NASA/TM-20230014837

GCAS Visualization Codebase Augmentation
Migration to Modern Standards and Feature Enhancements

Samuel J. Bloomingdale and Bryan W. Welch
Glenn Research Center, Cleveland, Ohio

February 2024

NASA/TM-20230014837

GCAS Visualization Codebase Augmentation
Migration to Modern Standards and Feature Enhancements

Samuel J. Bloomingdale and Bryan W. Welch
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

February 2024

Acknowledgments

The underlying software that makes up the GCAS was created by Dr. Bryan W. Welch. The initial versions of the visualization tool were developed by former interns Lucas Shalkhauser and Eric Henderson. The work to update the GCAS visualization tool to meet modern language standards and expand its capabilities, as described in this technical memorandum, was completed in collaboration with fellow interns Ana Granite, Mackenzie Martin, and Chloe Imbusch, each of whom filled a critical role in the research, development, and integration of these features.

GCAS Visualization Codebase Augmentation
Migration to Modern Standards and Feature Enhancements

Samuel J. Bloomingdale[*] and Bryan W. Welch
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

Summary

The Glenn Research Center Communication Analysis Suite (GCAS) includes many analysis tools that can be used to support a wide range of scenarios. It includes a visualization tool that implements the three.js graphics library to display a three-dimensional (3D) representation of its results. This software will enable researchers, engineers, and mission planners to interact intuitively with and understand the results of their analyses, which might not be apparent from raw data. With NASA's efforts to return humans to the Moon as a part of the Artemis missions, the GCAS has been used extensively for lunar terrain and landing system development analysis, which is vital to ensuring mission achievability and safety. This has created the need to add several significant features to the visualization tool, such as the ability to display the terrain of the lunar surface accurately and to provide information demonstrating how a given region might impact mission objectives.

Many of the changes made to the visualization tool can be separated into one of three general advancements: code restructuring to adhere to modern coding standards and practices, new user camera controls for first-person and third-person perspective views, and a terrain generation feature to enable rendering highly accurate terrains based on any celestial body's digital elevation model (DEM) in GeoTIFF format. These improvements notably elevate the visualization tool's functionality, accuracy, and user interaction while providing a robust foundation for future development.

Nomenclature

3D	three-dimensional
Blender	open-source 3D computer graphics software tool set
GCAS	Glenn Research Center Communication Analysis Suite
DEM	digital elevation model
ES5	Version 5 of the ECMAScript standard
ES6	Version 6 of the ECMAScript standard
ESLint®	static code analysis tool for identifying problematic patterns found in JavaScript code
GeoTIFF	file format for storing georeference information
NPM	Node Package Manager
three.js	open-source JavaScript library used for displaying 3D graphics
WebGL	JavaScript application programming interface for rendering interactive 2D and 3D graphics in a web browser
webpack®	module bundler used to create program files intended for production

[*]NASA Office of STEM Engagement (OSTEM) Spring 2023 intern, undergraduate at University of Nebraska.

Introduction

The tools in the Glenn Research Center Communications Analysis Suite (GCAS) enable the analysis of an extremely wide range of scenarios. To provide a clear perspective on the data produced within GCAS, it includes a visualization tool that implements the three.js graphics library to provide a three-dimensional (3D) representation of its results. This program allows researchers, engineers, and mission planners to interact intuitively with and understand the results of their analyses that might not be made apparent by raw data. With NASA's efforts to return humans to the Moon as a part of the upcoming Artemis missions, the GCAS has seen significant use for the analysis of the lunar terrain and landing system development, which are vital to ensuring mission achievability and safety. This has created the need to add several significant features to the visualization tool, such as the ability to display the terrain of the lunar surface accurately and information that demonstrates how a given region might impact mission objectives.

As the types of analysis that the GCAS can support continue to diversify, the range of scenarios that can be displayed effectively within its visualization software will need to grow accordingly. This will require the program's codebase to be conducive to new development, easily adaptable to modifications, and reliably maintained. This document discusses the primary capabilities added to the visualization tool, the challenges faced in their development, and the steps taken to aid in future development.

