Procedural Generation of 3D Planetary-Scale Terrains

Ryan J. Vitacion
College of Engineering and Computer Science
California State University, Northridge
Northridge, California, USA
ryan.vitacion.108@my.csun.edu

Li Liu
Department of Computer Science
California State University, Northridge
Northridge, California, USA
lliu@csun.edu

***Abstract*— The creation of three-dimensional geographical surface has been a primary concern at the forefront of the fields of space mission simulation. This paper introduces how to apply procedural terrain generation techniques to the creation of 3D terrains for a spherical object. The paper first identifies algorithms that can be used to generate terrains on a spherical surface. Then, the paper compares computational complexity and scalability of using these algorithms in 3D planetary scale simulation. The paper uses a benchmarking program created in virtual reality (VR) to evaluate the performance of these algorithms in the simulation and imaging of planetary bodies in VR including execution time, quality and memory usage.**

Keywords—Procedural Generation, Noise Algorithm, Planetary Terrain, Simulation, Imaging, Virtual Reality.

I. INTRODUCTION

The creation of three-dimensional virtual terrains is a constantly evolving topic in the field of computer graphics. Virtual terrains have diverse applications in many fields, ranging from movies and video games to simulation. These virtual environments are often hand-crafted by 3D artists and modelers. However, hand-crafting virtual terrains can prove to be an extremely complex task depending on the physical scale of the desired environment. Consequently, crafting vast virtual terrains on a large scale, such as that of an entire planet, can take a team of talented 3D artists many weeks or even months of manual modeling [12]. Such an endeavor is undoubtedly very costly with regards to the personnel and time required. Thus it is evident that the primary challenge of crafting such vast virtual environments is one of scalability.

Procedural terrain generation techniques alleviate the time, personnel, and scalability requirements of crafting virtual terrains by generating them algorithmically. These techniques mathematically shape each point of the terrain according to a deterministic function that can be applied over an infinite coordinate space. Consequently, a computer can utilize such techniques to create terrains an order of magnitude larger than those crafted by hand in a fraction of the time [3]. Given these advantages over manually modeling terrains, procedural generation is an integral component in the creation of large scale virtual terrains.

In recent years, rapid advancements graphics hardware and rendering technology have given rise to easily accessible virtual reality (VR) hardware available to 3D modeling. Evident from the already numerous experiences available to users, VR offers an unprecedented level of interactivity and immersion within virtual environments. Thus VR constitutes a new and rapidly evolving medium with great potential not only in entertainment and media, but also in practical applications such as simulation and training. For example, consider a VR simulation for training spacecraft pilots or a VR application for the imaging of the surfaces of distant planetary bodies. Such applications would directly benefit from the use of vast procedurally generated terrains.

II. PROCEDURAL TERRAIN GENERATION

Most procedural terrain generation techniques follow a consistent and systematic process in order to achieve geometry resembling natural terrain features. The process begins with a basic geometric shape represented by a 3D mesh. Traditionally, procedural terrain generation algorithms are applied to flat horizontal planes composed of a lattice of vertices. Each vertex is assigned an elevation value relative to its height above this plane. Initially, all vertices are assigned an elevation value of zero and thus form a completely flat terrain. At this point, the mesh is systematically deformed by introducing pseudorandom variation to each vertex's elevation value through the use of a deterministic mathematical function. Where all procedural terrain generation techniques differ is in the exact function used to calculate this variation in elevation and the specific ways through which this function can be parameterized. The end result is a completely new terrain geometry represented by the successfully deformed mesh.

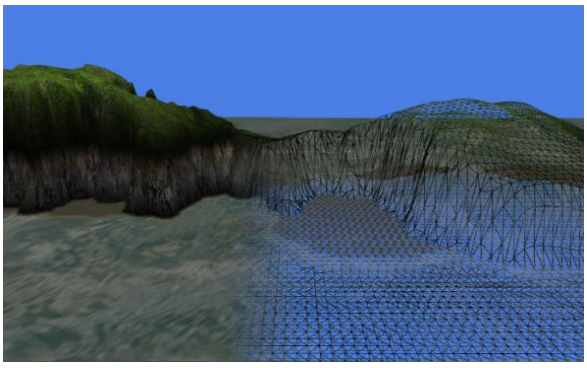

Figure 1. A procedurally generated terrain with the mesh visible. [1]

978-1-7281-1545-0/19/$31.00 ©2019 IEEE
DOI 10.1109/SMC-IT.2019.00014

A. Heightmaps

Heightmap is used as a principal data structure to store the elevation value of each vertex in the mesh's 3D geometry in procedural terrain generation. Heightmaps uses a two dimensional lattice of numerical values to represent the elevation value of a vertex at specific coordinates. The size and coordinate system of the heightmap mirrors that of the mesh targeted for deformation. Traditionally, heightmaps are created in the form of an image file, using the individual color values of each pixel to represent the height of the corresponding vertex in the mesh [11]. For example, traditional heightmaps take the form of grayscale images, with the brightness or darkness of each pixel representing the height of a specific vertex. Completely black pixels represent vertices at the lowest possible elevation in the mesh, while completely white pixels represent the highest possible elevation. Furthermore, heightmaps in the form of images provide a visual representation of the output terrain prior to application to the base mesh. However, a significant limitation of heightmaps is that each pixel can only represent a single elevation value. This renders heightmaps unable to represent more complex terrain features such as caves or overhangs, where any vertex located within would need to represent the height of the terrain both at the surface and below. In spite of this, heightmaps constitute a simple and straightforward structure that can be used to precisely assign and store elevation values for mesh deformation.

