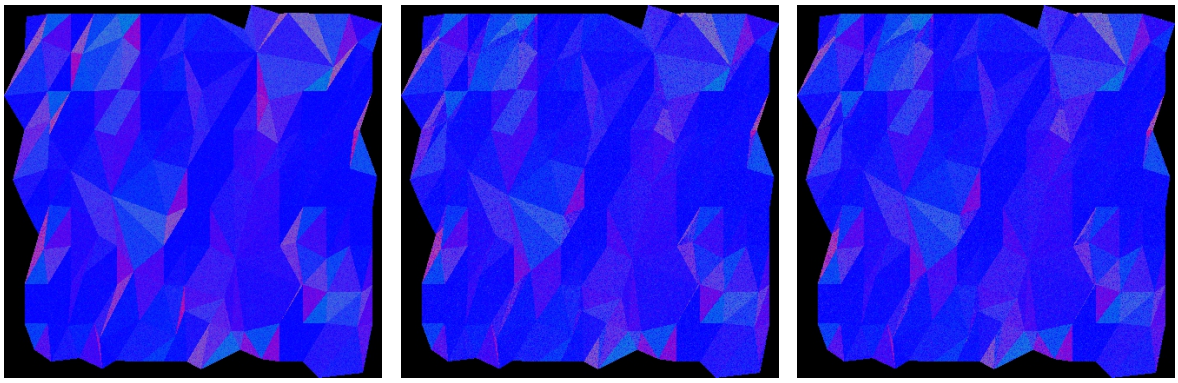


Figure 6.14: Force magnitude difference of the wrinkled model relative to the flat plate model.



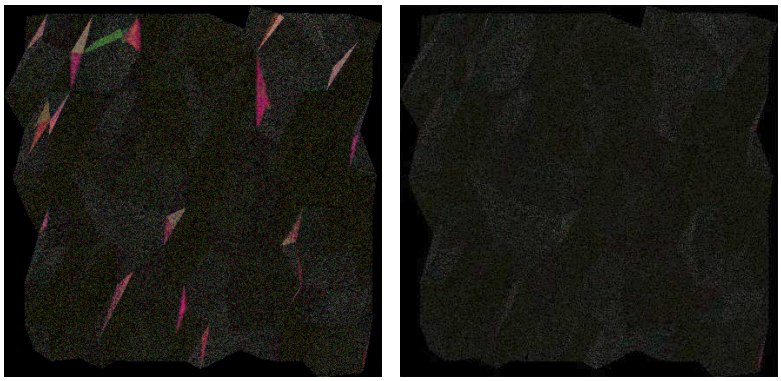
(a) One bounce (b) Two bounces (c) Three bounces

Figure 6.15: Rendered MLI mesh model for one and two where the difference between the rendered images is shown.

a sun heading perpendicular to coated side of the wrinkled mesh (colinear with the $\hat{\boldsymbol{z}}$ direction) is visualized in the rendered output of Figure 6.15. The difference between each bounce is visualized in Figure 6.16. It is evident that resolving the second ray bounces make a visible contribution to the force resolution, while resolving the third bounce shows a barely visible change.

6.2.3 MLI Case Study: Orbit Propagation

A GEO orbit simulation, developed in Basilisk, is used to simulate the evolution of the dynamics of each MLI mesh model over a single orbit. The orbital simulation is kept simple to emphasize the difference in evolution of the dynamics due to SRP on each mesh model. Of



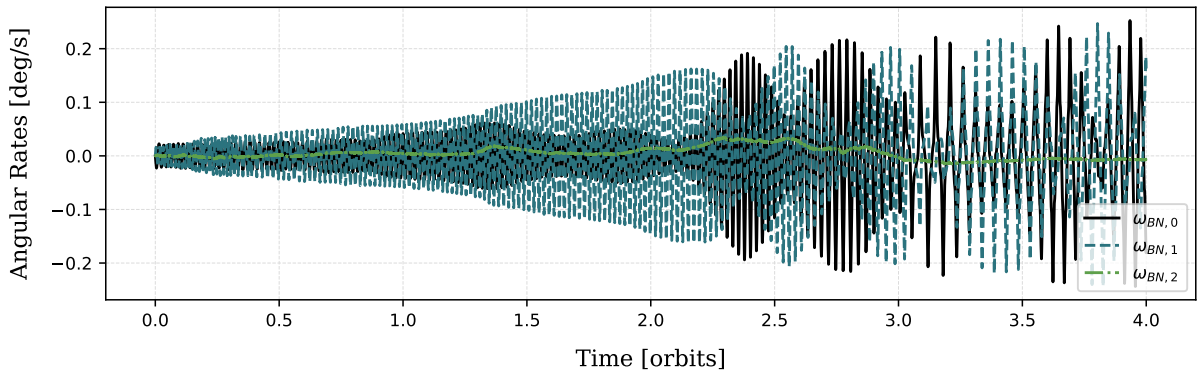
(a) Difference between bounce one and two.

(b) Difference between bounce two and three.

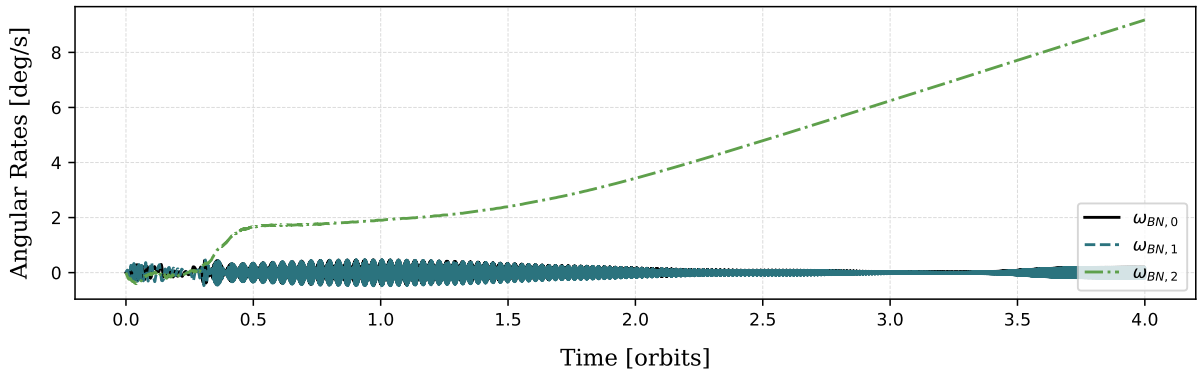
Figure 6.16: Rendered MLI mesh model differences between successive bounces.

particular interest is the ability of the ray tracing method to resolve the attitude dependent self shadowing and multiple reflection of the wrinkled MLI model as compared to the typically used flat plate model. The simulation's initial orbital parameters are $r_{\text{GEO}} = 42164$ [km], $e = 0$, $i = 0$, $\Omega = 0$, $\omega = 0$ and $\nu = 90°$. The initial attitude is given as the Modified Rodriquez Parameters set $\sigma_{BN} = [-0.44, 0.22, -0.44]$, which corresponds to a predominantly $\hat{\boldsymbol{z}}$ axes sun pointing attitude where the coated (highly reflective) side of the model is initially sun facing. The MLI object is given zero initial body angular rate.

The angular rate evolution of each model is shown in Figure 6.17 where both models develop a slow 'wobble' about the $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$ axes with the $\hat{\boldsymbol{z}}$ axis remaining roughly sun pointing. It is evident that both models demonstrate. The wrinkled plate model develops a spin about the sun pointing $\hat{\boldsymbol{z}}$ axis and it is notable that the spin rate continues to increase through to the end of the simulated four orbits. As shown in Figure 6.18 the attitude evolution between the two models differs in a manner commensurate to the angular rates. The flat plate maintains a consistent attitude earlier and then begins a slowly increasing wobble about the sun facing axis. The attitude variations of the wrinkled model exhibit a higher frequency wobble, compared to the flat plate model. Finally, the inertial frame force and body frame torque are shown in Figure 6.19 and Figure 6.20, respectively. The variability in the resulting force and torque due to the wrinkled model's surface variation is visible. When compared to the force and torque of the flat plate model, it is clear that the variability of



(a) Angular rate evolution flat plate.

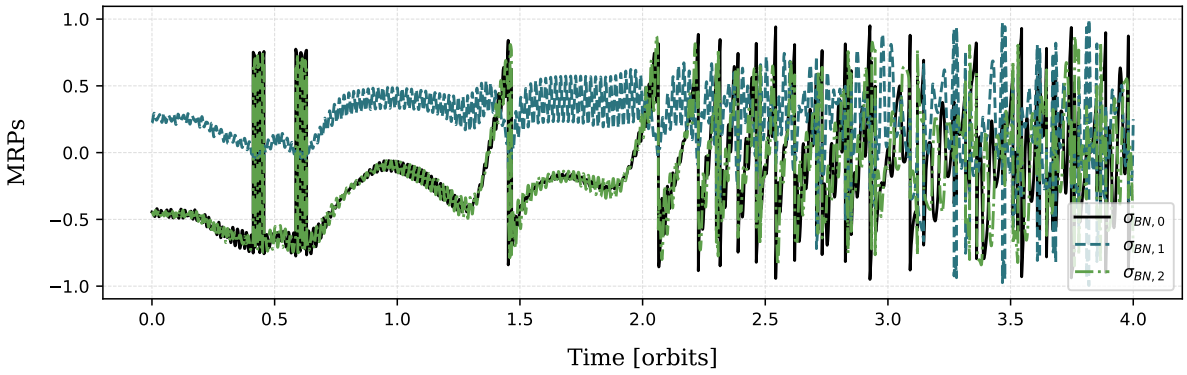


(b) Angular rate evolution wrinkled.

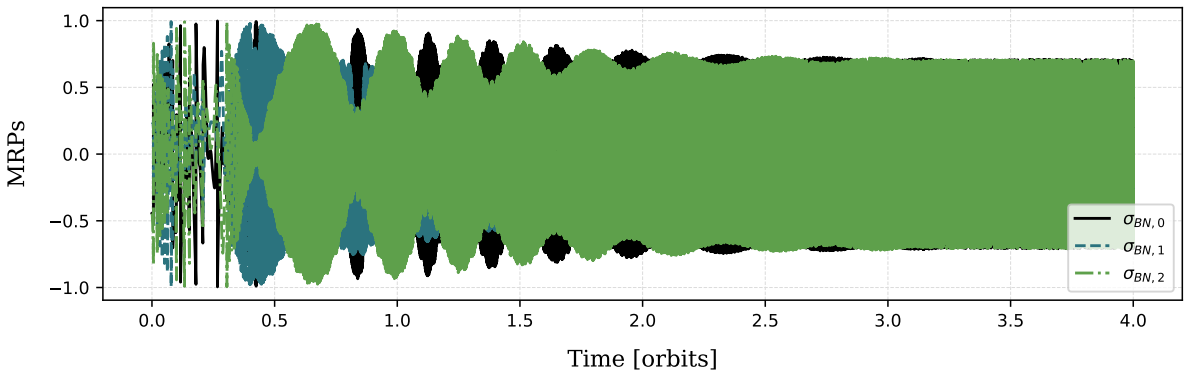
Figure 6.17: Ray tracing propagated angular rate evolution for MLI mesh model in GEO orbit.

the wrinkled model contributes to faster generation of body angular rates and significant attitude change over time.

