FAST FINITE SOLAR RADIATION PRESSURE MODEL INTEGRATION USING OPENGL

Jason Tichy[1], Abel Brown[2], Michael Demoret[3], Benjamin Schilling[4], and David Raleigh[5]

[1][2][3][4][5]*a.i. solutions, Inc., 10001 Derekwood Ln, Lanham, MD 20706, 301-306-1756, first.last@ai-solutions.com*

Abstract: *By coupling a common approach to vector graphics, OpenGL, high-fidelity solar-radiation pressure (SRP) effects are calculated easily and quickly with the power of graphics processing units (GPUs). For some missions SRP is a significant perturbation and a consideration wherein a simplified plate model does not suffice. OpenGL is a set of commands that interact with the GPU and model a system as many small polygons. By modeling the spacecraft in this environment, the fidelity of the SRP model is limited only by the polygon approximation of the spacecraft and not by the computational throughput of the software. This paper demonstrates how complex SRP forces are better extracted using OpenGL methods known as shaders that yield significant improvements in speed and fidelity with respect to existing tools.*

Keywords: *SRP, OpenGL, GPU, Solar Sail, JWST*

1. Introduction

Solar radiation pressure is exerted on any surface exposed to electromagnetic radiation. Specifically, it is found to be the force required for the change in momentum as the photons from the electromagnetic source interact with the surface. Generally, this force is too small to be detected in everyday activity; however, in the vacuum of space it can be a significant source of acceleration. Most mission designers model this force as a small perturbation on the desired orbit. Contrarily, SRP is the dominant perturbation to consider for both asteroid and comet studies as the force tends to exceed the forces of gravity nearby the smaller objects [1]. Some missions, such as those using solar sails, actually incorporate this force into their design of non-Keplerian orbits [2].

Solar radiation pressure is calculated from the solar constant and varies inversely by the square of the distance from the Sun. The pressure exerted by the Sun on a given surface is

$$P = \frac{W}{cR^2} \tag{1}$$

where R is the solar distance in terms of AU, c is the speed of light, and W is the solar radiance (W has a value of 1358 $^{W}/_{m^2}$ at 1.0 AU as chosen for this analysis). The resultant pressure, P, is determined as $^{N}/_{m^2}$ or Pascals after treating R as a unitless number.

The goal of this analysis is to develop an OpenGL environment where accurate three-dimensional spacecraft models may used to return a high-fidelity solution to the SRP encountered in a space environment. This paper begins with a review of different methods to model SRP. Next is a discussion on some of the useful tools OpenGL provides for simplifying this analysis. A demonstration on how to incorporate SRP methods into the modeling environment is provided with examples. Finally, a brief synopsis of the performance gains of modeling SRP perturbations in this environment is presented.

2. Finite SRP Modeling

A standard SRP model is the cannonball model. Sometimes denoted as the LAGEOS model due to the fact that it was used for that cannonball-shaped spacecraft, this model is based on the spacecraft being a simple sphere with all incident radiation being reflected directly back at the Sun. It is written as

$$\vec{a}_{SRP} = -C_R \frac{P(R)}{m} A \cdot \hat{u} \tag{2}$$

where A is the cross-sectional exposed area, C_R, the coefficient of reflectivity, is an SRP strength scaling factor, m is the mass of the spacecraft, and $\hat{u}$ is a unit vector pointing toward the radiation source. Nominally, C_R has a value of 2.0 for a perfectly reflecting spacecraft. However, it is more appropriately taken as $1+p$, where p is a value less than 1 from electromagnetic radiation being absorbed or transmitted through the spacecraft. This model is a common model in orbit determination as C_R is a variable which can be solved for in the process.

A slightly more complex model assumes that a spacecraft is approximated as a flat plate at an angle to the radiation source. This is written as

$$\vec{a}_{SRP} = -C_R \frac{P(R)}{m} A cos^2(\theta) \cdot \hat{n} \tag{3}$$

where θ is the angle between the surface normal,$\hat{n}$, and the source radiation. Fig. 1 is a freebody diagram which breaks down the flat plate model. A significant change from the cannonball assumption is that the reaction force is negatively parallel to the surface normal instead of the incident light source.

