NASA/TM-20230014788

Development of Level of Detail System and First-Person Camera for the GCAS Visualization Suite

Mackenzie Martin and Bryan W. Welch
Glenn Research Center, Cleveland, Ohio

December 2023

NASA/TM-20230014788

Development of Level of Detail System and First-Person Camera for the GCAS Visualization Suite

Mackenzie Martin and Bryan W. Welch
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

December 2023

Acknowledgments

Mackenzie M. Martin would like to thank Bryan W. Welch for his contributions and support. She would also like to thank her teammates—Chloe Imbusch, Sam Bloomingdale, and Ana Granite—for their support during the development of new functionality for the Glenn Research Center Communication Analysis Suite.

Development of Level of Detail System and First-Person Camera for the GCAS Visualization Suite

Mackenzie Martin[*] and Bryan W. Welch
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

Summary

The use of data-driven simulations has become standard practice as part of planning for future space missions. These simulations allow visualizing the data interactively to show what the data represents, as well as the importance of the data in the context of the mission. Using this visualized data can enhance users' understanding of it and accelerate analysis efforts related to missions planned around it. Three-dimensional (3D) visualization software was developed to allow creating 3D representations of various communication systems, as well as the physical terrain of the Moon, for upcoming missions.

The goal of this software development effort was to create interactive visualization capabilities in the Glenn Research Center Communication Analysis Suite (GCAS) using data exported from MATLAB® (MathWorks, Inc.) scripts. This software had the functionality to visualize the line of sight and dynamic link margins of the communication satellites orbiting the Earth and the Moon. One important addition to this was the visualization of the terrain data located within the GeoTIFF files, which were produced in an effort to understand the Moon's terrain. Proper displacement values of this data have to be visualized to showcase where craters are located and how the shadow casting works with said craters at different points of the day, as well as analysis of possible landing sites for future lunar expeditions. The graphics library coded in JavaScript, three.js, had been previously selected for developing this visualization software. The software was revised to conform to modern standards, then further developed to convert the MATLAB® data into JavaScript 3D objects and Blender GL Transmission Format Binary file (GLB) objects, which were to be imported into the scene. In the process, a variety of other testing projects were created to be combined with this project at a later point; these included the first-person camera movements around spherical objects to portray human movement around the Moon, GeoTIFF loading methods, data transfer methods for incorporating the elevation data into the scene, and level of detail (LOD) capabilities to decrease memory usage and rendering time.

Nomenclature

3D	three-dimensional
CORS	Cross-Origin Resource Sharing
GCAS	Glenn Research Center Communication Analysis Suite
GeoTIFF	format for storing georeference information
GLB	GL Transmission Format Binary file
LOD	level of detail
NPM	Node Package Manager
three.js	open-source JavaScript library used for displaying 3D graphics

[*]NASA Office of STEM Engagement Spring 2023 Intern, Youngstown State University undergraduate.

Introduction

Although the idea of traveling to the Moon and back is no longer new, a successful mission to another planetary body still demands a lot of planning and a lot of data, which must be accurate to ensure the safety of the astronauts involved. Modern technology simplifies collecting data and running simulations to help prepare these astronauts for their missions through the use of animations and video. Although the data itself is crucial, it may not convey the information necessary for future progress efficiently. Without a clear understanding of the context and content, the data may seem overwhelming and complex, with each individual piece possessing its own significance. Ultimately, this can make the data too voluminous and convoluted to be useful. To address this, data-driven simulations have become the new approach to testing and portraying the capabilities of space missions. The ability to condense vast amounts of simulation data into an interactive graphical format can aid in understanding the information and allow for faster testing. To this end, the Glenn Research Center Communication Analysis Suite (GCAS) serves as a tool for simulating spacecraft orbits, enabling NASA to optimize all aspects of missions through visually intuitive representation. In addition, geographical terrain data will also be incorporated to simulate the real-time terrain displacement on the lunar surface, simplifying mission planning.