B. Noise

A noise algorithms is used to assign actual elevation values to each pixel in the heightmap programmatically in most procedural terrain generation techniques. When generating heightmaps, the input to these noise functions are the coordinates of each vertex and the output is each vertex's elevation value. The most basic method to produce these values is to simply iterate through the entire heightmap and assign random values to each pixel, a concept known as white noise. This approach, however, results in volatile elevation values and coarse terrain since there are no constraints present on how much variation can exist between nearby pixels. This can result in unnaturally rapid changes from the lowest elevation value to the very highest.

The solution to this problem is to provide a mechanism through which each vertex can be given awareness of the elevation of other vertices nearby. Such a mechanism would allow constraints to be placed on sharp variations and ensure that any transitions in elevation are smooth and non-volatile. Noise algorithms designed with this concept produce what is known as coherent noise. Figure 2 illustrates the difference between white noise and coherent noise visually. Coherent noise functions exhibit three constraints. Any input value will always result in the same output value, small changes to the input value result in small changes in the output value, and large changes to the input value will result in more random changes in the output value [2]. Consequently, most procedural generation algorithms utilize the principles of coherent noise to ensure that the resulting terrain will be sufficiently smooth without any of the sharp spikes that are possible when utilizing white noise.

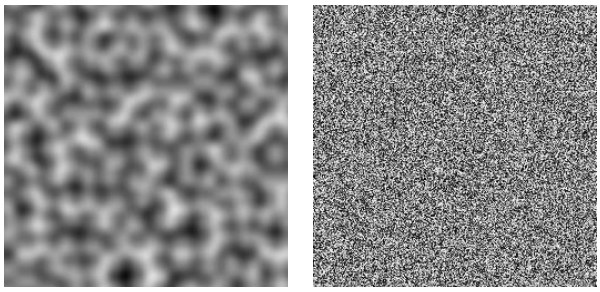

Figure 2. White Noise (Left) vs. Coherent Noise (Right)

Accurately simulating naturally occurring terrain phenomena presents the core challenge of designing an effective noise algorithm. While how well a noise algorithm mimics natural terrain is an arguably subjective matter, there are principles that can be followed in order to increase output quality. In observing naturally occurring terrain, polygonal shapes with hard angles and straight edges rarely occur. However, patterns resembling fractals often occur naturally in the form of coastlines, rivers, and mountains, for example [11]. Thus the self-similarity and chaotic appearance inherent to fractal patterns make them particularly useful when attempting to mimic natural terrain.

Many coherent noise algorithms are designed with this principle in mind, and can be applied iteratively on themselves through the use of a principle called Fractional Brownian Motion (fBm) in order to create fractal-like patterns [4]. When applying fBm to a noise algorithm, multiple iterations are run recursively, with each iteration increasing in frequency and decreasing in amplitude. This serves to introduce additional noise to the output and fill in the finer details of the terrain, resulting in higher overall fidelity. Using this technique, noise algorithms can be adapted to more accurately represent the appearance of terrain as it is found in nature.

C. Spherical Terrains

Traditional methods of generating heightmap with noise algorithm in procedural terrain generation do not work well when the surface becomes spherical since flat planar meshes only need rectangular terrains of limited length and width. In order to adapting a rectangular heightmap to a spherical object, an appropriate mesh representing a sphere must first be selected to act as the base geometry to be deformed. In choosing such a mesh, the primary concern is how evenly distributed the vertices are since the density of the vertices in a given area determines the amount of geometric detail that can be applied there. If a mesh concentrates too many vertices in a particular area and less in others, the geometric detail of the mesh will not be uniform and some areas will exhibit less of the detail present in the heightmap [7]. There are several approaches to representing a sphere in 3D space, perhaps the most familiar of which mirrors the geographic coordinate system of latitude and longitude. In this system, the sphere's vertices are placed at the intersections of the latitude and longitude grid. While simple and straightforward, this method

does not lend itself well to the application of heightmaps because the vertices become more dense and concentrated near the poles.

Another approach is to use a regular polyhedral shape and refine it in order to approximate a perfect sphere [7]. Regular polyhedral are three dimensional geometric shapes composed of faces of regular polygons, such as equilateral triangles or squares. Perhaps the most familiar regular polyhedron is the cube, which of course features six uniform squares as its faces. In order to approximate a sphere, each of the squares can be further subdivided into four new ones by adding new vertices at the midpoints of each edge and the center of the original square. This can be done recursively, resulting in exponentially finer geometric detail with each iteration. Once sufficient detail has been achieved, the position of each vertex can be adjusted such that it is one unit of distance away from the volumetric center of the cube. The result is an approximation of the unit sphere which can then be scaled and deformed using a heightmap.