Code Refactoring to Meet Modern Language Standards

Reasoning

ECMAScript (ES) is a standardized programming language specification developed and maintained by Ecma International. JavaScript is the most widely utilized implementation of this standard. Although the JavaScript name is a trademark of Oracle Corporation, the ECMAScript and JavaScript names have become synonymous in the field of software development and are often used interchangeably. Few significant changes were made to the ES standard from the release of the first version in 1997 through the release of ES5 in 2009. In the early 2010s, the rapid evolution of web development led to a dramatic upsurge in JavaScript's popularity. This sudden growth brought about a significant overhaul of the ES standard in 2015. Although five additional versions have been released since then, none of them have included changes as drastic as those made with ES6. The magnitude of the changes introduced by ES6 and the impact it had on web development has effectively separated JavaScript programs into two eras: Pre-ES6 and Post-ES6.

The visualization tool contained within GCAS was initially created in 2019. Although the ES6 standard was released in 2015, many browser engines and existing JavaScript libraries had not yet migrated their code to support and/or make use of the new features. One of those was three.js, a cross-browser JavaScript library and application programming interface (API) used to create and display animated 3D computer graphics in a web browser using WebGL. At the time, three.js had not yet completed its own transition to ES6, which meant the documentation and examples referenced in initial development followed pre-ES6 standards (Ref. 1). Several additional features had been successfully added to the visualization tool following its creation, all adhering to the same standards that were used in its initial development. Most browser engines and JavaScript libraries, including three.js, have now fully adopted post-ES6 standards (Ref. 2), and many even require their use. These requirements meant that any attempts made to update dependencies or the introduction of any feature that required more than a moderate amount of refactoring consistently led to bugs being created or the program breaking elsewhere. As a result, the decision was made to rewrite and restructure most of the existing codebase to increase the readability and maintainability of the code, as well as ensure that the program would not become deprecated as pre-ES6 code libraries and features continue to lose support.

Development Tools

Three major development tools—Node Package Manager (NPM), webpack® module bundler, and ESLint® static analysis tool (both OpenJS Foundation)—were used in support of the code restructuring to adhere to modern coding standards and practices, the new user camera controls for first-person and third-person perspective views, and a terrain generation feature to enable rendering highly accurate terrains based on any celestial body's digital elevation model (DEM) in GeoTIFF format.

NPM is a package manager for the Node.js runtime environment that allows adding, removing, or updating external libraries or tools within a project in a simple and unified manner. The webpack® module bundler is used to create program files intended for production. The separation of the source code files from the distribution files allows development and testing of the source code to occur without threatening the integrity of the production builds. The bundling process only includes dependencies that are explicitly defined in a configuration file or directly called within the code, eliminating unused code to minimize file sizes and preventing any unneeded complexity from being added to the distributed program. ESLint® is a linting tool implemented to enforce the uniform use of post-ES6 standards and configured styling rules, greatly improving the readability of code and reducing the likelihood that bugs or errors will be introduced into the program. These tools, in combination with several features that were introduced with ES6, have allowed for radical simplification of the file structure.

Refactoring for Modularity

Figure 1 shows the contents of the directories necessary to run a distribution-ready build of the visualization tool before and after migration. The webpack® module bundler integrates all the necessary JavaScript files into one bundle. The 'assets' and 'css' directories, as well as the HTML file used as the root of the program, are updated every time a new build is created, eliminating the risk of missing or unreachable dependencies being called within the program.