The GCAS three-dimensional (3D) visualization software program can transform raw data generated using scripts generated using the MATLAB® (MathWorks, Inc.) programming and numeric computing platform into accurate, modern, and interactive visualizations, so users can view the analysis results and generate simulation data simultaneously. The 3D visualization software was developed using open-source software elements, which has advantages such as low or no cost and accessibility to developers and users. The only cost associated with this project was the MATLAB® license because JavaScript and all associated open-source libraries are free. Furthermore, the software is highly accessible, operates on multiple computing platforms, and runs on Google Chrome, requiring no external programs to run other than MATLAB® software. This approach enables NASA to expand visualization functionality as new aspects of analysis become available, making it a useful tool for presenting data and helping users understand what it means.

Webpack Development

In order to test various cases without becoming entangled in the complexity of the original GCAS visualization code, separate testing projects were developed over the course of the development timeframe to demonstrate various capabilities and collect data to be implemented into the full project later. These projects are created in a webpack bundle and hosted using the webpack development server, which is necessary because of the multitude of imports used in the JavaScript files in both the testing projects and the primary GCAS software. Webpack (Ref. 1) is a static modular bundler used specifically for JavaScript applications, which takes the code written and makes it usable in a web browser. Webpack allows specifying how everything reads together and showcases specific inputs and outputs, depending on what is needed for the project. Imports such as orbit controls and the core three.js library were needed in most of the testing files and were imported directly into the primary index.js files by means of the webpack techniques already described. The direct importability and proper hosting server meant that testing files for character movement, GeoTIFF loading, and level of detail (LOD) rendering could be accomplished through the development period, each leading to the development of new capabilities for the GCAS software.

Movement on Spherical Terrain

One of the required additions to the GCAS 3D visualization software was the ability to explore the terrain data of the Moon in an organic way. This was needed to help visualize site possibilities in terms of sunlight exposure, current elevation, and angle of descent. The best way to achieve this was to implement a controllable character that could traverse the Moon's surface and help users orient themselves around a site. With the properly visualized GeoTIFF data portraying accurate elevation levels, as well as correct shadow transmissions to visualize sunlight placement throughout the cycle, missions could be planned more easily and quickly.

Individual testing files were developed early to figure out this movement; by the end of testing, a workable product was complete, but some aspects of it needed revision to support true organic controls. The end test code created a basic three.js scene containing basic lighting, as well as a sphere used as a stand-in for the Moon. Two cameras were then created to represent the main camera of the scene, as well as the first-person camera that is placed on the Moon that will be the primary target for most of the code. The camera is then tracked, and depending on which camera is active, the movement capabilities and the orbit controls are toggled on and off. A rover model acquired through the NASA 3D Resources page (Ref. 2) is then loaded into the scene and becomes the representation of the playable character. Figure 1 illustrates this top-down testing for first-person movement around a spherical object, with the rover being the primary focus point.

The goal is to have this model able to go around the sphere while maintaining its rotation, so that it appears to be driving around its surface. The camera should also follow the rover at a set distance, as well as never being forced to view the rover as if it were upside down. In other words, even when the rover is moving over the south pole, it must maintain a top-down appearance, no matter its placement around the Moon, as illustrated in Figure 2.

To achieve this, the theta and phi angles must be defined, as well as the radius of the sphere. Toggle buttons are added so that the software can determine which camera is active and what functions to use by setting the conditionals that are read throughout the code. Functions for updating model position and rotation are added to the scene so the model may move around the sphere while adjusting its rotation so it is always touching the sphere at its base, simulating the appearance of driving around the sphere. The update positions function takes the defined angles of phi and theta, as well as its defined radius, and increment its angular degrees, depending on which arrow keys are pressed. For example, if the left arrow key is pressed, the theta is incremented, moving the model in the correct direction. This can be visualized in rotational movement, much as is shown in Figure 3. Model rotation then constantly checks the model's current position in comparison to the target’s up vector and calculates the angle that needs to change, depending on the changes to these two factors. Although seemingly complicated, calculating the rotation of the model in this way is necessary because of limitations in the prebuilt methods in three.js.

A simple LookAt method appears to be a much less complicated fix; however, because of how LookAt functions work, traversing the poles of spheres becomes impossible as the calculated values approach zero, causing the model to decelerate movement constantly until it appears to be at a standstill. Also, even if the LookAt method is not on the model but on the camera instead, it still appears to have functionality issues in terms of inverting camera views at poles. Local topocentric cartesian vector calculations are the solution to these issues, so that angles do not converge to zero, resulting in a controllable model that can move around the sphere, as well as having a constant top-down view from the camera that follows it.