The comparison between the simulated flat plate model and the wrinkled model shows that capturing the effects of shape variation on SRP force and torque is important to simulating accurate evolution of the HAMR object's dynamics. The OpenGL-CL method is also capable of capturing the effect of shape variation (for the single radiation bounce) on SRP force and torque. The OpenGL-CL method is used to simulate the wrinkled model for the same four GEO orbits. This simulation allows one to develop a sense of whether simply capturing the equivalent first bounce with the OpenGL-CL method, is sufficient to yield similar dynamics evolution as that demonstrated by the multi-bounce ray tracing method. The OpenGL-CL resolution is 200 $\times$ 200 pixels which results in a resolution of the sun projected plane of 400 000 pixels. As shown in Figure 6.18 the

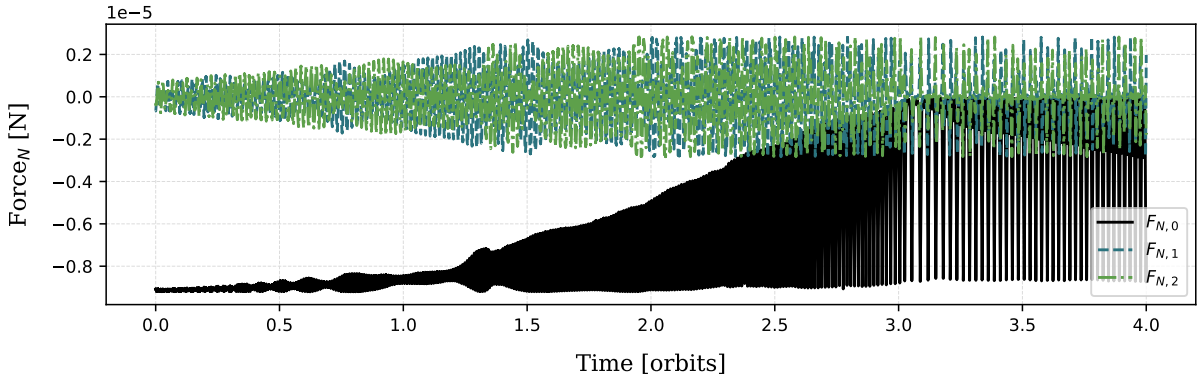


(a) Attitude evolution flat plate.

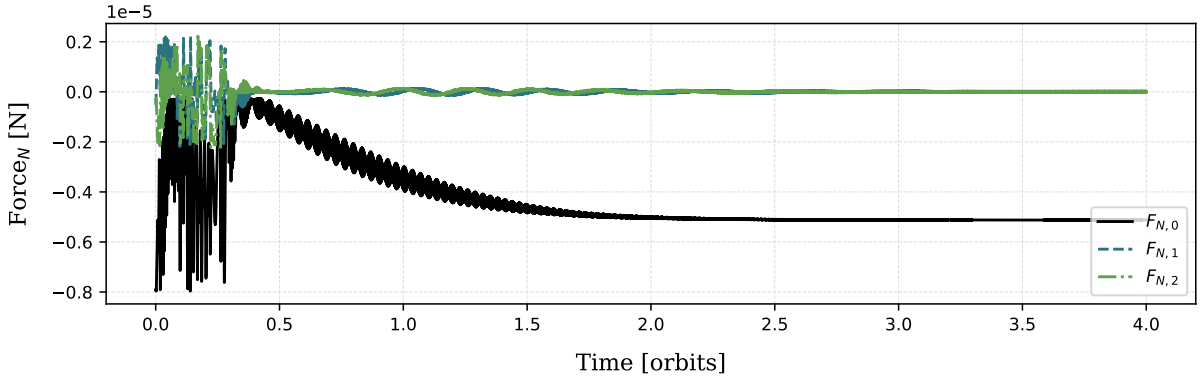


(b) Attitude evolution wrinkled.

Figure 6.18: Ray tracing propagated attitude evolution for MLI mesh model in GEO orbit.

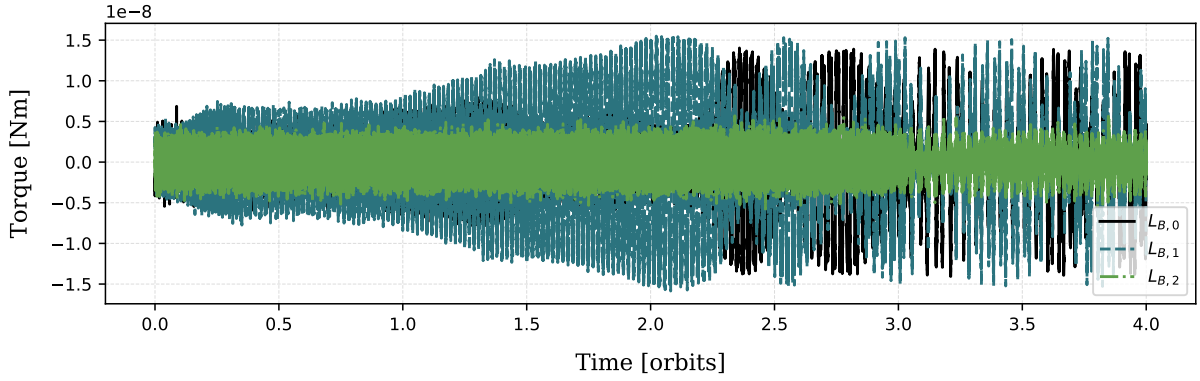


(a) Earth centered inertial frame force flat plate.

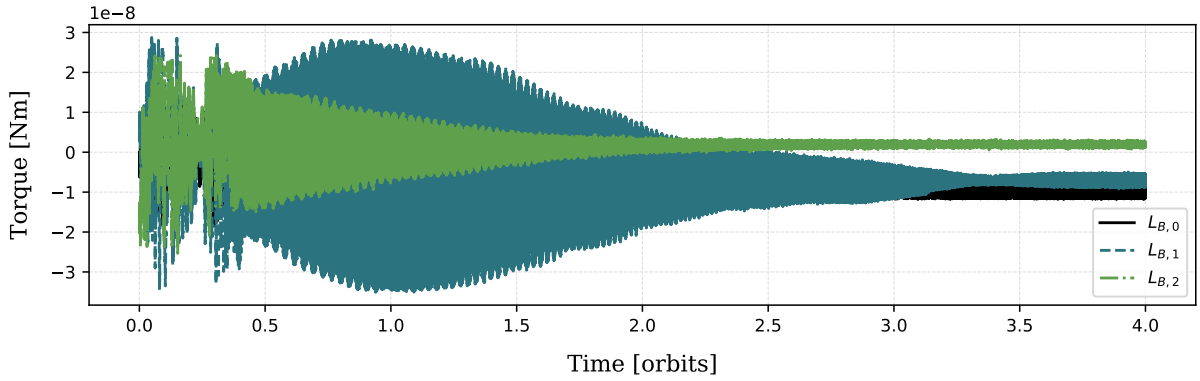


(b) Earth centered inertial frame force wrinkled.

Figure 6.19: Ray tracing propagated Earth centered inertial frame force evolution for MLI mesh model in GEO orbit.



(a) Torque body frame flat plate.



(b) Torque body frame wrinkled.

Figure 6.20: Ray tracing propagated body-frame torque evolution for MLI mesh model in GEO orbit.

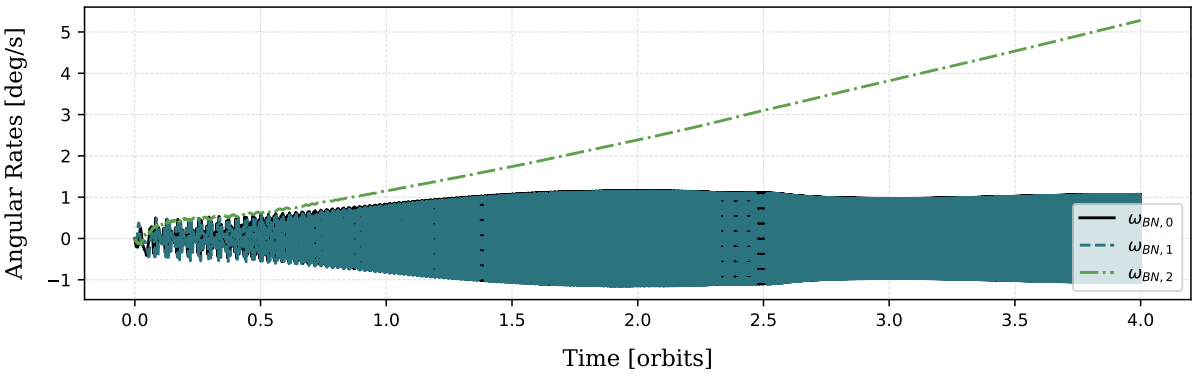


Figure 6.21: OpenGL-CL propagated angular rate evolution for MLI mesh model in GEO orbit.

attitude evolution between the two models differs in a manner commensurate to the angular rates. The flat plate maintains a consistent attitude earlier and then begins a slow spin about the sun facing axis. The wrinkled modeled attitude variations are an order of magnitude larger than the flat plate. Finally, the force and torque are shown in Figure 6.19 and Figure 6.20, respectively. The variability in the resulting force and torque due to the wrinkled models surface variation is visible. When compared to the force and torque of the flat plate model, it is clear that the variability of the wrinkled model contributes to faster generation of body angular rates and significant attitude change over time.

The ray tracing method applied in these simulations presents an online simulation capability currently not available using consumer grade computing resource. While high performance real-time computing systems are available and provide some of these rendering capabilities, these systems require specialized hardware configured specifically for the purpose of distributed computing. While computational speeds and ray tracing implementations vary greatly, a computational load commensurate to that of the OpenCL ray tracing approach would yield an approximate slow down by a factor of 100 compared to modest GPU hardware.

Efficient computational approaches which provide a great deal of insight into the long term propagation of uncontrolled objects can be achieved analytically via averaging theory techniques[74]. While highly effective, these approaches require precomputation assumptions about the spacecraft geometry and articulation state [63]. In contrast the OpenGL-CL and ray tracing models can be integrated with articulated mesh models that account for both the variation of shape and time varying material properties. Further, in the case of controlled object simulation the presented methods are also able to resolve the shadowing and reflection impacts from other objects in the case of close proximity operations and formation flying.

While the ray tracing method is used to demonstrate the dynamical effects of capturing a more realistic MLI shape model, the OpenGL-CL method presents an even greater opportunity to run Monte Carlo simulation of these orbit propagations because of its 10×-100× reduction in computation time compared to the ray tracing method (for similar resolutions on the first bounce).

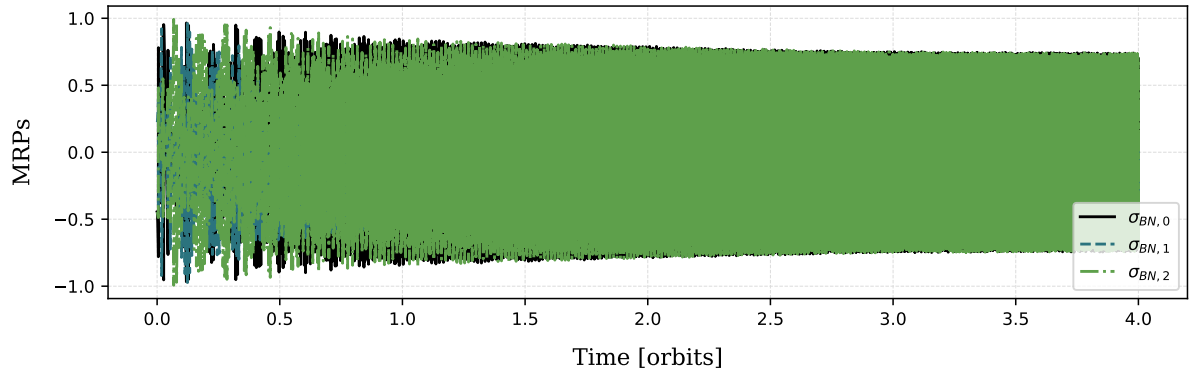


Figure 6.22: OpenGL-CL propagated attitude evolution for MLI mesh model in GEO orbit.

