Efficient client-side high-fidelity propagation and visualization for large numbers of RSOs

Bill McClintock[1], Duane Cornish[2], Chris Tapley[2], Sarar Aseer[2] and Tom Kelecy[1]
[1]*Stratagem Group, Aurora, CO USA*
[2]*Stratagem Group, Valley Forge, PA USA*

ABSTRACT

Propagation of Resident Space Objects (RSOs) with high-fidelity force models poses a challenge for most computational platforms when the number of RSOs exceeds a few 10's to (at most) 100's of RSOs. High fidelity modeling will require both Low Earth Orbit (LEO) models and Geosynchronous Earth Orbit (GEO) models to account for the Earth gravity, atmospheric drag perturbations, and solar radiation pressure (SRP) for High Area-to-Mass Ratio (HAMR) debris objects. Satellite break-ups can result in hundreds to thousands of objects requiring propagation in conjunction with the existing population of tracked RSOs (10's of thousands). This can present a daunting challenge for propagation and visualization of the dynamic space environment. This work applies modern software technologies to legacy propagation algorithms to enable low-latency client-side computation and visualization. Technologies utilized include Rust and Web Assembly (WASM) with an eventual goal of client-side processing via hosted Graphics Processing Units (GPUs) and modern browser-based visualization technologies to address this challenge. For this research, we implement a high-fidelity propagator employed in modern web technologies and compare performance to current server-based processing technologies. We utilize a GEO break-up dataset with 500 debris objects and propagate the objects over a week of time. Our work also includes an implementation of a 3-dimensional graphical visualization using Cesium to display the propagation. Results of this research have yielded a multi-threaded increase of 12.5 times speed up in execution performance over the single-threaded legacy variant.

Keywords: Debris break-up; Orbit propagation; High Area-to-Mass Ratio (HAMR), RUST, Web Assembly (WASM), Graphics Processing Unit (GPU).

1. INTRODUCTION

Propagation of orbits from a set of initial conditions, or in the context of orbit determination from the last measurement update, is a well-developed process. As the number of Resident Space Objects (RSOs) in Earth orbit grow, the ability to maintain accurate situational awareness requires timely updates and re-propagation as measurement updates occur. Much of the RSO population consists of orbital debris which can be challenging to maintain custody of, especially High Area-to-Mass Ratio (HAMR) debris objects [1]. When break-ups or collisions occur, timely orbit updates of the newly created debris objects can require the propagation of 100's if not 1000's of new RSO orbits**.** If a sigma point approach is used for covariance propagation as well [2], the dimension of the states requiring propagation increases to *2n+1* for each RSO thus compounding the problem.

Many existing propagation models utilize large server-based environments (Figure 1) to precompute and store large amounts of ephemerides. These ephemerides are available for query by external consumers where potentially large amounts of data are transmitted to the client for analysis or rendering. The computationally heavy operations (orbit propagation) are completed using server resources while the lighter operations (display, etc.) are handled on the client.

Figure 1: Typical processing paradigm for orbit propagation

Figure 2 shows the architecture of our proposed concept of operations. This paradigm shift moves the compute-heavy propagation operations to the client CPU and/or GPU. This is a significant shift that could have beneficial impacts when large numbers of RSO ephemerides need to be updated or re-propagated and visualized in near real-time.

Figure 2: Proposed processing paradigm for orbit propagation

2. PROBLEM STATEMENT, ASSUMPTIONS AND GOALS

The computational complexity in propagating a given orbit is a function of the force model fidelity, the number of RSO states to be propagated, the number of state elements, the programming language being used and the hardware resources available. Recent work has examined a variety of adaptive techniques to improve computational efficiency without sacrificing accuracy [3, 4]. The work presented in this paper focuses on the implementation challenges in propagating large numbers of RSO states. Specifically, it looks at the improvements that result in utilizing the modern programming language called Rust [5, 6].

Rust is a multi-paradigm, general-purpose programming language designed for performance and safety, especially safe concurrency [5]. It is known for enforcing "memory safety" where all references point to valid memory without a "garbage collector" or "reference counting" used in other memory-safe languages. Hence, Rust could be well suited for both high performance applications such as integrating large numbers of RSO states, as well as providing a robust computational platform appropriate for an operational environment.

Additionally, Rust has a built-in tool chain to generate web assembly modules (WASM). WASM offers the ability to execute at near native speed within the web browser, offering the benefits of native code execution within a web-based environment [7]. Benchmarks comparing WASM to native Rust executables show similar timing and consistency of results.

A goal of this research effort is to demonstrate the benefits of modern programming languages and rapid RSO propagation and visualization when updates are implemented using commodity compute resources on average client hardware. As the number of tracked space objects increases, and the demand for near real-time updates to the dynamic and physical attributes increases, so too will the need for rapid real-time visualization. Tracking and asynchronous updates for 10's of thousands of predicted ephemerides makes the traditional implementation of large datasets transmitted for visualization infeasible as a performant solution.

