NASA/TM-20230014794

Dynamically Rendering Rough Terrain With Minimal Memory Overhead

Joseph W. Symons and Bryan W. Welch
Glenn Research Center, Cleveland, Ohio

December 2023

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Technical Report Server—Registered (NTRS Reg) and NASA Technical Report Server—Public (NTRS) thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers, but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., "quick-release" reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Fax your question to the NASA STI Information Desk at 757-864-6500

- Telephone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Program
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM-20230014794

Dynamically Rendering Rough Terrain With Minimal Memory Overhead

Joseph W. Symons and Bryan W. Welch
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center
Cleveland, Ohio 44135

December 2023

Trade names and trademarks are used in this report for identification only. Their usage does not constitute an official endorsement, either expressed or implied, by the National Aeronautics and Space Administration.

Level of Review: This material has been technically reviewed by technical management.

This report is available in electronic form at https://www.sti.nasa.gov/ and https://ntrs.nasa.gov/

Dynamically Rendering Rough Terrain With Minimal Memory Overhead

Joseph W. Symons* and Bryan W. Welch
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

Abstract

Rendering highly detailed terrain is a process with the potential to consume a great deal of a computer's random access memory (RAM). In a browser-based application, this resource is limited even further, leading to the necessity to use alternative methods of rendering the large amount of data needed for high detail. This report describes one such method that places the onus of rendering on the speed of the graphics processing unit (GPU) rather than on the computer's memory. By removing attribute buffers, which contribute greatly to memory costs, from the rendering pipeline and generating the requisite attributes on the fly using a heightmap texture instead, it is estimated that memory usage can be cut down to one-sixth that of the previous method.

Introduction

When rendering rough terrain in the NASA Glenn Research Center Communications Analysis Suite (GCAS) visualization software (Ref. 1), several limitations disallow the use of a normal rendering process. The amount of data is large, with a dataset spanning 90° S to 80° S latitude of the Moon at a resolution of 80 m/pixel, which requires about 231 MB of memory when stored in the GeoTIFF format. When converted into a renderable geometry, complete with positions and normal vectors for each vertex, the memory cost jumps to approximately 1.6 GB. Although modern computers can easily handle this memory cost, the visualization software runs in a web browser, which imposes harsher memory restrictions. In practice, the full-resolution terrain was never renderable using the old method. This limitation was the main driving force in the development of a new technique for rendering terrain.

The technique developed to overcome the memory limitations of the browser circumvents the need to store positions and normal vectors for each vertex; rather, it calculates these at runtime on the graphics processing unit (GPU). This memory savings means the cost is reduced to just the size of the GeoTIFF file. In addition to the reduced memory cost, the new technique provides for a dynamic level of detail, which in turn allows for higher framerates when using the software.

Procedure

The terrain rendering technique described in this report has multiple optional features, including a dynamic level of detail that goes beyond the basic method of generating the terrain. Each of these add-on features is discussed in its own subsection.

*NASA Office of STEM Engagement Intern, Bowling Green State University undergraduate.

Basic Terrain Generation

This method foregoes the use of attribute buffers to hold information about the vertices that make up the geometry of the terrain. Doing this frees up all the memory that would otherwise be used to store vertex positions and vertex normal vectors. This also means that, instead of receiving these values from attribute buffers and assigning them to the corresponding vertex, the vertex shader must calculate all these values at runtime. To do this, some amount of data must still be sent to the vertex shader without using attribute buffers.

By passing a GeoTIFF file as a texture to the shader program, the vertex shader can access all the elevation values of the terrain, regardless of which vertex is being processed. Additionally, the vertex shader must be able to distinguish between all vertices; otherwise, exactly the same code will be applied to all the vertices and they would collapse to a single point. This is possible thanks to the *gl_VertexID* input variable, a feature of WebGL2, which is a JavaScript application programming interface (API) for rendering interactive two-dimensional (2D) and three-dimensional (3D) graphics within any compatible web browser without the use of plug-ins. This built-in input variable for the vertex shader represents the index currently being processed (Ref. 2), which essentially means it will be unique for each vertex. If not for this ID, it would be impossible to treat each vertex differently without using attribute buffers.