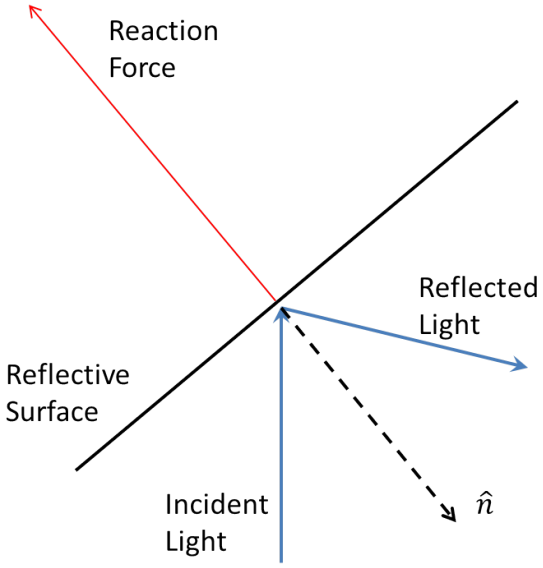


Figure 1: Forces on a reflector due to the photon flux

A detailed model of solar radiation pressure was used for the GRACE mission[3]. The GRACE model breaks the spacecraft into six separate plates, each having well-defined optical properties.

This model is written as

$$\vec{a}_{SRP} = -C_R \frac{P(R)}{m} \sum_{k=1}^{6} A_k cos(\theta_k) \left[(1 - s_k)\,\hat{u} + 2\left(\frac{\rho_k}{3} + s_k cos(\theta_k)\right)\hat{n} \right] \tag{4}$$

where k is the index for each surface. Each surface is represented by a normal, $\hat{n}$, an associated area, A_k, and specular and diffuse reflectivity coefficients, s_k and ρ_k, respectively. In this model each surface has a C_R value, but in this instance is nominally equal to 1.0.

The way GRACE was designed, as seen in Fig. 2, the 6-sided model described by Eq. 4 provides a highly accurate physical solution. Where the tangential components of the pressure are assumed to cancel each other perfectly in cannonball models, the GRACE model accounts for in the speculative and diffusive reflection properties.

For many applications a small number of plates suffices. Many spacecraft are designed with a simple exterior structure and solar panels that can be approximated using a limited model. However, even for a small number of plates the numerical complexity of computing Eq. 4 becomes burdensome considering the number of calls to the force properties involved in numerical propagation. Even so, each surface is not dependent on any other surface, meaning each surfaces SRP force can be computed independently and in parallel.

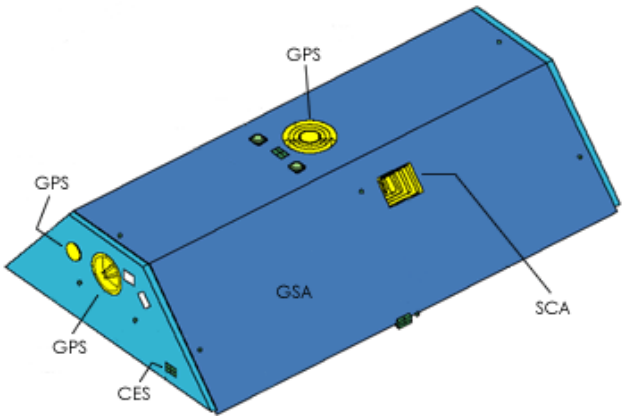


Figure 2: Basic model of the GRACE spacecraft

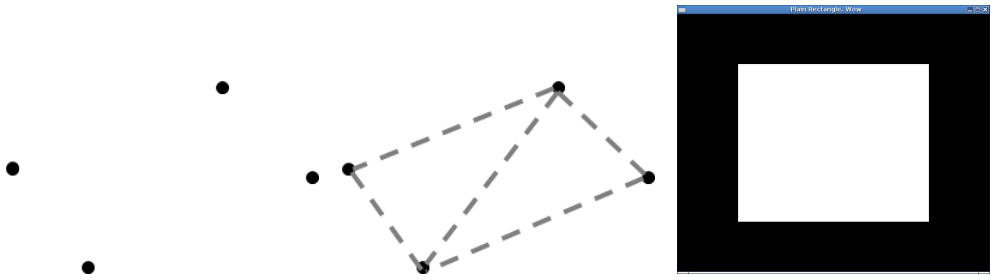
3. OpenGL

Though GPUs have more recently been used for highly parallel, general-purpose computations, they have a much longer history in optimizing vector graphics. Vector graphics is the process of projecting vectors and shapes in a modeled three-dimensional world space and determining its projection onto the viewing plane. This involves computing the direction and intensity of light as it is projected onto the model's surface and then computing how the object reflects upon the scene. The objective of high-fidelity SRP modeling is to determine how electromagnetic radiation (e.g. Sunlight) interacts with a surface, which is exactly what GPUs are originally designed to do. To display 3D graphics on the screen a set of programs, called shaders, is run through the GPU that