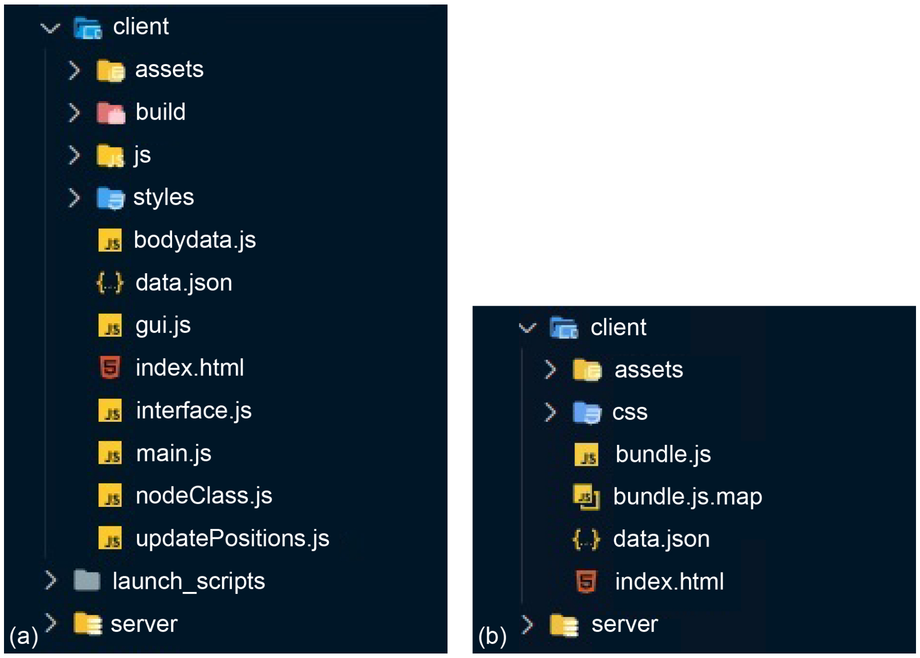
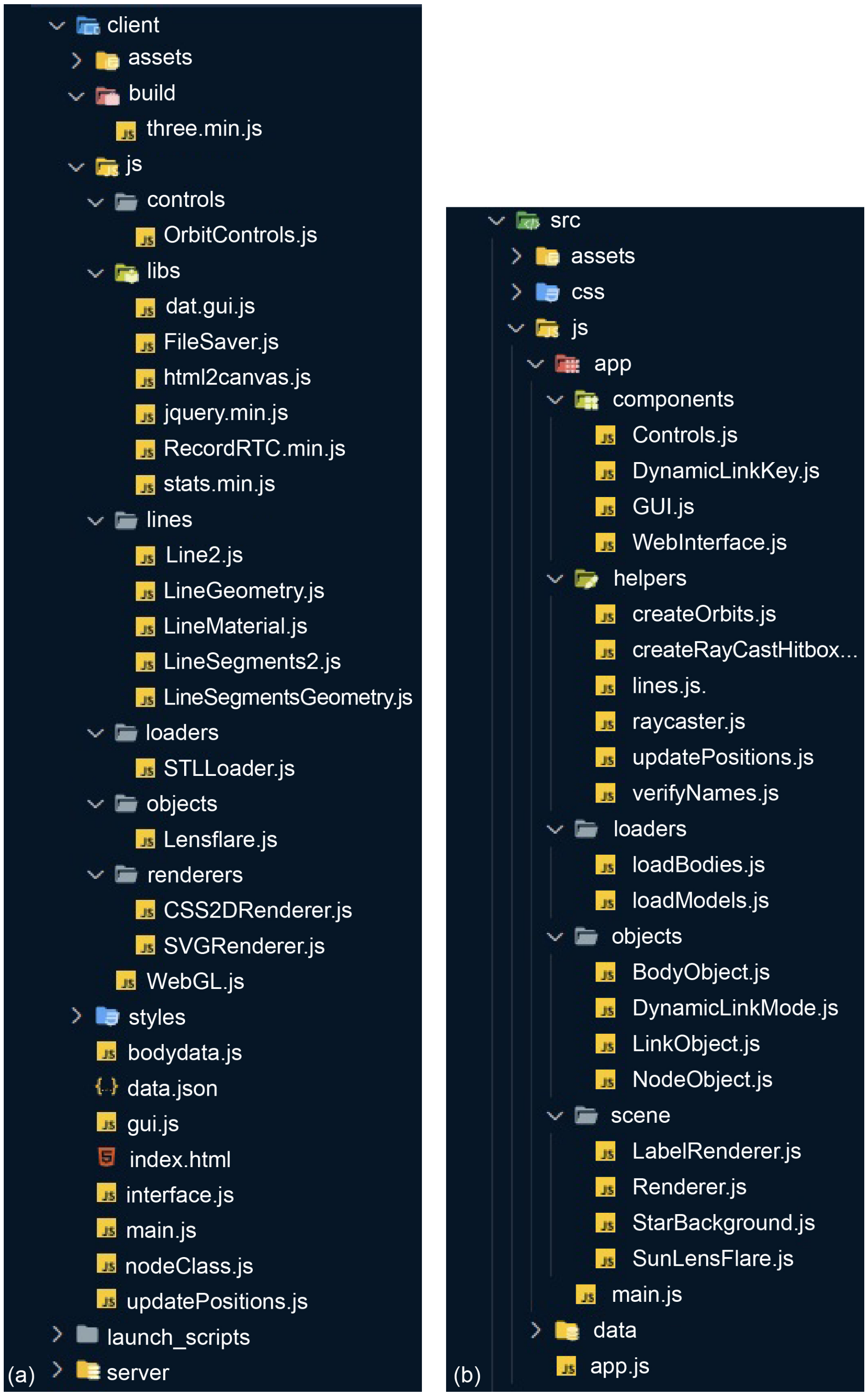