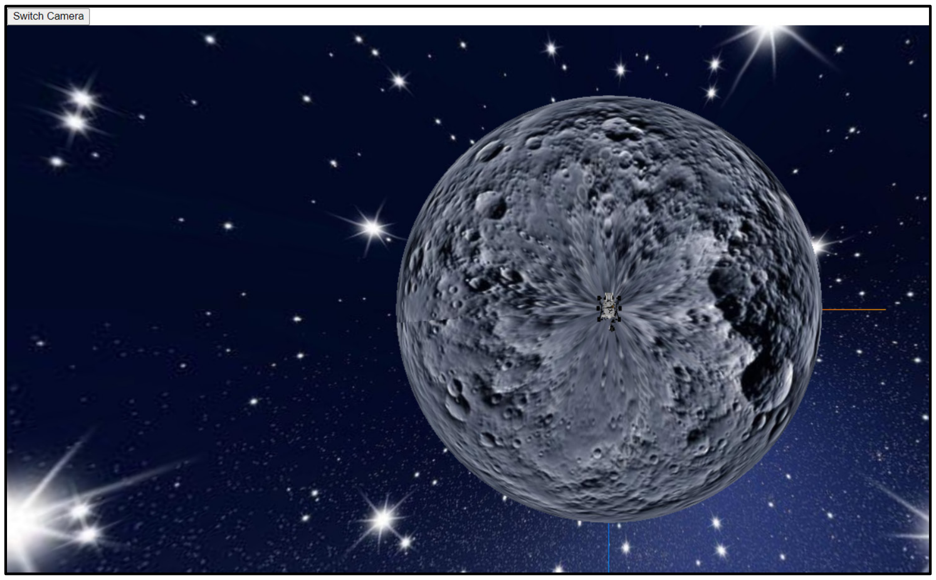


Figure 1.—Testing project view.

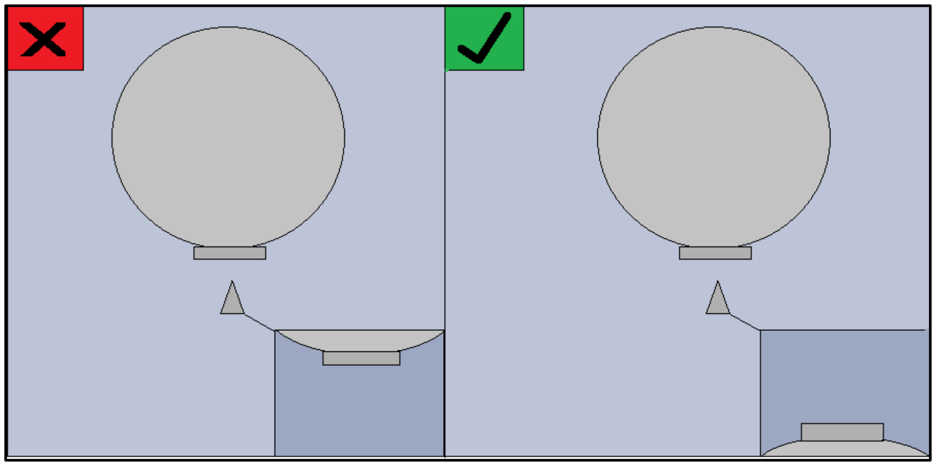


Figure 2.—Proper camera view example.