Building Blocks

The first step is to make a basic 2D structure from the vertices. By default, WebGL2 builds triangle primitives from sets of three consecutive vertices. By leveraging this behavior, a more useful quad structure can be built with two triangles, thereby using six vertices. To accomplish this, the shader operates on *gl_VertexID mod 6*, forcing every set of six consecutive vertices into a rectangle, as shown in Figure 1. The placement of vertices in the quad is somewhat arbitrary, but it does affect which side of the quad the renderer draws because of something called the winding order (Ref. 3).

Now, with a set of quads rather than vertices, each quad can be uniquely accessed with *floor(gl_VertexID / 6)*. The quads are then arranged into a grid, with the number of rows and columns being passed to the shader as uniforms. From what was once a set of dimensionless points, there is now a flat plane of arbitrary size. With some slight modifications to calculate quad size based on plane size and resolution and some extra uniforms passed in, the plane can be generated with a dynamic amount of subdivisions while being kept at the same size, allowing for variable resolution, as shown in Figure 2.

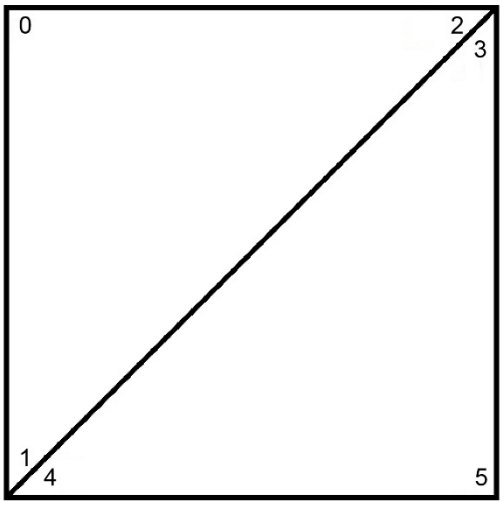

Figure 1.—Pixel vertices.

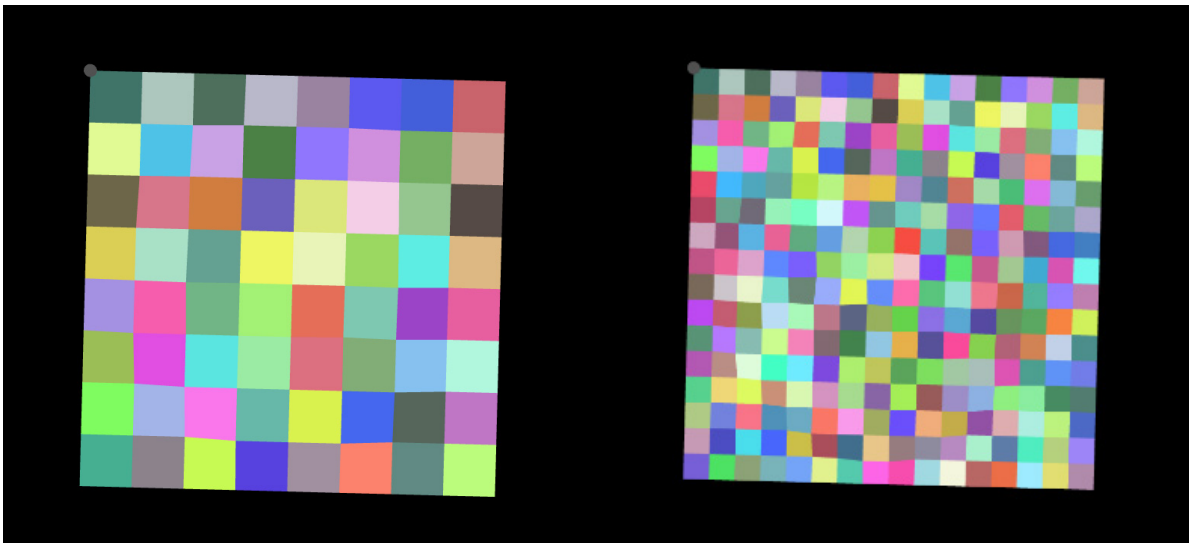

Figure 2.—Example of variable resolution.

