




# Open GL–Open CL Solar Radiation Pressure Modeling with Time-Varying Spacecraft Geometries

Patrick Kenneally\* and Hanspeter Schaub<sup>†</sup>   
University of Colorado, Boulder, Colorado 80303  
and  
Sergei Tanygin<sup>‡</sup>  
Independent Researcher, Long Beach, CA 90803

<https://doi.org/10.2514/1.1010869>

**A method for the fast computation of spacecraft force and torque due to solar radiation pressure (SRP) is presented. A faceted model is employed that tracks which elements of a time-varying geometry are exposed to the sunlight, but sunlight reflections are not modeled. The method uses the highly parallel execution capabilities of commodity graphics processing unit (GPU) and the Open Graphics Library (OpenGL) and Open Compute Language (OpenCL) to render a spacecraft mesh on the GPU. A custom-developed OpenGL render pipeline computes the per-model facet SRP forces and torques that are summed on the GPU before the resultant spacecraft force and torque vectors are copied back to the CPU bound process. The process is validated on spherical and cubic test shapes. The evaluation accommodates spacecraft self-shadowing and is capable of accounting for arbitrary spacecraft articulation. Material properties are encoded with the model to provide realistic specular, diffuse, and absorption surface light interactions. Numerical simulations illustrate the impact of geometric fidelity and articulated surfaces. The faceted OpenGL-OpenCL method is up to an order of magnitude faster on integrated GPU hardware than high-end graphics card as the process is not demanding on the GPU and benefits from the fast memory transfer of on-chip processors.**

## I. Introduction

**E**XPLORING novel mission concepts requires rapid numerical modeling of the spacecraft dynamics. The solar radiation pressure (SRP), the momentum imparted to a body by impinging solar photons, becomes a dominant nonconservative force above low Earth orbit (LEO) regime [1]. Given this importance of SRP, knowledge of the resultant forces upon a body due to SRP is a primary consideration in the modeling and analysis of spacecraft operating above the LEO region [2,3].

The video game and animation industries have driven the pursuit to create more vivid and realistic artificial worlds. This pursuit has resulted in highly optimized vector graphics software and graphics processing unit (GPU) computer hardware capable of carrying out many thousands of floating point operations in parallel [4]. Although these artificial worlds are visually persuasive, their implementation of electromagnetic radiation physics is understandably inaccurate. However, it is the parallel hardware and efficient vector graphics software implementations that may be used to simplify the steps of the SRP computation with great effect.

The ability to model and compute, at orders of magnitude faster than real-time, the SRP forces and torques on flexible and time-varying spacecraft structures presents compelling opportunities. Current SRP evaluation approaches are capable of modeling the resultant force of an articulated spacecraft where the articulation motion is known before evaluation [5]. However, there are many instances in which the articulation motion and the spacecraft state are dependent on the myriad spacecraft control inputs and constraints. Accounting for all possible permutations of the spacecraft dynamic

state is further challenged by the inclusion of flexing in large spacecraft structures. Further, the process of precomputing the SRP forces for all configurations can be a time-consuming initial task that does not lend itself well to mission design scenarios where multiple spacecraft configurations and scenarios are being considered.

It is evident then that a method of SRP evaluation characterized by an ability to include time-varying information of the spacecraft state has potential for a wide range of applications. Effective modeling of the SRP-induced perturbation of a spacecraft enables mission designers to consider SRP a valuable actuator rather than a disturbance. Such a novel use of the SRP force in maneuver and mission design is exemplified by the MERcury Surface, Space ENvironment, GEo-chemistry and RANGing (MESSENGER) mission. The MESSENGER mission designers employed a solar sailing technique to perform each trajectory change maneuver (TCM) and accurately target each of the mission's six planetary flyby maneuvers. Typically TCMs are performed using onboard thrusters. However, using SRP as the TCM actuator allowed the MESSENGER team to perform TCMs with more accuracy and finer control due to the smaller magnitude of the SRP-induced  $\Delta V$  [6]. Additionally, the MESSENGER team was able to reduce fuel and related structural accommodations in the spacecraft design to reduce overall mission cost [7].

Modeling the spacecraft SRP-induced force and torque with high geometric fidelity 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 in real-time with a simulation environment, spacecraft self-shadowing, and time-varying arbitrary material optical properties. Typically spacecraft geometries are kept simple, ignoring important spacecraft detail that has a significant degradation of a model's ability to more closely evaluate the true SRP force and torque. Further, methods that do capture changes in spacecraft articulations and self-shadowing do so as part of an offline evaluation that generates SRP force and torque lookup tables. Such offline evaluations are executed multiple times to accommodate the myriad different spacecraft configurations.

A survey of the current landscape of SRP research reveals a variety of approaches. The most basic model with regard to the analytic development is referred to as the cannonball model. It is often the case that the coefficient of reflection parameter is continually estimated

Received 6 June 2020; revision received 4 January 2021; accepted for publication 14 January 2021; published online 16 February 2021. Copyright © 2021 by the authors. Published by the American Institute of Aeronautics and Astronautics, Inc., with permission. All requests for copying and permission to reprint should be submitted to CCC at [www.copyright.com](http://www.copyright.com); employ the eISSN 2327-3097 to initiate your request. See also AIAA Rights and Permissions [www.aiaa.org/randp](http://www.aiaa.org/randp).

\*Graduate Research Assistant, Aerospace Engineering Sciences Department.

<sup>†</sup>Professor, Glenn L. Murphy Chair, Aerospace Engineering Sciences Department, Fellow AIAA.

<sup>‡</sup>Associate Fellow AIAA.

and updated by an orbit determination effort. This model is most notably used for the Laser Geodynamics Satellite or LAGEOS missions and continues to prove useful for initial mission analysis [8]. Increased modeling accuracy is often achieved by departing from the cannonball assumption and defining shape approximations of the spacecraft. A common shape approximation is to model the spacecraft bus and solar panels as a box and panels, respectively. Additionally, the individual reflection, absorption, and emission characteristics are kept distinct for each surface and set based on known spacecraft material properties [9]. However, common among shape approximation methods is that much of the modeling uncertainty occurs in an estimation process within the second step of the SRP evaluation. It is the model's computation, the second step of the process, in which much work is being done. Notably Ziebart details an evaluation procedure that requires precomputation of the body forces over all  $4\pi$  steradian attitude possibilities [10]. Ziebart's approach is also capable of modeling self-shadowing by using ray-tracing techniques and spacecraft re-radiation via reduced spacecraft thermal model. McMahon and Scheeres extend such a model by aggregating the resultant SRP forces into a set of Fourier coefficients of a Fourier expansion [9]. The resulting Fourier expansion is used for both online and offline SRP 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 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 [11]. Reference [12] discusses initial results exploring a GPU method of evaluating the SRP forces using a faceted spacecraft model. Tichy et al. use OpenGL, a vector graphics GPU software interface common in video games, to dynamically render the spacecraft model and evaluate the force of the incident solar radiation across a spacecraft structure approximated by many thousands of facets [13].

This paper studies the GPU-based numerical SRP modeling that addresses the three aforementioned modeling requirements. In this study the light interaction with each facet is accounted for individually and not in a coupled manner through reflection. The prospect of this faceted SRP method is a very fast computational evaluation. Although the application is for SRP force evaluations, the techniques presented can readily be applied to atmospheric drag force evaluation. Here the concern is also that facets are visible to the freestream for a given mesh geometry and what is the resulting force per facet. The OpenGL-CL modeling method is explored to yield an implementation that has a computational speed suitable for online execution. The technical challenge is how to implement the SRP evaluation on a time-varying spacecraft mesh while still benefiting from the massively parallel GPU evaluation pipelines. Of interest is how modern hardware such as on-chip and dedicated GPUs performs in evaluating the SRP. The on-chip GPUs are now a commodity hardware and readily available. The approach builds upon the author's previous OpenGL-faceted-based approaches [14] and extends the application of Open Compute Language (OpenCL) to allow for more flexible arbitrary computation. Additionally, the OpenGL-CL approach has parallels to the earlier work presented by Tanygin and Beatty in Ref. [12] and incorporates certain algorithmic decisions made by that work. Note that inherent with this non-ray-tracing method is that reflectance [15,16] and complex surface properties [17] are not included. These influences can contribute to 10–20% of the SRP evaluation. The benefit of the presented faceted SRP evaluation is that rapid prototyping applications are enabled, as well as integration with complex hardware-in-the-loop simulations where the speed of the dynamics evaluation tasks is critical.

The paper is outlined as follows. To start, the OpenGL Application Programming Interface (API) is introduced in the context of the render pipeline. This is followed by a description of the important OpenGL–OpenCL shared memory context functionality. An optimized OpenCL kernel is developed to perform a parallel reduction 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 that demonstrate the method's capability to capture the difference between spacecraft mesh models, while comfortably accommodating detailed meshes of many thousands of vertices.