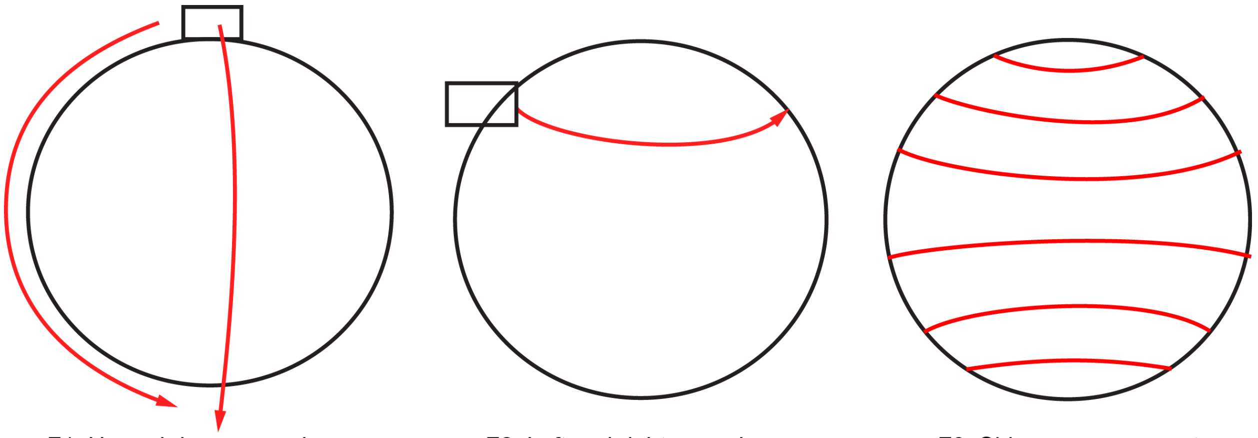


Figure 3.—Visualization of character movement.

Loading GeoTIFF Data

Webpack was used to build most of the testing projects described in the remainder of this report. It was used because, for many of these testing projects, the Cross-Origin Resource Sharing (CORS) error would appear, making testing impossible. CORS is a security feature implemented by web browsers to prevent websites from making requests to a different domain than the one that served the original content. To fix this error, the testing project can serve the file from a local web server running on a supported protocol like http or https; here, for simplicity and adaptability, webpack was used. However, by default, the webpack development server does not support loading .tif or .tiff files because it has no built-in loader for these file types. To enable the webpack development server to load GeoTIFF files, a loader that can handle these file types is needed. GeoTIFF-loader is a popular loader for GeoTIFF files that can be accessed through the Node Package Manager (NPM) install—or at least that was how loading the files was supposed to work.

Presumably, by running the command 'npm install GeoTIFF', developers can install the GeoTIFF JavaScript library as a dependency in a Node.js project via the NPM. The library's primary function is to facilitate the reading and writing of geospatial raster imagery stored in the GeoTIFF format. This file format is frequently used in Geographic Information Systems and remote sensing applications, which means the GeoTIFF library is a valuable tool for developers who work with geospatial data. However, the NPM install GeoTIFF loader was not working and the libraries that were supposed to be found in the "three" import were no longer there. Other NPM installs, such as the @loaders.gl/geotiff, were attempted to achieve similar functionality, but many errors were encountered with these approaches as well. Every new install was missing at least one functionality, which was detrimental to loading in the GeoTIFFs, and each new one instructed the user simply to download another to fix the issue. Eventually, a technique that involved reading in the raw data of the GeoTiff file and saving it to an array that would later be projected onto a plane was determined. However, it did take significant time to show results for something that should not have been so difficult.

Level of Detail

LOD (Ref. 3) is a technique used to improve rendering performance of 3D scenes by using simplified models of the object being rendered. These objects are then stored as children of the LOD object and rendered depending on the camera distance to the object. The purpose of this is to decrease memory usage and strain on the system when the user is far away from the primary body because there would be no need to see many details from that far away. This can allow faster rendering speeds and improved frame rates.

The proper LOD can be achieved in two primary ways. In the first way, the objects are cloned and subdivided, then vertices dependent on placement within the array are deleted. In the second way, multiple models can be created in an external platform, such as Blender, where the vertex count is changed, then three.js selects specific models to use, depending on camera placement.