Similar approaches utilize the recursive subdivision of other regular polyhedral, one such example being the regular icosahedron which is composed of twelve vertices and twenty equilateral triangles as faces. In this case, these equilateral triangles are subdivided in the same manner as the cubes in the previous approach, resulting in a slightly different geometric approximation of a perfect sphere. Due to their intrinsic uniformity, this method of recursively refining regular polyhedra ensures that the resulting mesh evenly distributes geometric detail across the entire sphere [7].

In the traditional application of heightmaps, each vertex's elevation value represents its vertical deviation from a flat reference plane with these values assigned by a noise algorithm. A similar technique can be applied in principle to spherical terrains. Since the reference mesh in a spherical terrain is a perfect sphere, the numerical value representing each vertex's height is its distance from the sphere's center, or its radius. Thus instead of representing vertical deviation from a flat reference plane, the values contained within a heightmap can be used to represent radial deviation from the surface of the reference sphere. With this principle in mind, the heightmap can then be mapped to the mesh using a spherical coordinate system. The end result is a procedurally deformed sphere utilizing an analogous approach to generating rectangular terrains.

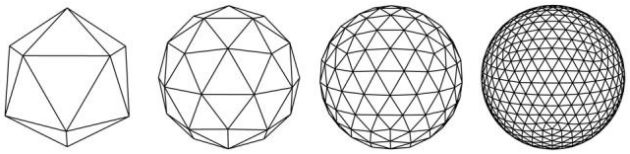

Figure 3. Icosahedral Approximation of a Sphere. [7]

III. NOISE ALGORITHMS

Five algorithms that are commonly used for procedural terrain generation were selected to evaluate their performance of producing noise for use in procedural terrain generation. They are Value Noise, Cubic Noise, Perlin Noise, Simplex Noise, and the Diamond Square Algorithm. This section provides a brief overview of how they each work to produce noise for use in procedural terrain generation.

A. Value Noise

Value Noise assigns pseudorandom values to each point in the heightmap. The final elevation value of each point is obtained by interpolating its value with those of the surrounding points. Doing so gives each point the awareness of its neighbors' elevation values that its characteristic of coherent noise. Different variations of Value Noise can use varying interpolation functions to achieve this result. In this case, the particular variant of Value Noise tested utilizes quintic interpolation to smooth out variations in each point in the heightmap.

B. Cubic Noise

The Cubic Noise algorithm generates a map of pseudorandom white noise larger than the desired heightmap. This map is then scaled down to the desired resolution using cubic scaling [13]. This process takes every pseudorandom value of the original white noise and combines it with nearby values through the cubic interpolation with the overall effect of smoothing out the variations in the final result. Through this mechanism, each position in the final heightmap is given awareness of the value of nearby positions, creating coherent noise.

C. Perlin Noise

Perlin Noise is one of the oldest coherent noise algorithms that can be used to create heightmaps. Designed by Ken Perlin in 1983, Perlin noise works by first generating a grid of unit-length gradient vectors originating at each intersection that point in pseudorandom directions. The position of each noise value being calculated is placed within each cell such that each coordinate is surrounded by four grid points, each with their own gradient vector. Next, four distance vectors are calculated representing the distance from the coordinates of the noise value being calculated and the corners of the surrounding cell. Finally, the dot products between each gradient vector and each corresponding distance vector are calculated and interpolated in order to produce the final noise value [8]. Because the computation of each noise value is dependent on the distance vectors, each point is given awareness of the value of its neighbors and coherent noise is achieved [10].

D. Simplex Noise

Simplex Noise is another coherent noise algorithm designed by Ken Perlin in 2001 that serves as an optimization of his earlier Perlin Noise algorithm [9]. Simplex Noise simplifies Perlin Noise by utilizing a simplex grid in which each cell is a triangle instead of a square. This results in less distance vectors needed to calculate each height value since each cell only has 3 corners. Furthermore, Simplex Noise utilizes a summation process instead of an interpolation to compute the final noise values [5]. Simplex Noise is less computationally complex than Perlin noise and thus enjoys better scalability. By using simplex grids, directional artifacts are also minimized.

E. Diamond Square Algorithm

The Diamond Square Algorithm is another coherent noise algorithm used to generate heightmaps. The algorithm utilizes the principle of midpoint displacement in order to assign elevation values to each coordinate [17]. The algorithm begins with a square coordinate grid with the four corner coordinates initialized to pseudorandom values. It then calculates the average of the elevation of these four point plus or minus a pseudorandom offset. This value is then assigned to the elevation of the point in the direct center of the grid. The grid is then subdivided into four smaller squares and the process is repeated for each of them. This is done until every value in the grid has been assigned an elevation value.