## II. OpenGL Render Pipeline

The Open Graphics Library (OpenGL) is a language-independent API for rendering computer vector graphics (<https://www.khronos.org/opengl/>). 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 that 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 Fig. 1 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 fragment (rasterized pixel). Each of the vertices or primitives is 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 Fig. 2. A minimally valid OpenGL pipeline requires the implementation of at least the vertex shader (VS) stage. The addition of further shader stages allows the software developer to create a custom render pipeline. The VS 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 (FS) computes per pixel operations following OpenGL's internal depth testing and rasterization processes. The two shader stages leveraged in this approach are the VS and FS stages.

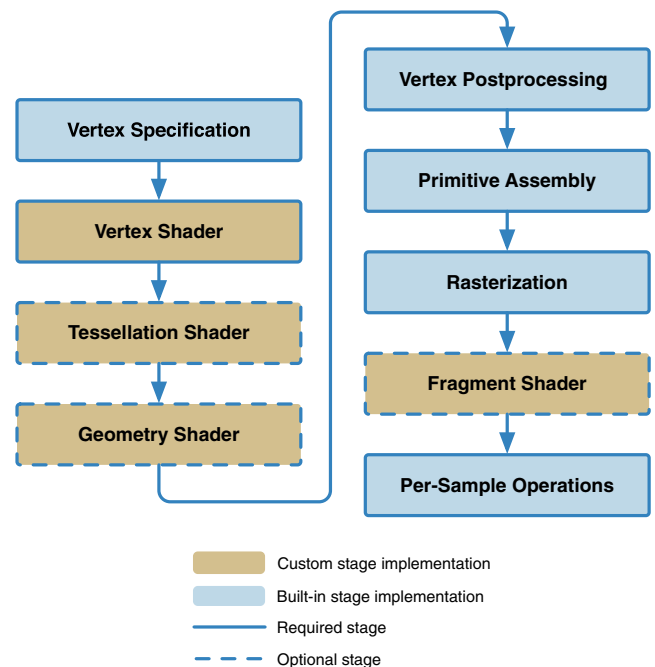


Fig. 1 The default OpenGL pipeline.

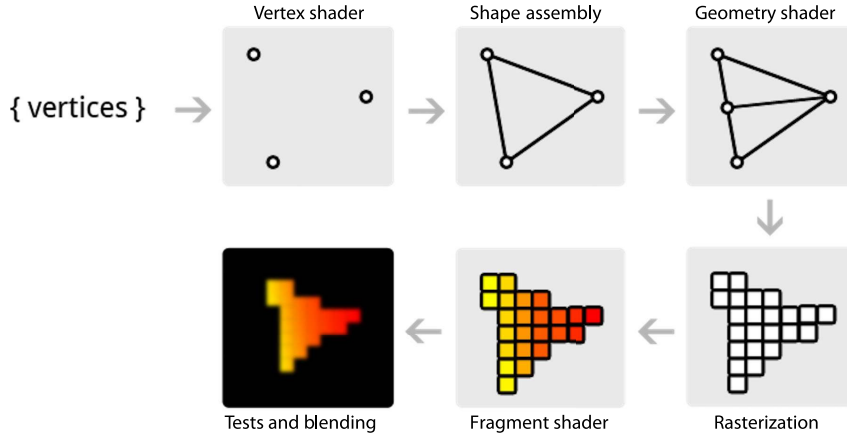


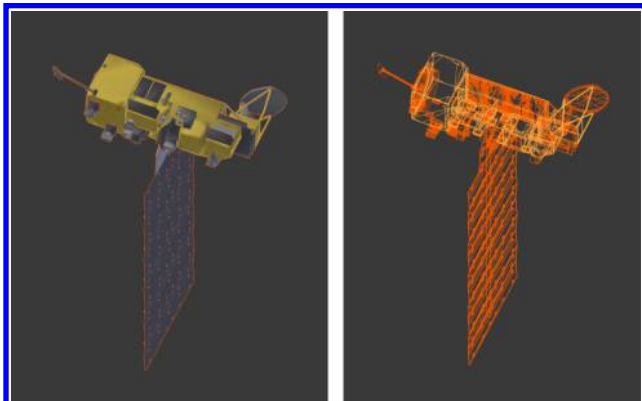
Fig. 2 Notional operations for each shader stage.

*Vertex shader:* The 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 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 Fig. 1 the VS is a programmable and required stage in the pipeline.

*Fragment shader:* The 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 single-precision floating point values per pixel. RGBA stands for red, green, blue and alpha channels, where the alpha channel controls transparency. Typically an RGBA value is output to a texture for each pixel. For a typical render it is the color texture that is displayed to the screen.

III. Mesh Definition

A triangulated mesh model is used 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 that handles a multitude of other primitive types. The model data format chosen is the Wavefront Object (.OBJ) (<http://paulbourke.net/dataformats/obj/>). 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 3a and 3b show the .OBJ mesh model of the Aqua spacecraft with complete materials and only mesh structure, respectively.



a) Aqua spacecraft with materials b) Aqua spacecraft mesh

Fig. 3 Aqua spacecraft .OBJ mesh model.

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 faceted SRP modeling the .MTL file is overloaded and a number of its 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 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.

IV. Custom OpenGL Render Pipeline

The custom OpenGL pipeline developed here builds upon Ref. [12], which employs 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

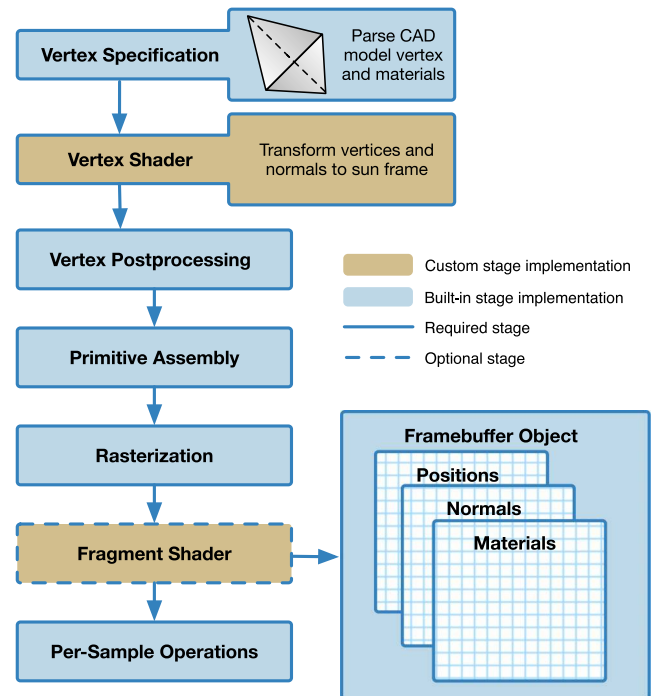


Fig. 4 Overview of OpenGL render pipeline with required custom vertex shader and fragment shader stages outputting to the textures held in a framebuffer object.

method implements custom VS and FS stages. An overview of this custom pipeline is shown in Fig. 4. The VS stage transforms mesh vertices to a projection frame and outputs the normal, position, and material parameters associated with the processed vertex. The pipeline depth testing and rasterization determines which vertex values are sunlit, and 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 a framebuffer object (FBO). An FBO allows a developer to define a nondefault destination for render output. Results from the FS stage, shown in Fig. 4, are written into the FBO's attached textures. Developers can manipulate the active FBO and the texture storage attached to the FBO.

The FBO is a destination for specific data types generated during the render process. To store these 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 four-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. The FBO and associated texture data structures are a key enabling portion of the OpenGL API that allows for the passing of OpenGL generated data to the subsequent OpenCL SRP computation. Data sharing between the OpenGL and OpenCL processes is made possible by declaring a shared data context between the initialized OpenGL and OpenCL computing contexts. This allows the two contexts to read and write to the same memory, where the OpenGL process writes data to memory that the OpenCL process then directly reads. Specifically, 2D textures, into which the spacecraft mesh model vertices, normal vectors, and material optical properties are written, are ultimately passed to the OpenCL SRP computation.

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`, whereas vector-based data are written to any one of a number of `COLOR_ATTACHMENTi` attachment points, where  $i$  is the index of the attachment point.

## V. 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 that 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 phase 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.

### A. 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  $\mathcal{B}$ :  $\{\hat{x}, \hat{y}, \hat{z}\}$ . 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 comprises multiple submeshes, 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 although individual submeshes may be articulated and therefore move within the model body frame, the vertices within each submesh are fixed. In other words, all submeshes in a model are rigid bodies. Most spacecraft with time-varying geometries satisfy this assumption, except for maybe solar sails. This constraint allows for the computation of an AABB for each submesh once at the start of a simulation. The AABB can now be computed recursively by computing the AABB of the vertices that define the corners of each submesh'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 submeshes),  $10^2$ .

