

# Multi-Mission, Interactive 3D Visualization in a Web Browser for Robotic System and Space Flight Mission Development and Operations

M. I. Pomerantz, A. Boettcher, M. Hans, M. Sandoval, S. Wenzel  
*Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 91109*

**Abstract**—Leveraging on our experience developing, engineering accurate, interactive 3D visualization systems for robotic system and flight mission formulation, development and operations, our team at the Jet Propulsion Laboratory has built a high performance, interactive 3D visualization system called Ranger, that runs in a web browser and has been designed for use during the entire mission life-cycle. This includes mission formulation through operations, robotic system research, flight mission design and development, flight mission operations, simulation test-bed experiments and in-the-field testing. Our Ranger system not only provides robotic missions with a to-scale situational awareness visualization capability, but also an interactive commanding tool whereby user interactions in the web browser can be communicated back through the system to robotic system or spacecraft control software. In this paper we will discuss the Ranger system architecture, rendering performance, current use cases and deployments, system development challenges, as well as a newly implemented fully immersive VR system capability.

## I. INTRODUCTION

In recent years, the performance of 3D graphics-intensive software, running inside a typical web browser, has improved dramatically through the use of WebGL [1] and JavaScript [2] and has allowed visualization system developers to move away from workstation-class systems with expensive discrete graphics cards, to more affordable and common place desktop/laptop systems and mobile devices such as iOS and Android systems.

With a goal to provide a re-useable, cost-effective, multi-mission 3D situational awareness capability across the various types of robotic and spacecraft missions developed at JPL and to continue to support the typical interactive, telemetry and simulation-driven robotic system and flight mission visualizations that our team has typically developed in the recent past, but with a simpler cross-platform deployment model, we have developed a WebGL/JavaScript rendering engine called Ranger with an accompanying back-end called Flow, that is completely data driven and can be used across our varied and multi-mission user base. Similar to our past systems used by the NASA/JPL Mars Science Laboratory [3] (MSL) and the NASA/JPL Low Density Supersonic Decelerator [4][5] (LDS) projects, our new system is currently in use by JPL Office of Naval Research swarms of autonomous surface boat tasks, JPL mission formulation and the NASA/JPL Mars M2020 (M2020) [6] mission. The

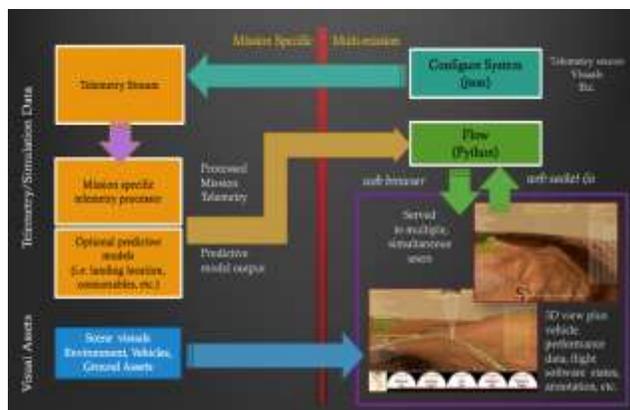


Figure 1. Mission specific data drives the multi-mission Ranger/Flow system.

Ranger system makes use of robotic and flight system CAD models, high resolution terrain models, planetary data for relevant bodies, streaming vehicle telemetry or simulation-generated data as well as ancillary information such as telemetry alarm limits, flight software states and environment data.

## II. GOALS AND ARCHITECTURE

Ranger's client-server system architecture, Fig. 1, allows multiple, simultaneous users to view telemetry data streamed live during vehicle operations, from test-bed simulations or from stored data during playback, and because our Flow server has been designed to stream data from multiple simultaneous streams, live telemetry can be intermixed with previously generated predicted data or other simulation generated data.

### A. System Goals

Based on our past experience working with JPL robotic system and flight mission customers, we converged on a set of goals that we used to drive our system design and development.

- Build an easily re-useable, multi-mission, data driven system that provides a cost-effective solution for building mission visualization applications more rapidly than previous legacy systems and is easy to deploy on a variety of desktop and hand-held devices.