Subdivision

Subdivision is a technique used to dynamically generate a new version of the object as it approaches the camera. It consists of breaking down the object into smaller, more refined parts to create a smoother appearance, increasing or decreasing the number of parts depending on placement in the scene. When the object is far away from the camera, a smaller number of vertices and faces are generated, producing a less detailed and less accurate version of the object. However, the closer the camera moves toward the object, the more faces it will develop until it is back to its original highest resolution model. This allows for

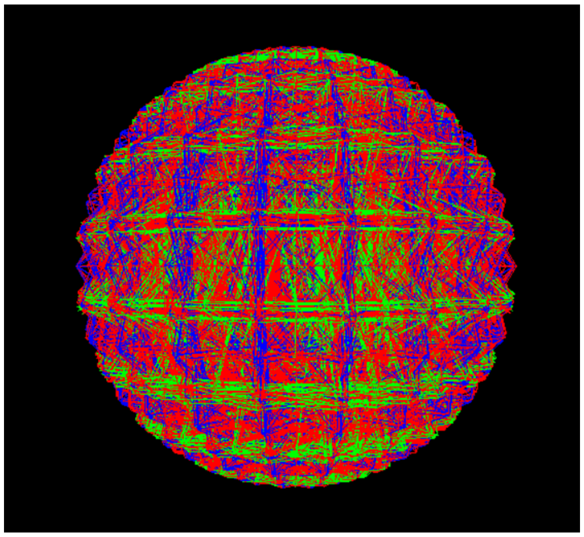


Figure 4.—Subdivided sphere.

better allocation of the objects so only what is needed is shown, leading to better performance in complex scenes. Figure 4 demonstrates this process using a basic sphere that was then subdivided and added to the scene. Its vertices' colors were then changed to allow for easier visualization of this division.

However, this process proved to come with a multitude of issues. The original way that this would work was to load the desired GL Transmission Format Binary file (GLB) model into the scene, then traverse through its textures. Next, based on the number of textures and their detailed amount, a different LOD was created based around them. However, with the models that are currently being made, this route was not possible. The models that were being tested only had an average of two textures each and did not support a built-in geometry, which meant that this system of decreasing the details was attempting to read something that did not exist. Theoretically, it would be possible to fix this step with Blender; however, the primary point of doing this exclusively in three.js was to avoid adding extra steps in an external software. To combat this, clones created via the SimplifyModifier were saved to be used as the LOD object children to traverse through.

The THREE.SimplifyModifier class is used to simplify the model's geometry by removing vertices based around a given tolerance. To do this, the model must have a distinguishable geometry, which is used to create other models based around it. However, there is a major issue when doing it this way, specifically with the models being used. Because of the high initial LOD, even the smallest sites can have more than 2 million vertices to go through, which makes reading all of the values and modifying them virtually impossible. To create the simplified geometry, the algorithm must first traverse each of the predefined points of the original site, which is difficult to achieve with so many vertices. By the time it was finished, it was too memory-intensive a process and was infeasible for full use in the GCAS.

Because there were so many issues with creating the geometries and reduced vertex models in three.js, the next testing project was done with multiple Blender models, each modified separately. The first type of testing relied on the three.js DRACOLoader (Ref. 4) to compress the models into smaller-file-sized variants, then load them into the scene based on the compressed version. In this way, the amount of compression could still be changed via the three.js code without requiring a heavy dependency on Blender for multiple compression rates. However, once again, this method had some implementation issues, so the results ended up being of multiple different models exported from Blender with the decimate modifier added on to them.

DRACOLoader

The DRACOLoader is a built-in loader for three.js that allows compressed files to become usable in a scene. It can make models significantly smaller at the cost of additional decoding time on the client side. These results can be achieved by adding in the proper imports for a DRACOLoader, then creating the loader in the scene. This way, any model already in a properly compressed format on export can be loaded in and viewed. The concept behind this was that it could become possible to read through the array of vertices with a simplified version of the model and use the original simplified modifier to create the LOD or it could simply be used to save space when creating the multiple GLB files to create their own LOD array. However, this approach was quickly found to be unusable, even at a lower compression rate, simply because of the appearance of the model, as shown in Figure 5.

The model appeared extremely blocky regardless of its compression rate in Blender, as well as its individual compression rate located in the code. This is theoretically because of the number of faces that it needs to simplify, making it almost impossible to do it this way while preserving any of the original model. The DRACOLoader-produced model still had more than 2 million vertices. Although it did save on file size, it did not appear to offer any major benefit in terms of memory usage or rendering capabilities, so it was also ultimately scrapped in the LOD testing project. Finally, the method that was used successfully was the multiple GLB models exported from Blender.