Figure 1.—Directories required to run a distribution-ready build of visualization tool. (a) Old distribution before migration to ES6. (b) New distribution after migration to ES6.



```
(a)
client
  > assets
  v build
      three.min.js
  v js
    v controls
        OrbitControls.js
    v libs
        dat.gui.js
        FileSaver.js
        html2canvas.js
        jquery.min.js
        RecordRTC.min.js
        stats.min.js
    v lines
        Line2.js
        LineGeometry.js
        LineMaterial.js
        LineSegments2.js
        LineSegmentsGeometry.js
    v loaders
        STLLoader.js
    v objects
        Lensflare.js
    v renderers
        CSS2DRenderer.js
        SVGRenderer.js
      WebGL.js
  > styles
    bodydata.js
    data.json
    gui.js
    index.html
    interface.js
    main.js
    nodeClass.js
    updatePositions.js
> launch_scripts
> server
```

```
(b)
v src
  > assets
  > css
  v js
    v app
      v components
          Controls.js
          DynamicLinkKey.js
          GUI.js
          WebInterface.js
      v helpers
          createOrbits.js
          createRayCastHitbox...
          lines.js.
          raycaster.js
          updatePositions.js
          verifyNames.js
      v loaders
          loadBodies.js
          loadModels.js
      v objects
          BodyObject.js
          DynamicLinkMode.js
          LinkObject.js
          NodeObject.js
      v scene
          LabelRenderer.js
          Renderer.js
          StarBackground.js
          SunLensFlare.js
        main.js
    > data
    app.js
```

Figure 2.—Expanded source directory trees. (a) Old source code. (b) New source code.

A similar reduction can be seen in the expanded source directory trees in Figure 2. In addition to simplifying dependency management, NPM manages the storage of any external libraries within its own ‘Node Modules’ directory. The webpack® module bundler will resolve the path of, and include in the bundle, any dependencies contained within this directory that are referenced within the program. This allows for the JavaScript files specific to the visualization to be separated and organized into smaller sections, according to the specific purpose of their contents.

This ability to separate these files lead to another significant benefit of the migration, by allowing for the use of the ‘Import’ and ‘Export’ module statements, which were introduced with ES6 (Ref. 3). These statements allow for related functions, objects, and variables to be accessed between files without requiring them to be globally accessible and/or mutable. For example, each file within the ‘data’ directory of the source code encapsulates all the code needed to read and store the simulation data and any objects that will be displayed. The entry point then imports these files, where the functions they contain are used to load and read the data into storage to be accessed in the visualization. With the data already loaded and handled, all code needed to run the simulation can be imported and called inside of ‘main.js’. Encapsulating relevant functions into these modules greatly improves the code’s readability and maintainability. In the old structure, ‘main.js’ contained 1,603 lines of code; in the new structure, it has been reduced to just 277. A developer can now simply follow along with the flow of the program and identify where some bit of code is executed; this greatly reduces the time required to debug, modify, or further develop the program’s features and capabilities.

Specialized Camera Configurations

GCAS is designed for analysis of a diverse range of scenarios. The ability to view all relevant aspects of a visualization precisely and intuitively is critical to the software’s ability to serve this purpose successfully. The most important aspect of this is ensuring that users have absolute control of a camera to adjust their view and how that view can adapt to best fit the needs of their analysis. In prior versions of the visualization package, the only controls available were traditional orbit controls, which allowed users to adjust the camera in three ways:

- Orbiting: Adjusting the azimuthal angle and polar angle of the camera with respect to the object in focus (i.e., rotating the camera about the object’s vertical and horizontal axes, respectively)
- Panning: Translating the camera’s position about its own vertical and/or horizontal axes
- Zoom: Increasing or decreasing the Euclidean distance between the camera and the object in focus

Although adjusting the distance and direction from the camera to the object in focus manually may be suitable, or even preferred, in many scenarios, feedback from GCAS users highlighted a need to implement finer camera control options.