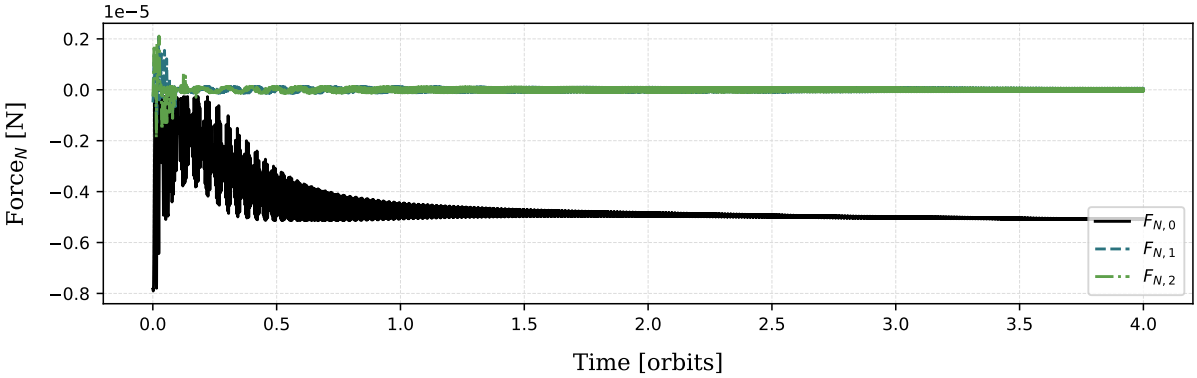


Figure 6.23: OpenGL-CL propagated Earth centered inertial frame force evolution for MLI mesh model in GEO orbit.

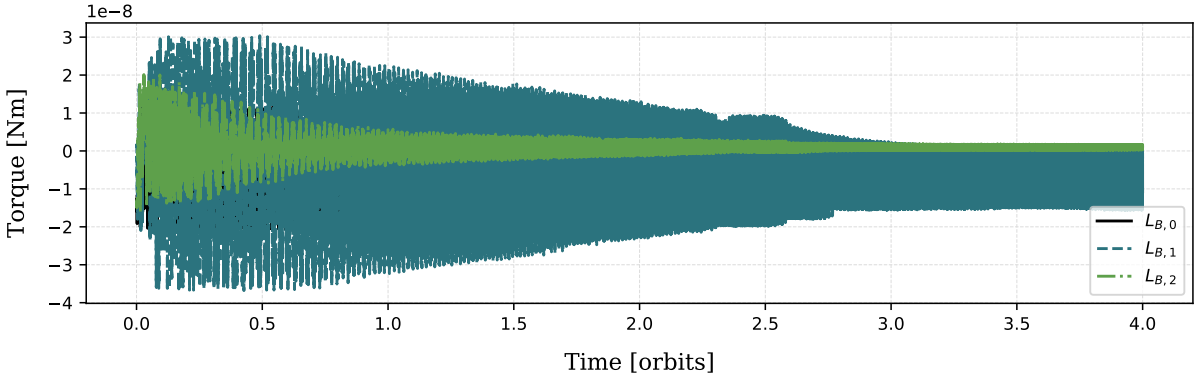


Figure 6.24: OpenGL-CL propagated body-frame torque evolution for MLI mesh model in GEO orbit.

For certain geometries only the ray tracing method captures the significant change in force direction due to a secondary bounce. For other geometries, such as that demonstrated here, only a very small portion of the force and torque is due to secondary reflection and thus the SRP dynamics can be resolved with high geometric fidelity by the OpenGL-CL method.

6.2.4 MLI Case Study: Conclusions

Capturing the realistic shape and it's coupling to the dynamics is important to propagation. The typical approach of using flat plate approximation for computing the SRP force and torque of an MLI HAMR object yields significant under and over prediction of the actual force and torque. The RT and OpenGL-CL methods both capture a MLI HAMR objects realistic shape variations. The random wrinkles which develop in an MLI blanket and the shape of the MLI sheet after becoming dislodged from its original object results in additional complexity and uncertainty on the propagation of the object's dynamics. This case study demonstrates that resolving a more realistic shape for a HAMR object can be a useful contributor to greater insight in long term propagation prediction.

Both the ray tracing and OpenGL-CL modeling methods carry out the propagation of a complex mesh model at computational speeds which make faster than real time online simulation and Monte Carlo simulation possible. Further, these simulation capabilities can be achieved easily on consumer grade computer hardware obviating the need to develop purpose built hardware. Each model is able to resolve model self shadowing, varying and complex material properties, spacecraft articulation and complex spacecraft mesh models. For mesh models where multiple reflections are unlikely the OpenGL-CL methods provide further reduction in time to solution and makes tractable Monte Carlo simulation of orbit prorogation.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Modern consumer grade computers possess significant and often under utilized computing resources. Modeling astrodynamic phenomena with high fidelity regularly demands increased computing power and novel approaches to each model's algorithmic design and software based implementation. In particular, modeling solar radiation pressure (SRP) with high fidelity, must account for the computationally expensive processing of detailed spacecraft models, complex material optical behaviors and time varying changes in the spacecraft's articulation and general dynamic state. High-fidelity SRP modeling methods contained in the literature demand that the models be pre-computed with all spacecraft state changes known prior to model evaluation. These pre-computed models output data as a lookup table which is subsequently used in any online simulation analysis tool.

This dissertation presents two novel solar radiation pressure (SRP) models which leverage the ubiquitous latent computational power available in consumer computing resources. The modeling approaches presented resolve SRP at significantly faster than real-time computational speeds while maintaining the capability to resolve the myriad inputs to the spacecraft's dynamic and structural state. This is achieved by both leveraging the computationally powerful graphics processing unit (GPU) hardware and developing computationally efficient algorithms.

The OpenGL-CL method makes use of the highly optimized OpenGL API vector graphics render pipeline. Doing so enables the method to model the equivalent of the first ray surface

intersection of a ray tracing approach. It inherently captures spacecraft self shadowing, varied material properties and spacecraft articulation. The recursive loose bounding box algorithm significantly reduces the number of operations required to compute the sun heading oriented view frame. An efficient parallel reduction algorithm is developed using the OpenCL API. The parallel reduction reduces OpenCL kernel launch overhead by having each Work Item sum multiple values and interleaves the SRP computation with the reduction operation.

Modeling SRP via a ray tracing approach has long been considered to offer the highest resolution of the SRP force and torque on a spacecraft. However, such approaches have come an large computational costs and therefore a slow time to solution. This dissertation details a faster than real-time SRP ray tracing model which employs the highly parallel environment of the GPU. The evolution of ray tracing techniques in the graphics rendering community has supplied a suite of efficient algorithms that are used with great effect in the SRP context. Techniques such as ray intersection search space reduction using bounding volume hierarchies, fast ray triangle intersection and importance sampling complex BRDFs all facilitate fast SRP evaluation. Furthermore, unlike graphics rendering, SRP evaluation is not concerned with the accuracy of visual representation, but the accuracy in force and torque resolution. This allows for assumptions such as non-transmissive materials to be introduced into the SRP evaluation which reduces computational complexity.

This dissertation presents a modular implementation for each modeling method which allows for direct integration and online faster than real-time simulation within an astrodynamics simulation software. The pursuit of modularity culminates with the development and demonstration the Blacklion distributed simulation middleware. The application of the Blacklion software architecture is novel in that it facilitates the execution of computationally demanding models on remote commodity computing resources. The Blacklion architecture integrates heterogeneous models into a distributed simulation and transparently manages data exchange and the advancement of time.

In an analysis situation determining which modeling approach to use is a decision which rests upon a few points of compromise. The first consideration is in regards to the impact of multiple spacecraft self reflections. Certain spacecraft geometries, coupled with particular sun headings,

may result in greater error between ray tracing and faceted approaches. If it is determined that the spacecraft CONOPS will not spend any appreciable time at these high error attitudes then analysts may choose to use the OpenGL-CL. This is demonstrated by the simulation and propagation of the complex wrinkled MLI mesh model. Under the assumption of a rigid model, such a mesh experiences little self reflection. Secondly, while both methods are capable of after than real-time evaluation, it is a benefit to choose the ray tracing method where there is access to higher performance GPU devices. Using the ray tracing model, it is shown that resolving, at least the secondary ray bounce, accounts for a significant portion of modeling error which otherwise would remain when using SRP models that ignore spacecraft self reflection. Further, the Blacklion architecture enables access to high performance GPUs via distributed simulation.

This work makes greater use of existing engineering data in the SRP modeling processes to reduce, where possible, uncertainty in the SRP force computation. Such pre-launch engineering data includes spacecraft geometry, material optical properties, and possible spacecraft time varying articulations. The computational speed of the models presented gives rise to the possibility of including high-fidelity SRP models in spacecraft ground software simulation and long-term dynamics propagation. Because the methods do not rely on an precomputed lookup tables, arbitrary time varying qualities of the spacecraft state (materials, articulations) can be accounted for in Monte Carlo simulation configurations.

7.2 Recommendations for Future Work

A particularly clear extension to this work exists in the area of thermal radiation modeling. Spacecraft thermal radiation induced force and torque operate via mechanisms similar to SRP. With some accommodation for thermal energy accumulation at the spacecraft surface, a faster than real-time thermal radiation model may be implemented by employing the same ray tracing architecture described in this dissertation. Similarly, the OpenGL-CL method could be modified to account for thermal accumulation and the resultant thermal radiation force.

While not a focus of this dissertation, both modeling approaches presenting interesting op-

portunities to include high-fidelity SRP evaluation in long-term dynamics simulation. Given the fast computation times of the SRP modeling methods an increased scope of spacecraft dynamics analysis can be conducted without the need for specialized and hard to access high performance computing resources.

Both modeling approaches can be extended to accommodate additional radiation sources as input. Sources such as planetary albedo, Earth infrared radiation and source of energy radiated from high power antenna. Beyond radiation sources, each modeling method may be extended to compute spacecraft drag forces and torques above altitudes where the particle mean free path distance is large enough to model the intersection of atmospheric particles in the same manner as reflecting photons.

Bibliography

[1] Advanced Solutions. Stk solis: Commercial plug-in to the analytical graphics, inc (agi) systems toolkit (stk). http://www.go-asi.com/solutions/stk-solis/, Oct 2018. Accessed 1 Oct 2018.

[2] Cody Allard, Manuel Diaz Ramos, Hanspeter Schaub, Patrick Kenneally, and Scott Piggott. Modular Software Architecture for Fully Coupled Spacecraft Simulations. Journal of Aerospace Information Systems, pages 1–14, oct 2018.

[3] Peter G. Antreasian and G W Rosborough. Prediction of radiant energy forces on the topex/-poseidon spacecraft. Journal of Spacecraft and Rockets, 29(1):81–90, 2019/04/26 1992.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. Commun. ACM, 53(4):50–58, April 2010.

[5] Ravishekhar Banger and Koushik Bhattacharyya. OpenCL Programming by Example. Packt Publishing, 2013.

[6] Y E. Bar-Sever. New and improved solar radiation models for gps satellites based on flight data final report. Technical report, Jet Propulsion Laboratory, 1997.

[7] P. Beckmann and A. Spizzichino. The Scattering of Electromagnetic Waves from Rough Surfaces. Artech House radar library. Artech House, 1987.

[8] O. Bertolami, F. Francisco, P. J S Gil, and J. Páramos. Estimating radiative momentum transfer through a thermal analysis of the pioneer anomaly. Space Science Reviews, 151(1-3):75–91, 2010.

[9] J J Biesiadecki, D A Henriques, and A Jain. A reusable, real-time spacecraft dynamics simulator. Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE, 2:8.2–8–8.2–14 vol.2, 1997.

[10] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. The unified modeling language user guide. J. Database Manag., 10:51–52, 1999.

[11] Paul Bourke. Data formats: Object files (.obj), http://paulbourke.net/dataformats/obj/.

[12] A. Jain C. Lim. Dshell++: A component based, reusable space system simulation framework. In Proceedings - 2009 3rd IEEE International Conference on Space Mission Challenges for Information Technology, SMC-IT 2009, pages 229–236, Pasadena, CA, July 19 - 23 2009. IEEE.

[13] Jonathan Cameron, Abhinandan Jain, Burkhart Dan, Erik Bailey, J Balaram, Eugene Bonfiglio, Havard Grip, Mark Ivanov, and Evgeniy Sklyanskiy. Dsends: Multi-mission flight dynamics simulator for nasa missions. Aiaa Space 2016, (September):1–18, 2016.

[14] Sittiporn Channumsin, Matteo Ceriotti, and Gianmarco Radice. A deformation model of flexible, hamr objects for accurate propagation under perturbations and the self-shadowing effects. Advances in Space Research, 61:1066–1096, 02 2018.