Blender Compression Rates

To achieve the desired vertex effect in Blender, the simplest way to decrease vertex count while still having a semblance of the original model was to use the decimate modifier and add it to the export of the model. The Decimate modifier (Ref. 5) allows the user to reduce the vertex and face counts of a mesh with minimal shape changes. It is primarily used to reduce polygon count to increase performance and to remove unnecessary vertices and edges. For example, from an original face count of 1,305,728 for the lunar south pole Digital Elevation Model (DEM) dataset, the user can decrease that by various percentages, then export each one to create a representation of multiple different models. Figure 6 showcases these exported variants of the models from Blender's perspective using the decimate modifier, ranging from 50 percent of the model being decimated to undecimated (100 percent) in 10 percent increments.

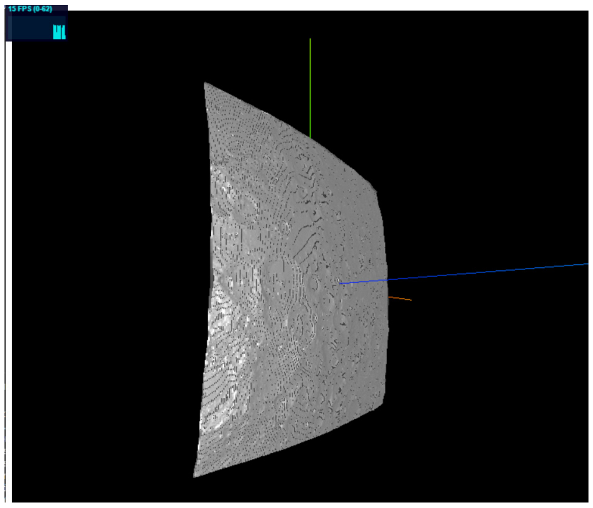


Figure 5.—DRACO loaded model.