Orbit Tracking

The effectiveness of specific camera options will vary greatly depending on the specific use case, with the most significant variable often being the type of object that is in focus. Figure 3 demonstrates one such scenario, in which the traditional orbit controls are less than ideal. In both Figure 3 and Figure 4, the camera is focused on CubeSat 3 as it orbits the Moon. The orbit controls are shown in Figure 3. The user has manipulated the camera’s position and distance relative to the satellite, where it will remain fixed, always looking in the direction of the satellite. The user is forced to maneuver the camera manually to prevent the Moon from obstructing the camera’s view of the satellite. To address this problem, an orbit tracking mode was developed, which allows a user to specify an object, separate from the object currently in focus, to be used as a reference point. When this mode is enabled, rather than waiting for a user to manipulate the camera’s position manually relative to the object in focus, the program will calculate a unit vector based on the position of the object in focus relative to the selected target. The calculated unit vector is then scaled by the camera’s current distance and added to the camera’s current position. Figure 4 demonstrates how the orbit tracking mode prevents the Moon from obstructing the camera's view of CubeSat 3. This same process can be used to position the camera using the node's velocity vector to place it behind the node, by setting the location used as a reference to a point in space further along the node's current orbit.

The orbit tracking mode can also be used to create a first-person perspective of the selected object. Whenever the camera is focused on a node, the node's 3D model will be hidden if the camera intersects its bounding box. When orbit tracking is enabled, the camera stays pointed in the direction of the selected reference object, providing the first-person perspective of the selected object from that node's perspective. This use case is only particularly useful to view objects in orbit around a body. If the same method were used to follow an object on the surface of a body, the camera would be stuck in a bird's-eye point of view, always looking down at the object, and the camera would frequently experience odd rotations as that object (or the body it is positioned on) moved through the world space, which is the WebGL terminology for the scene being rendered.

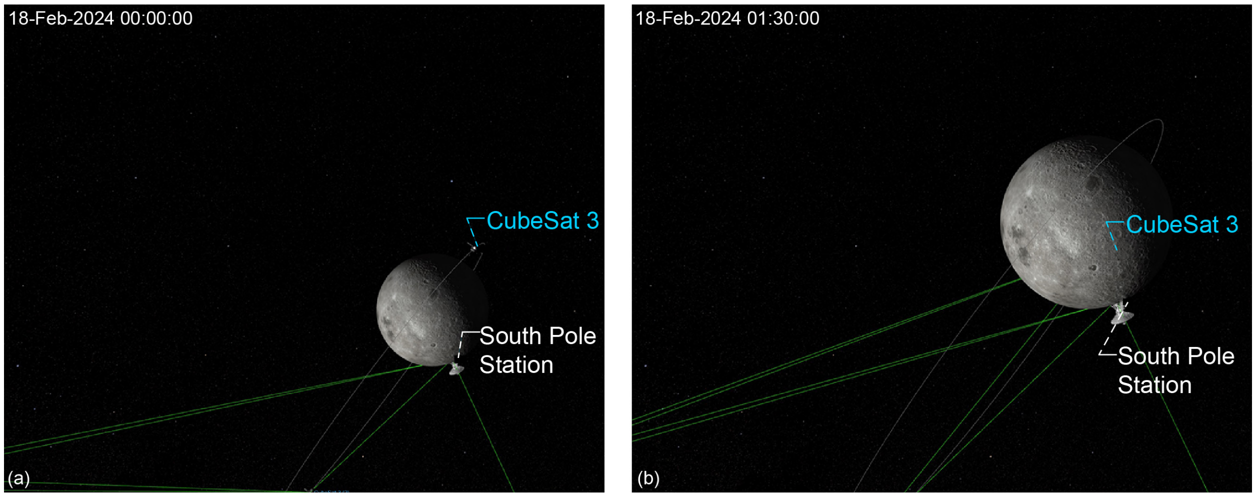


Figure 3.—User-defined offset direction using traditional orbit controls, with no manual adjustments made to camera between images. (a) Unobstructed view of CubeSat 3, taken at the first time step of the visualization. (b) Obstructed view of CubeSat 3, taken after 100 time steps.