### B. 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 of submeshes, each of which defines separate components of the spacecraft as shown in Fig. 5. Segmenting a spacecraft model into multiple submeshes 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 submesh. 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 submesh. The second reason for using submeshes is to facilitate arbitrary run-time articulation of spacecraft components such as solar panel structures, antenna, and instruments. Submeshes 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 submesh, and the transformation is applied to the mesh, thus performing the articulation.

All mesh vertex and normal data are defined with respect to the model frame body coordinate system  $\mathcal{B}$ :  $\{\hat{B}_1, \hat{B}_2, \hat{B}_3\}$ . However, it is

Algorithm 1: Recursive AABB algorithm

---

```

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) ← end ← tmpBoxVertices(8, 3);
11 end
12 node.bbox ← computeBBox(bboxUnion);
13 node.bbox.transform(node.transformation);
14 return

```

---

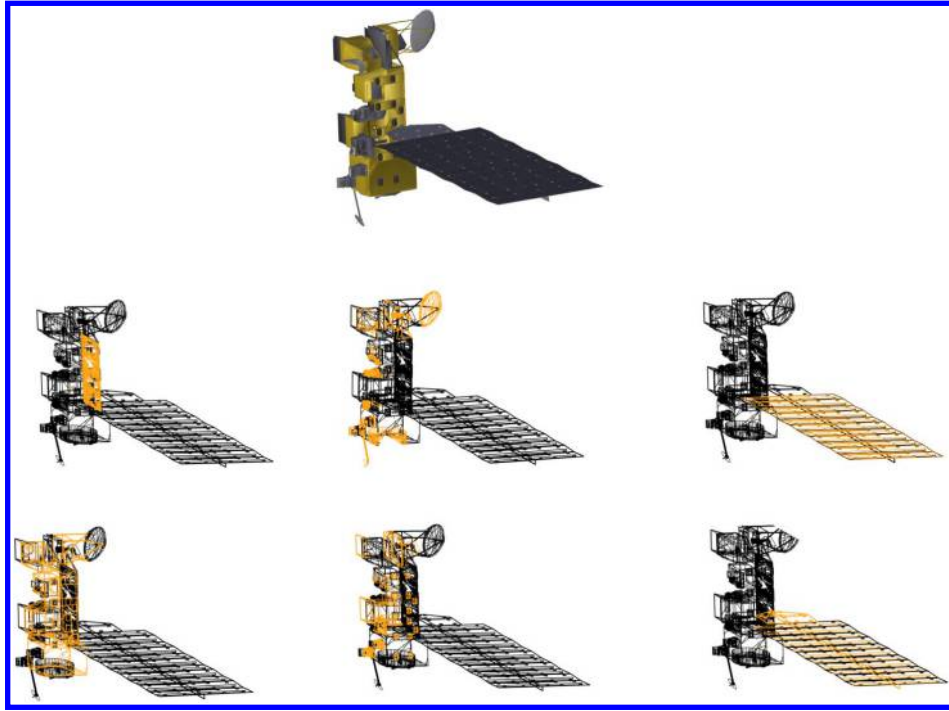


Fig. 5 Illustration of a range of submeshes (orange) used to model the Aqua spacecraft.

also convenient to allow the user to define submesh articulations with respect to a coordinate frame,  $C: \{\hat{C}_1, \hat{C}_2, \hat{C}_3\}$ , 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.

The transformation requires the definition of the origin and orientation of the submesh 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 rotation and translation:

$${}^B p = {}^B p_{B/C} + [BC]{}^C p \quad (1)$$

where  $[BC]$  is the direction cosine matrix (DCM) defining the orthogonal transformation of a  $C$  frame vector to the  $B$  frame [18]. Note that the left superscript denotes with respect to which frame the vector components are taken. This translation and rotation operation can be concisely expressed as a  $4 \times 4$  homogeneous transformation as

$$[BC] = \begin{bmatrix} [BC] & {}^B p_{C/B} \\ 0_{1 \times 3} & 1 \end{bmatrix} \quad (2)$$

Assuming that all mesh vertices are initially defined in the body frame, then the transformation described by Eq. (1) must be reversed. This is easily achieved by employing the inverse of the homogeneous transformation given as [18]

$$[BC]^{-1} = \begin{bmatrix} [BC]^T & -[BC]^T {}^B p_{C/B} \\ 0_{1 \times 3} & 1 \end{bmatrix} \quad (3)$$

This transform yields a vertex defined in the  $C$  frame as [18]

$${}^C p = [BC]^{-1} {}^B p \quad (4)$$

The homogeneous transformation matrix obeys the same successive transformation property as the direction cosine matrix as exemplified in the following transformation [18]. Here the inertial frame is defined as  $\mathcal{N}: \{x, y, z\}$ , and  $\mathcal{A}$  is a general intermediary frame.

$${}^{\mathcal{N}} p = [\mathcal{N}\mathcal{A}][\mathcal{A}\mathcal{B}]{}^{\mathcal{B}} p = [\mathcal{N}\mathcal{B}]{}^{\mathcal{B}} p \quad (5)$$

A result of this successive transformation property is that sequential frame definitions and the subsequent mappings from one mesh articulation frame to a submesh 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 submesh articulations where a parent submesh holds references to further child submeshes and the submesh transformation is defined relative to its parent mesh, rather than the body or inertial coordinate frames.

### C. 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:

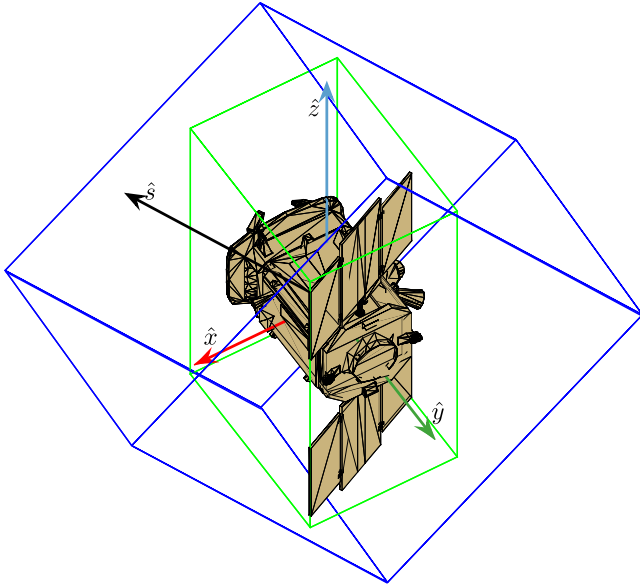
- 1) Body frame  $B$  to sun frame  $S$ ,  $[SB]$ ;
- 2)  $S$  frame to view frame  $V$ ,  $[VS]$ ;
- 3)  $V$  frame to projection frame  $P$ ,  $[PV]$ .

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

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 submesh 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 (eight 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 that lies between the sun and the model as illustrated in Fig. 6. The projection frame is constructed as an orthographic projection of the mesh model into this same plane.

Of the six sides of the loose sun frame bounding box, the centroid of the plane that has its normal as  $-{}^B \hat{s}$  is set as the vector  ${}^S e$  commonly referred to as the "eye" location. This is the position of the notional camera. The center of the bounding box is set as  ${}^S c$  and referred to as the camera target vector. It is necessary to set the target



**Fig. 6** Illustration of loose sun frame AABB (blue) and body frame AABB (green).

vector at the center of the loose sun frame AABB rather than the model body frame because this ensures that 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

$$\hat{f} = \frac{e - c}{|e - c|} \quad (6a)$$

$$\hat{u} = \hat{b}_2 \quad (6b)$$

$$\hat{s} = \hat{f} \times \hat{u} \quad (6c)$$

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

$$V = \begin{bmatrix} \hat{s}^T & 0 \\ \hat{u}^T & 0 \\ -\hat{f}^T & 0 \\ 0_{1 \times 3} & 1 \end{bmatrix} \quad (7)$$

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 where each of the three axes has a range of  $[-1, 1]$ . The orthographic projection matrix performs the mapping from view frame coordinates to NDC. The projection matrix  $P$  is constructed as

$$P = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (8)$$

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 VS. The body frame vertex is transformed to its submesh articulation frame as

$$\mathcal{R} p = [\mathcal{B}\mathcal{R}]^{-1} B p \quad (9)$$

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

$$\mathcal{R}' p = [\mathcal{R}'\mathcal{R}]^R p \quad (10)$$

and then transformed back to the body frame

$${}^B p = [\mathcal{B}\mathcal{R}']^{\mathcal{R}'} p \quad (11)$$

The VS next transforms the body frame vertex to NDC by

$${}^P p = [P][V][S][B]{}^B p \quad (12)$$

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 OpenGL 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, then the pixel represents a portion of the spacecraft's surface and the SRP force and torque computation continues; otherwise return a zero vector for force and torque.