[15] Michael K. Choi. Thermal assessment of sunlight impinging on osiris-rex ocams polycam, otes, and imu-sunshade mli blankets in flight. In Proc. SPIE 10401, Astronomical Optics: Design, Manufacture, and Test of Space and Ground Systems, volume 10401. SPIE, September 2017.

[16] John G. Cleary, Brian M. Wyvill, Graham M. Birtwistle, and Reddy Vatti. Multiprocessor ray tracing. Computer Graphics Forum, 5(1):3–12, 1986.

[17] Ryan D. Coder, Marcus J. Holzinger, and Moriba K. Jah. Space-object active control mode inference using light curve inversion. Journal of Guidance, Control, and Dynamics, 41(1):88–100, 2019/04/22 2017.

[18] M. Cols-Margenet, H. Schaub, and S. Piggott. Modular attitude guidance: Generating rotational reference motions for distinct mission profiles. Journal of Aerospace Information Systems, 15(6):335–352, 2018.

[19] Mar Cols Margenet, Hanspeter Schaub, and Scott Piggott. Modular platform for hardware-in-the-loop testing of autonomous flight algorithms. In International Symposium on Space Flight Dynamics, Matsuyama-Ehime, Japan, June 3–9 2017.

[20] Douglas E. Comer. Internetworking with TCP/IP. Addison-Wesley Professional, 6th edition, 2013.

[21] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. In Proceedings of the 8th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '81, pages 307–316, New York, NY, USA, 1981. ACM.

[22] CS Systèmes d'Information. Orekit: An accurate and efficient core layer for space flight dynamics applications. https://www.orekit.org, 2018. Accessed 15 Oct 2018.

[23] J. Cuseo. Stk/solis and stk/odyssy flight software: Supporting the entire spacecraft lifecycle. In Workshops on Spacecraft Flight Software. Johns Hopkins University Applied Physics Laboratory, Laurel, MD, October 2011.

[24] DARTS Shell (Dshell). Jet propulsion lab darts lab. https://dartslab.jpl.nasa.gov, 2018. Accessed 1 Oct 2018.

[25] Joyce Dever, Kim K. deGroh, Jacqueline Townsend, and L Len Wang. Mechanical properties degradation of teflon(trademark) fep returned from the hubble space telescope. Technical report, National Aeronautics and Space Administration, 02 1998.

[26] Donald J. Dichmann, Cassandra M. Alberding, and Wayne H. Yu. Stationkeeping monte carlo simulation for the james webb space telescope. In 24th International Symposium on Space Flight Dynamics, volume 1, pages 1–21, 2014.

[27] Kayvon Fatahalian and Mike Houston. A closer look at gpus. Commun. ACM, 51(10):50–57, October 2008.

[28] Henry F. Fliegel and Thomas E. Gallini. Solar force modeling of block IIR Global Positioning System satellites. Journal of Spacecraft and Rockets, 33(6):863–866, 1996.

[29] FreeFlyer. a.i. solutions. https://ai-solutions.com/freeflyer/, 2018. Accessed 21 Oct 2018.

[30] Carolin Frueh and Moriba Jah. Coupled orbit-attitude motion of high area-to-mass ratio (hamr) objects including self-shadowing. Acta Astronautica, 95:227–241, 02 2014.

[31] Carolin Früh, Thomas M Kelecy, and Moriba K Jah. Coupled orbit-attitude dynamics of high area-to-mass ratio (hamr) objects: influence of solar radiation pressure, earth's shadow and the visibility in light curves. Celestial Mechanics and Dynamical Astronomy, 117(4):385–404, 2013.

[32] Ryu Funase, Yoji Shirasawa, Yuya Mimasu, Osamu Mori, Yuichi Tsuda, Takanao Saiki, and Jun'ichiro Kawaguchi. On-orbit verification of fuel-free attitude control system for spinning solar sail utilizing solar radiation pressure. Advances in Space Research, 48(11):1740 – 1746, 2011. Solar Sailing: Concepts, Technology And Missions.

[33] J. Gal-Edd and A. Cheuvront. The osiris-rex asteroid sample return mission operations design. In 2015 IEEE Aerospace Conference, pages 1–9, March 2015.

[34] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2013.

[35] Jeroen L. Geeraert, Jason M. Leonard, Patrick W. Kenneally, Christian W. May, Peter G. Antreasian, and Michael C. Moreau. Osiris-rex navigation small force models. In Astrodynamics Specialist Conference. AIAA, 2019.

[36] General Mission Analysis Tool. Nasa goddard space flight center. https://software.nasa.gov/software/GSC-17177-1, 2018. Accessed 27 Oct 2018.

[37] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C. C. Luk. Performance characterisation and simulation of intel's integrated gpu architecture. In 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 139–148, April 2018.

[38] EKA Gill and O Montenbruck. Satellite orbits: Models, methods and applications. Springer, 2013.

[39] Goddard Space Flight Center. 42: A comprehensive general-purpose simulation of attitude and trajectory dynamics and control of multiple spacecraft composed of multiple rigid or flexible bodies. https://software.nasa.gov/software/GSC-16720-1, Oct 2018. Accessed 2018-10-1.

[40] D. Guarnera, G.C. Guarnera, Abhijeet Ghosh, Cornelia Denk, and Mashhuda Glencross. BRDF Representation and Acquisition. Computer Graphics Forum, 35(2):625–650, may 2016.

[41] Eric Heitz. Understanding the masking-shadowing function in microfacet-based brdfs. Journal of Computer Graphics Techniques (JCGT), 3(2):48–107, June 2014.

[42] Eric Heitz. A Simpler and Exact Sampling Routine for the GGX Distribution of Visible Normals. Research Report Unity Technologies, page 4, 2017.

[43] Eric Heitz and Eugene D'Eon. Importance sampling microfacet-based bsdfs using the distribution of visible normals. Computer Graphics Forum, 33(4):103–112, July 2014.

[44] Pieter Hintjens. ZeroMQ Messaging for Many Applications. O'Reilly Media, 2013.

[45] ISC License. Open source initiative. https://opensource.org/faq, 2018. Accessed 15 Oct 2018.

[46] M. Jah and R. Madler. Satellite Characterization: Angles and Light Curve Data Fusion for Spacecraft State and Parameter Estimation. In Advanced Maui Optical and Space Surveillance Technologies Conference, page E49, 2007.

[47] A. Jain and G. Rodriguez. Recursive flexible multibody system dynamics using spatial operators. Journal of Guidance, Control, and Dynamics, 15(6):1453–1466, Nov 1992.

[48] Robert Karam. Satellite Thermal Control for Systems Engineers. American Institute of Aeronautics and Astronautics, 2019/04/22 1998.

[49] T L Kay and J T Kajiya. Ray Tracing Complex Scenes. Computer Graphics (SIGGRAPH '86 Proceedings), 20(4):169–278, 1986.

[50] Patrick W. Kenneally. High geometric fidelity solar radiation pressure modeling via graphics processing unit. Master's thesis, University of Colorado, Boulder, 2016.

[51] Patrick W. Kenneally and Hanspeter Schaub. High geometric fidelity modeling of solar radiation pressure using graphics processing unit. In AAS/AIAA Spaceflight Mechanics Meeting, Napa Valley, California, Feb. 14–18 2016. Paper No. AAS-16-500.

[52] Patrick W. Kenneally and Hanspeter Schaub. Modeling solar radiation pressure with self-shadowing using graphics processing unit. In AAS Guidance, Navigation and Control Conference, Breckenridge, CO, Feb. 2–8 2017. Paper AAS 17-127.

[53] Patrick W. Kenneally and Hanspeter Schaub. Fast spacecraft solar radiation pressure modeling by ray-tracing on graphic processing unit. In AAS Guidance and Control Conference, Breckenridge, CO, Feb. 1–7 2018. Paper AAS 18-096.

[54] Khronos Group. OpenGL Documentation, 10 2015.

[55] H. Kuninaka and J. Kawaguchi. Lessons learned from round trip of hayabusa asteroid explorer in deep space. In 2011 Aerospace Conference, pages 1–8, March 2011.

[56] David M. Lucchesi. Reassessment of the error modelling of non-gravitational perturbations on LAGEOS II and their impact in the Lense–Thirring derivation—Part II. Planetary and Space Science, 50(10-11):1067–1100, 2002.

[57] J A Marshall, S B Luthcke, P G Antreasian, and G W Rosborough. Modeling Radiation Forces Acting on TOPEX/Poseidon for Precision Orbit Determination. Technical report, 1992.

[58] Marshall Space Flight Center. Marshall solar activity future estimation. https://sail.msfc.nasa.gov, 2018. Accessed 1 Oct 2018.

[59] Yaroslav Mashtakov, Stepan Tkachev, and Mikhail Ovchinnikov. Use of external torques for desaturation of reaction wheels. Journal of Guidance, Control, and Dynamics, 41(8):1663–1674, 2018.

[60] MATLAB/Simulink. Mathworks. https://www.mathworks.com/products/matlab.html, 2018. Accessed 23 Oct 2018.

[61] Jay W. McMahon and Daniel J. Scheeres. New solar radiation pressure force model for navigation. Journal of Guidance, Control, and Dynamics, 2010.

[62] Jay W. McMahon and Daniel J. Scheeres. Improving space object catalog maintenance through advances in solar radiation pressure modeling. Journal of Guidance, Control, and Dynamics, 38(8):1366–1381, 2019/04/22 2015.

[63] Jay W. McMahon and Daniel J. Scheeres. Improving space object catalog maintenance through advances in solar radiation pressure modeling. Journal of Guidance, Control, and Dynamics, 38(8):1366–1381, 2019/04/24 2015.

[64] Tomas Moller and Ben Trumbore. Fast , Minimum Storage Ray / Triangle Intersection. Journal of Graphics Tools, 2(1):21–28, 1997.

[65] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Geometrical considerations and nomenclature for reflectance. Technical report, Final Report National Bureau of Standards, Washington, DC. Inst. for Basic Standards, 1977.

[66] OpenCL Working Group Khronos. The OpenCL Specification Version: 2.2, 06 edition, March 2016.

[67] Daniel J. O'Shaughnessy, James V. McAdams, Peter D Bedini, Andrew B Calloway, Kenneth E Williams, and Brian R Page. Messenger's use of solar sailing for cost and risk reduction. Acta Astronautica, 93:483–489, January 2014.

[68] J.D. D Owens, M. Houston, D. Luebke, S. Green, J.E. E Stone, and J.C. C Phillips. GPU Computing. Proceedings of the IEEE, 96(5), 2008.

[69] Matt Pharr and Greg Humphreys. Physically Based Rendering: From Theory to Implementation, Third Edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, third edition edition, 2017.

[70] Tomas Plachetka. Perfect load balancing for demand-driven parallel ray tracing. In Euro-Par, 2002.

[71] Proceedings of the AIAA/USU Conference on Small Satellites. NASA Operational Simulator for Small Satellites (NOS3): Tools for Software-based Validation and Verification of Small Satellites, 2016.

[72] Benny Rievers. High precision modelling of thermal perturbations with application to Pioneer 10 and Rosetta. PhD thesis, University of Bremen, 2012.

[73] Hyung-Jin Rim, Charles Webb, Sungpil Yoon, and Bob Schutz. Radiation Pressure Modeling for ICESat Precision Orbit Determination. AIAA/AAS Astrodynamics Specialist Conference and Exhibit, (August):1–7, 2006.

[74] Aaron J. Rosengren and Daniel J. Scheeres. Long-term dynamics of high area-to-mass ratio objects in high-Earth orbit. Advances in Space Research, 52(8):1545–1560, 2013.

[75] Hanspeter Schaub. 14th US National Congress of Theoretical and Applied Mechanics, 23 - 28 June 2002. Modelling and Simulation in Materials Science and Engineering, 9(4), jul 2001.

[76] Hanspeter Schaub and John L. Junkins. Analytical Mechanics of Space Systems. AIAA Education Series, Reston, VA, 3rd edition, 2014.

[77] Issac D. Scherson and Elisha Caspary. Multiprocessing for ray tracing: a hierarchical self-balancing approach. The Visual Computer, 4(4):188–196, Jul 1988.