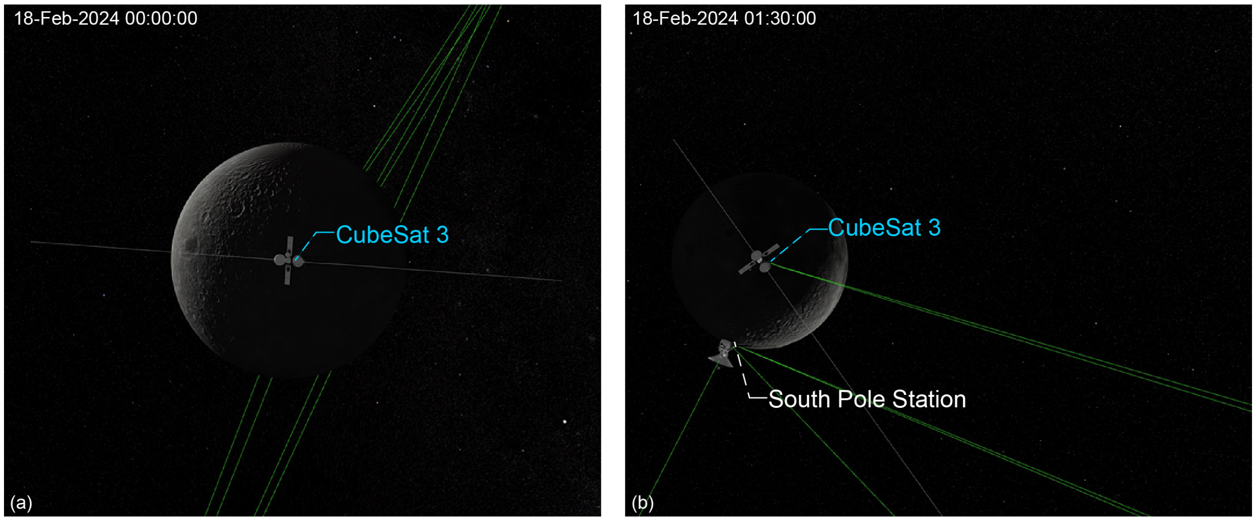


Figure 4.—Unit vector offset direction, with no manual adjustments made to camera between images. (a) Unobstructed view of CubeSat 3, taken at the first time step of the visualization. (b) Unobstructed view of CubeSat 3, taken after 100 time steps.

Surface Object Perspectives

Although both the original orbit controls and the orbit tracking options allow users to focus the camera on an object located on the surface of a body, neither can accurately represent that object's own perspective, as explained previously. An entirely separate camera system was developed to meet this need. This alternate camera works much like the character controllers used in video games. This camera presents the user with two primary methods that could prove extremely useful in communications analysis.

The first of these methods automatically adjusts the camera position within the animation to mimic the movement of an object as it travels across a surface. This method requires a surface object with position data to be present in the visualization. The camera can be configured always to look in the direction of travel, preventing users from manipulating its position or the direction it is looking. In first-person perspective, this effectively allows users to view the visualization from the perspective of that object or any equipment, such as a camera or communications array, that could be carried by the object. Users can also choose to take control of the direction the camera is pointing while its position remains updated automatically.

The second method provides users with absolute control of the position and orientation of the camera, eliminating the requirement for a surface object to be defined in the visualization data. This includes a first-person and third-person perspective on the surface. In the first-person perspective, the camera can be moved forward, backward, left, or right, using the W, S, A, and D keys, respectively, and the cursor can be locked to the window, allowing mouse inputs to control the camera's rotation. The third-person perspective is similar, except the movement keys will move the location of the chosen 3D model, and the camera's position will automatically be updated to maintain a set distance between itself and the model. The camera orientation remains controlled via mouse input, but any rotations will be made about the axes of the model, rather than the camera's own axes.

Terrain Generation

A new 3D model with displacement data applied was created using the Blender open-source 3D computer graphics software tool set to take the place of the current Moon model. This model more accurately represents the entire lunar surface, although memory constraints prevent loading the highest quality model created into the visualization; a lower resolution export of that model is used instead. To compensate for this lower quality terrain and provide additional terrain views, the visualization software can now process a DEM in GeoTIFF format to generate a 3D mesh that accurately represents the data it contains.

Reading a GeoTIFF File

The geotiff.js library is used to load a DEM file, interpret its metadata, and read the elevation values into an array buffer (Ref. 4). Once the entirety of a file's data has been read, it is passed to a function that removes any unnecessary information and returns an object containing the array elevation values and the properties needed to create the terrain accurately. If more than one DEM is provided to be displayed, the program will need to complete the entire reading, processing, and geometry creation processes for each file individually before it will move on to the next. The process implemented in this feature can be used to generate isolated terrain from nearly any standard DEM in GeoTIFF format, although any provided

GeoTIFF files must meet some requirements to be displayed correctly in relation to any other geospatial data used in a scene.

1. If the pixels in the file represent any area other than 1 m, the ‘fileDirectory’ attribute must contain a value for the ‘ModelPixelScale’ property with the correct pixel area.
2. If the terrain represented by the data is centered anywhere other than (0,0,0) in its corresponding body’s cartesian coordinates, the ‘fileDirectory’ attribute must contain a value for the ‘ModelTiePoints’ property containing the coordinates of the first pixel’s location.
3. If the displacement values are represented in any units other than 1 m, the ‘fileDirectory’ attribute must contain the proper scale factor within the ‘GDAL_METADATA’ property.
4. The ‘geoKeys’ attribute must contain a value for either the ‘GeogSemiMajorAxisKey’ or ‘GeogSemiMinorAxisKey’ properties storing the radius of the represented body in meters.