Adding Elevation Data

Now that a variable-resolution plane can be generated with no use of attribute buffers, the elevation data can be added to the plane to displace the vertices accordingly. The GeoTIFF file is formatted as a texture, ensuring that the type and format match the type of data the file contains. This texture is then passed to the shader program as a uniform sampler2D. To look up the correct elevation value in the texture, the vertex coordinates need to be converted into U-axis and V-axis (UV) coordinates in the range of 0 to 1. The x- and y-coordinates are divided by the corresponding dimensions of the plane, making sure this calculation is independent of the current level of subdivisions. Now with the correct elevation value, the geometry can be brought properly into the third dimension, as shown in Figure 3, although it does not interact properly with light.

Light Interaction

The terrain now has elevation but no normal vectors, which tell the fragment shader which direction the geometry is facing at a given vertex. Without this, the standard lighting system cannot interact with the terrain. To calculate the normal vectors, the elevations of neighboring vertices are used to obtain an approximation. Note that neighboring vertices, not texture pixels, must be used because the varying resolution means that these are not necessarily the same. The shader accesses the neighboring elevation values by shifting the texture lookup by the size of a single quad converted into UV space. Four neighbors are used, and the difference vectors between opposing neighbors are then normalized. The cross product between these two vectors serves as the normal vector. With this addition, the terrain can react to lighting, as shown in Figure 4.

Spherical Mapping

The GeoTIFF data that the elevation values are read from is a stereographic projection of the appearance of the surface. To account for this, all the vertices need to be unmapped back into latitude and longitude, then converted again into an XYZ format for the shader to process. See the calculations in Figure 3.

```
vec3 map_to_sphere(vec3 stereo) {

      float radials = distance(vec2(0.0), stereo.xz);
      float lon = atan(stereo.x, stereo.z);
      float lat = (-PI / 2.0) + 2.0 * atan(0.5 * radials, radius);

      vec3 mapped = vec3(
          (radius + stereo.y) * cos(lat) * cos(lon),
          (radius + stereo.y) * sin(lat),
          (radius + stereo.y) * cos(lat) * sin(lon)
      );
      return mapped;
 }
```

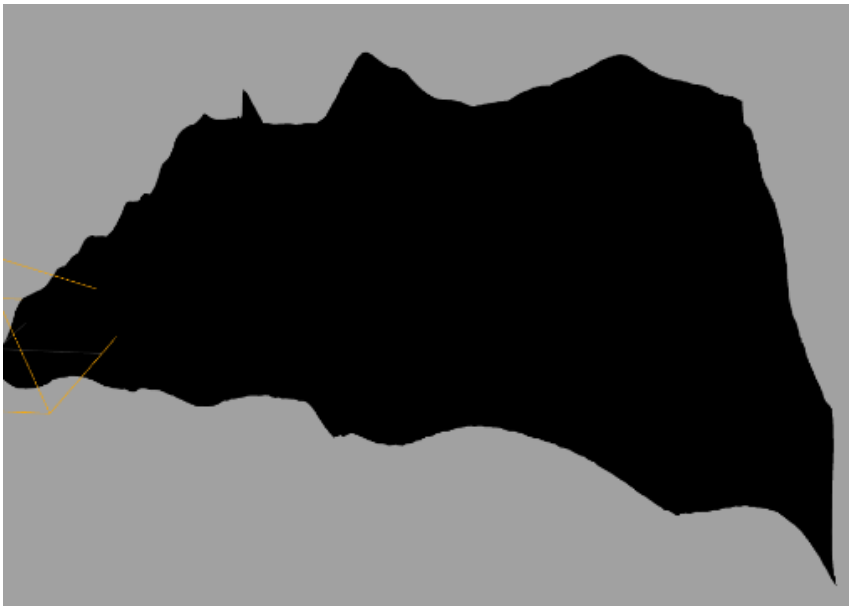

Figure 3.—Initial elevation data added to render.