then breaks the model into individual portions and computes light interactions and displays the resulting projection. As 3D gaming has become more popular, GPU manufacturers have built more powerful devices to handle a greater computational load of more complex 3D environments. In this case, thousands of cores are available on these devices and are capable of computing the light properties on millions of plates at a single instance in time, far more than the 6 to 10 plates that normally represent a spacecraft.

OpenGL (Open Graphics Library) is a multi-platform, cross-language application programming interface (API) for rendering vector graphics. The API is specifically designed to interact directly with the GPU to achieve hardware-accelerated rendering. Major hardware manufacturers ensure each of their designs is compatible with OpenGL specifications. Since 1992 OpenGL has been a core utility in which many programming languages implemented and thousands of applications depend on for support. Other graphics languages are also suitable for this analysis, but OpenGL is exploited for its wide adoption verses a platform specific implementation, like DirectX or Cocoa.

Building an OpenGL model is a relatively simple process. An OpenGL model is created by vertices and defining faces as sets of vertices. The most simple example is a flat-plate. An infinitely thin flat plate is defined as 4 vertices in 3D Cartesian space. Given 4 vertices, the plate is created by passing into OpenGL two arrays which contain three vertices apiece. In OpenGL terminology the two sets of points form polygons, for the purpose of this analysis each polygon represents a plate for an SRP calculation. Fig. 3 shows a basic building of two plates using OpenGL defined polygons.



(a) Vertices of plate (b) Face definition of plate (c) OpenGL display

Figure 3: Defining a simple plate in OpenGL

Normals associated with each vertex indicate to OpenGL how to apply its shading techniques(OpenGL shading will be discussed in the next section). Normals associated with the vertices indicate how a plate is oriented with respect to light sources. When a single vertex is associated with multiple faces several different normal vectors can be associated with that vertex. Figure 4 uses a simple cube to demonstrate different shading patterns determined by OpenGL by using multiple normals associated with shared vertices.

(a) Defining the cube face normals (b) OpenGL Rendering the shaded cube

Figure 4: OpenGL Shaded cube

In building an OpenGL model, vertex normals do not necessarily need to be perpendicular to the surface. One of the powerful tools which come with the OpenGL API is an automatic interpolation between surfaces for increased fidelity even with fewer plates. Associating normals with vertices defines the direction the light reflects off of the surface but also how this interpolation occurs between the plates. This allows the program to interpolate between surfaces smoothly and create more complex shading than would normally be possible. Figure 5 demonstrates this effect on a simple polyhedral model.

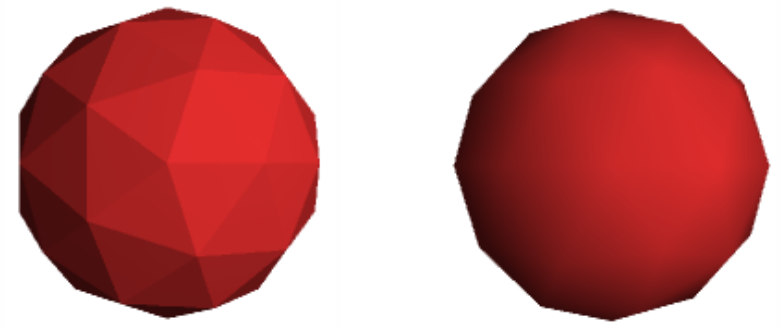


Figure 5: Shadow difference between surface perpendicular and smoothed normals

The main power gained in using OpenGL is the availability of code heritage. The main complexity of performing the SRP analysis using OpenGL is developing the accurate 3D model. There are many graphics programs available to develop 3D models in the same fashion as AutoCAD, such as Autodesk's MAYA. Once the analyst has obtained an accurate model the math is straightforward and OpenGL handles the parallelism of the analysis.

4. OpenGL Shaders

Shaders, or OpenGL Shading Language (GLSL), was developed with the OpenGL C-like syntax to provide programmers with direct control over the graphics pipeline. This feature frees the programmer from the need to use machine language to accomplish custom tasks and provides several math functions to perform complex display events. Figure 6 is a flowchart that depicts the graphics pipeline, which is the process that occurs after the CPU executes the OpenGL API. The graphics pipeline handles all of the vector graphics computations and performs all functions until the model is rendered to the screen.