[78] Conrad Schiff and Edwin Dove. Monte carlo simulations of the formation flying dynamics for the magnetospheric multiscale (mms) mission. Journal of Aerospace Engineering, Sciences and Applications, 4(4):66–78, 2012.

[79] Christophe Schlick. An inexpensive brdf model for physically-based rendering. Computer Graphics Forum, 13:233–246, 1994.

[80] Brandon T. Bailey Scott A. Zemerick, Justin R. Morris. Nasa operational simulator (nos) for v and v of complex systems. In Modeling and Simulation for Defense Systems and Applications VIII, volume 8752, 2013.

[81] Dave Shreiner, Graham Sellers, John Kessenich, and Bill Licea-Kane. OpenGL Programming Guide: The Official Guide To Learning OpenGL, Version 4.3. Addison-Wesley, Upper Saddle River, NJ, 2013.

[82] B. Smith. Geometrical shadowing of a random rough surface. IEEE Transactions on Antennas and Propagation, 15(5):668–671, Sep. 1967.

[83] Brian Smits. Efficiency Issues for Ray Tracing. Journal of Graphics Tools, 3(2):1–14, 1999.

[84] T. A. Springer, G. Beutler, and M. Rothacher. A new solar radiation pressure model for gps satellites. GPS Solutions, 2(3):50–62, 1999.

[85] Graeme L. Stephens, Deborah G. Vane, Ronald J. Boain, Gerald G. Mace, Kenneth Sassen, Zhien Wang, Anthony J. Illingworth, Ewan J. O'connor, William B. Rossow, Stephen L. Durden, Steven D. Miller, Richard T. Austin, Angela Benedetti, and Cristian Mitrescu. The cloudsat mission and the a-train. Bulletin of the American Meteorological Society, 83(12):1771–1790, dec 2002.

[86] Rao Surampudi, Julian Blosiu, Paul Stella, John Elliott, Julie Castillo, Thomas Yi, John Lynos, Mike Piszczor, Jeremiah McNatt, Chuck Taylor, Ed Gaddy, Simon Liu, Ed Plichta, Christopher Iannello, Patricia M. Beauchamp, and James A. Cutts. Solar power technologies for future planetary science missions (JPL D-101316). Technical Report December, Strategic Missions and Advanced Concepts Office Solar System Exploration Directorate Jet Propulsion Laboratory, 2017.

[87] Systems Tool Kit. Analytic graphics inc. https://www.agi.com/products/engineering-tools, 2018. Accessed 20 Oct 2018.

[88] Sergei Tanygin and Gregory M Beatty. Gpu-accelerated computation of drag and srp forces and torques with graphical encoding of. In Conference: 26th AAS/AIAA Space Flight Mechanics MeetingAt: Napa, CA, number February, pages 1–20, 2016.

[89] ISO/IEC JTC 1 Information technology. Iso/iec 7498-1:1994 information technology – open systems interconnection – basic reference model: The basic model. Technical report, International Organization for Standardization, 1994.

[90] Trick Simulation Environment. Nasa johnson space center. https://github.com/nasa/trick/wiki/FAQ, 2018. Accessed 20 Oct 2018.

[91] David Vallado. Fundamentals of astrodynamics and applications. Springer, New York, 2007.

[92] Eric Veach. Robust monte carlo methods for light transport simulation. Dissertation at the Department of Computer Science of Stanford University, 134(December):759–764, 1997.

[93] Bruce Walter, Sr Marschner, Hongsong Li, and Ke Torrance. Microfacet models for refraction through rough surfaces. Eurographics, pages 195–206, 2007.

[94] Charles J. Wetterer, Richard Linares, John L. Crassidis, Thomas M. Kelecy, Marek K. Ziebart, Moriba K. Jah, and Paul J. Cefola. Refining Space Object Radiation Pressure Modeling with Bidirectional Reflectance Distribution Functions. Journal of Guidance, Control, and Dynamics, 37(1):185–196, 2014.

[95] M Ziebart, S Adhya, a Sibthorpe, S Edwards, and P Cross. Combined radiation pressure and thermal modelling of complex satellites: Algorithms and on-orbit tests. Advances in Space Research, 36(3):424–430, 2005.

[96] Marek Ziebart. High Precision Analytical Solar Radiation Pressure Modelling for GNSS Spacecraft. PhD thesis, University of East London, 2001.

Appendix A

Simulation Architecture Basilisk

Spacecraft simulation software tools are an indispensable part of modern spacecraft design processes. The continual increase in complexity of spacecraft mission and maneuver design, dynamical and kinematic design verification, and post-launch telemetry analysis all heavily rely on software simulation tools. These simulation tools provide engineers with the ability to increase the quality of design and testing by reducing cost and duration of development. For example, proposed changes to a mission's configuration, parameter tuning or in-flight anomalies may be explored via Monte Carlo simulation [78, 26]. Additionally, hardware-in-the-loop (HWIL) testing allows for verification and validation of the spacecraft hardware and software systems in a controlled laboratory environment. Hardware-in-the-loop testing can expose technical faults and system integration problems saving considerable project financial and personnel resources before launch to space. While there are both commercial and open source tools available that solve some of these challenges, there hasn't been an open astrodynamics software tool to address all.

Astrodynamics simulation tools can be broadly categorized into three groups; Commercial off the shelf (COTS), Government off the shelf (GOTS) and general open source. A number of tools have their origin in the GOTS category and subsequently moved to the open source category. Popular tools to either simulate full missions, orbits, attitude motion, flight algorithm, or that perform hardware- and software-in-the-loop capabilities include

- MATLAB/Simulink [60] combination to simulate algorithms and auto-code to flight C-code
- Analytic Graphics Inc (AGI) Systems Tool Kit (STK) [87] to model orbital simulations

and mission scenarios

- a.i. FreeFlyer [29] to simulate spacecraft dynamics
- NASA General Mission Analysis Tool (GMAT) [36] to perform orbital trajectory optimizations
- NASA Trick [90] to simulate complex spacecraft physics
- OreKit [22] to simulate spacecraft using a Java library using open tools
- Jet Propulsion Laboratory's Dynamics Algorithms for Real-Time Simulation (DARTS)/Dshell [24] software to simulate complex spacecraft behaviors and and control solution using proprietry software
- NASA 42 [39] to simulate spacecraft with open source software

Each tool is developed with a specific subset of space asset simulation purposes in mind. For example the OreKit, GMAT and STK tools were initially developed with a focus on high fidelity orbit dynamics, orbit estimation, orbit propagation and trajectory design. As a result these tools include a range of different propagators, complex multi body gravity models, drag, solar radiation pressure and orbit determination tools. For example the Orekit tool includes six optional methods to model atmospheric density ranging from simple exponential models to empirical predictive models such as the Marshall Solar Activity Future Estimation [58].

When assessing software packages in the context of their ability to simulate full spacecraft dynamics it is important to identify how the dynamics are computed and how this impacts the modularity of the implementation. For example, tools such as OreKit and STK have increased their ability to accommodate spacecraft attitude. STK can be paired with the SOLIS plugin, a commercial plugin to STK which models spacecraft translational and attitude dynamics. And while the SOLIS plugin enhances STK's spacecraft dynamics, it does not model disturbances which may alter the spacecraft's center of mass[1]. Similarly, OreKit models the spacecraft as a rigid body,

and the dynamics are primarily focused on defining perturbations as uncoupled external forces and torques.

Two tools which do provide increased modularity, coupled dynamics and the ability to customize the spacecraft dynamics are JPL's DARTS environment and NASA's "42" software package [12, 39]. The DARTS tool uses spatial operator algebra for the development of multi-body dynamics to generate a spacecraft system mass matrix in a form that is efficiently solved recursively [47]. Similarly, the simulation package "42" allows for spacecraft composed of multiple rigid or flexible bodies using a tree topology to formulate the dynamics. Both of these formulations allow developers to add arbitrary models to the simulation without significant change to the code base.

It seem an unreasonable requirement to expect a tool, which simulates the high complexity of a spacecraft system, to accommodate all possible missions configurations and spacecraft subtleties as out-of-the-box modeling functionality. On this basis, it is reasoned that extensibility of a simulation tool via means of scripting and custom code development is needed to allow engineers to adapt the tool to the particular specification and requirements of their mission. All of the tools listed include some basic level of scriptability while others enable significantly more customization. For example AGI's STK offers their Connect and Object Model APIs which facilitate the addition of custom simulation models (except for coupled spacecraft dynamics). In contrast JPL's DARTS tool allows a user to compile and add a custom model to any part of the simulation framework. This may include a model of the flexible dynamics of a large solar panel boom or the addition of a simulated ground station.

While it is not surprising that none of the COTS tools use a version of an open source license, it is interesting to note that COTS tools are typically not cross platform in so far as they support installation on only one or two of the primary operating systems; macOS, Windows, and Linux. A notable exception to this is the MATLAB suite. This limits the reach of the tool to research labs that are already using a particular operating system.

Finally, a large feature which is not available by default in most tools is HWIL and Software-in-the-loop (SWIL) functionality. Of the tools listed, MATLAB/Simulink and the DARTS/Dshell

tools support HWIL and SWIL functionality without significant modification. Hardware-in-the-loop and SWIL functionality allows engineers to use of the same set of tools and flight algorithms through multiple phases of the mission and in multiple engineering teams across an organization.

Basilisk[1] is a novel astrodynamics framework that simulates complex spacecraft systems in the space environment. While many simulation tools possess overlapping features with Basilisk, none others possess the combined characteristics of Basilisk. The Basilisk framework is a highly modular, Python user-friendly, open-source simulation framework that provides accurate (fidelity is configurable) coupled vehicle position and attitude dynamics, along with optional structural flexing, imbalanced momentum exchange device, and fuel slosh dynamics, with at least a 365 times speedup (one mission year in one compute time day). Furthermore, Basilisk is equally well employed during early mission design phases as it is later on during detailed design phases and further in post-launch telemetry analysis and spacecraft command sequence validation. The software has also been used in distributed hardware-in-the-loop simulations [19]. Basilisk is available online as open source software.[2]

This appendix describes the Basilisk framework and the underlying message passing interface that enables a very modular approach to open spacecraft simulation development. In Section A.1 the Basilisk framework's software stack is introduced. Section A.2 gives a detailed account of the aforementioned novel Basilisk system architecture, which allows for the rapid development of a simulation for of a wide variety of complex spacecraft systems. The key architectural components discussed are the Basilisk message system which facilitates data passing between models and the Basilisk spacecraft dynamics implementation. Section A.3 outlines the simulation execution flow of control and how the fundamental components of Modules, Tasks, and Task Groups work together to provide the user with flexible control over simulation design, integration rates and message passing. Sections A.4 and A.5 provide an overview of the Basilisk multi-processing Monte Carlo tools and the ability to log, process, and analyze large (multi-gigabyte) data sets. In the final section of this

[1] `https://hanspeterschaub.info/bskMain.html`
[2] `https://bitbucket.org/avslab/basilisk`

paper an example Basilisk simulation configuration will be presented to illustrate how Basilisk's modular design allows the end-user engineer to build a detailed simulation scenario from simple Basilisk building blocks.

A.1 Software Stack and Build

The core Basilisk architectural components and physics simulation modules are written in C++ to allow for object-oriented development and fast execution speed. However, Basilisk Modules can also be developed using Python, for easy and rapid prototyping, and C to allow flight software modules to be easily ported directly to flight targets. Future developments will include modules developed in Fortran to accommodate legacy space environment models.

Whereas Basilisk Modules are developed in a number of programming languages, Basilisk users interact and script simulation scenarios using the Python programming language. Python bindings are available for all Modules and supporting simulation utilities and core functionality as indicated in Fig. A.1. The Python bindings are auto-generated at compile time using the Software Interface Generator (SWIG) tool. A typical Module is defined by a header file (.h) a source file (.c/cpp) and most importantly a SWIG interface file (.i). The SWIG interface file contains compiler directives, which at compile time are parsed and determine the class interface in the target language (Python). At compile time three build products are produced for each Module's compilation. These three build products are a Module library (.so or .dll), a Python interface to the underlying library (.py), and a Python-to-source language translation file (.cxx). The Python interface mirrors the underlying C++ class variables and functions. The Python bindings allow users to employ the Module's functionality within the Python environment through the typical package import mechanism as demonstrated below in Listing. A.1.