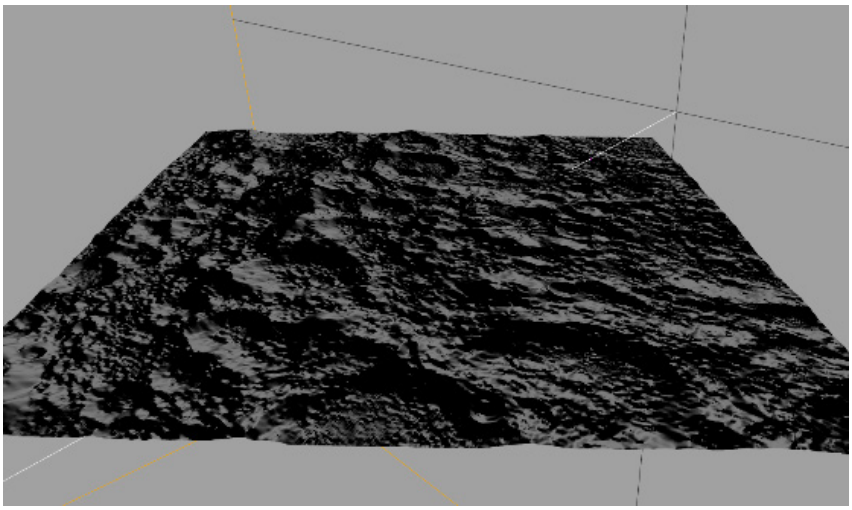

Figure 4.—Terrain with normal vectors to handle light interactions.

Before applying this mapping, the terrain should be scaled to the appropriate size according to the scaling factors in the GeoTIFF metadata, then centered so that the mapping applies correctly. Note that because the positions of all vertices change in this process, the normal vector calculations must include the mapping process for each of the neighboring vertices before using their positions. Additionally, the elevation data texture requires that the coordinates not be centered.

With this step, the terrain curves along the body's surface, as shown in Figure 5. This is the final step of the base terrain rendering method. Additional features used in the GCAS visualization software can improve functionality and performance.

Additional Terrain Features

Level of Detail

When viewing the terrain at full resolution, it is likely that hundreds of millions of vertices will be rendered simultaneously, which is enough to slow down even reasonably fast GPUs. To solve this, the level of detail is implemented, which shows more vertices only when the camera is close enough for the resulting image to benefit from that detail. Most of the work is already done here, as the resolution can be set dynamically from outside the terrain shader. The final step in implementing a very basic level of detail is to calculate the distance from the camera and use that as a parameter in a function that sets the resolution of the terrain. This solution enhances performance when the camera is far from the terrain but suffers the same issues when close up.

A better solution, and the one implemented in GCAS, is to split the terrain into a grid of tiles, as shown in Figure 6, each of which has its own geometry with its own resolution. The tile size and dimensions of the tile grid are factored into the vertex shader so that position, UV, and normal vector calculations can remain largely unaltered and so the same elevation texture can be sent to all tiles. Linear filtering should be used on the texture to avoid artifacts near the edges of tiles. The result is a dynamic level of detail that greatly improves the performance when viewing the terrain up close.

Auxiliary Terrains

Often, the data resolution of a large GeoTIFF will not be fine enough for extreme closeup viewing. In these cases, auxiliary terrain data with finer resolution is used to supplement the terrain. The base shader code still works with the auxiliary terrains, with one exception—tie points. The large terrain is centered at the pole and so needs no special placement, but the auxiliary terrains are not centered at the pole. Fortunately, the GeoTIFF metadata includes tie points, which indicate where the terrain should be placed in a stereographic view relative to the pole. By factoring this in before doing the spherical mapping, the auxiliary terrains can be added on in the correct location.

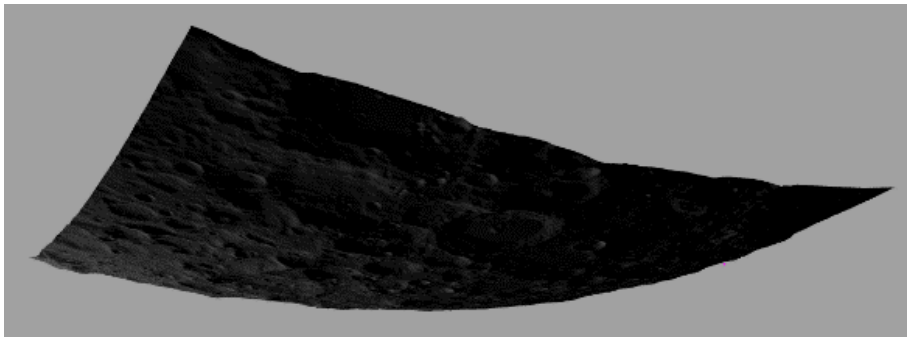

Figure 5.—Spherical corrections to lunar south pole region.