- Maintain engineering accuracy and to-scale visuals with enough precision to allow mission engineers to safely command vehicles, develop and test vehicle control software and algorithms and determine state and health of the vehicle(s) during tests and operations.
- Support multiple, simultaneous visualization application clients in web browsers on a variety of hardware devices.
- Support data provided by multiple simultaneous data sources. Either streamed live, from log file playback or a combination of both.
- Provide an easy to use server interface for providing mission-specific data processing, visuals and/or vehicle commanding.
- Provide robust exception handling and connect/re-connect capability for clients, to ensure stable system operation during critical mission events.
- Support the typical flight and robotic mission hardware configurations encountered at JPL and other NASA centers, including support to accurately visualize vehicle articulation.
- Peer review new system features with code reviews and use continuous, automated integration and unit test execution during system development.

### B. Ranger System Architecture, Accuracy and UI

The Ranger system is essentially a client-server system with a Python v3.7 backend called Flow and a WebGL/JavaScript rendering engine called Ranger. The Ranger rendering engine core is written in C++ and OpenGL [7] and makes use of features available to a typical OpenGL ES 2.0 renderer, including GLSL [8] shaders that are used to re-program the target device's graphics processing unit (GPU). The Ranger C++ code does the heavy lifting regarding ephemerides calculations, memory allocation and other computational utilities such as graphics primitive generation (cones, spheres, etc.). Because the core, multi-mission part of Ranger is C++ code, in the future we plan to investigate linking existing code-bases such as the JPL NAIF [9] library and customer provided predictive models directly into Ranger. Because C++ and OpenGL code does not directly execute inside a web browser, we use the open source Emscripten [10] tool to compile Ranger's C++ and OpenGL into WebGL and asm.js [11] or WebAssembly [12]. As a note, asm.js performance is substantially better than pure JavaScript and is ~10-50% of native C++ code. While we have not done extensive performance testing, our Eclipse app can render at ~60fps on a Mac laptop. Fig. 3. WebAssembly performance is advertised to be near native C++. We plan to confirm these number in future Ranger performance testing.

The JavaScript code generated during the Emscripten compile process is responsible for managing Ranger's main event loop, user interactions, viewing camera actions (such as transitions from one object to another), scene-graph traversal

and evaluation and provides the interface to the user application layer. See Fig. 2.

User specific application-side logic can be built on top of the Ranger JavaScript using the system's provided JavaScript API. In addition to this API, Ranger supports a complete JSON [13] scene description and scene management system, whereby users can describe their scene without writing JavaScript at all. Simple data entry for planetary bodies and CAD-model based visuals are supported as well as GLSL code that can be embedded directly in the JSON scene description. Note that for multi-device support, users can provide a scene description with branches for desktop and mobile devices, with the desktop branch providing high resolution visual assets if desired. Ranger can detect the device at run time and choose the appropriate scene branch based on device performance.

Regarding visual asset data, Ranger supports the typical data types supported by most 3D rendering systems, such as triangle meshes, line sets, geometric primitives such as cubes, spheroids, cones, as well as texture imagery and CAD models in glTF 2.0 [14] format with Physically-Based Rendering. Ranger can also process and render streamed triangle mesh data, which can be useful when displaying the output of a simulation that may be modifying a robotic system's configuration on the fly for example. The Ranger-Eclipse application is shown in Fig. 3.

1) *Accuracy*: System accuracy is an important component of any visualization system developed to display engineering data and especially when users may make decisions regarding the health and safety of the robotic system or spacecraft that is visualized. To address this accuracy issue, and with the understanding that at best, there is limited hardware support for OpenGL and WebGL double precision, we use techniques in the Ranger engine to mitigate the lack of double precision support, especially when depicting scenes containing large Solar System magnitude numeric values.

