Terrain Generation with Continuous Level of Detail

Lucian-Valentin Deaconu
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
valentin.l.deaconu@gmail.com

Constantin Nandra
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
constantin.nandra@cs.utcluj.ro

Dorian Gorgan
Technical University of Cluj-Napoca
Cluj-Napoca, Romania
dorian.gorgan@cs.utcluj.ro

ABSTRACT

This paper describes the implementation of a terrain generation technique, featuring a solution meant to address the problem of continuous level of detail. The implemented technique is employed as part of a custom-built rendering engine that can be extended by programming-proficient users to create desktop applications with 3D rendering capabilities. Throughout this paper, we focus on the challenges posed by generating and rendering terrain with variable level of detail, dependent upon the camera position. We analyze the roots of the problem and provide a tessellation-based solution to solve it, while maintainig the accuracy of the topology. The description of the various implementation aspects will also include some insight into the basic data structures employed.

Author Keywords
Dynamic Terrain Generation; Continous Level of Detail Problem; OpenGL.

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

General Terms
Algorithms; Dynamic Terrain generation; 3D rendering; GPU rendering.

DOI: 10.37789/rochi.2021.1.1.8

INTRODUCTION

Rendering engines are arguably some of the most important elements of any real-time rendering application. Their main purpose is to transpose a virtual scene into a sequence of digital images through the process called rasterization. A virtual scene consists of multiple objects. Some of them are so small that it is acceptable for them not to be displayed due to the very large distance between them and the camera. Others can be so large that they are visible from any point within the virtual scene. These require special optimizations, based on the reasonable assumption that it is not necessary to have high quality, detailed renderings of far away surfaces. Appliyng this simple principle in an effective manner could lead to significant performance gains when displaying complex scenes. This is the case with terrain rendering. The terrain is often one of the most important parts of the perceived environment in open-world scenes. It should be visible from far-off distances while also appearing detailed enough when observing patches which are close to the camera. In some scenes, the terrain is flat (simple) and does not require much computational power to render. However, significant problems may arise when land mass generation is required, because of details like mountains, valleys or simple bumps, that require the setup of additional vertices.

The problem with of detailed terrain rendering is somehow mitigated by the fact that, usually, only a few small parts of the terrain are close to the observer and therefore have to be rendered in high quality. The majority of the terrain will be far from the camera, and its detail quality could decrease with the distance. This kind of implementation optimization can be achived by dividing the surface into multiple patches, with varying levels of details, where low level of detail (high quality) will be displayed when the observer is close and high levels (low quality) will be displayed when the observer is far away.

When dividing the terrain according to varying levels of detail, patching them together comes with a visual artefact – cracks. This is due to neighbouring nodes that are found at the border of different levels of detail, where their resolution does not match perfectly. This problem is called "continuous level of detail" – CLOD and represents one of the main topics of this paper.

Throughout this paper we will present a method of generating a dynamic terrain model using the GPU. This model will be able to support the usage of height maps, lighting (using computed normals) and multiple textures. In the end, we will also provide a performance analysis of the proposed method that was implemented.

RELATED WORK

Land mass generation requires the definition of infinite meshes. Generating vertices along the x and z axis (latitude and longitude) does not represent a problem due to terrain's property of being a grid of equally distant vertices. Generating values along the y axis requires an infinite function that associates any pair of (x, z) coordinates to a floating-point value. Perlin noise represents a method of achieving this function in a pseudo-random manner as described within [1] and [2]. It is the recommended to be used in terrain generation due to its efficiency and its property of generating smooth noise.

If infinite terrain is not a requirement, the altitude values can be pre-processed and stored in a digital image named, called a height map, which contains normalised values (between 0 and 1). One such example is described within [3]. The authors propose a solution for generating terrain from small resolution height maps, in a bid to allow users lacking sophisticated tools to create detailed features with minimal input.

In the case of large-scale land masses, the mesh can become very complex, with a large number of vertices implying a direct impact on performance. Dividing the mesh into different levels of details can improve the execution time. In [4] a hierarchical division is described. The author uses a quad-tree data structure to split terrain into patches with varying levels of details. The authors of [5] also tackle the problem of variable levels of detail by using a hierarchical model, based on a construction tree representation. In this model, the leaves represent generic terrain features that can be instantiated with provided parameter sets, such as river beds, cliffs and ridges. Meanwhile, the intermediary nodes act as operations, meant to combine the leaves in different manners. This allows the creation of complex terrain models starting from a set of primitive features and operations.

Generally, the number of vertices involved in the mesh definition is directly proportional to the quality of the object, but inverse proportional to its performance. This is especially true in the exponential growth achieved by a hierarchical division. Some aditional geometry can be provided using the graphics card's tessellation functionality. This method is described in [6] and suggests the creation of new primitives inside of the existent primitive available in the mesh structure. The new verticies' properties can be obtained by interpolating those of the original vertices.