At each subdivision level, the magnitude of the pseudorandom offset decreases, ensuring that sharp variations do not occur since any changes between coordinates in close proximity are calculated by the smaller subdivision steps. Through this mechanism, the Diamond Square Algorithm achieves coherent noise. It is important to note that the Diamond Square Algorithm is unique from the other algorithms in that it is inherently fractal [11]. Consequently, it is unnecessary to utilize a method like Fractional Brownian Motion in order to achieve fractal-like noise. However, the algorithm has several limitations. In order for the original grid to subdivide completely, the original grid must have dimensions of 2n+1. In addition, the algorithm can only produce square heightmaps due its design.

IV. Noise Benchmarking

A. Technical Approach

In order to obtain a practical understanding of each noise algorithms' performance ramifications for procedurally generating planets in VR, each algorithm was put through a benchmark testing on the dimensions of computational complexity, memory requirements, and dependency on hardware within actual VR settings. The engine used in this paper is Unity, a cross-platform development environment for developing 3D applications. Unity was chosen as the target engine due to its native support for VR integration, mature feature set, and thorough documentation. Unity comes with a fully featured scripting engine and API that supports C# as a primary scripting language. Thus the programming language used to develop both the benchmarking application and the rendering prototype is C#.

In order to measure each noise algorithms' computational complexity, memory requirements, and dependence on hardware, a comprehensive test suite named "NoiseBenchmark" with a graphical user interface was created in Unity that iteratively generates heightmaps using each algorithm.

Whenever a heightmap is generated, the benchmarking program also tracks the execution time of the algorithm in milliseconds, the memory usage of the heightmap, and the overall size of the managed memory heap allotted by the Unity engine. The benchmarking program can not only be used to run a the full test suite but also to generate individual heightmaps with variable parameters. The program's graphical user interface features multiple fields for manipulating these various parameters affecting the output of the noise algorithms. By isolating certain parameters and holding all others constant, the benchmarking program is able to analyze the performance of each algorithm along each of the desired dimensions.

To generate a single heightmap, the user simply sets the desired values for each parameter and clicks the "generate" button. Once this is complete, the user can then click the "export" button and the program will output a .png image file to the directory it is located within. The program will then generate the heightmap and display the output in the rendering window. The rendering window also displays the generation time, memory usage, and heap size for the completed operation. The user can also run a specific test case by setting the parameters and clicking the "Run" button. The "Run All" button will run the complete test suite for all noise algorithms while setting the parameters automatically throughout its execution. These functions will output the results of each test case to a .csv text file, which can then be converted to a spreadsheet for analysis.

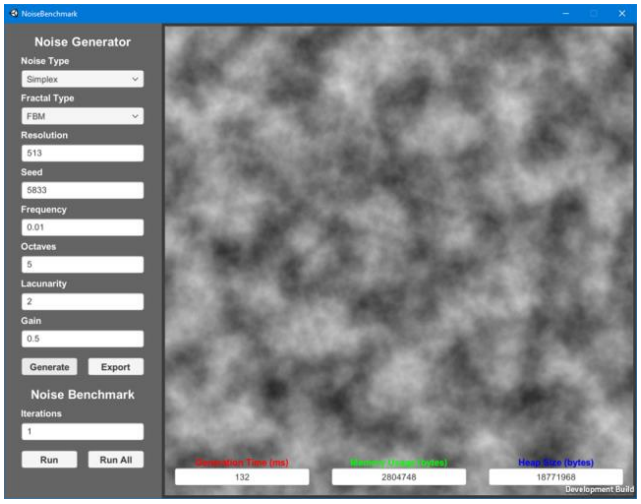

Figure 4. The NoiseBenchmark GUI

B. Methodology

In order to systematically compare each noise algorithm along the desired performance dimensions, the benchmarking program follows a completely automated test suite that runs each algorithm at different sets of fixed parameter values. The test suite generates heightmaps for each algorithm at four levels of resolution and five levels of fractal octaves for the specified number of iterations. This process is executed for four resolutions, 256×256, 512×512, 1024×1024, and 2048×2048. Furthermore, in order to eliminate any computational advantages that may occur if the seed value was held constant for each run, the seed value is automatically randomized by the application before generating each height map.

To capture the values for generation time, memory usage, and heap size, the benchmarking application makes use of several features native to C# and the Unity engine. The C#

System.Diagnostics namespace features a Stopwatch class, which allows breakpoints to be placed at certain points in the code in order to measure the elapsed time between them in milliseconds. To capture memory diagnostics, the Unity API features the UnityEngine.Profiling namespace which includes the Profiler class. This class has several methods that can be used to track memory usage in code.

In order to obtain a sufficiently large dataset for analysis, a test suite consisting of one hundred iterations was performed. For each combination of resolution and fractal octaves, the results of the one hundred iterations were then averaged and plotted to gain a visual representation of each algorithm's performance curves. In addition, in order to explore each algorithm's dependence on hardware, the one-hundred-iteration test suite was run on three different computers, each having a diverse set of hardware components. These three computers were classified into three types, a low-end machine, a mid-range machine, and a high-end machine. The separate datasets from these three test suites were also plotted and compared in order to examine how each algorithm performs on different hardware setups.