If any of these are not present in the metadata, but the correct values are known, they can be implemented without needing to modify the source file by passing them as arguments to the ‘createGeometry’ function.

Creating the three.js Geometry

Upon reading a DEM and storing all the requisite data, the object holding that data is passed to the function responsible for creating the three.js geometry. The arguments must include the object containing the terrain data, a resolution scale factor, and any properties listed previously that were absent from the file’s metadata.

The dimensions of the displacement array are multiplied by the pixel scale value to determine the dimensions of the geometry in meters, and the pixel scale is multiplied by the provided scale factor to determine the distance between vertices. For instance, if a 5-meters-per-pixel (mpp) DEM with a 1,000 by 1,000 pixel image were to be provided with a scale factor of 2, the resulting geometry would be a plane measuring 5,000 by 5,000 m, containing a vertex every 10 m.

Next, the newly created geometry will be translated along its local horizontal and lateral axes according to the distances contained by the ‘ModelTiePoints’ property. The function will then iterate through each vertex of the geometry. Each vertex is first displaced along the plane’s vertical axis by the value contained in the respective index of the displacement array, followed by a calculation to set the x, y, and z coordinates to their respective locations on a sphere with the specified radius.

Much like the models generated in Blender, this method can quickly become restricted by the memory constraints of JavaScript running in the browser when rendering terrain at higher resolutions. All source files currently used to create the lunar terrain displayed in the program store the displacement data in single-precision floating-point format, which requires 32 bits (4 bytes) for every value stored. This means that the positions array of the geometry alone will require nearly 3 times the amount of memory occupied by the source file. The three.js geometries store additional attributes for each vertex (Ref. 5), which further increases the total memory needed to store each terrain mesh. The benefit to rendering these heightmaps individually is that, for such small regions, a significantly higher resolution terrain can be used while staying within the memory limits. Although the need for scale factors could eventually be eliminated entirely with the implementation of geometry shading (Ref. 6), WebGL does not yet natively support them. Depending on the number of terrain geometries that need to be loaded into a scene, the ideal scale factor seems to be around 3 to 4 for DEM files of approximately 150 MB or smaller, and anywhere from 5 to 10 for files larger than 250 MB. Figure 5 shows how increasing the scale factor can drastically reduce the number of vertices that need to be stored, with the total memory required to display