In [7], a solution for using tessellation in terrain generation is described with an example of graphic pipeline implementation, using OpenGL's shader programs. The author aimed at developing an adaptive tesselation algorithm that would balance the capability to display detailed terrain features and the computing resource utilization that it would entail. The authors of [8] also describe the employment of tesselation to add surface detail to terrain meshes generated from height maps. The solution presented within the paper is meant to tackle the memory limitations that the detailed visualization of such models could entail, while also compensating for the limited availability of large-scale elevation maps. In [9], the authors describe a rendering approach utilising a tile-based division of the terrain, with each tile associated to a quad-tree structure, whose nodes are then being fed to the GPU pipeline for tesselation. The described implementation can utilize both height maps and geometry images for generating surface detail data.

When working with terrain models with varying levels of detail across its surface, one of the most important problems to solve would be dealing with the transition from high-level to low-level areas. This tends to affect the smooth transitions of the height (Z axis) dimension, often leading to artifacts seen as gaps into the mesh. This issue is tackled in various souces, in manners pretty much dependent on the chosen terrain representiation model. For example, the authors of [10] describe a solution based on Dynamic Stitching Strips (DSS) used in conjunction with terrain tesselation to achieve smooth transitions between levels. This solution, the authors argue, is well suited for real-time rendering. A quad-tree representation model is described within [11]. With the terrain organized as a quadtree of tiles, the authors propose a morphing-based approach to solve the level transition problem.

Within this paper, we describe our own implementation of a solution meant to address, among others, the problem of gaps between the terrain tiles within a quad-tree. It has the benefits of being adaptable to different heightmaps and featuring a logarithmic search time for neighbouring tiles to determine the how to divide the patches along the boundry.

TERRAIN GENERATION

The solution described in this paper was developed as a module for a rendering engine, designed for use by the developers of 3D applications. Within the engine, this component can provide a default implementation of the ground upon which the virtual scene can be built. Figure 1 shows a screenshot of a demo scene that was built and rendered using the above-mentioned engine.

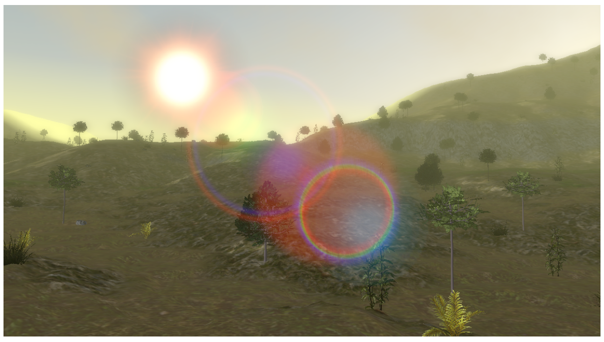

Figure 1. Landscape formed by the terrain with decorations, skybox, fog and sun with lens flare effect

In terrain generation, the basic approach suggests generating a simple two-dimensional grid of vertices on the X and Z axes, ignoring the height (Y axis). The obtained mesh, in a three-dimensional space, is represented in the form of a simple plane, or flat terrain (Figure 2).

In order to add bumps, height information has to be added on each vertex. In order to achieve this, a height function should exist. It can be defined as

$$h: \mathbb{Z}^2 \rightarrow [0, \infty)$$

and should return for each pair of coordinates (x, z) a positive height value. Such a function can be obtained by pre-processing a height map using noise generators and storing them in a digital single-channel image, normalising the values (Figure 3).

When a height map is read, the values should be multiplied with a constant N, representing the maximum height allowed in the terrain mesh (minimum height allowed will be 0).

Terrain can be mapped 1:1 to the height map, meaning that their dimensions should be equal and any pixel in the height map should have an equivalent as vertex in the terrain's mesh (Figure 4). This approach can yield terrain of scalable quality, dependent upon the complexity of the height map. The main benefit is the possibility of generating new terrain by changing the height map, without the need to make changes to the underlying code. However, the quality of the mesh overall is limited by the image resolution (high resolution height maps occupy more memory).