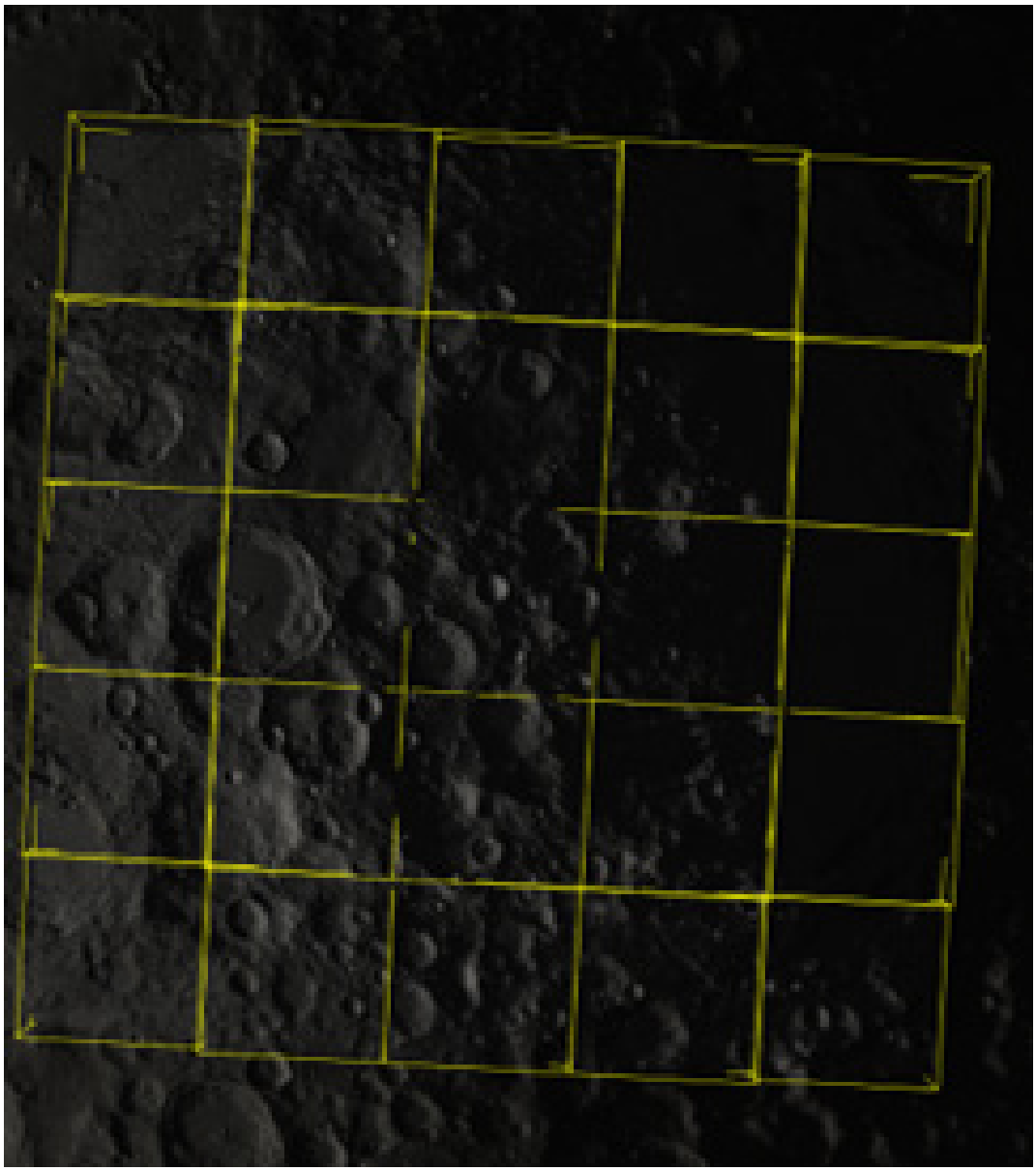

Figure 6.—Level of detail grids in lunar south pole terrain.