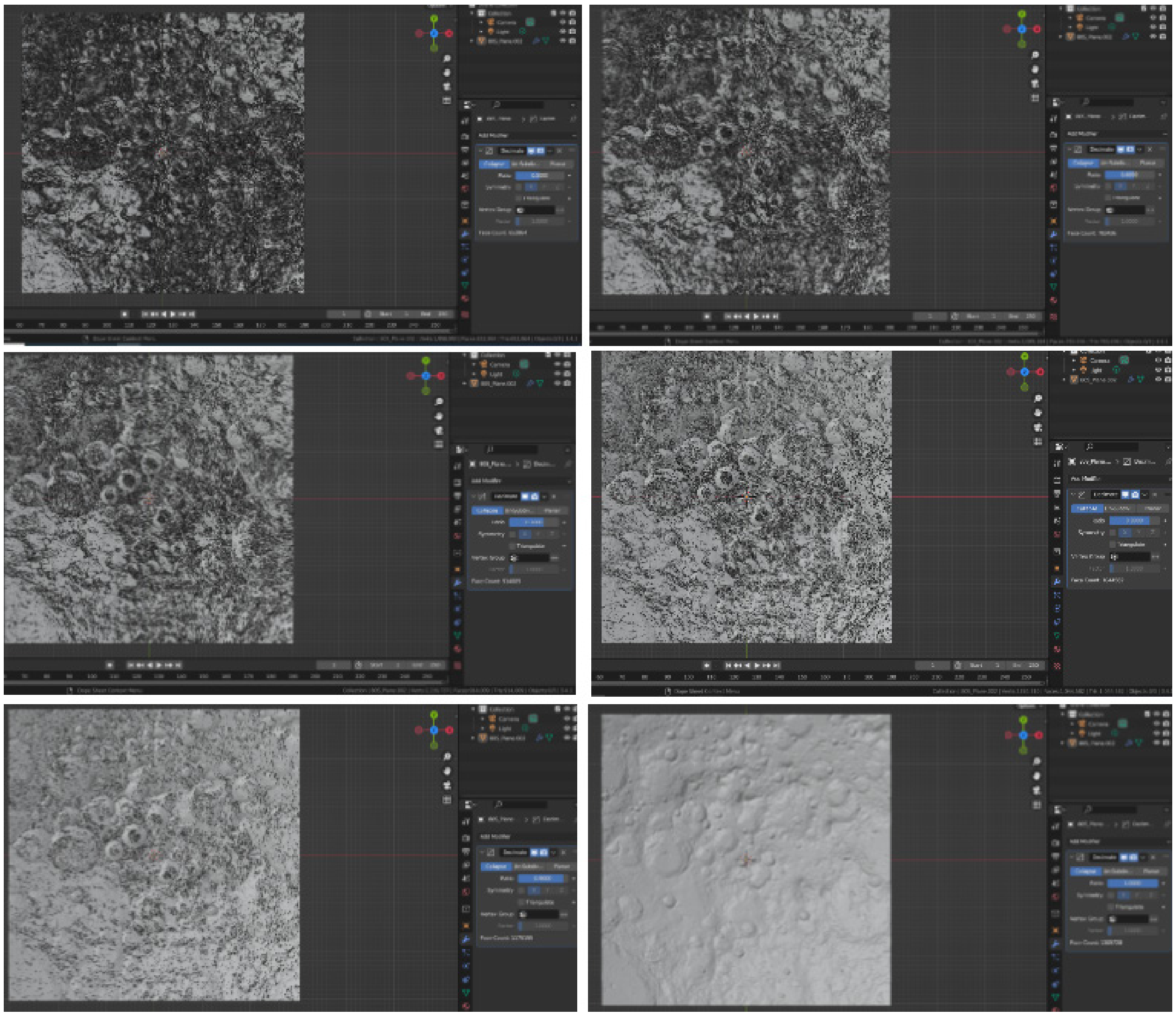


Figure 6.—Exported models across various decimation levels from 50 to 100 percent.

For these exported models, an array of file paths is created for them, then traversed through, adding each of them to the LOD object. An array of distances is also created, indicating when each LOD child should be presented based on how far away the camera is from the model. Finally, the distance is constantly checked in the animate function and changes based on the information are recorded. The objects on screen then change based on the terms defined in the code and a working LOD model is presented in the scene, as shown by Figure 7.

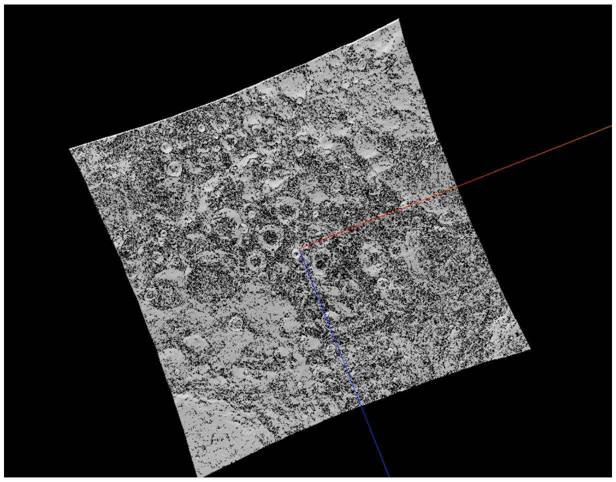
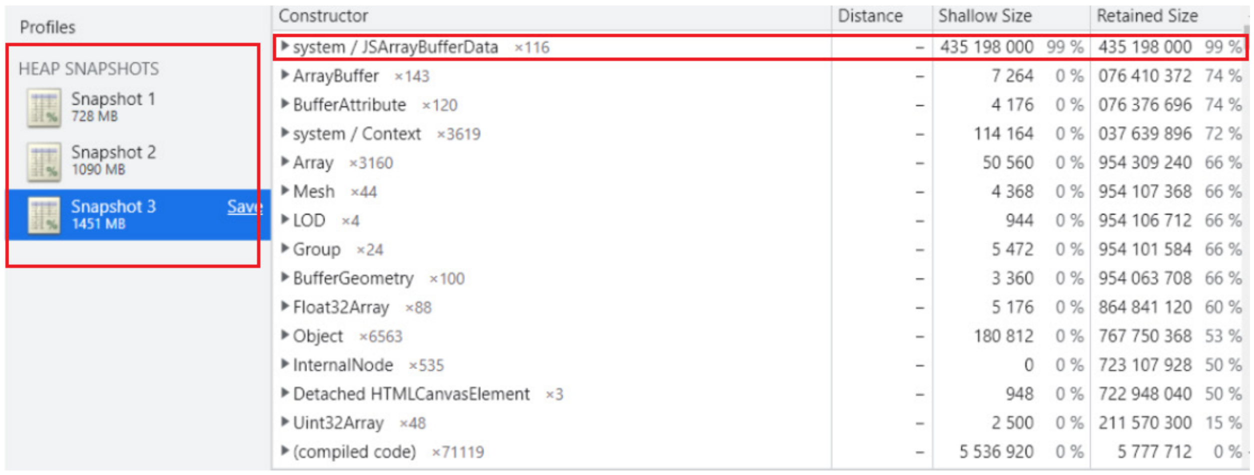