## VI. 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 (the occurrences of conditional statements in code which result in alternate execution pathways), and remove unnecessary instruction overhead by unrolling loops. For each pixel the force computed as

$$F_{\odot_k} = -P(|r_{\odot}|)A_k \cos(\theta_k) \left\{ (1 - \rho_{s_k}) \hat{s} + \left[ \frac{2}{3} \rho_{d_k} + 2 \rho_{s_k} \cos(\theta_k) \right] \hat{n}_k \right\} \quad (13)$$

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

$$L_{\odot_k} = r_{P/C} \times F_{\odot_k} \quad (14)$$

where the position vector  $r_{P/C}$ , 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. Figures 7 and 8 provide a simplified example of the stages of this sequential addressing procedure. In this example there are two OpenCL work groups (WGs), each of which contains two work items (WI). At each iteration, a WI sums two values, where the values are denoted in Figs. 7 and 8 as

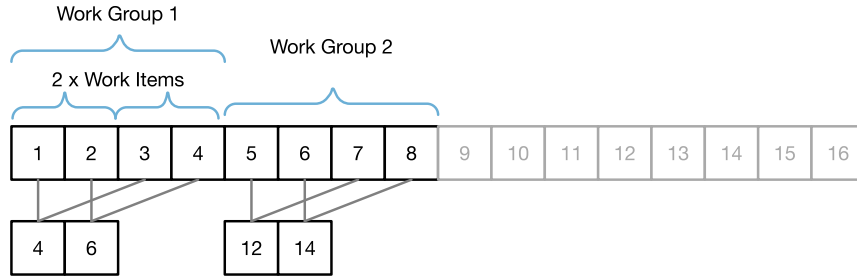


Fig. 7 Notional parallel reduction by summing sequenced addressing.

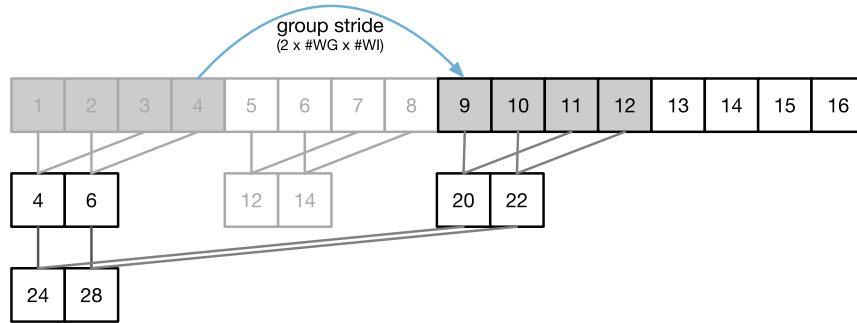


Fig. 8 Notional parallel reduction striding over all allocated addresses to continue summing sequential blocks of memory in a the `while()` loop.

an integer within a cell. As a result, summing cells one and three results in adding values  $1 + 3 = 4$ . After one iteration the number of values to sum has been reduced by half. The next iteration sums  $\{4, 6\}$  and  $\{12, 14\}$ , and so on. Although 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 Fig. 8. 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 1.

Additionally, time overhead is incurred by loop instructions. To avoid unnecessary instruction overhead, at the completion of a single WG 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 single instruction, multiple data, or Single Instruction, Multiple Data (SIMD) architecture by optimizing memory loads and scheduling of instructions given the fixed width of the unrolled loop iterations [19]. An

Listing 1 Abbreviated excerpt of the while loop in the OpenCL parallel reduction kernel

```

unsigned int i = group_id * group_stride + local_id;
while (i < texture_size)
{
    // compute x, y coords for lookup in square image map (texture)
    int y = i / tex_width;
    int x = i % tex_width;
    float4 nHat_B = read_imagef(normalsMap, coords);
    if (nHat_B[3] > 0) {
        // Perform position and material read_imagef and
        // compute force and torque for pixel
    }
    // If the mesh size is smaller than the group_size then we
    // have to stop trying to compute the second facet in the
    // parallel reduction because there will be no
    // more facets in the mesh.
    unsigned int secondPixelIdx = i + group_size;
    if (secondPixelIdx < textureSize) {
        float4 nHat_B_1 = read_imagef(normalsMap, coords_1);
        if (nHat_B_1[3] > 0) {
            // Perform position and material read_imagef and
            // compute force and torque for pixel
        }
    }
    i += local_stride;
}