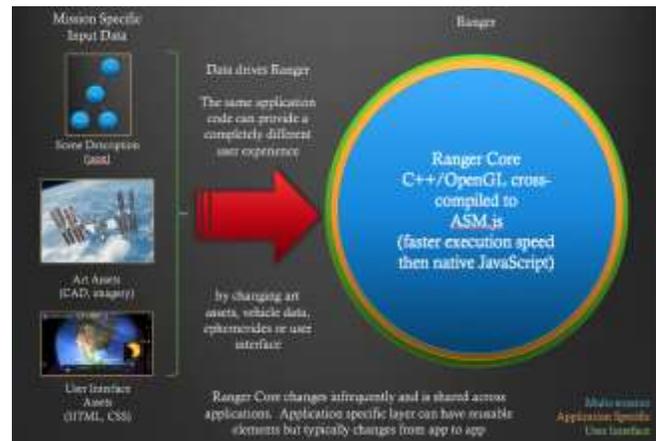


Figure 2. Ranger multi-mission core and mission specific layer



Figure 3. Ranger 2017 Solar Eclipse Application.

These techniques include:

- A frame centric system that eliminates the need for a world coordinate system. Essentially, all scene object are rendered with respect to the viewing camera and the viewing camera is always parented to the object of interest. This helps preserve numerical accuracy.
- All Ranger renderer calculation are double precision until the final transformation is sent to the GPU.
- Use of a Log Depth Buffer allows for Solar System-scale scenes while minimizing Z-fighting.

2) *Templated Scenes Description*: Ranger has a scene description template system designed to make the development of scene elements simpler and faster and with reusable and shareable components. Templates are JSON data that provide default parameters and an abstraction mechanism for scene objects, such as planetary bodies and spacecraft. Users can choose to override template default parameters and expand on existing templates to generate custom scene objects in a fashion not too dissimilar from common object-oriented language inheritance. Ranger provides pre-made Templates for planetary bodies, geometric primitives, mathematical calculations and conversions, and input data types while also allowing users to create their own templates for uses beneficial to their application. Scene construction becomes simpler because scenes can be created once and then re-used in future applications, either as-is or with changes to the Template parameters as needed. For complex scenes, Template nesting is also supported. Fig. 4.

3) *User Interface (UI) Elements*: UI elements for Ranger applications are rendered on top of an HTML5 Canvas and can be provided statically or contained in a Ranger scene description JSON blob and with Cascading Style Sheets (CSS) [15] and art asset references as needed. Whether loaded statically or read in a JSON blob, users can communicate UI interactions back to the Ranger engine using

Ranger's JavaScript API. For system-level interactions such as mouse motion and button presses, Ranger function handlers track those event and report back to the user's application code. How those events are handled is up to the application developer.

4) *Label system*: Ranger provides a basic labeling system for 3D objects in the scene, that are designed to uniquely identify objects by name, especially at distances from the viewing camera where object geometry may be difficult or impossible to resolve, by mapping user provided scene object text labels from three-dimensional space to screen space and with screen clipping determination. The look and feel of the labels is determined by user provided HTML5 and CSS.

### C. System Architecture – Flow

Flow is the server-side component of the Ranger system and is a set of Python classes that can process multiple telemetry data source streams, each on a unique Python thread, aggregate that data based on contained telemetry time, and provide that data to the Ranger renderer via a WebSocket interface. Similar to the Ranger renderer, Flow has been designed with a multi-mission core and a Python plugin/API layer that provides users with an interface to their custom, mission specific telemetry processing code. This user provided code typically is written in a way to access or provides data to Flow from some user data source and depending on user needs, can also receive messages from Flow that may be generated by user interaction in the web browser running the Ranger renderer or by computations in Flow. Because the user provided mission-specific code is written in Python, typical data structures, threads of execution, file i/o for additional configuration files, and all other typical Python constructs are supported by Flow.

At run-time, the Flow core reads a JSON configuration file that provides a mapping between expected telemetry items and visuals to be rendered in Ranger. For example, in our Mars 2020 visualization, the Flow JSON config file specifies that each spacecraft position and orientation data processed by Flow and time ordered, is to be applied to the 3D CAD model of the Mars 2020 flight system rendered in Ranger. Similar configuration items are used by Flow to change the rendered flight system configuration based on received flight software states, such as rendering an entry parachute or descent thruster visuals.