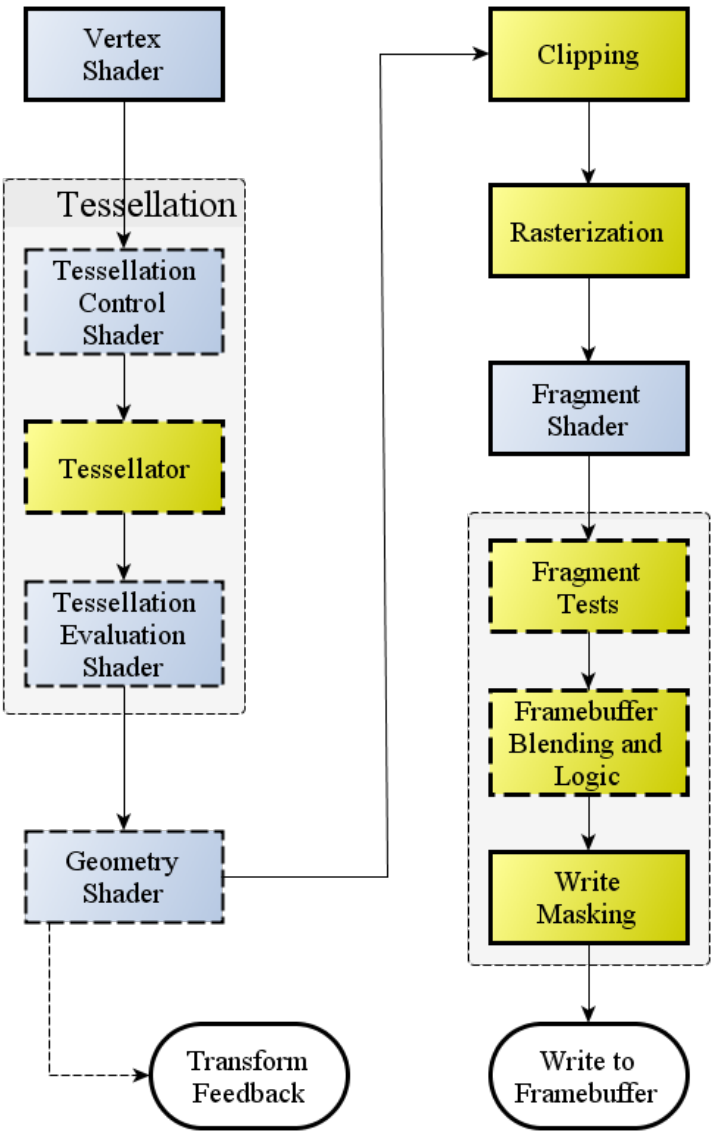


Figure 6: OpenGL Graphics Pipeline (image credit: OpenGL.org)

First in the graphics pipeline is the Vertex Shader. This shader is used to take the vertices the user supplied and perform basic transformations on them to place them in the Cartesian world space.

Listing 1 is a simple example of a vertex shader.

Listing 1: OpenGL Basic Vertex Shader

```
// Vertex Shader
layout(location = 0) in vec3 vertexPosition;
varying vec4 color;
uniform vec3 projectionMatrix;
uniform vec3 viewCameraMatrix;
uniform vec3 modelSizeMatrix;
out vec3 worldPosition;

void main()
{
  gl_Position = projectionMatrix * viewCameraMatrix *
     modelSizeMatrix * vertexPosition;

  worldPosition = modelSizeMatrix * vertexPosition;
}
```

In this example of a Vertex Shader matrices are used to change a vertex position with respect to the desired model's size. Then a matrix is used to change the vertex position with respect to a viewing camera. Finally, a matrix is used to project the vertex position in a 3D space onto a two dimensional viewing plane. *gl_Position* is an automatic variable recognized by the GPU for the graphics display. For SRP calculations, vertex information must be represented for the physical world also. The *worldPosition* variable is calculated and declared as an output variable to control the model with respect to the physical world.

In the tessellation phase the GPU itself has built-in routines to interpolate between vertex points in a quick, but highly accurate, way [4]. In short, tessellation subdivides an object into a smoother surface by generating more plates, providing a greater amount of detail.