```

**Listing 2** 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);
}

```

abbreviated portion of the unrolled loops is shown in Listing 2. The number of unrolled loops is controlled at kernel compile time by 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 that match the hardware's maximum WG size.

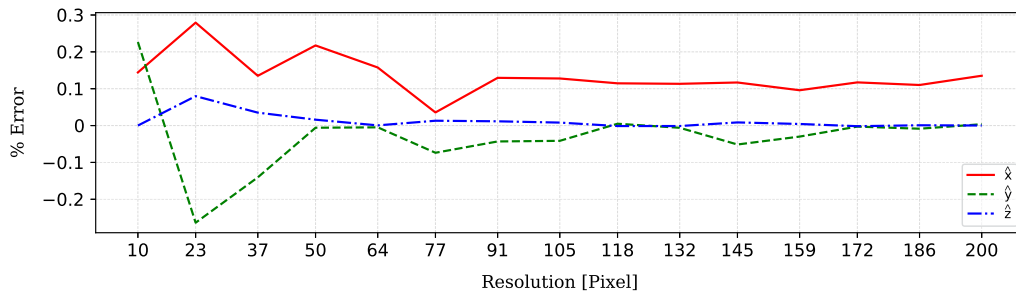
At the end of the first kernel execution the number of computed force and torque vectors is equal to the number of WGs  $\times$  WIs. A second simple parallel reduction kernel is launched for all marshaled WIs to return the final resultant force and torque vectors.

## VII. Model Validation

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 the analytic cannonball model. The spacecraft model used in the computation is a sphere of radius 1 m made up of 5120 triangular facets. Evaluated at a distance of 1 AU the analytic cannonball model computes a force of  $\mathbf{F}_{\odot} = [-1.42559 \times 10^{-5} \text{ N}, 0.0, 0.0]$ . The percent error for increasing pixel resolutions of the mesh model force evaluation with respect

to the analytic cannonball evaluation is shown in Fig. 9. It can be seen that the error decreases rapidly as the resolution increases to  $64 \times 64$  pixels. For resolutions of beyond  $190 \times 190$  pixels the error remains less than 0.01% for the  $\hat{y}$  and  $\hat{z}$  force components. The  $\hat{x}$  component maintains an offset in error of approximately 0.1% for all pixel resolutions. This is because the projected area of the mesh model is not precisely that of a circle, whereas for the analytic cannonball model the projected area is exactly  $A = \pi r^2$ . This offset demonstrates the importance of a sufficiently accurate mesh model spacecraft representation. Increasing the sphere mesh to 20,480 faces results in a lower offset of 0.034%. Further increasing the mesh to 81,920-facets reduces the offset to 0.0076%. For the symmetric cannonball model the torque is expected as zero. Given that the mesh models are not exact spheres, there is a small torque computed. This torque is of the order of  $10^{-10} \text{ N} \cdot \text{m}$  for the 5120 sphere, decreasing to  $10^{-12} \text{ N} \cdot \text{m}$  for the 20,480 sphere, and  $10^{-14} \text{ N} \cdot \text{m}$  for the 81,920 sphere.

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  $\hat{s} = [0.7071, 0.7071, 0]$ . The resulting percentage force error with

**Fig. 9** Error with respect to analytic cannonball evaluation for 1 m sphere mesh.**Fig. 10** SRP force evaluation error with increasing resolution.



respect to a faceted evaluation of the same model is shown in Fig. 10. The error decreases with increased resolution where the error remains below 0.1% for resolutions above  $1400 \times 1400$  pixels.

### A. 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 headings  ${}^B\hat{s}$  over the  $4\pi$  steradian sphere of possibilities. The three spacecraft models are shown in Fig. 11. Each model represents an increase in the detail of the modeled spacecraft. The first model in Fig. 11a is a simple box-and-wing model with a single large face oriented in the  $\hat{x}$  body frame direction in order to approximate the sun-pointing projected area of the spacecraft. The second spacecraft model in Fig. 11b incorporates larger spacecraft components, including the high-gain antenna (HGA), thruster ring, and sample return module. The final model in Fig. 11c 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 ( $\rho_d = 0.102$  and  $\rho_s = 0.408$ ) and general solar panels ( $\rho_d = 0.022$  and  $\rho_s = 0.088$ ).

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. (15). 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. (16). This approach is used for plotting both the force and torque differences.

$$F_{\text{base}} = \frac{1}{N} \sum_{n=1}^N |F_n| \quad (15)$$

$$\Delta F = \frac{F_{\text{model}} - F_{\text{hifi}}}{F_{\text{base}}} \times 100 \quad (16)$$

The force percentage differences of both low-fidelity models relative to the high-fidelity model are shown in Fig. 12. It is evident that the box-and-wing model overpredicts the resultant force for sun headings in the  $+\hat{x}$  and  $-\hat{x}$  direction while significantly underpredicting the force for headings in the  $+\hat{y}$  and  $-\hat{y}$ . The torque percentage difference of both low-fidelity models relative to the high-fidelity model are shown in Fig. 13. It is clear that the absence of the HGA from the box-and-wing model results in an underprediction of torque for a large region of sun heading midlatitude and longitudes.

The force and torque percentage differences for the box-and-wing model relative to the high-fidelity model, in each of the body frame components, are shown in Figs. 14 and 15, respectively. The approximation of the box-and-wing model is most evident in the  $\hat{x}$  force component of Fig. 14a. In the region spanned by latitude range  $-40$  deg to  $+40$  deg and longitude  $-50$  deg to  $+50$  deg the box-and-wing model overpredicts the force due to the absence of the HGA. Additionally, at sun headings of longitude  $-90$  deg and  $+90$  deg the absence of the thruster ring, sample return module, and depth due to the HGA result in over- and underpredictions of greater than 20%.