## III. USE CASES AND RECENT DEPLOYMENTS

### A. Use Cases and Target Platforms

Based on many years of experience developing real-time, interactive 3D visualization systems at JPL and in industry, we identified a core set of use cases that we felt our Ranger system should support to cover most of the typical robotic and space mission scenarios that we would encounter at JPL. These include single and multiple vehicle situational



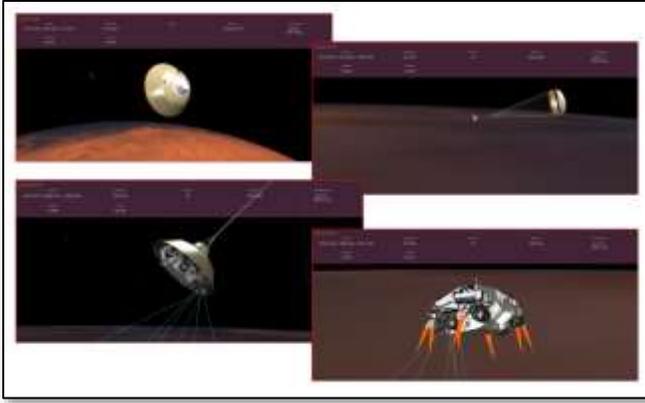


Figure 6. Ranger display depicting various stages of the M2020 flight system during simulated EDL

and complete data bundle is repeated. When displaying simulation data the process is much simpler as there is no possibility of receiving data not time-ordered. In this case, data bundles are created, from the simulation data and sent to Ranger at a rate specified by the user at run time. Typically at frame rates of 30-60hz. In addition to spacecraft telemetry data, the application will display high-resolution terrain data for the landing region, as well as visuals showing the landing error ellipse. Finally, as a departure from our 2012 MSL landing night visualization system, there is no direct coupling of a specific physics-based mission simulation environment in the Ranger system. We have chosen to support both JPL's DSEDS and NASA Langley's Program to Optimize Simulated Trajectories (POST) through the use of M2020 mission provided simulation files in Matlab format [17]. Having large amounts of simulation data provided in this manner was a challenge, but through the use of the Python SciPy [18] module, we were able to read and process the very large (300MB) simulation files directly in Flow.

3) *Ranger-Eclipse*: One of the most complex Ranger applications developed to date was built for the JPL Education and Public Outreach organization to depict and help educate the public regarding the 2017 solar eclipse seen across much of the United States. The eclipse application [19] used correct orbital dynamics to display the Sun, Earth and Moon system for the entire duration of the 2017 eclipse across much of North America. To access the application, users would enter the URL of the site serving the application on Amazon Web Services (AWS) [20], and the application code and data, including planetary body position data, texture imagery, and geometry for bodies, would download to the user's browser running on their device of choice. When using the application, users could choose a location in the path of the eclipse, scrub back and forth over time, and view a rendering of how much of the Sun would be covered by the Moon. Other interesting views were also available such as a "virtual telescope" view from the user's selected location and

looking directly at the application's virtual Sun; An "Earth, Sun, Moon" view showing Earth and Moon orbits, as well as the Ecliptic plane; An Moon/Earth view with displayed shadow umbra and penumbra cones. During the actual eclipse in August, 2017, the application had over one million users on devices ranging from iPhone 5s to more modern Android, iPhone, iPad and desktop systems. Fig. 3.

#### IV. RANGER SYSTEM DEVELOPMENT CHALLENGES

During development of the Ranger rendering engine and during initial system testing, we identified a number of challenges centered around differences in rendering performance, memory allocation and the ability to gracefully recover from application errors in the various web browsers that we had planned to support (Safari, Edge, Chrome, Firefox). We quickly found that across browser versions performance could change dramatically. To address these issues, we found it necessary to system test across all browser versions and hardware devices that we planned to support, prior to releasing a new version of the Ranger code. The large number of tests required quickly became a challenge for our small team, as we were supporting four operating systems (iOS, Android, Windows, Mac OSX), four web browsers, multiple mobile devices (iPhones 5, 6, 7, iPad variations, latest Android) and desktop Mac and Windows systems. In addition, we found that some platforms did not support the GLSL EXT\_frag\_depth which we required for our log depth buffer calculations [21]. For these platforms, we let the hardware perform native depth buffering, but in cases where z-fighting or alpha blending problems were apparent, we rendered objects in order based on distance from the viewing camera.