Figure 7.—Working LOD model.



Profiles

HEAP SNAPSHOTS

- Snapshot 1 728 MB
- Snapshot 2 1090 MB
- Snapshot 3 1451 MB Save

Constructor	Distance	Shallow Size		Retained Size	
system / JSArrayBufferData ×116	–	435 198 000	99 %	435 198 000	99 %
ArrayBuffer ×143	–	7 264	0 %	076 410 372	74 %
BufferAttribute ×120	–	4 176	0 %	076 376 696	74 %
system / Context ×3619	–	114 164	0 %	037 639 896	72 %
Array ×3160	–	50 560	0 %	954 309 240	66 %
Mesh ×44	–	4 368	0 %	954 107 368	66 %
LOD ×4	–	944	0 %	954 106 712	66 %
Group ×24	–	5 472	0 %	954 101 584	66 %
BufferGeometry ×100	–	3 360	0 %	954 063 708	66 %
Float32Array ×88	–	5 176	0 %	864 841 120	60 %
Object ×6563	–	180 812	0 %	767 750 368	53 %
InternalNode ×535	–	0	0 %	723 107 928	50 %
Detached HTMLCanvasElement ×3	–	948	0 %	722 948 040	50 %
Uint32Array ×48	–	2 500	0 %	211 570 300	15 %
(compiled code) ×71119	–	5 536 920	0 %	5 777 712	0 %

Figure 8.—Memory leak.

Memory Issues

Even after the LOD was completed, a memory leak was detected in the code. After each time the page was reloaded, certain aspects of the scene were not disposed of correctly, which led to eating up memory continuously. This is because the garbage collector (Ref. 6) in three.js is not instantaneous with all kinds of objects; depending on the occasion, certain objects must be disposed of separately from the automatic garbage collection. Figure 8 shows the representation of these objects not being collected properly, showcasing the continuous growth in memory consumption after each new load.

In an attempt to fix this leak, the code will check which LOD child is active in every frame. Depending on the answer, it will dispose of the other objects' geometry and material so that it does not constantly keep this data after it is finished with it. Presumably, this will fix most of the memory issues, but how successful this approach will be in the overall scheme of the project is yet to be determined.

Concluding Remarks

The methods and implementations of the testing projects described in this report are all being created to enhance the capabilities of the GCAS visualization suite. Most of these findings were developed with individual test files, which will likely be incorporated into the visualization suite later. The goal is to include the various test capabilities in order to give the user the full capabilities needed to analyze future missions. Further developments needed to increase usage of the software include a proper mission planning stage that includes both the camera functionality and proper data collecting tools to showcase exactly what is needed, depending on the scene. The current software version provides a platform to build upon and an environment to offer a representation of the data that has been collected over the years.

References

1. AG Grid: Webpack. https://webpack.js.org/guides/getting-started/ Accessed Mar. 30, 2023.
2. National Aeronautics and Space Administration: 3D Models—Search Results for “rover.” https://nasa3d.arc.nasa.gov/search/rover/model Accessed Mar. 2, 2023.
3. three.js: LOD. https://threejs.org/docs/index.html?q=lod#api/en/objects/LOD Accessed Mar. 30, 2023.
4. three.js: DRACOLoader. https://threejs.org/docs/index.html?q=draco#examples/en/loaders/DRACOLoader Accessed Apr. 13, 2023.
5. Gitea: Decimate modifier. https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/decimate.html Accessed Apr. 20, 2023.
6. three.js: Dispose things correctly in three.js. https://discourse.threejs.org/t/dispose-things-correctly-in-three-js/6534 Accessed May 3, 2023.