V. Results

A. Computational Complexity

The primary metric used by the test suite to assess each algorithm's computational complexity is generation time. First, the average generation time of each algorithm was plotted against the number of fractal octaves performed. In this test case, the resolution was held constant at 2048×2048 as well as the hardware setup the test was run on. Note that because the Diamond Square Algorithm is inherently fractal by design, fractal octaves are not applicable and thus the algorithm is omitted from this test. It is clear from the data that for each algorithm, generation time appears to scale roughly linearly with respect to the number of fractal octaves. Based on Figure 5, conclusions about how well each algorithm scales with respect to the number of fractal octaves can be made based upon the slope of each curve, which represents how quickly generation time increases with each fractal iteration.

Cubic Noise displays the worst scalability having higher overall generation time than the other algorithms and the steepest performance curve. Perlin Noise comes in second with a shallower performance curve and slightly worse generation times than Value and Simplex Noise. Value and Simplex Noise exhibit very similar performance curves, with Value Noise performing slightly better. Increasing the number of fractal octaves when generating heightmaps provides the benefit of a corresponding increase in the finer details of the output results. For large scale heightmaps such as those used when generating planetary terrains, the ability to generate heightmaps with more fractal octaves in less time is a valuable benefit to have.

Another metric through which the computational complexity of each algorithm can be analyzed is the actual resolution of the heightmap itself. In this test case, generation time was plotted against the increasing resolution of each heightmap, with the number of fractal iterations being held constant at four octaves. Four fractal octaves is a reasonable compromise between generation time and detail and provides sufficient visual parity to the inherent fractal quality of the Diamond Square Algorithm. Figure 6 on the following page displays each algorithms' performance curves for this test. The data shows that for each algorithm, generation time scales in a markedly nonlinear fashion with respect to the resolution of the target heightmap.

Cubic Noise once again exhibits the worst performance, with Perlin Noise performing slightly better. Simplex Noise and Value Noise display very similar performance, though Value Noise scales slightly better than Simplex Noise. The Diamond Square Algorithm exhibits the fastest generation times and the shallowest performance curve. Thus it is clear that the Diamond Square Algorithm has the best scalability in terms of the resolution. Increasing the resolution of the target heightmap provides it with finer detail by allowing more overall elevation values to be stored. The density of these coordinates also increases, allowing a higher resolution heightmap to generate greater geometric detail when applied. This is a significant benefit when generating vast terrains such as those on a planetary scale.

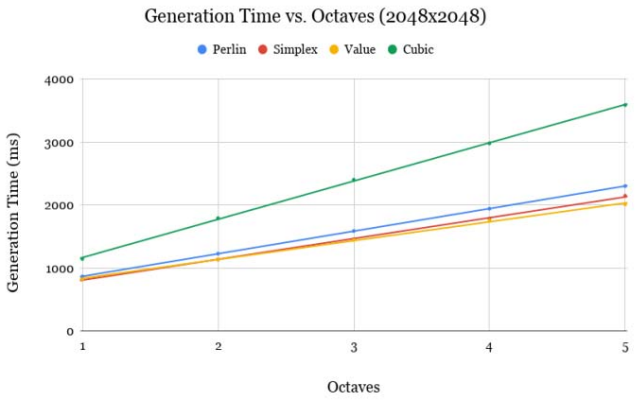

Figure 5. Generation Time vs. Octaves

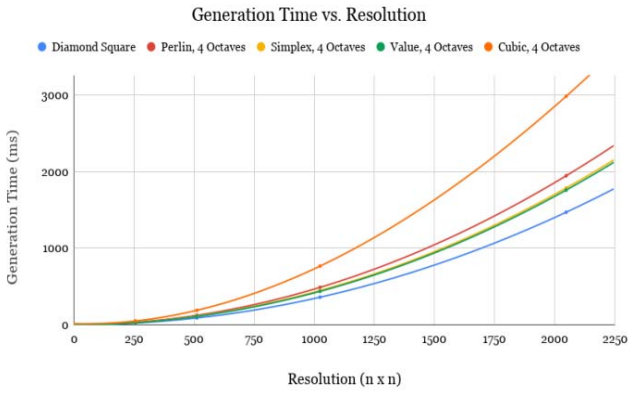

Figure 6. Generation Time vs. Resolution

B. *Memory Usage*

The primary factor in determining how much memory a heightmap consumes within the Unity engine is its resolution. Whenever a heightmap is being generated, the engine instantiates a two-dimensional texture object at the specified resolution [16]. This texture file is composed of a total number of pixels equal to the resolution squared, each with their own color values. Unity automatically allocates a section of the managed memory heap to this texture object, which the benchmarking application measures along with the total size of the heap itself. Recall that the Diamond Square Algorithm operates with the significant caveat that it can only generate heightmaps of 2^n+1 resolution. The resolution of heightmaps generated with the Diamond Square Algorithm can only be increased in discrete steps, which themselves increase exponentially. Thus, when comparing the memory usage of the Diamond Square Algorithm against that of the other four algorithms at a 2n resolution, the closest possible resolution heightmap the Diamond Square Algorithm can generate will be this value plus one.