Generating a separate SWIG wrapped library object (.so or .dll) for each Module obviates any compile time dependencies between one Module and another. This modularity relieves the user of needing any software compilation knowledge and provides the ability to rapidly reconfigure their simulation scenario during runtime at the Python language level of the technology stack. This crit-

Listing A.1: Selective Python imports of Basilisk Modules.

```
from Basilisk.simulation import reactionWheelStateEffector, rwVoltageInterface, simple_nav,
    spacecraftPlus
```

ical feature allows a spacecraft simulation to be modified later on by selectively replacing Modules and connecting them with the input and output message passing of the existing Modules. Each Module has its own unit and integrated tests. As a result of this decoupling Module replacement is achieved without requiring the re-validation of other simulation.

A.2 Modularity In Basilisk

The types of missions that Basilisk can be used to simulate lay on a spectrum with Earth orbiting cubesats at one end and interplanetary probes and spacecraft constellations at the other. The hallmark of the Basilisk framework is its highly modular system architecture. Modular design has been the guiding principle throughout Basilisk's development. The result is that Basilisk implements only two core system components, the Basilisk message system and Basilisk simulation controller. Basilisk's modular design is achieved by three key design choices. The first is the complete decoupling of model and run loop dependence. The second design choice is to use a messaging system approach to manage Module input and output data and inter-Module data requirements. Finally, a Dynamics Manager is implemented to manage the fully-coupled nature of a spacecraft rigid body dynamics in a computationally efficient manner [2].

A.2.1 Components

Spacecraft onboard computers typically employ a real-time operating system (RTOS) which executes algorithms at both a fixed rate and within a fixed allocation of time. Similarly, a dynamic simulation employs either a fixed or variable time step integration of the equations of motion (EOM). Both of these time rate driven processes motivate the conceptualization of the core Basilisk

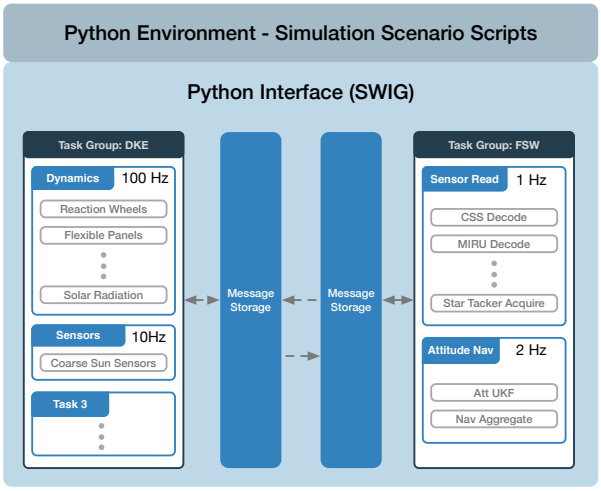


Figure A.1: An example layout of a complete Basilisk simulation where each element of the system has SWIG generated Python interfaces available in the Python environment.

components introduced in this section. The result is a unique flexibility and configurability of a Basilisk simulation scenario's timing and flexible integration rates.

A Basilisk simulation is built up from Modules, Tasks and Task Groups. These fundamental abstractions are depicted in their relationship to each other in Fig. A.2. A Basilisk Module is stand-alone code which typically implements a specific model (e.g. an actuator, sensor, and dynamics model) or self-contained logic (e.g. translating a control torque to a reaction wheel command voltage). Modules receive input data as messages by subscribing to desired messages available from the Messaging System. Similarly, a Module publishes output data as messages to the Messaging System.

Tasks are groupings of Modules. Each Task has an individually set integration rate. The Task integration rate directs the update rate of all Modules assigned to that Task. As a result a simulation may group modules with different integration rates according to desired fidelity. Furthermore, the configured update/integration rate of each Task can be adjusted during a simulation to capture increased resolution for a particular phase of the simulation. For example a user may increase the integration rate for the Task containing a set of spacecraft dynamics Modules, such as flexing solar panels and thrusters, in order to capture the high-frequency flexing dynamics and thruster firings during Mars Orbit Insertion (MOI). Otherwise the integration time step may be kept to a longer

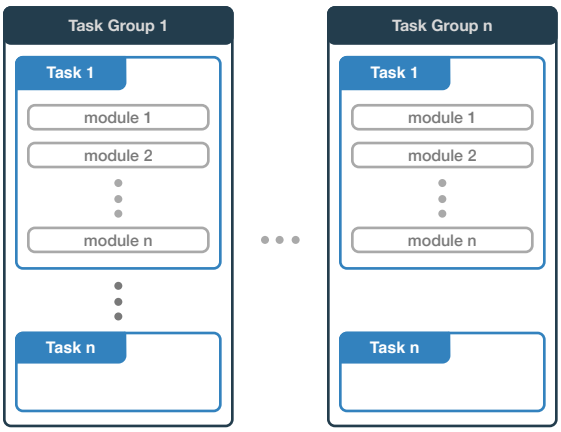


Figure A.2: Basilisk Task Group, Tasks and Module layout.

duration during the less dynamically active mission phases such as interplanetary cruise.

The execution of a Task and therefore the Modules within that Task is controlled by either enabling or disabling the Task. A Task's enabled status can be toggled any time during a simulation. This feature is particularly useful for enabling or disabling FSW specific Modules contained within a Task related to the simulated spacecraft's FSW mode e.g. Safe Mode, Sun Pointing.

Task Groups are the highest level grouping of Basilisk components. Task Groups act as a container for Tasks and provide a mechanism for resolving message dependencies between Modules as discussed in greater detail in Section. A.2.2. Task Groups can be considered silos of Tasks and the messages published and subscribed by Modules within the Task Group.

A.2.2 Message System

The Basilisk messaging system facilitates the input and output of data between simulation Modules. The messaging system decouples the data flow between Modules and Task Groups and removes explicit inter-Module dependency resulting in no run-time Module dependencies.

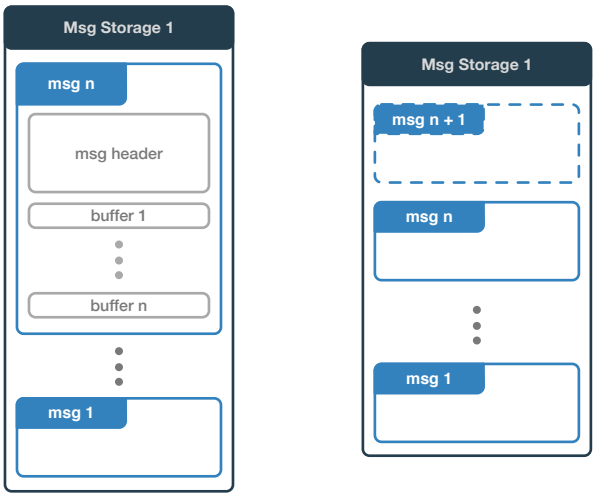
A Basilisk Module reads input messages and writes output messages to the Basilisk messaging system. The message system acts as a message broker for a Basilisk simulation. The messaging system employs a publisher-subscriber message passing nomenclature. A single Module may read and write any number of messages. A Module that writes output data, registers the 'publication'

of that message by creating a new message entry within the message system. Similarly, a Module that requires data output by another Module subscribes to the message published by the other Module. The messaging system then maintains the messages read and written by all Modules and the network of publishing and subscribing Modules.

A message is defined by a unique message name, a message ID and a payload data structure (typically a C/C++ struct). The messaging system maintains meta-data for each message in a message header. The message header meta-data includes a list of allowed message publishers, subscribers, buffer memory locations and read and write statistics.

The messaging system implements the message storage as directly managed memory. As shown in Fig. A.3(a) a region of memory is allocated and managed as a message storage container. The messaging system manages multiple storage containers, one for each Task Group. The size of the allocated memory for each storage container is determined by the combined size of the number of created messages, their associated headers and the number of message buffers allocated for each message. It is important to note, that all messages are at least double buffered in the messaging system. Ring buffer logic is used for message entries that are registered with more than two buffers. Multiple buffers per message entry helps to protect data integrity during message writes and facilitates the, albeit rare, use case in which two Modules must write a single message or when Basilisk operates in a multi-process/threaded configuration. However, as shown in Fig. A.3(a), a Module can declare to increase the number of buffers for a specific message.

A message is created in the message system when a Module invokes the function call, shown in Listing A.2, on the SystemMessaging singleton instance. This function call takes a unique message name, the maximum size in bytes of the message payload struct, the number of buffers into which an entry of the message may be written, the type of message payload struct and the unique identifier of the Module creating the new message. As demonstrated by Fig. A.3(b), the memory allocated in the Task Group's message storage container is increased and existing message entries are moved within the allocated memory to accommodate the new message 'msg n + 1'. A Module 'creates' a message in the message system by passing a message payload type and a unique



(a) Message system memory layout

(b) Message system memory layout upon new message creation

Figure A.3: Basilisk messaging system memory layout and organization.

name and receives in return a unique message identifier generated by the message system.

At simulation initialization a three stage process resolves the message subscription and publication pairs. Simulation initialization and the associated resolving of message pub-sub pairs is discussed in greater detail in Section. A.3. However, the functionality of a Task Group Interface (a unidirectional message 'bridge' from one Task Group to a second Task Group) is described here. Each Task Group has a single associated message storage container. This one-to-one design seeks to accommodate simulation configurations where the dynamic and environment Modules remain wholly separate from the flight software Modules. This separation, while being useful to organize related Modules within a simulation, becomes significantly useful when operating Basilisk as a distributed simulation across multiple compute resources. For example in a SWIL configuration the dynamics and environment Module's execute on a desktop computer while the FSW Modules execute on a separate flight target processor or processor emulator. However, there are further less stereotypical instances in which a simulation developer would like for messages in one Task Group to be available to Modules in a second Task Group. As a result, to facilitate the exchange of messages between Task Groups, Task Group Interfaces are available to make this connection. A Task Group Interface is a unidirectional message exchange from one Task Group to a second Task

Listing A.2: Register a new message with the messaging system.

```
uint64_t msgId SystemMessaging::CreateNewMessage(
    std::string messageName,
    uint64_t maxSize,
    uint64_t numMessageBuffers,
    std::string messageStruct,
    int64_t moduleID)
```

Group. This allows for Modules in a first Task Group to publish messages to a second Task Group and, as implied, Modules in the second Task Group to subscribe to messages published in the first Task Group.

A.2.3 Dynamics Manager

The third and final piece of Basilisk's modular design is the implementation of the Dynamics Manager. The spacecraft dynamics are modeled as fully coupled multi-body dynamics with the generalized EOMs being applicable to a wide range of spacecraft configurations. The implementation, as detailed in Reference [2], uses a back-substitution method to modularize the EOMs and leverages the resulting structure of the modularized equations to allow the arbitrary addition of both coupled and uncoupled forces and torques to a central spacecraft hub.

A Module which impacts the translational or rotational dynamics is called an Effector. Effectors are classified as either a State Effector or a Dynamic Effector. State Effectors are those Modules which have dynamic states to be integrated and therefore contribute to the coupled dynamics of the spacecraft. Examples of State Effectors are reaction wheels, flexible solar arrays, variable speed control moment gyroscopes (VSCMGs) and fuel slosh. In contrast, Dynamic Effectors are Modules which implement dynamics phenomena that result in external forces or torques being applied to the spacecraft. Examples of Dynamic Effectors include gravity, thrusters, solar radiation pressure (SRP) and drag.