#### V. EMBEDDING RANGER IN A WEB PAGE

A unique deployment of Ranger which is typically unavailable to applications build to run natively on a specific operating system, is the ability to embed a Ranger application in a web page with related content. We can accomplish this using an HTML IFrame [22], which allows users to embed a Ranger web site, referenced by the site's URL, into an existing web page or by embedding a Ranger application directly into an existing web page using an HTML Div [23] container. Either method (IFrame or Div) allows for full interactivity with the embedded Ranger application, which includes both user interaction via mouse or touch events and application-to-application interaction using JavaScript function calls into Ranger. This application-to-application interface for example, can be used by the parent web page to control Ranger's camera position and pointing and provides a nice way for users to build a web page with a variety of related content, including a Ranger interactive 3D display and with control over much of Ranger's built in functionality. Fig. 7.



Figure 7. Ranger application embedded in a NASA web page

## VI. RANGER SYSTEM DOCUMENTATION

Because Ranger has evolved into a complex system designed for rapid development of applications, we have compiled over 90 Wiki pages of user-level documentation, hosted in our Github repo, with example code and images and a primary focus on system setup, application tutorials and the Ranger JavaScript API. The repo is currently only available to JPL users but there have been discussions regarding open-sourcing Ranger.

## VII. IMMERSIVE VR AND FUTURE DEVELOPMENT

In the summer of 2018, we added an immersive VR capability into the Ranger core (Fig. 5) through the use of WebXR [24], with the ultimate goal of supporting commercial VR headsets for viewing and interacting with Ranger visualizations. We were able to successfully integrate WebXR into Ranger by implementing the following.

- Recognize our HTC Vive [25] VR device and start a VR session.
- Render models in three-space in Ranger for left and right eyes and with correct eye separation and with shearing terms in the projection matrix to build the non-symmetric camera frustums [26] needed to eliminate image distortion.
- Recognize VR device position and orientation (6 degrees of freedom).
- Recognize two controller's position and orientation and render a 3D representation of the controllers in the scene.
- Support ray intersection for each controller for selection/picking in the 3D scene.
- Ability to move freely in the scene via controller gestures.
- 

We did encounter challenges during VR development in Ranger, as WebXR was relatively new and documentation and code examples were not as readily available as we would have liked, plus the only web browser that supported WebXR

was Chrome 66 and later versions. For the HTC Vive controllers, WebXR always returned the same event code no matter which button was pressed on a given controller, so in essence, we were limited to a single button per controller. Button press events were unique across the two controllers. Regarding interaction with the rendered scene, we implemented both scene translation and scene scaling based on controller gestures and a controller button press. Overall, rendering performance for our solar system-scale visualizations was good and frame rates were high enough that no users experienced nausea or dizziness during use.

## VIII. CONCLUSION

The JPL developed Ranger, web browser-based visualization system has been developed as a high-performance, easily re-configurable and cost-effective visualization system for displaying flight mission and robotic system data for use by mission development engineers and operations personnel. Because the Ranger system has been designed to be multi-mission and extensible, user provided physics-based simulations, telemetry processing and data analytics software can be integrated into the system to provide value-added capability to assess, predict and report vehicle health, status and the safety during testing and operations, and in the context of the mission flight rules and the surrounding environment. In addition, the Ranger team emphasizes system accuracy during development to ensure that computed and displayed mission data is accurate and correct and verified through extensive regression and integrated system testing and with input and data provided by flight and robotic mission domain experts.

Future work will focus on further reducing application development time as well as providing support for tight integration with the JPL Mission Operations Division software suite. Additional work will continue regarding user interface design, including VR/AR for use in mission operations.