3. RESEARCH METHODOLOGY

The first step of our research is to translate an existing RSO orbit propagation baseline written in CSharp (C#) to Rust. After testing and validating the new the Rust propagator, we leverage the Rust – WASM integration to execute the orbit propagator code in a modern web browser. Lastly, we implement a web-based UI and visualize the data using Cesium [14].

Starting with a C# codebase, a baseline implementation of a high-fidelity orbit propagator, we refactor and streamline to get the best possible performance without fully re-architecting. Following that process, we implement new main driver logic to allow the integrators to be executed on demand with varying input arguments, options, and output formats. This baseline serves as the basis for all runtime and computational comparisons to the Rust and WASM solutions.

The Rust implementation is nearly a line-by-line port of the code from C#. Differences in memory management, syntax and execution platforms make C# and Rust very different languages. Porting from C# to Rust requires many changes to the baseline; the most prominent of which is the migration away from an object-orientated (OO) paradigm to a data-oriented paradigm. Additionally, Rust implements other strategies for memory safety and data ownership which requires some rather large deviations from the original C# design. Ultimately, this paradigm shift may prove to be valuable as the research enters the WebGPU phase where the software will require more refactoring.

The Rust programming language provides WASM hooks to migrate the Rust code to a browser-based platform with minimal effort. Leveraging the Rust/WASM build process, we migrate the Rust propagator implementation to WASM and furthermore, we develop driver logic in JavaScript that calls the WASM implementation. We implement only minor alterations to move from file-based Rust operations for suppled inputs in data files to the JavaScript centric model of loading resource data across a network. For execution within a browser, we develop a simple web page for setting data inputs and configurations. This web page allows us to execute orbit propagation using the WASM/Rust module and provide results to the browser for analysis.

Lastly, we leverage the Cesium platform to integrate a 3D graphical display into our web-based front-end. Results from the Rust/WASM propagations are then injected into the Cesium platform. Within the Cesium display, we render each RSO as a sphere where the sphere radius is an indicator of the Area to Mass ratios (C_rA/m) and the sphere's color represents the RSO's speed (magnitude of the velocity vector). This helps a user more easily visually classify different objects based on attributes tied to the dynamics. For example, larger RSOs are more greatly affected by solar radiation pressure, and a user or analyst can see relationships between the propagated orbit and resulting velocity of RSOs with a greater C_rA/m as compared to propagated orbits of RSOs with a lesser C_rA/m. Additionally, coloring is implemented in a traffic light style to indicate faster and slower moving object where red is slow, and green is fast.

4. SIMULATED BREAKUP STATES AND PROPAGTION MODEL

Our propagator relies on an integration method to propagate each state through time. The integrator combines the various forces acting on each RSO.

Our high-fidelity RSO propagation model includes the following forces:

- EGM96 Earth Gravity model (using a 12 x 12 model)
- Sun and Moon third-body gravitational influences
- Solar Radio Pressure (using the 'cannonball' model)

We exclude forces that have trivial effects on an RSO in a GEO orbit (atmospheric drag, tide models, etc.). We implement our propagation using two variants of integrators: 1) Shanks 8th order and 2) the Runge-Kutta 4th order. Shanks 8th order integration is used for all performance metrics. Furthermore, propagation is limited just to object states and excludes covariance propagation. However, if the sigma point covariance propagation technique is used, our framework will simply increase the total number of states to propagate. Hence our analysis can easily extrapolate to including state covariances.

Inputs to the integrator are supplied from a user interface and are adjustable to simulate various integration techniques or considerations. Propagation is controlled by a time range, ephemeris step sizes, output units and reference frames.

To test performance, we use simulated data from a geosynchronous breakup event where 500 smaller debris objects originated from a single RSO in geosynchronous (GEO) orbit. This data is derived from NASA's breakup model [10, 11]. These 500 states consist of simple state vectors and area to mass ratios at the instant of breakup.

Each state contains:

- 3D position and 3D velocity vector
- Solar radiation parameter $\gamma = C_rA/m$

The solar radiation parameter γ is an area-to-mass-ratio, where C_r is the solar radiation pressure coefficient, A is the effective RSO cross-sectional area and m is the RSO mass. Figure 3 shows the distribution of the area-to-mass values within the breakup model. The longer-term evolution of the orbital elements will result from the combined perturbations over extended propagation periods for the HAMR objects [12]. Figure 5 shows the distribution of ΔV magnitudes for the 500 debris pieces derived from the NASA break-up model. Figure 7 is a dispersion diagram for the Cartesian vector break-up ΔV components. The dispersion illustrates the resulting breakup debris objects caused by an impact with the "outlier", likely the originating impactor. The histograms shown in Figure 4 - Figure 8 show the eccentricity, period, and inclination distributions of the initial orbital states.