A problem with this solution arises from the fact that there is now a coarse-resolution terrain sharing the same space as a fine-resolution terrain. Visually, this means that in some spots, the original terrain from the larger dataset will show through, covering up the desired auxiliary terrain. To avoid this, a list of the tie points and dimensions of all currently used auxiliary terrains is sent to the shader, which parses these into masks with which to check against the vertices. If all points of a triangle fall within the masked area, all points of that triangle are set to the origin. This sets the triangle's area to zero, causing it not to be drawn. What this looks like visually is a hole punched out where the auxiliary terrain would be, which is then covered by said auxiliary terrain. Figure 7 shows an example of rendering auxiliary data for Malapert Massif, a mountain near the lunar south pole.

Data Mapping

The visualization so far only displays a representation of the physical appearance of the terrain, but there is no reason it should be limited in this way. Most physical features of the terrain can be represented in a more explicit manner. In the current implementation, the minimum and maximum elevation values are precomputed and passed to the shader as uniforms. This allows the terrain to be colored as a gradient based on elevation, which provides further insight to users of the software, as shown in Figure 8. The principle of this can be extended to other physical characteristics, such as slope and normal vectors.

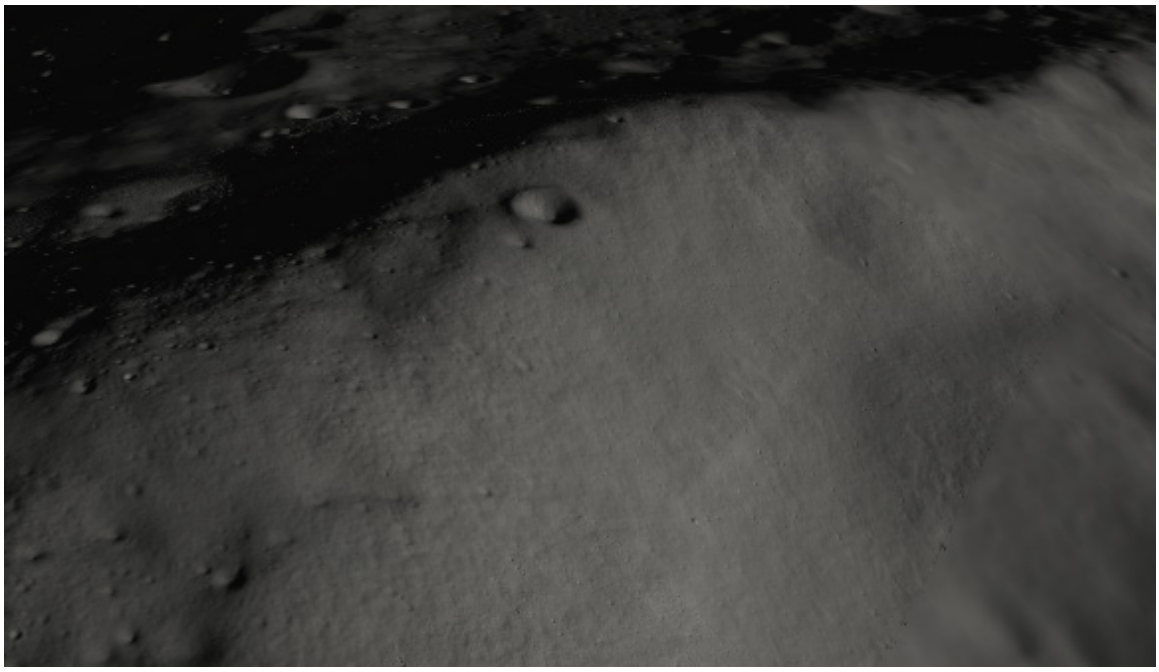

Figure 7.—Auxiliary data view of Malapert Massif.