The next step in the OpenGL pipeline is the Geometry Shader. This optional step is the only opportunity to interact with the three vertex positions that form a plate. The plate which is being passed into the Geometry Shader may be a superset of the input vertices, as the tessellation process has broken the object down into a large set which the GPU is capable of handling. However, the geometry shader provides access to only the points that form the local plate.

Listing 2 is an example of the simplest form of a Geometry shader for the purpose of demonstrating its functionality.

Listing 2: OpenGL Passthrough Geometry Shader

```
// Passthrough Geometry Shader
layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

void main() {
  for(int i = 0; i < 3; i++) {
    // Triangles coming in, so it's always 3
    gl_Position = gl_in[i].gl_Position;
    EmitVertex();
  }
  EndPrimitive();
}
```

The Geometry shader is provided with the plate which was yielded by the tessellation process. The shader can perform computations on the plate and have access to its associated vertices and normals for the last time in this step. In order to move to the next step the Geometry shader needs to release the plate geometry back into the pipeline one vertex at a time. Therefore, the array of information that is received is looped over and *"emitted"* back out into the overall process. This was the important step needed to compute SRP information but the rest of the pipeline is required to create a functioning program.

The Clipping and Rasterization processes take the calculations performed in the previous steps and turn the scene into pixels and dots to prepare the graphical output on the GPU. While the process is straightforward, it is greatly accelerated by the GPU. At this point the device has performed as many as several million vector math operations and is ready to take a slice of the image, generally occurring at 24 frames per second or faster. With modern displays and high performance GPUs it is possible to do the whole OpenGL process at 240 frames per second. For a 4K definition scene that equates to 1.99 billion pixels per second, which includes all the vector math to be performed to define each pixel. To represent performance metrics, each frame represents a round of SRP calculations and a pixel describes the resulting fidelity of the model.

The final step required for rendering is the Fragment shader. This shader colors the scene to be displayed on the monitor. Listing 3 demonstrates the most simple way to create a Fragment shader. For every portion of the model developed by the Vertex and Geometry shaders, the Fragment shader colors that portion. In this example, that color is always defined to be red based on a vector representing Red, Green, Blue and Intensity values. All vector calculations have been performed already and this is required to complete the pipeline.

Listing 3: OpenGL Passthrough Geometry Shader

```
// Fragment Shader
out vec4 colorOut;

void main(){
    colorOut = vec4(1.0, 0.0, 0.0, 1.0);
}
```

The Framebuffer is the monitor or other video device designated to render the image to the screen. This is the end of the pipeline where the GPU sends the signals to the device actually producing what the analyst will see as the output.

5. SRP in the OpenGL Pipeline

The ability to load a complex model and do simultaneous calculations for each surface is an enabling factor for higher fidelity SRP calculations. It can be imagined that there are other ways of performing the SRP calculations at other parts of the graphics pipeline, the Geometry shader environment is the most natural fit. What is handed to the Geometry shader is reconfigured triangles in a three-dimensional space. Furthermore, any information generated during the Vertex shader process also accompanies the triangle vertex points. In the previous section, the sample code in the Vertex shader Listing 1 also computed world-position information for the model. While that information was ignored previously in the Geometry and Fragment shader examples (Listings 2 and 3) in this section a new Geometry shader is developed to make use of that information.

The first step to developing the SRP calculation is to isolate the parameters involved. A simple calculation for SRP in the cannonball model in Eq. 2 requires that C_R data be passed in with the vertices of the model. A Geometry shader that performs this calculation is built as an example in Listing 4.

Listing 4: OpenGL Cannonball SRP calculation inside of a Geometry Shader

```
// Cannonball Geometry Shader
layout(triangles) in;
// In variables come in as arrays of 3
in vec3 worldPosition[];
in float CR[];

layout(triangle_strip, max_vertices = 3) out;
// Out variables are individual variables emitted 3 times
out float F_SRP;

float P = 1361.0 / 299792458.0;

void main() {

  vec3 side1 = worldPosition[1] - worldPosition[0];
  vec3 side2 = worldPosition[2] - worldPosition[0];
  float area = 0.5 * length(cross(side1,side2));

  // Triangles coming in, so it's always 3
  for(int i = 0; i < 3; i++) {
    // First do what the Geo Shader is supposed to do
    gl_Position = gl_in[i].gl_Position;

    // Add the SRP calculation
    if (i == 0)
    {
        F_SRP = -CR[i] * P * area;
    }
    else
    {
        F_SRP = 0.00;
    }
    // Emit data as though the vertex shader did
    EmitVertex();
  }
  EndPrimitive();
}
```

Because SRP information is associated with the input vertices to the Geometry shader the resulting SRP force on the plate created by the vertices is exported with the first of the exported vertices. This reduces complexities when summing the returned values of SRP across all plates inside the

model. Because the Geometry shader is required to produce three values of *F_SRP*, it reduces the complexities of have 3 repeated forces with each plate. Considerations for retrieving the SRP force values are discussed in a subsequent section.