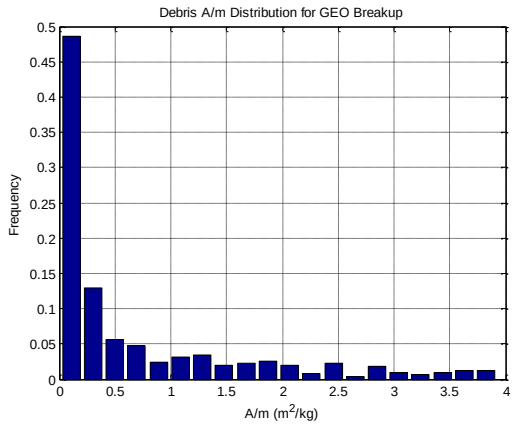


Figure 3: Debris cloud area-to mass distribution

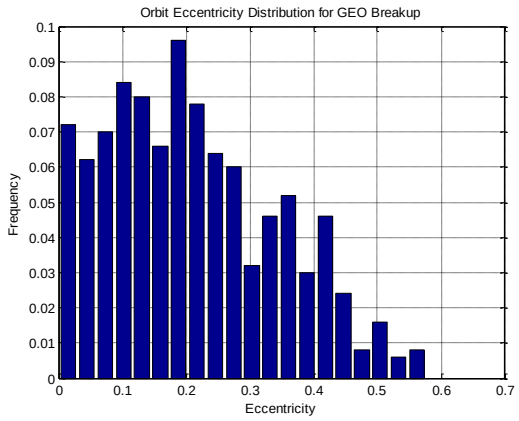


Figure 4: Debris cloud eccentricity distribution

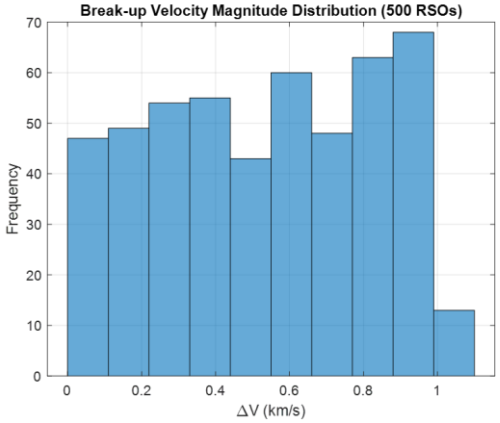


Figure 5: Debris ΔV histogram

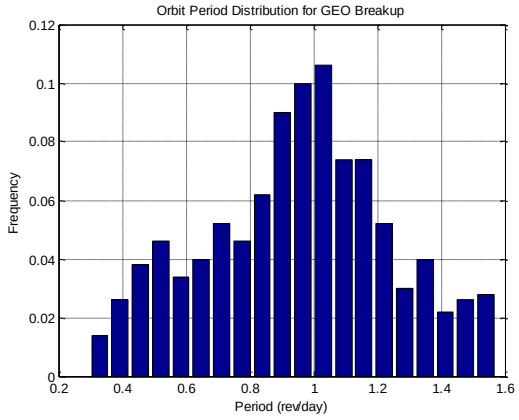


Figure 6: Debris cloud eccentricity distribution

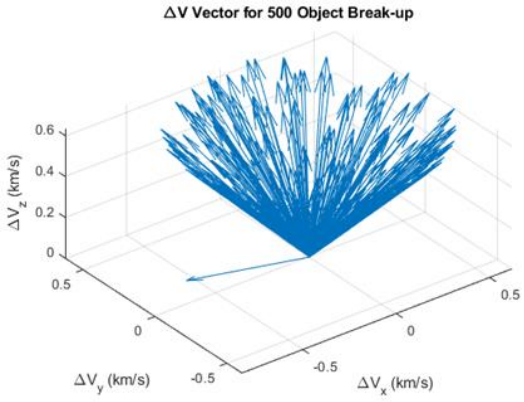


Figure 7: Debris ΔV vector dispersions

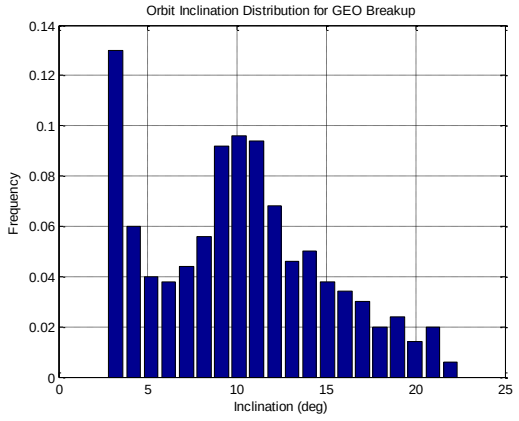


Figure 8: Debris cloud inclination distribution

5. PROPAGATION COMPARISONS AND PERFORMANCE RESULTS

This section provides the details on the runtime analysis conducted in order to compare C#, Rust and WASM implementations. Results quantify the execution runtime in seconds for the various implementations and reveal the processing time advantages attainable with modern software implementations and architectures. We create separate input files with 1, 5, 10, 50, 100, and 500 states from the full 500-state GEO breakup data. We propagate each of the state files from the RSO breakup model five times for each propagator implementation (C#, Rust, and WASM). This approach allowed us to compute an average result that is less susceptible to a single run that could potentially be affected by the computer running background services, other tasks, etc. Table 1 provides the specifications of the execution hardware we used for all analysis. Table 2 shows the single set of execution parameters and inputs that we used for all executions.

Table 1: Runtime hardware specifications

Manufacturer / Model	Hewlett Packard / HP ZBook Fury 15 G7
CPU / RAM	Intel Core i9-10885H @ 2.40 GHz / 32.0 GB
GPU #0	Intel UHD Graphics – Driver 30.0.101.1404
GPU #1	NVIDIA Quadro T2000 with Max-Q Design – Driver 30.0.14.7298
Browser	Microsoft Edge Version 103.0.1264.71 (Official build) (64-bit)