The force and torque percentage differences for the HGA model relative to the high-fidelity model, in each of the body frame components, are shown in Figs. 16 and 17, respectively. By comparison to

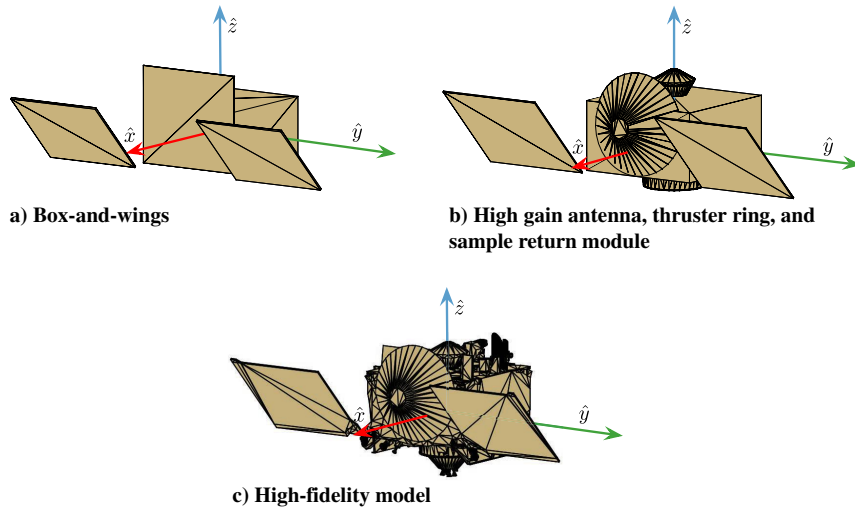


Fig. 11 OSIRIS REX spacecraft mesh models.

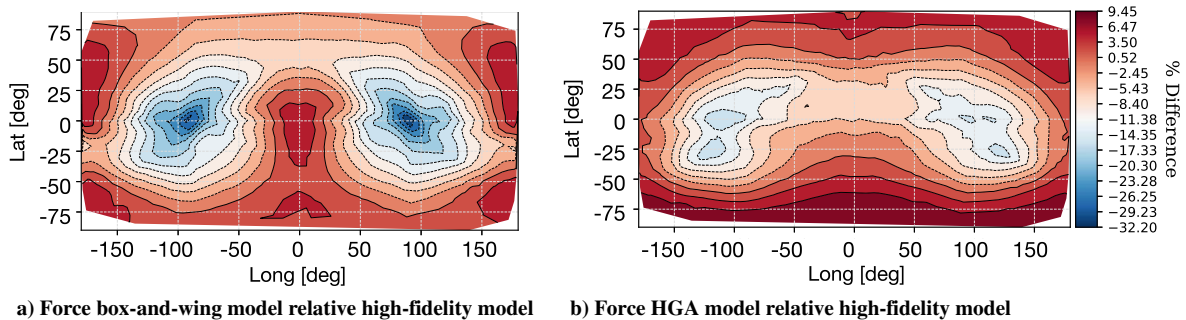
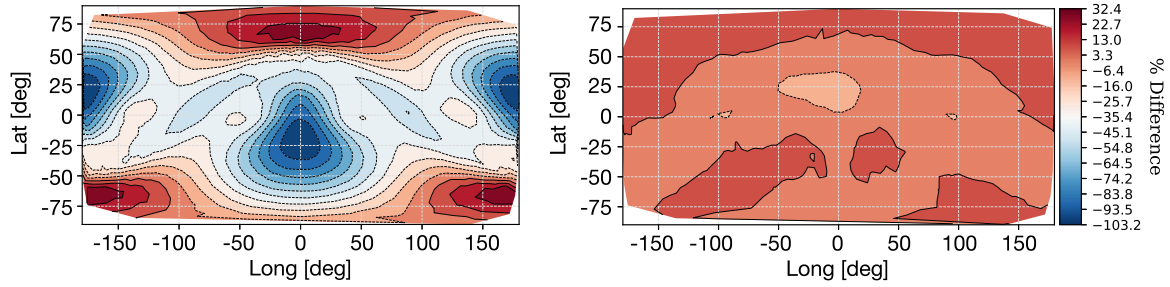
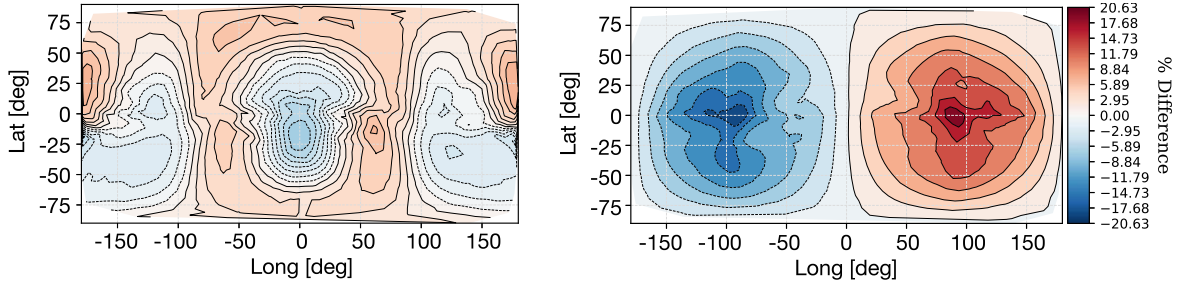


Fig. 12 Force percentage difference between low-fidelity models relative to the high-fidelity model with baseline value  $5.73361 \times 10^{-5}$  N.



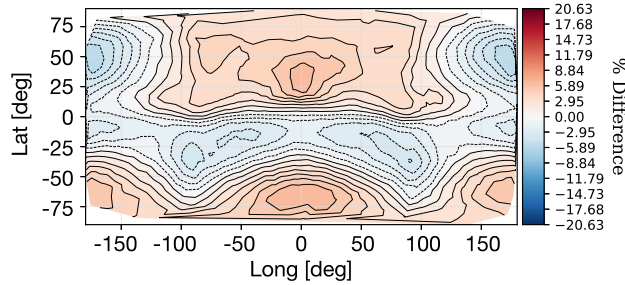
a) Torque box-and-wing model relative high-fidelity model    b) Torque HGA model relative high-fidelity model

Fig. 13 Torque percentage difference between low-fidelity models relative to the high-fidelity model with baseline value  $6.57817 \times 10^{-5} \text{ N} \cdot \text{m}$ .



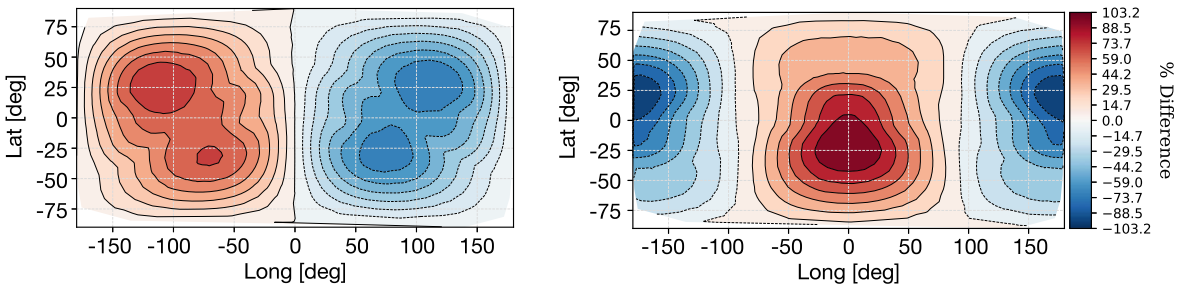
a) Force  $\hat{x}$  % difference

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



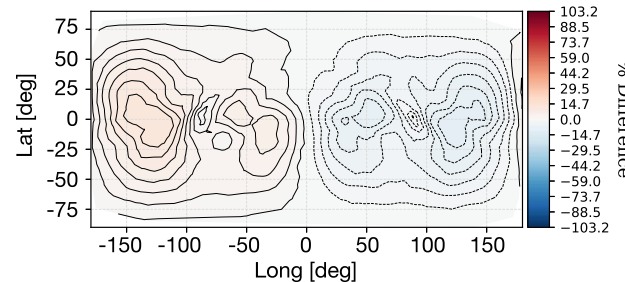
c) Force  $\hat{z}$  % difference

Fig. 14 Force percentage difference between box-and-wing model relative to the high-fidelity model with baseline value  $5.73361 \times 10^{-5} \text{ N}$ .



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

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



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

Fig. 15 Torque percentage difference between box-and-wing model relative to the high-fidelity model with baseline value  $6.57817 \times 10^{-5} \text{ N} \cdot \text{m}$ .

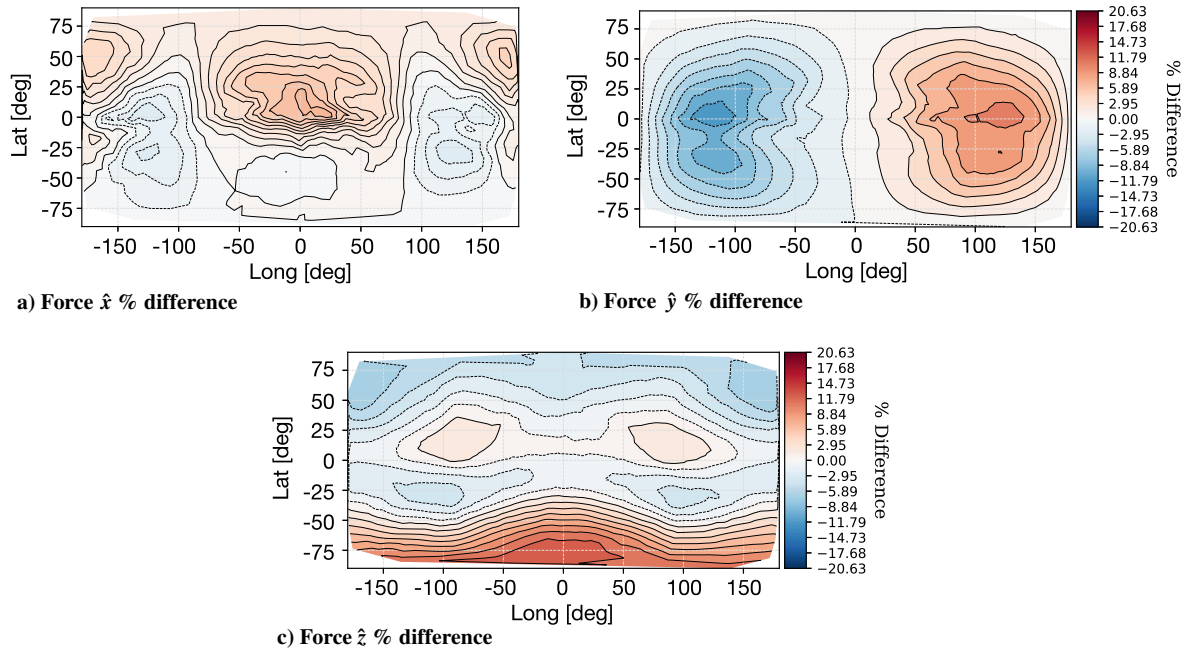


Fig. 16 Force percentage difference between HGA model relative to the high-fidelity model with baseline value  $5.73361 \times 10^{-5}$  N.

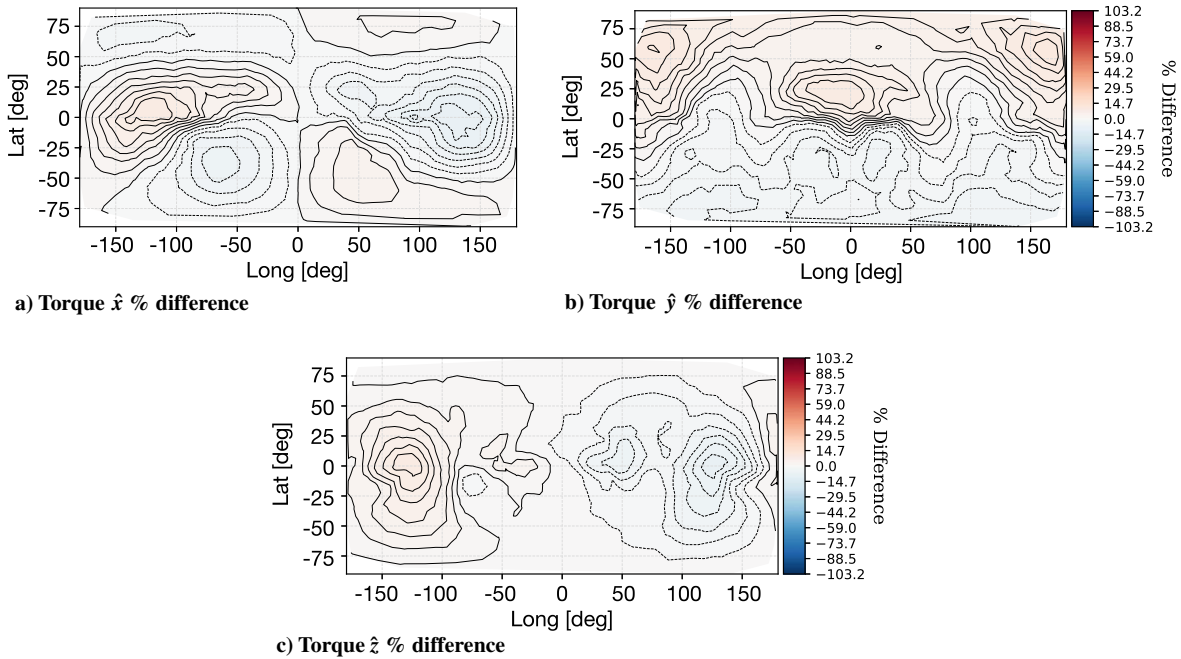


Fig. 17 Torque percentage difference between HGA model relative to the high-fidelity model with baseline value  $6.57817 \times 10^{-5}$  N · m.

the percentage force error of the box-and-wing model shown Fig. 14, the HGA model shows less error across midlatitudes. The largest SRP improvement is along the body  $\hat{y}$  axis for this particular spacecraft geometry. The presence of the HGA, thruster ring, and sample return module results in a significant improvement of torque modeling. Because of the symmetry in this particular spacecraft geometry the torque differences in the  $\hat{z}$  axis are already small with the lower-fidelity model. The torque modeling improvements in the  $\hat{x}$  and  $\hat{y}$  axes with a higher-fidelity model are very evident. However, the model does show a slight increase in force difference when the sun heading predominates in the  $+\hat{z}$  and  $-\hat{z}$  body frame components. Note that these results are an example of how the spacecraft geometry modeling can impact the SRP force and torque modeling results. The value of the fast SRP evaluation, given a CAD model, is that there is less need to spend time doing a reduced-order model tradeoff. Rather,

the GPU evaluation allows numerical analysis to be performed with a high geometric fidelity.

## B. Model Articulation and Detailed Material Properties

To demonstrate the articulation capability of this modeling method the Aqua spacecraft, shown in Fig. 3b, is simulated in Basilisk (<http://hanspeterschaub.info/basilisk>). The open-source Basilisk astrodynamics framework is a modular spacecraft simulation architecture [20]. The OpenGL-CL method is implemented as a Basilisk dynamic effector module that allows it to be integrated into the general propagation of a spacecraft rigid-body hub [21]. The simulation orbit is a 1000-km-altitude polar orbit, and the Keplerian orbital elements are listed in Table 1. The spacecraft's solar panel is controlled to articulate in a manner that causes the panel normal vector to track the

**Table 1** Spacecraft orbit parameters for polar orbit

Parameter	Value
$a$ , km	7378
$e$	0
$i$ , deg	90
$M_0$ , deg	90
$\Omega$ , deg	0
$\omega$ , deg	0

**Table 2** Spacecraft submesh material optical parameters

Material	Specular ( $\rho_s$ )	Diffuse ( $\rho_d$ )
Gold MLI [22]	0.184	0.736
Silver MLI [23]	0.66	0.16
Germanium MLI [24]	0.3	0.3
Solar array rear [25]	0.1	0.3
Solar array front [25]	0.023	0.092
Solar array boom [25]	0.3	0.3

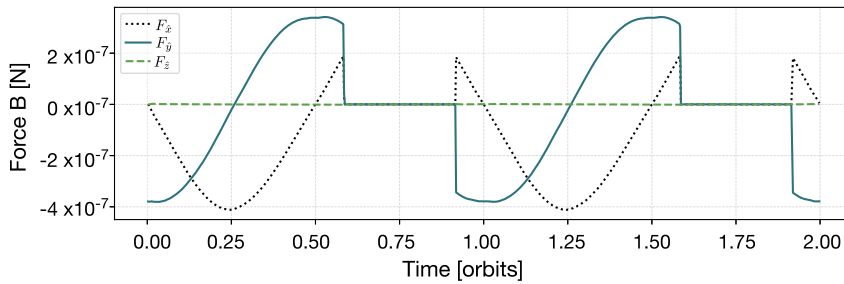
inertial heading  $[1, 0, 0]$ . The spacecraft is assigned three reaction wheel control devices that serve to control the spacecraft attitude to the computed reference attitude. The spacecraft maintains a nadir pointing attitude for its instrument deck, and the computed reference attitude is given by the orbital Hill frame.

The spacecraft is assigned material parameters for each submesh defined and shown previously in Fig. 5. The material optical properties are given in Table 2. The material parameters are chosen loosely to provide variation among materials rather than to serve as an exact reference for the optical properties of each material.

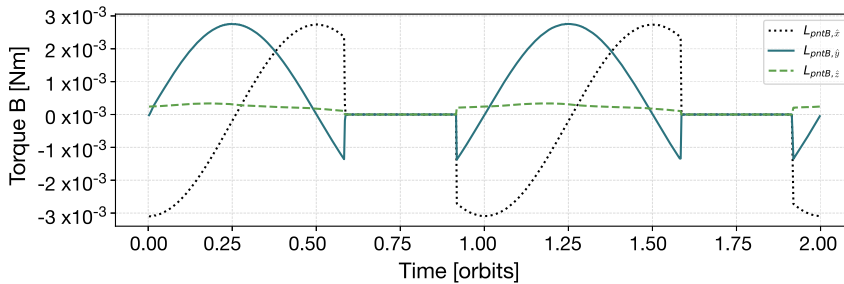
The body frame force over two orbits is shown in Fig. 18. The body torque over two orbits is shown in Fig. 19. The eclipse period is visible in both plots where the force and torque return zero.

The rendered output at three time steps is shown in Fig. 20. The viewing orientation is looking in the  $-\delta^S$  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.