Consequently, heightmaps utilizing the Diamond Square Algorithm will require slightly more memory than those generated with the other four algorithms at similar resolutions, which can be scaled continuously. Figure 7 compares the average memory usage on the high-end system of Simplex Noise at five octaves and the Diamond Square Algorithm along with the peak size of the Unity managed memory heap. The performance curves in this graph illustrate how a heightmap generated using the Diamond Square Algorithm will consume more memory on average than the other four algorithms due to the discrete resolution limitation. The graph also illustrates how the managed memory heap grows as the resolution increases. This raises the question of what percentage of the managed memory heap the heightmap consumes.

The Unity engine will continue to draw memory from the system in order to grow the managed memory heap as necessary until no more physical memory is available. If the amount of memory used ever exceeds the maximum heap size, the application will crash. It is thus beneficial to minimize the percentage of the heap the heightmap consumes in order to allow the remaining portion to be allocated for other 3D objects or rendering functions. Figure 8 illustrates the percentage of heap consumed on the high-end system by heightmaps generated with five octave Simplex Noise and the Diamond Square Algorithm at different resolutions.

From this graph, it is evident that heightmaps generated using the Diamond Square Algorithm will consume a larger percentage of the heap at similar resolutions when compared to the other four algorithms. By extension, the Diamond Square Algorithm will reach the upper memory limit much faster than the other four algorithms due to the discrete resolution limitation. From a practical standpoint, less managed memory will be available for other 3D objects or rendering functions [16]. This is an important factor to consider if the 3D application being developed is to render a scene with the procedural planet alongside many other 3D objects or rendering functions. It thus follows that though the Diamond Square Algorithm has several performance benefits over the other algorithms due to its inherent fractal nature, developers must consider how much memory they wish to be allocated to the heightmap itself versus other assets being rendered, as well as if one of the discrete output resolutions of the Diamond Square Algorithm will provide sufficient benefits in terms of detail given this memory trade off.

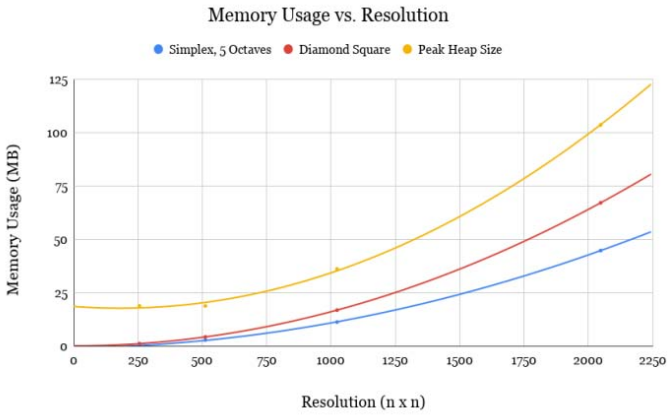

Figure 7. Memory Usage vs. Resolution

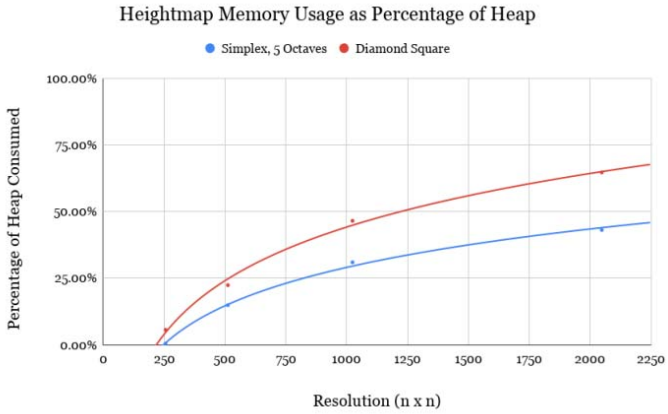

Figure 8. Memory Usage vs. Resolution

TABLE I. TEST SYSTEM HARDWARE SPECIFICATIONS

	Low-End	Mid-Range	High-End
CPU	Intel Core 2 Duo E8400	Intel Core i5-2500k	AMD Ryzen 7 1700X
Core Count	2	4	8
Logical Processors	2	4	16
CPU Frequency	3.0 GHz	4.6 GHz	3.9 GHz
RAM	4 GB DDR3	16 GB DDR3	16 GB DDR4
RAM Frequency	1333 MHz	1600 MHz	3200 MHz
GPU	Nvidia GTX 260	Nvidia GTX 670	Nvidia GTX 1070
VRAM	2 GB	6 GB	8 GB
Year	2009	2011	2017

C. Hardware Dependency

As mentioned previously, the test suite was run on three separate systems with varying specifications in order to test each algorithm's dependence on hardware. These three systems were classified as a low-end system, a mid-range system, and a high-end system based on their computational capabilities. The results for each system were then plotted against one another for each algorithm, allowing for a general performance comparison. Overall, the greatest performance gap in terms of generation time between the low-end system and the high-end system reached as high as one and a half seconds, with this gap increasing rapidly with heightmap resolution. Based on the hardware tested, CPU clock speed, amount of available RAM, and RAM frequency had the most significant effects on generation times. It is important to note that for all algorithms, systems with more RAM are able to generate higher resolution heightmaps. Table 2 summarizes each algorithm's hardware dependency.