Table 2: Propagator input configuration for all executions

Start / Stop Time	2004-06-30 11:58pm / 2004-07-07 11:58pm
Integrator	Shanks 8th Order
Integration Step Size	300 seconds
Force Models	Sun / Moon / Srp
Reference Frame	J2000 ECI
State Output Units	KM

Table 3 and Table 4 show the mean and standard deviation of runtimes in seconds across 5 executions of each propagation implementation for increasing RSO counts (1-500). The results indicate a performance improvement (over all 5-runs) for Rust compared to C#. Execution times of the Rust (WASM) compared to the Rust native times showed little difference. The single-threaded implementation of Rust is roughly twice as fast as the single-threaded C# implementation. The Rust (WASM) executions are within 95% of the run time of native Rust executions. The Rust threaded performance is an order of magnitude faster than the threaded C# implementation. This may be a factor of the managed vs. native platforms and thread handling differences. Figure 9 shows a similar representation of the data on a linear scale chart of the average run times in seconds. Notice the order of magnitude improvement of multi-threaded rust compared to single threaded rust.

Table 3: Single-threaded propagation execution times (seconds) averaged across 5 runs

RSO Count	1	5	10	50	100	500
C#	0.56 ± 0.008	1.97 ± 0.007	3.67 ± 0.045	17.45 ± 0.065	34.45 ± 0.093	170.83 ± 0.513
Rust	0.17 ± 0.003	0.84 ± 0.006	1.66 ± 0.009	8.23 ± 0.023	16.43 ± 0.026	82.23 ± 0.234
Rust/WASM	0.20 ± 0.006	0.88 ± 0.006	1.73 ± 0.013	8.48 ± 0.151	17.00 ± 0.075	85.42 ± 0.309

Table 4: Multi-threaded propagation execution times (seconds) averaged across 5 runs

RSO Count	1	5	10	50	100	500
C#	0.75 ± 0.090	0.82 ± 0.004	1.47 ± 0.059	7.56 ± 0.156	16.96 ± 0.493	117.64 ± 4.364
Rust	0.17 ± 0.002	0.20 ± 0.004	0.24 ± 0.003	1.03 ± 0.005	1.94 ± 0.007	9.41 ± 0.014