The cannonball model serves as a building block for a more complicated model and an excellent stopping point for debugging code issues. The next step is to compute the model in Eq. 4. To accomplish this Texture objects and properties available through the OpenGL API are used to provide the necessary surface properties to employ this calculation.

Textures are available in OpenGL to standardize the way materials are computed across the graphics industry. Technically, a texture is part of a Material object where material properties provide for color and reflectivity, including both the diffusive and specular properties. Listing 5 is an example declaration of the Material made available by the OpenGL program as a structure with all of the extracted properties.

Listing 5: OpenGL Geometry shader accessing Material properties

```
// Geometry Shader
layout(triangles) in;
// In variables come in as arrays of 3
in vec3 worldPosition[];
in float CR[];

layout(std140) uniform Material{
        vec3 diffuse;
        vec3 ambient;
        vec3 specular;
        vec3 emissive;
        float shininess;
        int texCount;
};

uniform vec3 Origin2SunVec;

layout(triangle_strip, max_vertices = 3) out;
// Out variables are individual variables emitted 3 times
out vec3 F_SRP;
```

Notice that *diffuse* and *specular* are vector objects. OpenGL is sophisticated enough to specify the red, green, and blue reflective properties of a material. However, for this study the magnitude of the reflective properties for use in Eq. 4 is employed. A uniform (unchanging) vector *Origin2SunVec* is included in Listing 5 in which the host program has provided to the shader a vector from the center of the model reference to the Sun. OpenGL passes the vertices defined in the order in which the

face is defined into the Geometry shader; obtaining the surface normal is as simple as crossing *side2* with *side1*.

$$cos(\theta) = \hat{n} \cdot \hat{u} \tag{5}$$

where $\hat{n}$ is the surface normal unit vector and $\hat{u}$ is a unit vector representing the direction of the Sun vector. For most situations it is safe to assume the Sun is sufficiently far away that the angle between the surface normal and the Sun does not change greatly across the surface of the triangle plate, and that the angle between the origin of the model and the Sun and the angle between the plate and the Sun also does not change greatly. It is straightforward to include the vector math into the Geometry shader to accomplish those tasks.

A common function to perform a quick shadow determination is to monitor the angle of the surface normal to the Sun vector and enforce that it be considered for only angles of inflection less than 90 degrees. Anything greater than 90 degrees should result in not calculating the SRP of that plate. That is,

$$cosTheta = \begin{cases} cos(\theta) & \text{for} \quad 0 < cos(\theta) \leq 1 \\ 0 & \text{for} \quad cos(\theta) \leq 0 \end{cases} \tag{6}$$

OpenGL provides a fast hardware solution for performing this calculation quickly by making a call to the *clamp* function. *Clamp* functionally enforces the calculation be between two values. By *clamp* enforcing Eq. 6 the software can numerically determine both Eq. 3 and Eq. 4 to be zero for all surfaces not pointing toward the light.

6. Retrieving OpenGL Data

There are multiple methods for retrieving the SRP values computed in the Geometry shader which resides in the GPU memory. Inside the Geometry shader the *F_SRP* was computed and exported along with the vertex information. This exporting process places the values into a memory buffer located on the GPU. Normally, the OpenGL pipeline goes one way from the API calls to writing the Framebuffer; however, in OpenGL version 3.3 functionality exists to retrieve the buffer of memory on the GPU and bring it back to the main program. This process is known as Transform Feedback [5].

Transform feedback allows the user to declare to the GPU that a chunk of memory not be destroyed at the end of the graphics pipeline. That allows the user to swap the memory buffer back to the main program loop. Swapping the memory back this way results in an array of *vec3* objects whose values line up in the same order as the vertices of the model input into the process. Summing along the axis of this array provides a Cartesian total solar radiation pressure force for the entire model. Torque may also be easily determined by multiplying the *F_SRP* values by the input vertex array.