For a Module to operate as either a State or Dynamic Effector, the implemented Module

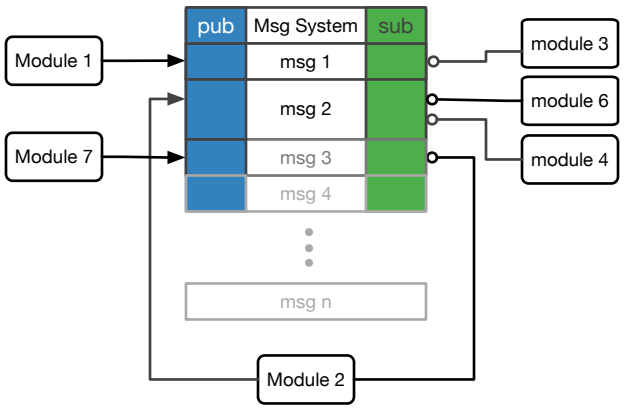


Figure A.4: A notional messaging system publish and subscribe map for a message storage container of a single Task Group.

class must inherit from the StateEffector or DynamicEffector parent classes. The developer of a dynamics Module is responsible for implementing only the dynamics of the Effector model. For a State Effector a developer must provide a custom implementation of the three functions shown in Listing A.3. Listing A.4 shows the single method for the non-coupled Dynamic Effector that a developer must override.

The Dynamics Manager transparently organizes and aggregates the various dynamic contribution of each Effector Module in a simulation. It ensures all dynamic states are updated and propagated. The user may select from various numerical integration schemes to propagate the spacecraft dynamics. Moreover, the interface between the Dynamics Manager and the integrator has been generalized to allow other developers to implement their own desired numerical integration scheme.

A.3 Execution Control

A Basilisk simulation steps through a number of distinct initialization, integration, and shut down phases. The high level flow of control for a Basilisk simulation is shown in Figure. A.5. Basilisk Modules, Tasks, Task Groups, and their associated message storage and linkages are initialized by a three stage process. Each Basilisk Module inherits from the `SysModel` class. As shown in Listing. A.5, the `SysModel` abstract class defines an interface of four functions, which a Module

Listing A.3: StateEffector required methods.

```
// Provide contributions to the spacecrafts mass and inertia properties.
virtual void updateEffectorMassProps(double integTime);
// Provide coupled contributions to the back-substitution matrices.
virtual void computeStateContribution(double integTime);
// Compute the Module's own state derivatives.
virtual void computeDerivatives(double integTime,  Eigen::Vector3d rDDot_BN_N, Eigen::Vector3d
    omegaDot_BN_B, Eigen::Vector3d sigma_BN);
```

must implement. These functions are called on each Module as part of the overall simulation flow of control process.

The three stages of simulation initialization are self-initialization, cross-initialization and reset. During self-initialization each Module's `selfInit()` function is called allowing a Module to register the messages it intends to publish with the messaging system. Next, each Module's `crossInit()` function is called allowing a Module to subscribe to messages that were made available as published messages in the previous self initialization stage. As the last major step before beginning the run-loop, `Reset()` is called for each module. The `Reset()` function provides each module an opportunity to setup to a 'clean' known initial state.

The simulation flow of control is governed by three loop iterations. The outermost loop iterates through each of the instantiated Task Groups according to each Task Group's assigned priority level. Within each Task Group, each Task is looped through. Subsequently within each Task, all Modules within a task are iterated through according to their priority within the Task. For each Module in a Task the Module's `updateState()` function is called. The logic contained

Listing A.4: DynamicEffector required method.

```
// Compute the body or inertial frame force and/or torque due to the Effector.
virtual void computeForceTorque(double integTime);
```

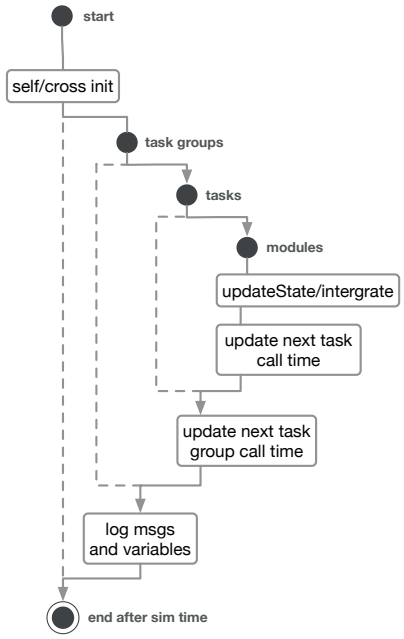


Figure A.5: Basilisk high level flow of control for simulation execution.

in the `updateState()` function is custom to each Module. However, a typical sequence of many `updateState()` implementations is to read subscribed input message, perform a computation defined by the Module and then write published output messages for use by other Modules. Of particular importance is the special `SpacecraftDynamics` Module which implements the aforementioned Dynamics Manager. The `updateState()` of the `SpacecraftDynamics` Module is responsible for triggering the dynamics integration process and in doing so determines the integration rate of the spacecraft dynamics.

Following the iteration through each of the Task Group and Task loops, the next call time for a Task and Task Group are set. This is required because each Task within a Task Group may have a different update rate and Tasks may be enabled or disabled at various times during the simulation. As a result, the next call time for a Task and Task Group and therefore the modules can change from one loop to the next loop and updating the next call time allows the simulation to skip forward to the next expected update time according to the combined Task update rates.

Listing A.5: SysModel abstract class interface definition.

```
virtual void SelfInit();
virtual void CrossInit();
virtual void UpdateState(uint64_t currentSimNanos);
virtual void Reset(uint64_t currentSimNanos);
```

A.4 Data Logging

Data output by Modules through messages or internal Module variables (which have a declared `public` scope in their C++ class definition) may be logged. Data to be logged is determined prior to a simulation run where a user may specify complete messages, a single variable within a message or internal simulation variables to be logged and the logging rate desired. The highest logging frequency is driven by the highest frequency at which the Task, containing the Module producing the data, is executed. No interpolation is done for data logged at a frequency higher than the frequency at which data samples are produced. As shown in Fig. A.5, the simulation Data Logger reads the requested messages and variables at the end of each loop through all Task Groups. At the conclusion of the simulation the user may retrieve the data with each message and variable made available as a time stamped series. This returned data format may be directly used in post processing scripts developed in Python using tools like Numpy, Matplotlib and PANDAS.

A.5 Monte Carlo Capability

A key benefit of Basilisk's Python interface is the ability to take any simulation script and with minimal code changes configure that script as a Monte Carlo (MC) simulation. A Basilisk Monte Carlo simulation can be executed in a serial or parallel multi-processing fashion. When executing in parallel the MC simulation can be executed on multiple local CPU cores or in a parallel remote execution environment. Additionally, the MC functionality includes run-time generated variable dispersions, logging and saving of each MC iteration's simulation dispersed initial conditions and

resulting simulation data. The logged simulation data is made available in the portable Dataframes data structure from the PANDAS Python module.

Variable dispersions are built upon base Python implementations of scalar, vector, and tensor variable type dispersion classes. Currently Basilisk maintains uniform and normal dispersion for Cartesian variables, Euler angles, and Modified Rodrigues Parameter (MRP) descriptions[76]. However, each of these individual base dispersion can be inherited by a user's custom dispersion implementation allowing users to generate dispersions for variables with different physical bounds, variances and specific statistical distributions.

The initial conditions, including the dispersed variables and random number seeds are saved in a JSON file format for each MC run. This allows a user to rerun and examine closer, one or more, particular runs of interest from a MC simulation, with bit-for-bit repeatability.

Multi-process capability is a key benefit of the MC tools. The MC controller uses the Python Multiprocessing module to spawn and manage as many Python Basilisk simulation processes as the user or host machine allows. For example a computer with a 4 core CPU, each physical core with two virtual cores, will be used by the MC controller as a machine with 8 processors. The controller will launch 8 simulations at once and continue to provide simulations to the worker pool of processes until all simulation work is complete. Each simulation execution is handled individually with data logging, initial conditions and failures all logged for later analysis. Post processing of MC data makes use of the convenient PANDAS statistical and data manipulation functions. While single simulation plotting is done with the more traditional Matplolib package, plotting of large multi-gigabyte data sets is achieved using the DataShaders plugin to the Bokeh plotting library. This module employs a rasterized plotting approach to display plots after a few seconds of execution time and is capable of of plotting extremely large data sets.

A.6 Development Approach - Open Source

The initial motivation for the development of Basilisk was to support the design and development of the attitude determination and control system for an interplanetary spacecraft. The

intention was to use Basilisk as an early mission Phase A/B design and analysis tool, a flight algorithm verification and validation tool during latter Phase C, and finally as the space environment and dynamics simulator for HWIL and SWIL testing during Phase D. Basilisk has been utilized in all these mission phases. Basilisk's increasing utility has prompted the original development team at the Laboratory for Atmospheric and Space Physics (LASP) and the AVS Lab to make the project available as an open source project. Basilisk uses an Internet System Consortium (ISC) License which is a permissive software license simply requiring attribution and relinquishing the creator of liability [45]. It is anticipated that such a permissive license will help to encourage experimentation and contribution back to the main Basilisk project.

The Basilisk framework does not contain any export controlled components. Rather, all the included simulation and astrodynamics control algorithms are from open published literature. If a user needs to create modules that contain company proprietary tools or export controlled solutions, then the user would create these modules outside the regular Basilisk framework and import them separately in the Python simulation script. This allows Basilisk to model several common dynamical systems such as reaction wheels dynamics in a very general fashion, but no reaction wheel specific communication interfaces are included as these are vendor or mission specific. To use Basilisk for mission development and analysis purposes, this modularity allows for a clean separation of general purpose modules as well as user custom-developed modules.

Basilisk has undergone an internal verification and validation effort within LASP and the AVS Lab. As an open source project, Basilisk will benefit from the strong ongoing engagement from a wide community of users. The community shall provide further validation, bug fixes and functionality additions. In the short time that Basilisk has been openly available a number of fixes and functionality additions have been made by the user community.

A.7 Example of a Basilisk Simulation Configuration

Constructing a Basilisk simulation scenario requires the creation of Task Groups, assigning Tasks to these Task Groups, and the instantiation of Modules within each Task. The following

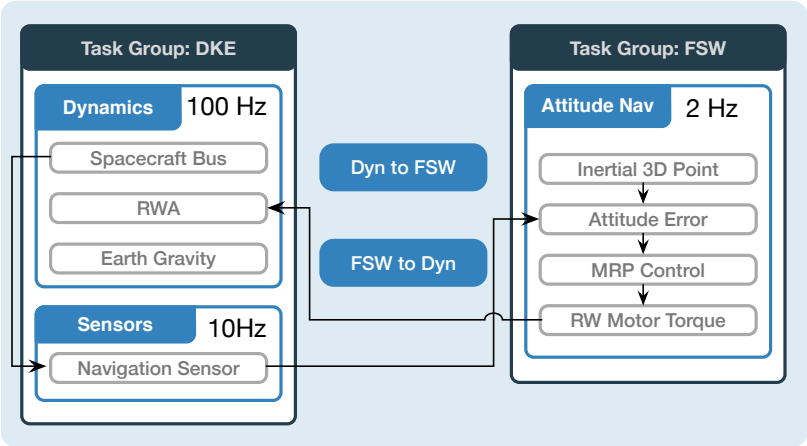


Figure A.6: Concept diagram of simple multi body gravity orbiter Basilisk simulation configuration.

example demonstrates the key Basilisk function calls which configures a simple Earth orbiting spacecraft whose attitude control system must align to a chosen inertial attitude. The simulation uses two Task Groups. The first Task Group contains all dynamics, kinematics and environment (DKE) Modules. The second Task Group contains all flight software algorithm Modules. This example simulation configuration closely exemplifies a Basilisk orbital simulation developed for SRP analysis in Chapters. 4 and 3. The arrangement of Task Groups, Tasks and associated Modules is presented in Fig. A.6. The Modules within the DKE Task Group are separated into two Tasks. In the first Tasks the Modules included are spacecraft hub, reaction wheels (fully coupled to the hub) and the Earth gravity field. In the second Task a Module called SimpleNav is included. It receives the spacecraft states through the output message of the spacecraft module. The SimpleNav Module perturbs the truth state of the spacecraft using a gauss-markov error model. For simple simulations, such as this example, the SimpleNav Module is used in place of the more complex nominal spacecraft navigation system output. There is a single Task in the FSW Task Group and it contains all the Modules required to implement a MRP inertial pointing controller using reaction wheels [76]. The FSW determines the attitude error by reading the navigation sensor output message, while the reaction wheel motor torque module outputs a message that drives the resulting reaction wheel assembly (RWA) dynamics.