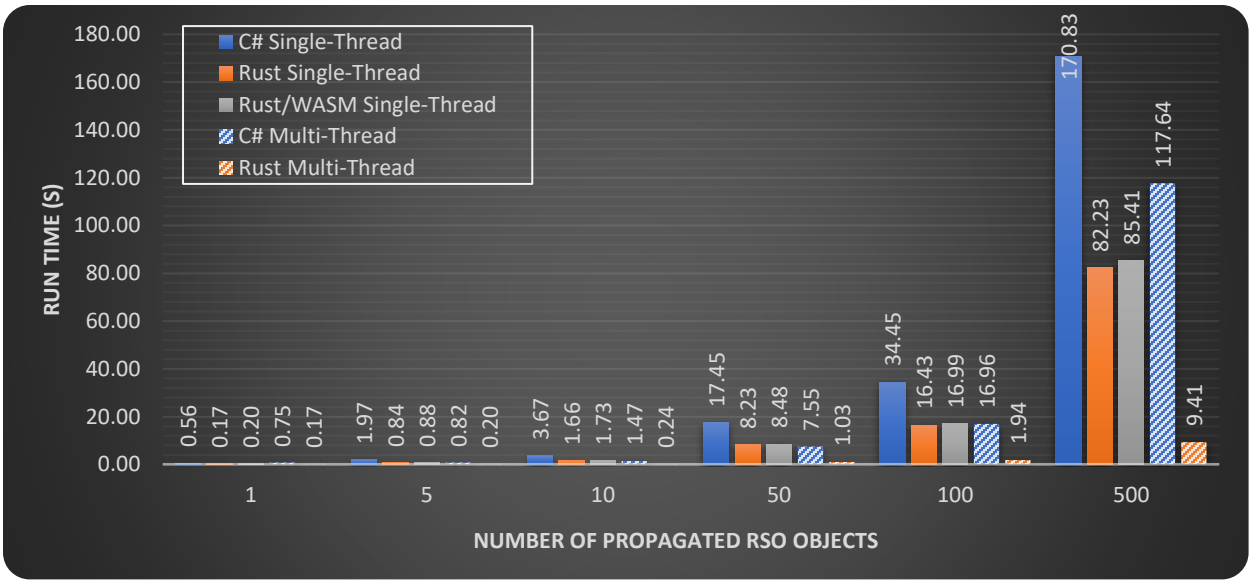
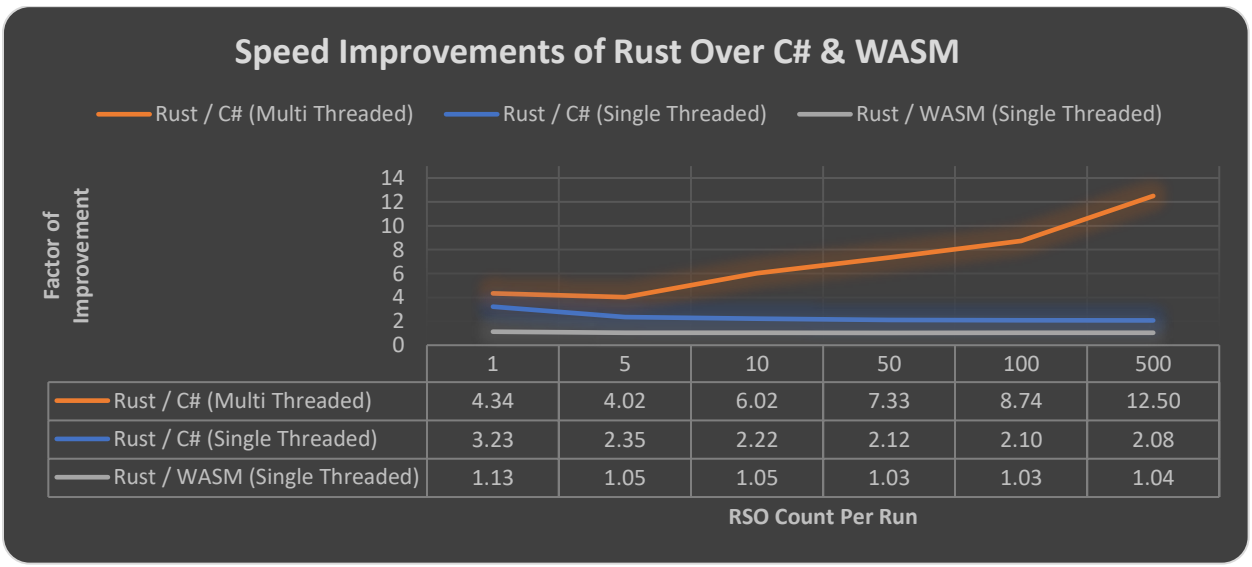


Figure 9: Average runtime (seconds) of C# (blue), Rust (orange), Rust/WASM (gray) orbit propagation implementations. Solid-colored bars are single-threaded, while lined-bars are multi-threaded. Results are clustered by the number of RSOs propagated.

Figure 10 shows the factor of improvement gained from using Rust over C#. The units for this chart are a simple multiplier based on the ratio of relative runtimes, where the Rust multithreaded implementation is roughly 12.5 times faster than its equivalent threaded implementation in C#. These comparisons are for matching threading configurations: (a) single threaded vs. single threaded and (b) multi-threaded vs. multi-threaded. Additionally, the Rust vs. WASM comparison is for single threaded runs and shows how similar the WASM (in-browser implementation) compares against the native runtime at a nearly 1.0 average runtime. This indicates that the WASM solution is nearly as fast as Rust native implementations.



	1	5	10	50	100	500
Rust / C# (Multi Threaded)	4.34	4.02	6.02	7.33	8.74	12.50
Rust / C# (Single Threaded)	3.23	2.35	2.22	2.12	2.10	2.08
Rust / WASM (Single Threaded)	1.13	1.05	1.05	1.03	1.03	1.04

Figure 10: Runtime improvements with Rust. This table shows the average runtime of rust divided by each of the other implementations, categorized by the number of RSOs propagated.