Transform Feedback is the OpenGL way of returning the SRP information from the GPU. OpenGL also maintains a high level interoperability between many other mathematical utilities such as *NVidia's* Compute Unified Device Architecture (CUDA) or an open source language *Open Compute Language* (OpenCL). Writing a function using one of these languages saves time by not bringing the memory back from the GPU and performing the calculation in parallel directly on the memory block.

7. Results

Implementing the SRP calculation using OpenGL on the video device provides for many orders of magnitude of speed increase which is a function of model size. For instance, a six sided object such as a cube shaped satellite or GRACE the increase is negligible. The real increase in performance comes in model fidelity. A small laptop video device is capable of performing the SRP calculations at the same speed for an object defined with thousands of plates as it is for a six sided cube object using the OpenGL implementation on a dedicated GPU. This is because of the efficiency in performing these calculations in parallel on the GPU.

The performance of this tool can be measured in frames per second, since an SRP calculation is performed each time the model is rendered to the screen. In some ways this can be limiting as operating systems attempt to control the quality of the screen image. Windows 7, for instance, forces the screen to render only as many times as the monitor is capable of refreshing the image. This feature can be bypassed with special settings to allow the program to run at the full speed of the program.

For choosing material properties which result in matching the reflectivity coefficient, C_R, the SRP acceleration found by using the OpenGL method matched with that determined by the astrodynamics tool *FreeFlyer* [6]. *FreeFlyer* uses a special version of the Cannonball Model, Eq. 2, which includes a scaling factor for shadow conditions. At a distance of $1AU$ from the Sun *FreeFlyer* determined the SRP acceleration of a 1000 kg satellite with 10 m^2 of reflecting surface with a C_R of 2.0 to have an acceleration of $9.3708e-11$ $^{km}/_{s^2}$. Inputting the same parameters into an OpenGL SRP simulation using rectangle model of varying plates yields the same $9.3708e-11$ $^{km}/_{s^2}$ for as little as two plates and tested with as many as 1000 plates. Future analysis will be performed to compare the OpenGL tool against SRP analysis currently being performed on JWST.

Using a Windows 8 desktop computer with dual AMD FirePro video devices results for SRP calculations are able to be performed at a rate of 200 frames per second (200 total SRP calculations) for a model of the Hubble spacecraft containing $661,000$ plates. That yields an SRP calculation every 0.005 seconds. In comparison, Matlab on the same system takes 15.39 seconds to perform just a single SRP force calculation with all vector math having been pre-computed. Therefore, performing the SRP calculation utilizing OpenGL has gained three orders of magnitude in performance gain for a model of this fidelity.

Figure 7 shows the OpenGL application performing an SRP analysis inside of a performance monitoring environment provided by NVidia. This is a model of the James Webb Space Telescope with $25,300$ faces. When performed on a 2013 MacBook Pro with an NVidia graphics processor it is using less than 15% of the computing resources. The graphics card in this instance is bounded by the refresh rate of the monitor which is 60 frames per second. The 60 frames per second is equivalent to performing 60 full fidelity SRP calculations each second, which is three orders of magnitude faster than Matlab would perform the same calculation on the same computer.