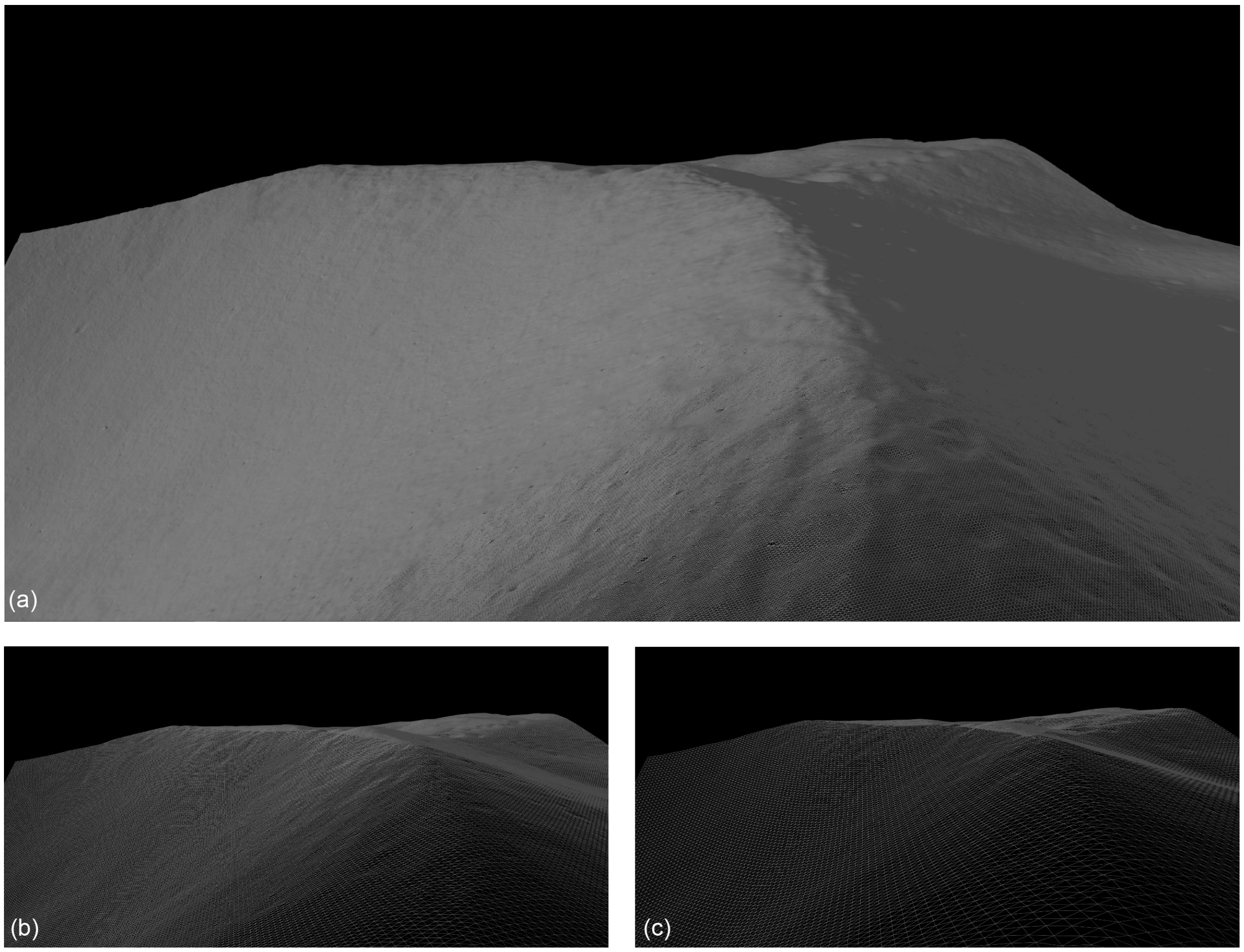


Figure 5.—Wireframe views of terrain depicting rim of Shackleton Crater, all derived from same 5-mpp source DEM. Each camera is positioned at an altitude of 2,500 m directly over lunar south pole, facing approximately 180° E. (a) Generated with scale factor of 2 and effective resolution of 10 mpp. (b) Generated with scale factor of 10 and effective resolution of 50 mpp. (c) Generated with scale factor of 20 and effective resolution of 100 mmp.

each mesh coming out to 190, 52.9, and 48.5 MB, respectively. As the scale factor increases, the more intricate details of the terrain can become obscured. Any scale factor required to generate terrain that stays within the program's memory limits will have very little effect on how the larger, more important features of a landscape are portrayed.

Concluding Remarks

Overall, the enhancements made to the Glenn Research Center Communication Analysis Suite GCAS software have significantly improved the functionality and accuracy of its visualization tool, particularly with regard to the lunar surface, and have successfully implemented the features and modifications its users requested. The restructuring and adherence to modern coding standards provide a solid base for the future development and maintenance of the software. The new user camera controls provide users with a more intuitive experience, while also providing finer control of the visualization. The terrain generation feature, which now supports data from any celestial body that contains the metadata described previously, can create highly accurate terrain views. These advancements are expected to be critical in achieving user-oriented analysis and visualization tools to further the capabilities of the GCAS and contribute to future applications in space research and exploration.

References

1. Shalkhauser, Lucas D.; Henderson, Eric; and Welch, Bryan W.: On Development of Three-Dimensional Visualization Capabilities in Glenn Research Center Communication Analysis Suite. NASA/TM-20205000040, 2020. https://ntrs.nasa.gov
2. three.js: Docs. https://threejs.org/docs/ Accessed May 9, 2023.
3. JavaScript: Modules. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules Accessed May 9, 2023.
4. geotiff.js: Home. https://geotiffjs.github.io/geotiff.js/ Accessed May 09, 2023.
5. National Instruments Corp.: Using Meshes to Draw 3D Objects. 2023. https://www.ni.com/docs/en-US/bundle/labview/page/lvconcepts/3d_meshes.html#:~:text=A%20mesh%20is%20a%20collection,delineate%20or%20connect%20the%20vertices Accessed May 9, 2023.
6. Geiss, Ryan: Chapter 1. Generating Complex Procedural Terrains Using the GPU. NVIDIA Corporation, 2023. https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu/ Accessed May 9, 2023.