6. VISUALIZATION PERFORMANCE

For this research, we use a web-based visualization tool called Cesium to animate the propagation results from our WASM implementation. The ephemerides computed by the web-assembly Rust module are passed to the Cesium component of our web-based front-end for display.

Table 5 shows the runtime performance numbers as mean values in seconds plus or minus a standard deviation of the various parts of the execution process. The table defines the compute, convert, extract, transfer, and overall execution times of the runs. The compute phase of the operation is the Rust/WASM logic executing the initial propagation of the elements over the time specified. The convert phase is the WASM operation that converts the raw ephemeris data to a textual representation of the data suitable for transmission back to JavaScript across the WASM/JavaScript bridge. The extract and transfer phases are the times to get the data across that bridge and back down to the visualization implementation used in this paper. In future development, the team expects that the convert, extract, and transfer phases will be eliminated when the need for transference to 3rd party visualization is eliminated.

Table 5: Performance numbers of WASM executions through visualization

Samples	Compute	Convert	Extract	Transfer	Overall
1	0.185+/-0.007	0.005+/-0.001	0.001+/-0.000	0.006+/-0.003	0.197+/-0.007
5	0.846+/-0.011	0.026+/-0.001	0.004+/-0.000	0.021+/-0.002	0.897+/-0.011
10	1.701+/-0.019	0.050+/-0.003	0.008+/-0.001	0.038+/-0.001	1.798+/-0.019
50	8.622+/-0.051	0.328+/-0.114	0.057+/-0.005	0.198+/-0.004	9.209+/-0.135
100	17.082+/-0.223	2.387+/-0.221	0.146+/-0.010	0.387+/-0.001	20.016+/-0.395
500	84.514+/-1.607	18.532+/-5.875	0.655+/-0.056	1.996+/-0.025	105.763+/-5.888

Figure 11 and Figure 12 show the distribution of the debris field after roughly 24 hours of propagation as presented by our Cesium visualization. In Figure 12, RSO objects are depicted as spheres, yet we also have turned on "tracks" which show the trajectory of each RSO over the past 4 hours. This distribution clearly shows the spread of the debris from GEO orbit to orbits much closer and further away that could impact spacecraft in all orbital regimes. The color distribution shows the objects with slower speed (Red) and objects with a greater speed (Green). We define speed as the magnitude of the velocity vector in the Geocentric Celestial Reference Frame (GCRF) which is an inertial reference frame. Additionally, object size is representative of the relative C_rA/m of the object which is constant throughout the simulation. Figure 11 shows where the RSOs and their tracks project down onto the earth in a 2D map, while Figure 12 shows a 3D model of the earth.

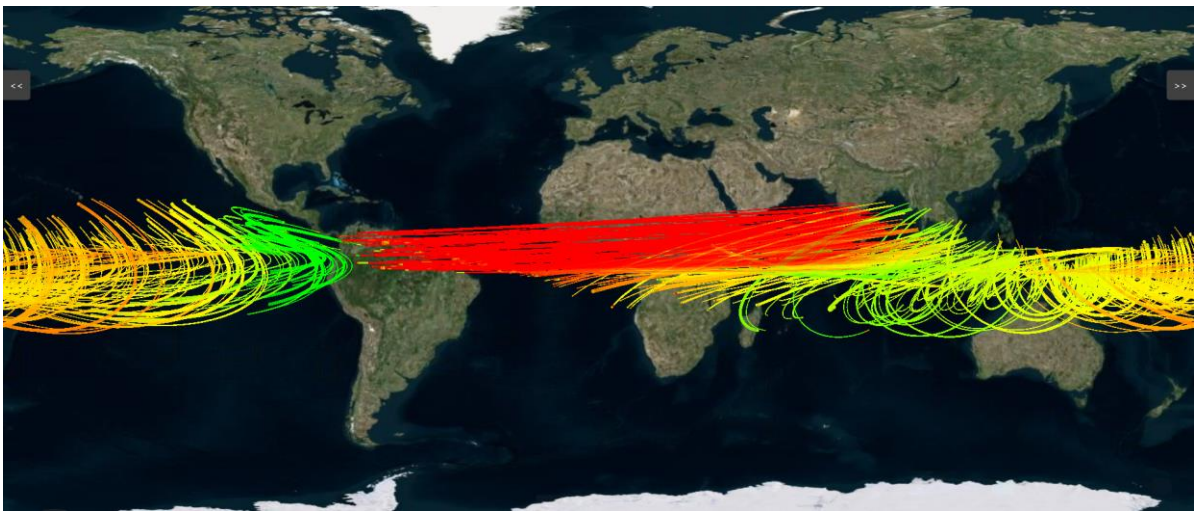


Figure 11: Rendering from our Cesium UI that displays ground traces projected onto the Earth from a 500-RSO GEO break-up model after roughly 24-hours of propagation.