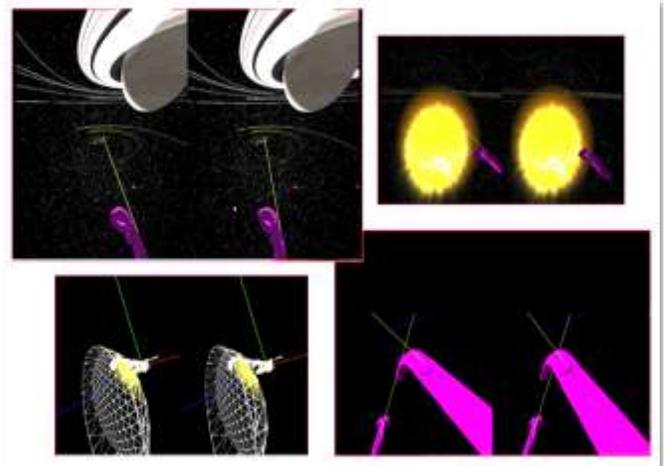


Figure 8. Stereo renderings from immersive VR development

## IX. ACKNOWLEDGMENTS

The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Copyright 2019. California Institute of Technology. Government sponsorship acknowledged.

We would like to thank our sponsors and collaborators: the NASA/JPL Mars 2020 mission, the JPL Mission Concept Systems Development Group, the JPL Education and Public Outreach organization and the JPL Mission Operations Division.

## REFERENCES

- [1] WebGL, [https://developer.mozilla.org/enUS/docs/Web/API/WebGL\\_API](https://developer.mozilla.org/enUS/docs/Web/API/WebGL_API)
- [2] JavaScript, <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
- [3] Mars Science Laboratory, <https://mars.nasa.gov/msl/>
- [4] Low Density Supersonic Decelerator, [https://www.nasa.gov/mission\\_pages/tm/ldsd/index.html](https://www.nasa.gov/mission_pages/tm/ldsd/index.html)
- [5] Marc Pomerantz, George Chen, Mark Ivanov, Christopher Lim, Tom Hyunh, “Applied Multi-Mission Telemetry Processing and Display for Operations, Integration, Training, Playback and Event Reconstruction”, Conference and Exhibition, Pasadena, California, 2015
- [6] <https://mars.nasa.gov/mars2020/>
- [7] OpenGL Graphics Library, [www.opengl.org](http://www.opengl.org)
- [8] OpenGL Shading Language, [https://www.khronos.org/opengl/wiki/Core\\_Language\\_\(GLSL\)](https://www.khronos.org/opengl/wiki/Core_Language_(GLSL))
- [9] The Navigation and Ancillary Information Facility, <https://naif.jpl.nasa.gov/naif/toolkit.html>
- [10] Emscripten toolchain, <https://kripken.github.io/emscripten-site/>
- [11] asm.js JavaScript subset, <http://asmjs.org>
- [12] WebAssembly, <https://webassembly.org>
- [13] JavaScript Object Notation, <https://www.json.org>
- [14] GL Transmission Format, <https://www.khronos.org/gltf/>
- [15] Cascading Style Sheets, <https://developer.mozilla.org/en-US/docs/Web/CSS>
- [16] M.I. Pomerantz, C. Lim, S. Myint, G. Woodward, J. Balaram, C. Kuo, “Multi-Mission Simulation and Visualization for Real-time Telemetry Display, Playback and EDL Event Reconstruction”, AIAA Space Conference, Pasadena, California, 2012
- [17] matlab, <https://www.mathworks.com/products/matlab.html>
- [18] <https://www.scipy.org>
- [19] Nasa Eye’s Eclipse 2017, <https://eyes.jpl.nasa.gov/eyes-on-eclipse-web-app.html>
- [20] Amazon Web Services, <https://aws.amazon.com>
- [21] Logarithmic Depth Buffer, <https://developer.nvidia.com/content/depth-precision-visualized>
- [22] HTML IFrame, <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe>
- [23] HTML Content Division Element, <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div>
- [24] WebXR, <https://developers.google.com/web/updates/2018/05/welcome-to-immersive>
- [25] HTC Vive, <https://www.vive.com/us/>
- [26] Paul Bourke, “Calculating Stereo Pairs”, 1999, <http://paulbourke.net/stereographics/stereorender/>