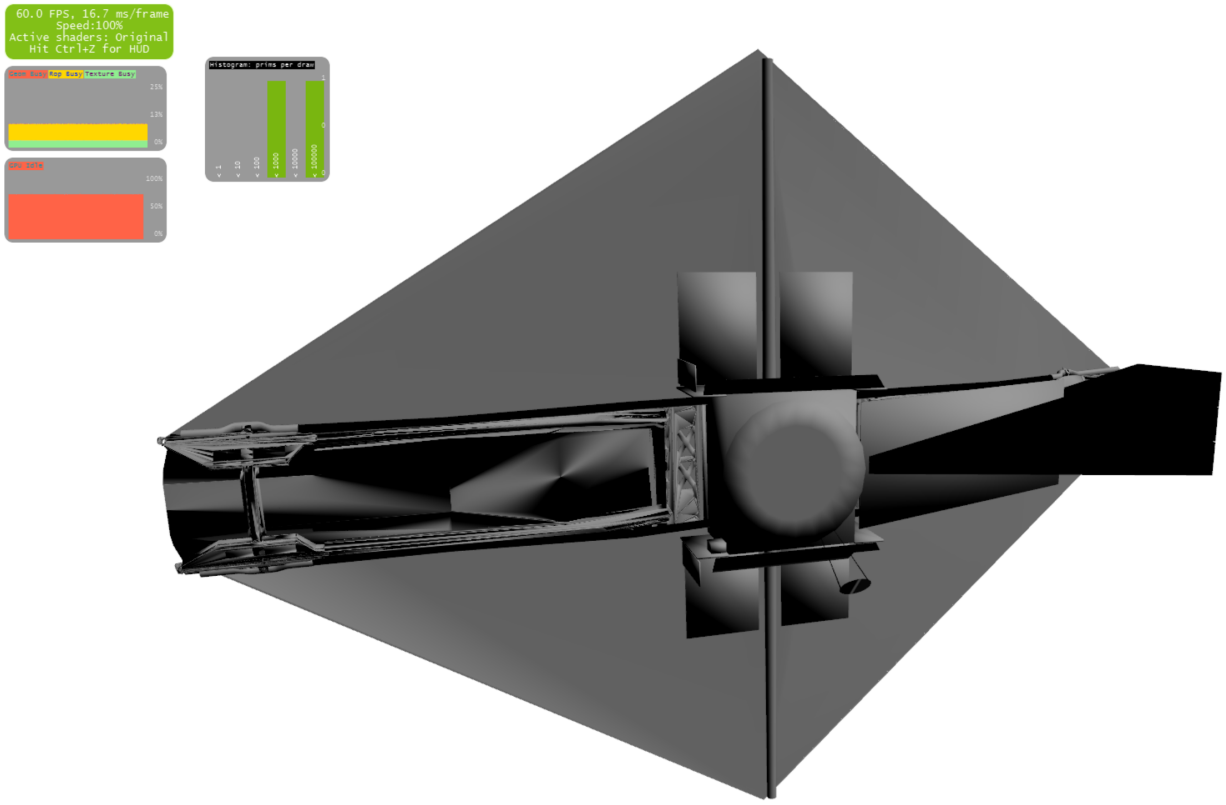


Figure 7: Graphics Profiling of SRP analysis on the sun-facing portion of the James Webb Space Telescope

8. Conclusions

Modern graphics processing units are designed and built to perform vector math as efficiently as possible, so it comes as no surprise that they perform these calculations at several orders of magnitude faster than the CPU counterparts which are designed to handle a broader range of problems. This study has shown that they provide a significant advantage to computing a high fidelity SRP analysis. Other methods for performing this computation on the GPUs exist. CUDA and OpenCL are both options for performing the parallel computations as well. OpenGL was chosen for its heritage in graphics computing. There are several optimizations built into the language itself for loading complex models which relieves the user from performing many other calculations for the models. The main benefit of using OpenGL was that the analysis could focus on the math involved in performing the SRP computation for a single polygon correctly and the hardware and software structure it is built on top of handled complex interpolation and parallelization automatically. A CUDA or OpenCL solution might provide the program with more direct control over how the model is interpolated, but the programmer would not gain the advantage of having the hardware support for the interpolated structure. This analysis has shown a distinct advantage to using GPUs and the OpenGL support to perform high fidelity solar radiation pressure analysis on extremely complex models, obtaining several orders of magnitude in performance increase for the complex models.

9. References

[1] Russell, R. "Survey of Spacecraft Trajectory Design in Strongly Perturbed Environments." Journal of Guidance, Control, and Dynamics, Vol. 35, No. 3, pp. 705–720, 2012.

[2] McInnes, C. R. Solar Sailing: Technology, Dynamics and Mission Applications. Springer, 2004.

[3] Cheng, M., Ries, J., and Tapley, B. "Assessment of the Solar Radiation Model for GRACE Orbit Determination." Advances in the Astronautical Sciences, Vol. 129, pp. 501–510, 2008.

[4] Wright, R. S., Haemel, N., Sellers, G. M., and Lipchak, B. OpenGL SuperBible: Comprehensive Tutorial and Reference. Pearson Education, 2010.

[5] Wolff, D. OpenGL 4 Shading Language Cookbook. Packt Publishing Ltd, 2013.

[6] a.i. solutions Inc. "FreeFlyer 6.9.1." `http://www.ai-solutions.com/FreeFlyer`, 2014.

[7] Montenbruck, O. and Gill, E. Satellite orbits. Springer, 2000.

[8] MATLAB. version 8.2.0.701 (R2013b). The MathWorks Inc., Natick, Massachusetts, 2013.