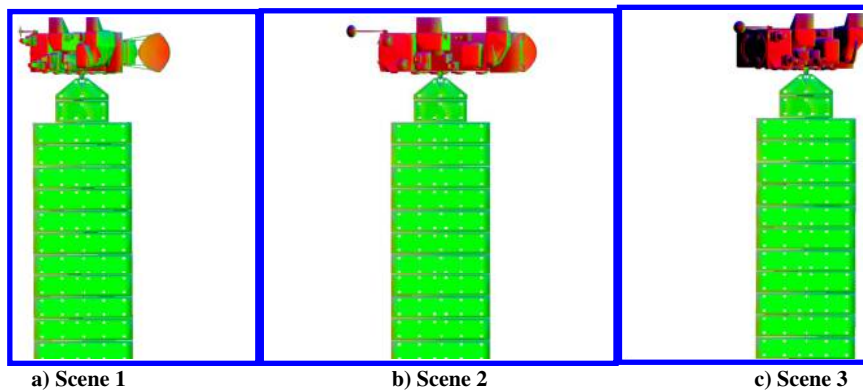
It is clear that different modeling approaches will yield different simulated dynamics. To demonstrate the potential difference in fidelity offered by the OpenGL-CL method, a comparison is made between the detailed Aqua spacecraft model and a box-and-wing approximation of Aqua spacecraft model. The detailed Aqua spacecraft model is shown



**Fig. 18** Body frame force components over two orbits.



**Fig. 19** Body frame torque components over two orbits.



**Fig. 20** Sequential rendered spacecraft in sun frame.

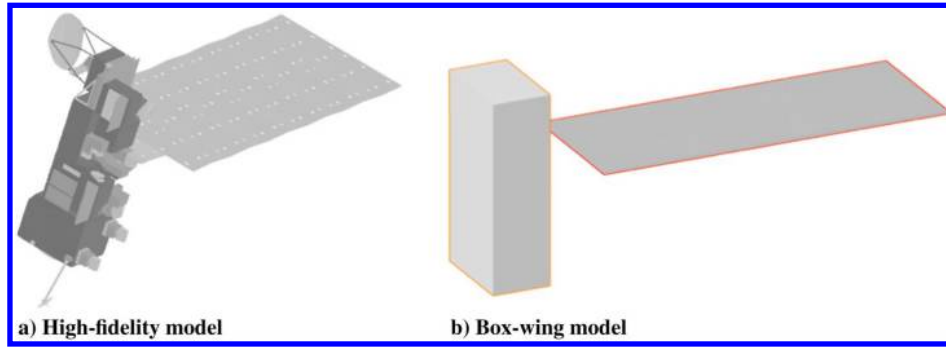


Fig. 21 Illustration of the two Aqua spacecraft models considered, each with an articulated solar panel substructure.

Table 3 Box-and-wing aqua spacecraft model material optical parameters

Material	Specular ( $\rho_s$ )	Diffuse ( $\rho_d$ )
Bus	0.32	0.5
Solar array	0.11	0.16

in Fig. 21a and the box-and-wing model shown in Fig. 21b. The high-fidelity Aqua spacecraft model is assigned the material optical properties given in Table 2, whereas the box-and-wing model is assigned the properties given in Table 3. The values in Table 3 are computed as the surface-area-weighted average of the material optical properties given to bus and solar panel components of the high-fidelity model's material properties. The simulation scenario is again the polar orbit with reaction wheel spacecraft control as previously described in this section. To convey the variation in the two modeling approaches the difference of the high-fidelity model relative to the box-and-wing model is computed for both the inertial position vector and the reaction wheel angular velocities, and shown in Figs. 22 and 23, respectively. This result serves as an illustration of the type of analysis that can be performed with this fast SRP modeling tool. The SRP modeling impact is naturally tied to the spacecraft being studied. In this case over a period of two orbits the translational differences are still small but growing in an unstable manner. In contrast, the difference in the reaction wheel momentum is already significant after two orbits.

## VIII. 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 key 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 significant performance benefits on GPU hardware that shares a common direct random access memory (DRAM) space with the CPU [26]. 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.

The computation time for resolutions from  $10 \times 10$  pixels to  $2048 \times 2048$  pixels is shown in Fig. 24. Although the Intel HD Graphics 630–integrated GPU possesses less computation capability than the other GPUs, it is able to outperform the other GPUs up to resolutions of  $1029 \times 1029$  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

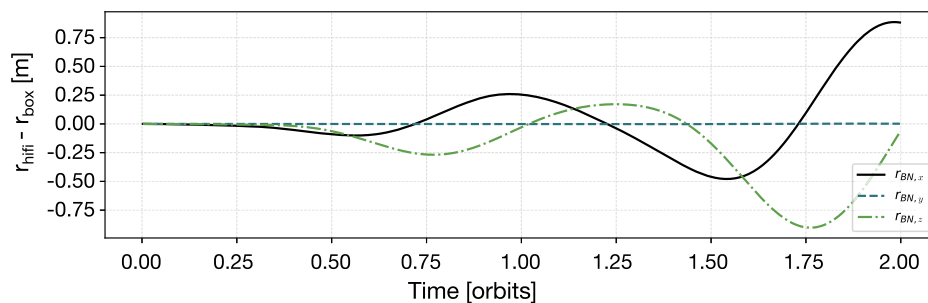


Fig. 22 Inertial position difference of the high-fidelity model relative to the box-and-wing model.

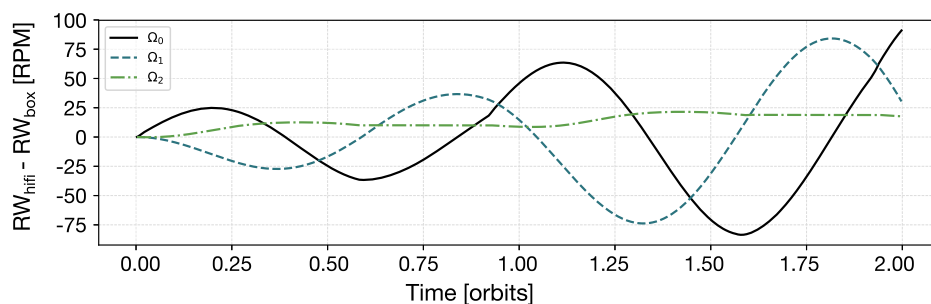


Fig. 23 Reaction wheel speed difference of the high-fidelity model relative to the box-and-wing model.

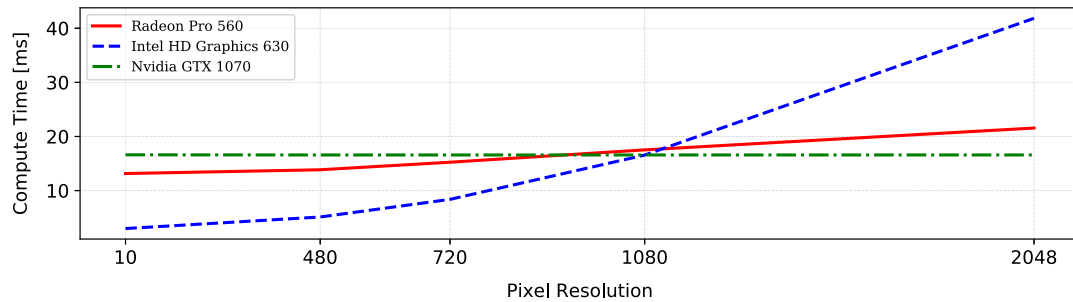


Fig. 24 OpenGL-OpenCL computation times for three different GPUs.

computational latency. The computational capability of the Radeon Pro 560 and NVIDIA GTX 1070 exceeds that of the built-in Intel graphics card for resolutions slightly above the 1080 range. The greater computational capability of the NVIDIA GTX 1070 is evident as there is no appreciable increase in computation time for all resolutions tested. All the test cases considered returned effectively the same computation time, indicating that it is the communication overhead that is determining the evaluation time, not the SRP evaluation, whereas for the Radeon Pro 560 the computation time steadily increases with increased resolution, illustrating that for this graphics card the computation time for this spacecraft case is a function of both communication overhead and SRP evaluation.

## IX. Conclusions

This paper provides a detailed description of the faceted 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. Further, a general parallel reduction algorithm is described in which the computation of SRP is tightly integrated to seek best use of GPU computing resources through the OpenCL API. These enhancements provide a method for computing high-geometric-fidelity SRP in a computationally performant and configurable manner. The OpenGL-OpenCL approach provides significant modeling capability and enables previously computationally prohibitive analysis on modest personal computing hardware. 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.

## References

- [1] Vallado, D. A., and McClain, W. D., *Fundamentals of Astrodynamics and Applications*, 2nd ed., Springer, The Netherlands, 2001, Chap. 9.
- [2] Fliegel, H. F., and Gallini, T. E., "Solar Force Modeling of Block IIR Global Positioning System Satellites," *Journal of Spacecraft and Rockets*, Vol. 33, No. 6, 1996, pp. 863–866. <https://doi.org/10.2514/3.26851>
- [3] Marshall, J. A., and Luthcke, S. B., "Modeling Radiation Forces Acting on TOPEX/Poseidon for Precision Orbit Determination," *Journal of Spacecraft and Rockets*, Vol. 31, No. 1, 1994, pp. 99–105. <https://doi.org/10.2514/3.26408>