TABLE II. HARDWARE DEPENDENCY COMPARISON

Algorithm	Dependence on Hardware Specification		
	CPU Clock	*RAM Frequency*	*Available RAM*
Value	Low	High	Moderate
Cubic	High	High	Moderate
Perlin	Moderate	Moderate	Moderate
Simplex	Low	Low	Moderate
Diamond Square	High	Low	High

VI. PROCEDURAL PLANETS IN VR

A. Planetary Scale Rendering

The procedural generation of a spherical terrain begins with a base shape that is then refined through recursive subdivision. While this recursive subdivision greatly increases the geometric detail available in the mesh, it also increases the number of vertices and triangles exponentially [14]. It is critical to note that the resulting increase in geometric detail is distributed uniformly throughout the mesh. However, depending on the geometry of the planet itself and the location of the camera viewpoint within the scene, many of these vertices and triangles may be very far away or obscured by the planet's horizon where increased geometric detail is unnecessary [15]. The number of triangles and vertices being rendered in a 3D scene directly affects the application's framerate, and most VR applications recommend high framerates in order to minimize motion sickness and tracking errors. It thus follows that minimizing the total triangle count while preserving geometric detail is critical in rendering procedural planets in VR.

The solution to this problem is to devise a system where geometric detail is dynamically increased where it is most needed. Areas that are distant or not visible to the camera do not benefit at all from any increased geometric detail. Thus, the resources that would be used to render increased geometric detail in these two cases would be put to better use rendering what the camera can actually see and areas of the terrain that exhibit volatile variations in elevation. This dynamic adjustment of geometric detail would greatly reduce the overall triangle count of the scene being rendered at any given frame, increasing the applications framerate and optimizing performance [6]. Such systems are known as Level-of-Detail systems or LOD.

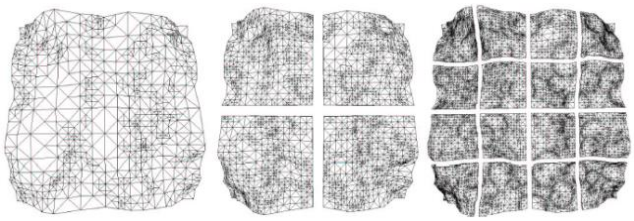

Figure 9. Dynamic Levels of Detail [14]

One example of a dynamic LOD system is the Adaptive Quadtree system. The system begins with a square mesh with the capability to be subdivided into four smaller square meshes. These four child meshes can be further recursively subdivided as well, for as many levels as necessary. A data structure known as a quadtree is used to keep track of each mesh and its children. Quadtrees are a type of tree data structure in which each node has four children. Thus each node in the quadtree perfectly corresponds with each square mesh in the system as well as its children. Upwards and downwards traversals of the quadtree decrease and increase the level of geometric detail, respectively [15]. Applying this process in appropriate areas results in the conservation of rendering resources that is crucial in rendering procedurally generated planets at scale.

B. Rendering Prototype

The VR rendering prototype, named "PlanetGenVR", was also developed using the Unity engine with C# as the primary scripting language. In order to integrate VR functionality for rendering on the HTC Vive, the SteamVR API was included in the program as well. In addition, a Unity plugin named "Planetary Terrain", which provides an Adaptive Quadtree LOD system, was integrated into the prototype, maximizing performance. Combining these software packages, a functional prototype was created that is capable of rendering large procedural planets at a high enough framerate for use in VR.

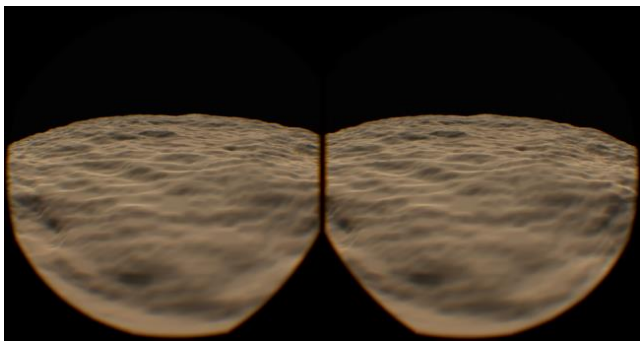

Figure 10. Perlin Noise VR Sample

C. Output Quality

To obtain samples of each algorithm's output quality in VR, heightmaps were generated at an 8192×8192 resolution with five octaves where applicable. The seed value and all multiplier values were held constant to ensure a consistent comparison between each algorithm's visual output. Every sample ran with a framerate of at least ninety frames per second, which is ideal for VR. While how well an algorithm simulates natural terrain is a largely subjective matter, the samples still provide useful insight when taken in the context of the performance analysis conducted previously. Each algorithm resulted in planets with unique degrees of geometric variation, fine detail, and presence of noticeable blocks or directional artifacts. Table 3 summarizes the general differences in output quality for each algorithm and Figure 10 displays a VR rendering sample of Perlin Noise.