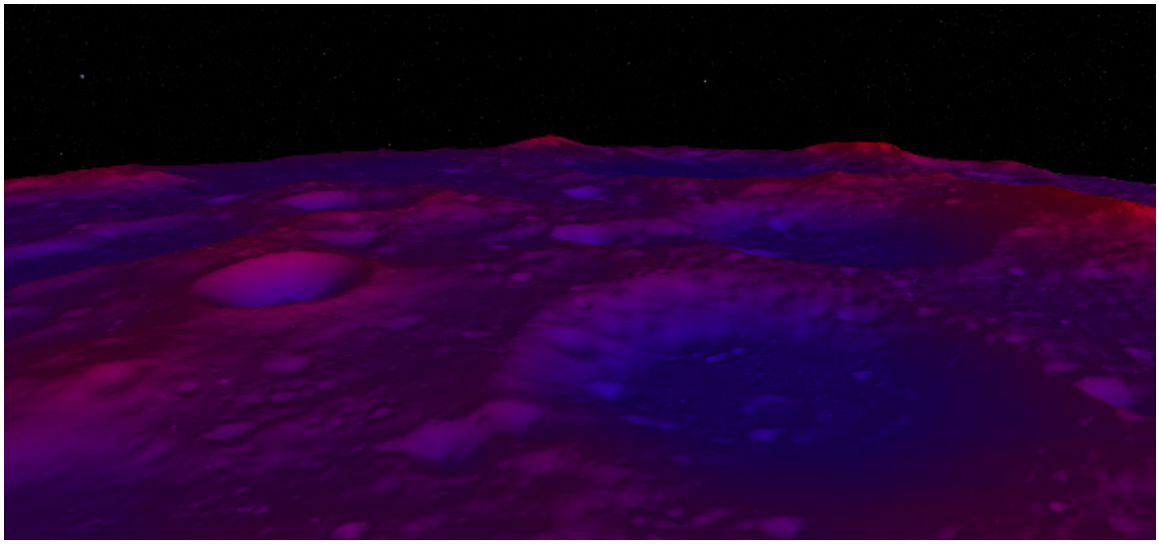

Figure 8.—Elevation data colormap gradient.

Results and Discussion

The result of replacing the old method of terrain rendering with this new method is a theoretical sixfold decrease in memory usage. The full resolution of data was never successfully displayed under the old method, so the memory usage reduction is an estimate. Without the attribute buffers for positions and normal vectors, the only memory cost is the GeoTIFF image itself, which is only one-sixth the size of the combined attribute buffers. In terms of performance, the new method allows displaying terrain at a much higher resolution without crashing the program. Rendering at full resolution reduces the framerate significantly, but the dynamic level of detail brings this to a usable, if still low, framerate.

However, this method is much more complex and difficult to understand, potentially making code maintenance more difficult in the future. Many rendering engines abstract away the need to deal directly with graphics application programming interfaces (APIs) or shaders, but this technique relies heavily on understanding the intricacies of how vertex shaders work. The increased complexity is ultimately outweighed by the benefits it provides, but it is worth considering when judging the usefulness of this new method.

Another point to consider before implementing this method is the narrow use case. It is rare that the bottleneck in rendering a three-dimensional scene is the memory usage, as it was for the specific case for which this was used. More often, it is performance, which the new technique all but ignores in its quest to reduce random access memory (RAM) usage. Nonbrowser applications generally have both more memory and more options to optimize memory and performance (tessellated geometry), which outperform the method described in this report in essentially every regard. In the near future, even browsers may have better options with the adoption of WebGPU, a new graphics API for the web. In other words, although the implementation of this rendering technique was successful in solving the immediate memory limitation issues, it might not be the best path to continue to pursue, other than in similarly exceptional circumstances.

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. Wikipedia contributors: Built-in Variable (GLSL). OpenGL Wikipedia, The Free Encyclopedia, 2015. http://www.khronos.org/opengl/wiki_opengl/index.php?title=Builtin_Variable_(GLSL)&oldid=12444 Accessed Aug. 23, 2023.
3. Wikipedia contributors: Face Culling. OpenGL Wikipedia, The Free Encyclopedia, 2015. http://www.khronos.org/opengl/wiki_opengl/index.php?title=Face_Culling&oldid=14600 Accessed Aug. 23, 2023.