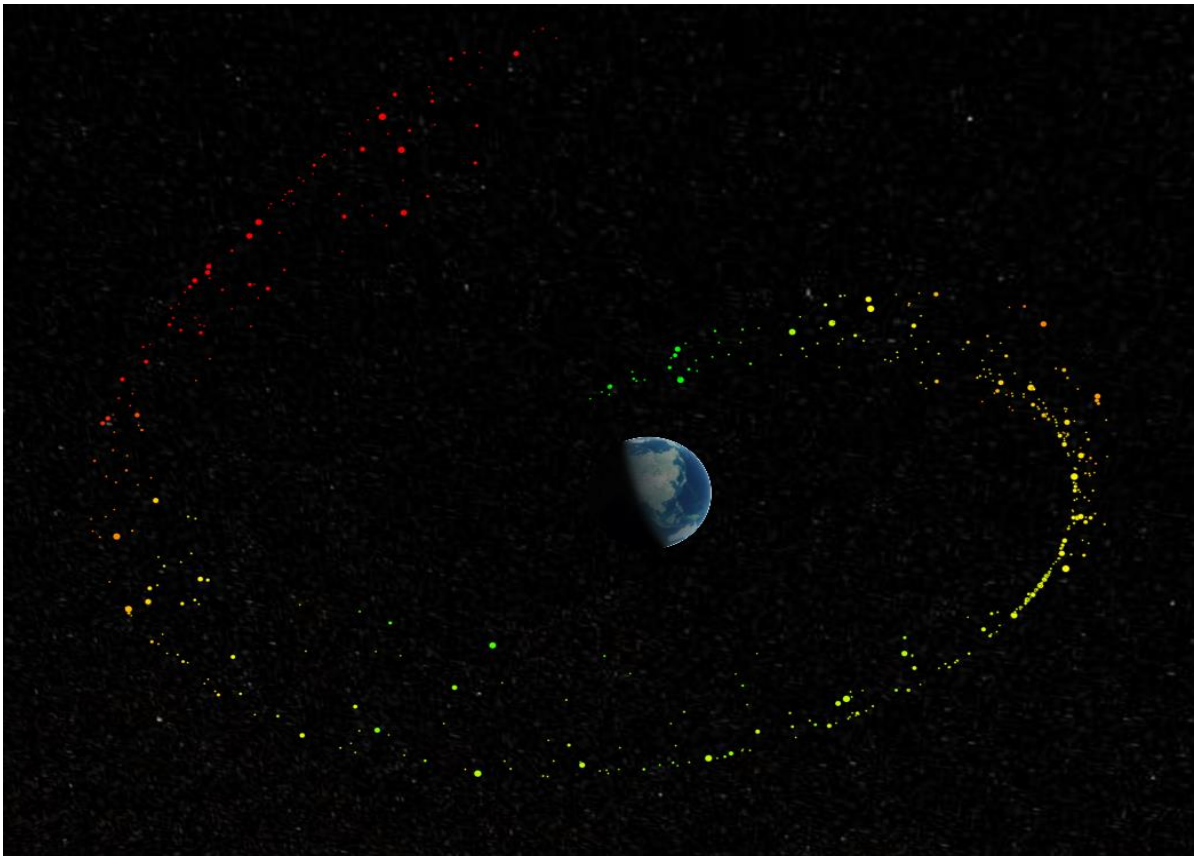


Figure 12: A 3D Cesium rendering that displays 500 RSOs from the GEO break-up model which have been propagated 24 hours from the initial breakup.

7. CONCLUSIONS

Operators and analysts requiring rapid updates and propagation of states benefit from both the improved propagation speed and efficiency. This work employs modern visualization mechanisms that can leverage the computational speed of Rust and display large numbers of RSOs being simultaneously propagated and visualized.

The primary benefit from this work is the ability to receive ephemeris updates for many RSOs and visualize orbits and metadata in real-time. RSO metadata can include country of origin, physical attributes, operational status, etc. In this way, we can manage a dynamic, ever-changing environment in near real-time and enable operational users to react quickly to potential flight safety risks.

This research has shown that the implementation of modern technologies within the client web browser has the potential to offer significant improvement in orbit propagation as compared to legacy implementations. We have shown that multi-threaded implementations can deliver upwards of 12 times the performance. Additionally, the WASM implementation nearly matches the performance of the native Rust implementation, which demonstrates the potential for real-time propagation in the browser.