TABLE III. OUTPUT QUALITY COMPARISON

Algorithm	**Output Quality**		
	Geometric Variation	***Fine Detail***	***Directional Artifacts***
Value	Moderate	Moderate	Moderate
Cubic	Low	Low	High
Perlin	Moderate	Moderate	Low
Simplex	High	High	Low
Diamond Square	Low	Low	High

VII. CONCLUSION

In summary, this paper provided great insight into the subtleties of procedural generation of planetary terrains in VR. Based on this research, it can be concluded that it is indeed feasible to utilize this method of procedural generation to successfully generate 3D planets for use in VR simulations and imaging applications. However, several limitations presented themselves highlighting specific areas for further research. Based on the data generated during the benchmark tests, resolution is a considerably limiting factor in how quickly heightmaps can be generated and directly affects the quality of generated terrain. Faster and more efficient noise algorithms that can generate higher resolution heightmaps in less time would be invaluable in this regard In addition, alternative data structures specialized for spherical terrains that can efficiently store more elevation values with less memory than traditional heightmaps constitute another area for improvement. Finally, the existing parameterization of the noise algorithms tested are rather abstract and tend to affect the output globally, resulting in a degree of uniformity in the terrain with limited control over placing particular terrain features in specific areas. Thus clearer and more straightforward ways to parameterize these noise algorithms is key and would provide developers with more definitive control over the terrains they create.

REFERENCES

[1] F. Bevilacqua, C. T. Pozzer, and M. C. Dornellas, "Charack: Tool for Real-Time Generation of Pseudo-Infinite Virtual Worlds for 3D Games," *2009 VIII Brazilian Symposium on Games and Digital Entertainment,* 2009.

[2] J. Bevins, "Libnoise Glossary," *Libnoise*. [Online]. Available: http://libnoise.sourceforge.net/glossary/index.html#coherentnoise. [Accessed: 05-Sep-2018].

[3] D. M. D. Carli, F. Bevilacqua, C. T. Pozzer, and M. C. Dornellas, "A Survey of Procedural Content Generation Techniques Suitable to Game Development," *2011 Brazilian Symposium on Games and Digital Entertainment*, 2011.

[4] A. Cristea and F. Liarokapis, "Fractal Nature-Generating Realistic Terrains for Games," *2015 7th International Conference on Games and Virtual Worlds for Serious Applications (VS-Games)*, 2015.

[5] S. Gustavson, "Simplex Noise Demystified," 22-Mar-2005. [Online]. Available: http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf. [Accessed: 15-Sep-2018].

[6] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner, "Real-time, Continuous Level of Detail Rendering of Height fields," *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*, 1996.

[7] R. Kooima, J. Leigh, A. Johnson, D. Roberts, M. Subbarao, and T. Defanti, "Planetary-Scale Terrain Composition," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 5, pp. 719–733, 2009.

[8] K. Perlin, "An Image Synthesizer," *12th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '85)*, 1985.

[9] K. Perlin, "Improving Noise," *29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '02)*, 2002.

[10] D. Maung, Y. Shi, and R. Crawfis, "Procedural Textures Using Tilings with Perlin Noise," *2012 17th International Conference on Computer Games (CGAMES)*, 2012.

[11] T. J. Rose and A. G. Bakaoukas, "Algorithms and Approaches for Procedural Terrain Generation-A Brief Review of Current Techniques," *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*, 2016.

[12] R. Smelik, J. de Kraker, S. Groenewegen, "A Survey of Procedural Methods for Terrain Modelling," *TNO Defence, Security, and Safety*, 2009.

[13] J. Talle, "Job Talle," *Cubic Noise*, 31-Oct-2017. [Online]. Available: https://jobtalle.com/cubic_noise.html. [Accessed: 27-Sep-2018].

[14] T. Ulrich, "Rendering Massive Terrains using Chunked Level of Detail Control," *29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '02)*, 2002.

[15] T. Ulrich, "Continuous LOD Terrain Meshing Using Adaptive Quadtrees," *Continuous LOD Terrain Meshing Using Adaptive Quadtrees*, 28-Feb-2000. [Online]. Available: https://www2.cs.duke.edu/courses/cps124/spring08/notes/14_datastructures/lod_terrain.html. [Accessed: 25-Aug-2018].

[16] "Understanding Automatic Memory Management," *Unity – Manual*. [Online].Available:https://docs.Unity.com/Manual/Understanding/AutomaticMemoryManagement.html. [Accessed: 04-Oct-2018].

[17] H.-R. Wang, W.-L. Chen, X.-L. Liu, and B. Dong, "An Improving Algorithm for Generating Real Sense Terrain and Parameter Analysis Based on Fractal," *2010 International Conference on Machine Learning and Cybernetics*, 2010.