- [4] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C., "GPU Computing," *Proceedings of the IEEE*, Vol. 96, No. 5, 2008, pp. 879–899. <https://doi.org/10.1109/JPROC.2008.917757>
- [5] Ziebart, M., Adhya, S., Sibthorpe, A., Edwards, S., and Cross, P., "Combined Radiation Pressure and Thermal Modeling of Complex Satellites: Algorithms and On-Orbit Tests," *Advances in Space Research*, Vol. 36, No. 3, 2005, pp. 424–430. <https://doi.org/10.1016/j.asr.2005.01.014>
- [6] O'Shaughnessy, D. J., McAdams, J. V., Bedini, P. D., Calloway, A. B., Williams, K. E., and Page, B. R., "Messenger's Use of Solar Sailing for Cost and Risk Reduction," *Acta Astronautica*, Vol. 93, Jan. 2014, pp. 483–489. <https://doi.org/10.1016/j.actaastro.2012.10.009>
- [7] O'Shaughnessy, D. J., McAdams, J. V., Williams, K. E., and Page, B. R., "Fire Sail: Messenger's Use of Solar Radiation Pressure for Accurate Mercury Flybys," *Advances in the Astronautical Sciences*, Vol. 134, 2009, pp. 1527–1539.
- [8] Lucchesi, D. M., "Reassessment of the Error Modeling of Non-Gravitational Perturbations on LAGEOS II and Their Impact in the Lense-Thirring Derivation—Part II," *Planetary and Space Science*, Vol. 50, Nos. 10–11, 2002, pp. 1067–1100. [https://doi.org/10.1016/S0032-0633\(02\)00052-1](https://doi.org/10.1016/S0032-0633(02)00052-1)
- [9] McMahon, J., and Scheeres, D. J., "New Solar Radiation Pressure Force Model for Navigation," *Journal of Guidance, Control, and Dynamics*, Vol. 33, No. 5, 2010, pp. 1418–1428. <https://doi.org/10.2514/1.48434>
- [10] Ziebart, M., "High Precision Analytical Solar Radiation Pressure Modeling for GNSS Spacecraft," Ph.D. Dissertation, Univ. of East London, London, 2001.
- [11] Tanygin, S., and Beatty, G. M., "GPU-Accelerated Computation of SRP Forces with Graphical Encoding of Surface Normals," *AAS/AIAA Astrodynamics Specialist Conference*, AAS Paper 15-688, San Diego, CA, 2015.
- [12] Tanygin, S., and Beatty, G. M., "GPU-Accelerated Computation of Drag and SRP Forces and Torques with Graphical Encoding of Surface Normals," *AAS/AIAA Space Flight Mechanics Meeting*, AAS Paper 16-313, San Diego, CA, Feb. 2016.
- [13] Tichy, J., Brown, A., Demoret, M., Schilling, B., and Raleigh, D., "Fast Finite Solar Radiation Pressure Model Integration Using OpenGL," *24th International Symposium on Space Flight Dynamics (ISSFD)*, 2014.
- [14] Kenneally, P. W., and Schaub, H., "Modeling Solar Radiation Pressure With Self-Shadowing Using Graphics Processing Unit," *AAS Guidance, Navigation and Control Conference*, AAS Paper 17-127, San Diego, CA, 2017.
- [15] Kenneally, P. W., and Schaub, H., "Parallel Spacecraft Solar Radiation Pressure Modeling Using Ray-Tracing On Graphic Processing Unit," *International Astronautical Congress*, Paper IAC-17,C1,4,3,x40634, 2017.
- [16] Kenneally, P. W., and Schaub, H., "Fast Spacecraft Solar Radiation Pressure Modeling by Ray Tracing on Graphics Processing Unit," *Advances in Space Research*, Vol. 65, No. 8, 2020, pp. 1951–1964. <https://doi.org/10.1016/j.asr.2019.12.028>
- [17] Kenneally, P. W., and Schaub, H., "Spacecraft Radiation Pressure Using Complex Bidirectional-Reflectance Distribution Functions On Graphics Processing Unit," *AAS Spaceflight Mechanics Meeting*, AAS Paper 19-531, San Diego, CA, 2019, pp. 1631–1650.
- [18] Schaub, H., and Junkins, J. L., *Analytical Mechanics of Space Systems*, 4th ed., AIAA Education Series, AIAA, Reston, VA, 2018, Chap. 3. <https://doi.org/10.2514/4.105210>
- [19] Banger, R., and Bhattacharyya, K., *A Comprehensive Guide on OpenCL Programming with Examples*, Packt Publ., Birmingham, England, U.K., 2013, Chap. 7.
- [20] Kenneally, P. W., Piggott, S., and Schaub, H., "Basilisk: A Flexible, Scalable and Modular Astrodynamics Simulation Framework," *Journal of Aerospace Information Systems*, Vol. 17, No. 9, 2020, pp. 496–507. <https://doi.org/10.2514/1.1010762>
- [21] Allard, C., Diaz-Ramos, M., and Schaub, H., "Computational Performance of Complex Spacecraft Simulations Using Back-Substitution," *Journal of Aerospace Information Systems*, Vol. 16, No. 10, 2019,

- pp. 427–436.  
<https://doi.org/10.2514/1.1010713>
- [22] Kubo-Oka, T., and Sengoku, A., “Solar Radiation Pressure Model for the Relay Satellite of SELENE,” *Earth, Planets and Space*, Vol. 51, No. 9, 1999, pp. 979–986.  
<https://doi.org/10.1186/BF03351568>
- [23] Darugna, F., Steigenberger, P., Montenbruck, O., and Casotto, S., “Ray-Tracing solar Radiation Pressure Modeling for QZS-1,” *Advances in Space Research*, Vol. 62, No. 4, 2018, pp. 935–943.  
<https://doi.org/10.1016/j.asr.2018.05.036>
- [24] Choi, M. K., “Thermal Assessment of Sunlight Impinging on OSIRIS-REx OCAMS PolyCam, OTES, and IMU-Sunshade MLI Blankets in Flight,” *Proceedings of SPIE 10401, Astronomical Optics: Design, Manufacture, and Test of Space and Ground Systems*, Vol. 10401, SPIE, 2017.  
<https://doi.org/10.1117/12.2276642>
- [25] Rauschenbach, H. S., *Solar Cell Array Design Handbook*, 1st ed., Springer, Dordrecht, The Netherlands, 1980, Chap. 5.  
<https://doi.org/10.1007/978-94-011-7915-7>
- [26] Gera, P., Kim, H., Kim, H., Hong, S., George, V., and Luk, C.-K., “Performance Characterisation and Simulation of Intel’s Integrated GPU Architecture,” *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Belfast, 2018, pp. 139–148.  
<https://doi.org/10.1109/ISPASS.2018.00027>

J. P. How  
Associate Editor