The key takeaways from this work include:

1. Modern software languages and techniques show greater performance over legacy approaches.
2. The results show great promise for providing near real-time propagation and visualization updates to support rapid maintenance and visualization capabilities.
3. Leveraging existing rendering techniques is sufficient for interaction, yet we believe even greater improvement can be made by co-locating both the propagation and the visualization on the GPU. See section 8 for a future research explanation.

8. FUTURE WORK

For future research we propose the use of WebGPU which is the working name for a cutting-edge web standard and JavaScript API for accelerated graphics and compute, aiming to provide "modern 3D graphics and computation capabilities". Unlike WebGL (OpenGL for the web), WebGPU is not a direct port of any legacy native API [8]. It is based on modern APIs provided by Vulkan, Metal, and Direct3D 12 and WebGPU intended to enable “high-performance 3D graphics and data-parallel computation on the web” across both mobile and desktop platforms [9]. This could enable more robust and portable visualization implementation from a variety of platforms to support operations and timely situational awareness.

WebGPU will open a new range of possibilities from GPU computation and modern graphical rendering and performance improvements. WebGPU offers the possibility of offloading the computational burden to the client-side GPU and could offer exponential gains in performance. Modern GPUs have many of thousands of smaller compute cores that are designed for number computations. If the power of this hardware can be realized for propagation, we can parallelize execution and greatly accelerate orbit propagation for large numbers of RSOs.

Additionally, integrating this GPU resident data with new visualizations could offer a new dynamic to real time data manipulation. We also believe there is great benefit to co-locating a new visualization methodology with the propagation code on the GPU. This approach would reduce the amount of data transfer between a host CPU and the GPU, and it could more easily enable the ability to render RSO states on demand. By keeping this data resident on the GPU, the potential gains for display may be realized and real-time or near real-time manipulation of the data on the GPU can be accomplished by augmentations on the input parameters. This ability to change the data on large volumes quickly for immediate display could offer areas of post-analysis yet realized, included improved collision detection.

With the growing number of mega-satellite constellations planned for LEO, some including as many as 12,000 operational satellites [13], there will be much greater challenges for propagation and visualization of RSOs in that orbit regime. Hence, future work will expand on this work to include propagation and visualization of large numbers of LEO objects. Specifically, a useful future system will be able to update orbit ephemerides and provide graphics that enable distinction between constellations operated by commercial entities and the large number of other operational satellites and debris.

9. ACKNOWLEDGEMENTS

We would like to thank the Stratagem Leadership team – Brian Bontempo, Ben Avicolli, Jason Putnam, and Jared Hershman – for having the trust and confidence in supporting this research through precious internal funding. We also thank Brian Bontempo and Rad Widman for their valuable review and helpful comments.

10. REFERENCES

1. Schildknecht, T., “Optical surveys for space debris,” Astron Astrophys Rev (2007) 14:41-111 (DOI 10.1007/s00159-006-0003-9).
2. J. Stauch and M. Jah, On the Unscented Schmidt-Kalman Filter Algorithm. Journal of Guidance, Control, and Dynamics 38(1): 117-123, 2014.
3. Wang, X., T. Elgohary and S. Atluri, “An Adaptive Local Variational Integration Method for Orbit Propagation and Strongly Nonlinear Dynamic Systems,” AIAA Scitech 2020 Forum, Orlando, FL, January 5, 2020.
4. Jones, B., E. Delande, E. Zucchellie, and M. Jah, “Multi-Fidelity Orbit Uncertainty Propagation with Systematic Errors, 20th AMOS Technical Conference, Sept. 17-20, 2019, Wailea, Maui, HI.
5. Hoare, Graydon (December 28, 2016). "Rust is mostly safety". Graydon2. Dreamwidth Studios. Archived from the original on May 2, 2019. Retrieved May 13, 2019.
6. "FAQ – The Rust Project". Rust-lang.org. Archived from the original on June 9, 2016. Retrieved June 27, 2019.
7. WASM: https://webassembly.org/
8. Legacy native API: https://gpuweb.github.io/gpuweb/
9. Chrome Developers: https://developer.chrome.com/docs/web-platform/webgpu/
10. NASA/JSC orbital debris models: https://ntrs.nasa.gov/citations/19980005919.

11. Bade, A. & Jackson, Albert & Reynolds, R.C. & Eichler, P. & Krisko, Paula & Matneyi, M. & Anz-Meador, Phillip & Johnson, Breakup model update at NASA/JSC. 125-138, 2000.
12. Kelecy, T. and M. Jah, "Analysis of high area-to-mass ration (HAMR) GEO space object orbit determination and prediction performance: Initial strategies to recover and predict HAMR GEO trajectories with no a priori information," Acta Astronautica, Volume 69, Issues 7-8, September-October 2011 (pp 551-558).
13. Mega-constellations: https://www.fromspacewithlove.com/satellite-mega-constellations/
14. Cesium: https://www.cesium.com