Task Groups and Tasks are created and linked for the DKE and FSW Task Groups. This

is done by creating a Task Group, also referred to as a Process, and then adding a Task to this Task Group as shown in Listing. A.6. The Dynamics Task integration rate is set to 0.1 seconds and the Sensors Task rate set to 0.5 seconds. In Basilisk, the base time scale is nanoseconds and so the sec2nanos() conversion utility is used for convenience. Recall that Task Groups are message containers. Messages within a Task Group may only be published and subscribed to by Modules within that Task Group. To facilitate message passing between Task Groups, a Task Group interface must be created for each of the desired message flow directions. In this simulation, it is desired that certain messages generated in the DKE Task Group are available to the Modules in FSW and vice-versa. As shown in Figure. A.6, two Task Group interfaces are created to facilitate this transparent message exchange between the two Task Groups.

With the core Basilisk structures instantiated, the next step is to populate the simulation with various Basilisk Modules. As shown in Listing. A.7, the DKE Modules are instantiated and assigned to their respective Tasks. The `SpacecraftPlus()` Module instantiates the rigid body hub to which other `StateEffectors` and `DynamicEffectors` can be associated.

The FSW Modules are created and populated in the FSW Task Group as shown in Listing. A.8. While Modules can be development in either Python, C++ or C, the FSW modules employed in this simulation are developed in C. Developing these modules in C allows analysts to run the same code and algorithm in simulation, SWIL and eventually HWIL. The FSW Modules included are:

- `Inertial3dPoint` - compute the spacecraft reference attitude.
- `AttitudeError` - determine the spacecraft attitude error from the reference attitude.
- `MRPControl` - compute required control torques according to MRP based feedback control law.
- `RWMotorTorque` - map the attitude control torque onto a set of reaction wheel torque commands.

Listing A.6: Simulation, Task Group, Tasks, and Task Group message sharing interface instantiation.

```
# Instantiate simulation container
scSim = SimulationBaseClass.SimBaseClass()

# Create Task Groups (Processes)
dynProcess = scSim.CreateNewProcess("dynProcess")
fswProcess = scSim.CreateNewProcess("fswProcess")

# Create and add Tasks to each Task Group
dynProcess.addTask(scSim.CreateNewTask("dynTask", sec2nanos(0.01)))
dynProcess.addTask(scSim.CreateNewTask("sensorTask", sec2nanos(0.1)))
fswProcess.addTask(scSim.CreateNewTask("fswTask", sec2nanos(0.5)))

# Create interfaces and define message sharing directionality
intDynToFsw = sim_model.SysInterface()
intFswToDyn = sim_model.SysInterface()
intDynToFsw.addNewInterface("dynProcess", "fswProcess")
intFswToDyn.addNewInterface("fswProcess", "dynProcess")

# Add interfaces to Task Groups
dynProcess.addInterfaceRef(intDynToFsw)
fswProcess.addInterfaceRef(intFswToDyn)
```

Listing A.7: Instantiate DKE Modules and assign to respective Tasks.

```
# Create spacecraft hub effector
scObject = spacecraftPlus.SpacecraftPlus()
scSim.AddModelToTask("dynTask", scObject, None, 1)
# Create earth gravity body DynamicEffector
gravBodies = gravFactory.createBodies(['earth'])
scObject.gravField.gravBodies = spacecraftPlus.GravBodyVector(gravFactory.gravBodies.values())
# Create reaction wheel StateEffectors
RW1 = rwFactory.create('Honeywell_HR16', [1, 0, 0], maxMomentum=50., Omega=100.)
RW2 = rwFactory.create('Honeywell_HR16', [0, 1, 0], maxMomentum=50., Omega=200.)
RW3 = rwFactory.create('Honeywell_HR16', [0, 0, 1], maxMomentum=50., Omega=300.)
# Add reaction wheel StateEffectors to spacecraft object and Task
rwStateEffector = reactionWheelStateEffector.ReactionWheelStateEffector()
rwFactory.addToSpacecraft("ReactionWheels", rwStateEffector, scObject)
scSim.AddModelToTask("dynTask", rwStateEffector, None, 2)\
```

The novel utility of Basilisk's modularity is demonstrated by the arrangement these FSW algorithm Modules. Each of these FSW Modules computes a specific kinematic or control related quantity. As such each Module can be used as a building-block to compose complex FSW behaviors. In this simulation scenario four Modules are used to create an inertial pointing control scheme. The output data generated and the input data required by these Modules is facilitated by published and subscribed messages. Greater detail of the application and theory enabled by this building-block approach is contained in Reference [18].

To begin the simulation three function calls are made. The first initializes the Task Groups, Tasks and Modules by calling the `SelfInit(), CrossInit()` and `ResetInit()` functions. Following this, the simulation stop time is set and then the simulation is launched.

External changes to the simulation configuration can be made conditionally triggered by Events or more simply after a set duration of execution as demonstrated in Listing. A.10. This is useful to simulate specific spacecraft sequence instruction sets and FSW mode changes. Events are available and can be configured to trigger a custom user provided function. This user provided function allows an analyst to trigger and change any variable/state in the simulation that is available through the Python interface of each Basilisk Module.

Plots are created from the simulation generated data using numpy, Matplotlib and PANDAS packages. For the presented simulation the evolution of the spacecraft attitude is shown in Figure. A.7. It is evident that the spacecraft controls to the reference attitude with convergence achieved after eight minutes. Figure. A.8 shows the computed control torques and the resulting actuated reaction wheel control torques. In this simulation each reaction wheel's maximum available torque has been set as 0.2 [Nm]. As shown, reaction wheels 2 and 3 saturate their actuated torque early in the simulation. Finally, the resulting reaction wheel control speeds are shown in Figure. A.9.

Listing A.8: Instantiate FSW Modules and assign to respective Task.

```
# Setup the attitude determination Module.
inertial3DConfig = inertial3D.inertial3DConfig()
inertial3DWrap = scSim.setModelDataWrap(inertial3DConfig)
scSim.AddModelToTask("fswTask", inertial3DWrap, inertial3DConfig)

# Setup the attitude tracking error evaluation Module.
attErrorConfig = attTrackingError.attTrackingErrorConfig()
attErrorWrap = scSim.setModelDataWrap(attErrorConfig)
scSim.AddModelToTask("fswTask", attErrorWrap, attErrorConfig)

# Setup the MRP Feedback control Module.
mrpControlConfig = MRP_Feedback.MRP_FeedbackConfig()
mrpControlWrap = scSim.setModelDataWrap(mrpControlConfig)
scSim.AddModelToTask("fswTask", mrpControlWrap, mrpControlConfig)

# Setup the control torque to RW torque translation Module.
rwMotorTorqueConfig = rwMotorTorque.rwMotorTorqueConfig()
rwMotorTorqueWrap = scSim.setModelDataWrap(rwMotorTorqueConfig)
scSim.AddModelToTask("fswTask", rwMotorTorqueWrap, rwMotorTorqueConfig)
```

Listing A.9: Launching a simulation.

```
scSim.InitializeSimulation()
scSim.ConfigureStopTime(simulationTime)
scSim.ExecuteSimulation()
```

Listing A.10: Spacecraft mode changes made after the simulation executes for a specified duration.

```
scSim.ConfigureStopTime(sec2nanos(20))
scSim.ExecuteSimulation()
# Command the FSW to go into safe mode and advance to ~ periapsis
scSim.modeRequest = 'safeMode'
scSim.ConfigureStopTime(sec2nanos(60))
scSim.ExecuteSimulation()
# Command the FSW to go into Nav only mode
scSim.ConfigureStopTime(sec2nanos(60 * 11 * 1 + 30))
scSim.modeRequest = 'navOnly'
scSim.ExecuteSimulation()
```

A.8 Conclusion

The Basilisk astrodynamics framework provides a new open source alternative for fully coupled spacecraft dynamics mission simulation with integrated flight algorithm emulation. Among the suite of other available simulation tools, Basilisk provides an enabling mix of usability, extensibility and computational speed. Basilisk is able to achieve this usability by providing a Python user interface for each Basilisk component. The Python interface enables users to leverage the depth of the Python math and data analysis package ecosystems. Basilisk's modular architecture of Modules, Tasks, Task Groups, and the Messaging system supports this usability by enabling users to configure simulation scenarios from the very simple early feasibility analysis to complex mission verification and validation.

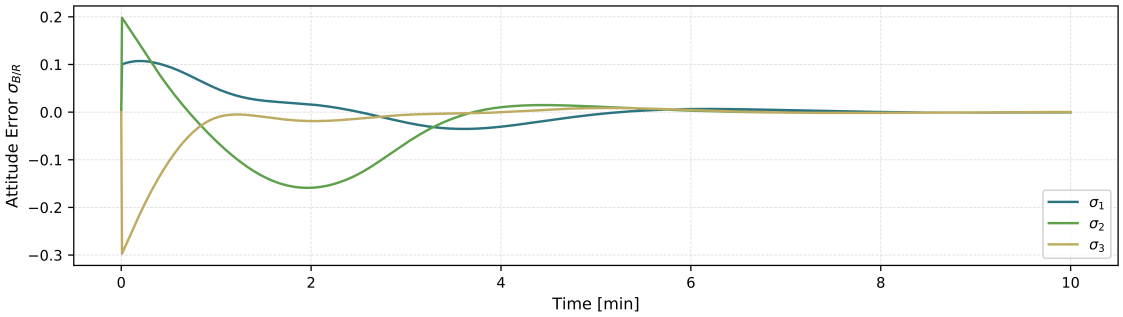


Figure A.7: Evolution of attitude error in each MRP component.

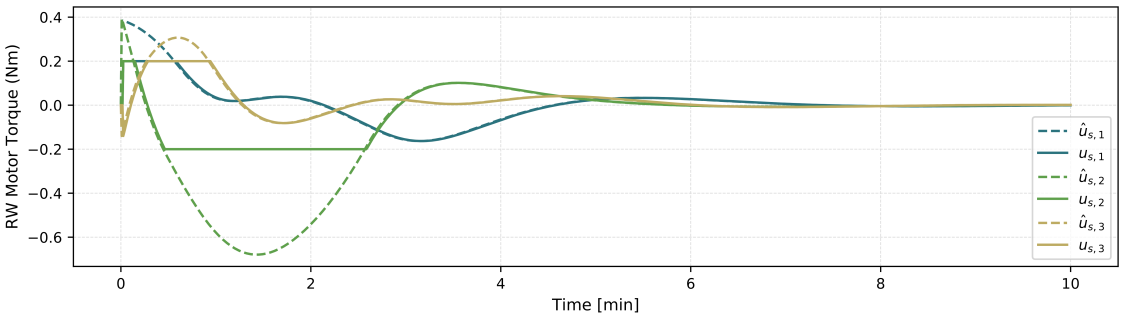


Figure A.8: Evolution of computed reaction wheel torques (dashed) and the actual reaction wheel torques.

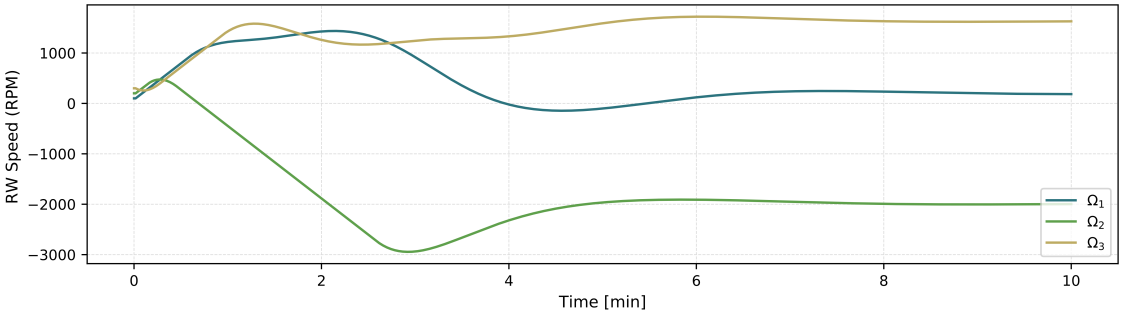


Figure A.9: Evolution of reaction wheel speeds.