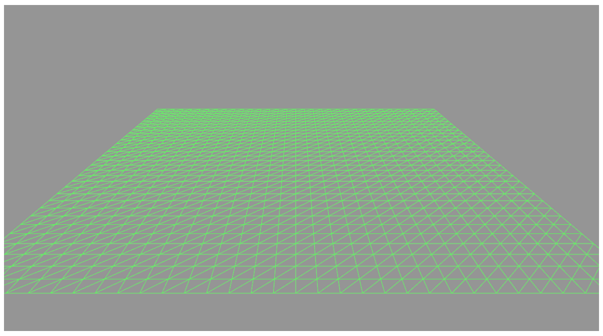
Figure 2. Flat terrain (without any height information)

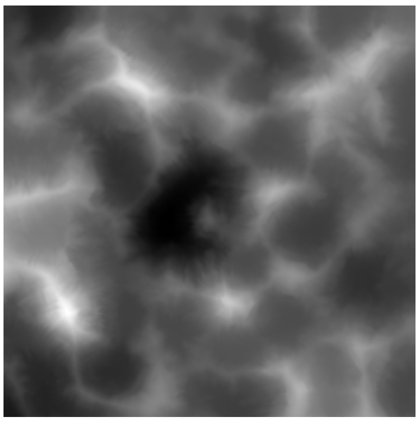
Figure 3. Example of height map (greyscale)

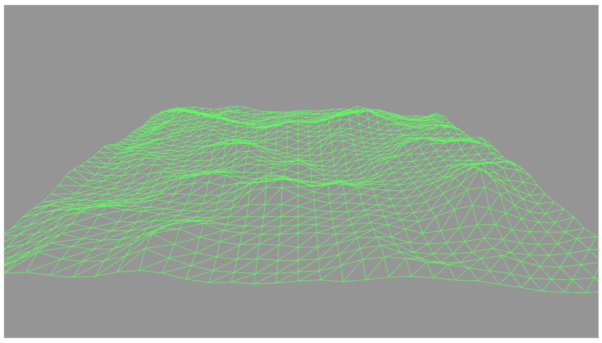
Figure 4. Simple terrain, with height informations

Navigation
To navigate over the terrain it is necessary to know the bi-dimensional coordinates (axes x and z) and using those to compute the y-axis value, by using the height function described previously, $y = h(x, z) * N$, where N is the height constant. The height value will always be positive (see the function definition) so the scaled value will also be positive.

In order to keep an object on the surface of the ground while moving, it is only necessary to keep track of the two axes, computing the third only when their value is changed. This method will allow the object to follow the terrain surface.

Increasing terrain quality
To obtain a high-quality terrain with an admissible resolution height map some vertices must exists without being mapped to exactly one pixel, implying that the height function should also provide values for those "sub-pixels". A simple approach to achieve this behaviour is using interpolations. On digital images, bilinear and bicubic interpolations obtain the best results [12]. This method also comes with limitations due to pixels' depth but compared with the previous approach it can provide a higher quality.

Tessellation
Tessellation or tiling represents the process of covering a plane (flat surfaces) using one or more geometric shapes with no gaps. Graphic cards come with the ability of tessellating primitives (triangles and quads) into multiple primitives. By taking advantage of this feature, new vertices can be created inside the terrain's mesh. Those new vertices will obtain their height value by interpolating the original primitive vertices' height values.

Optimization
Combining the methods mentioned until this point will result in a decent terrain representation, but the performance is highly impacted by the large number of vertices that must be processed for every frame.

The mesh processing step can be optimized based on the observation that it is not necessary to display the same number of vertices everywhere; the density may be different, based on the observer position in the virtual scene. The quality may decrease with distance because less pixels are covering more vertices in the respective areas.

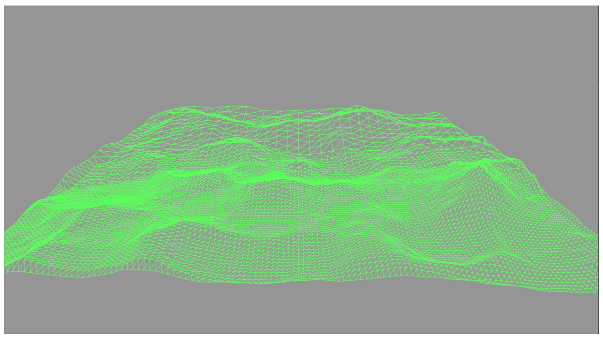

Figure 5. Simple terrain divided into quad-tree structure

Dividing the mesh into multiple levels of detail can achieve the desired behaviour, displaying the highest level of detail on nodes that are closer to the observer and the lowest level of detail on nodes that are far away from the observer (Figure 5).

Using a hierarchical structure, such as a quad-tree, the terrain can be properly displayed on each level of the tree, even when levels are combined, due to the property of the tree, where each list of sub-nodes can be entirely replaced by the super-node they belong to. The requirement in using such a technique is that the terrain can only have square shapes initially. This is because we need to divide each node in 4 similar sub-nodes, each of which occupy 25% of the surface.

Rendering

Due to the tree hierarchical structure, the rendering process can be achieved using a recursive method. Starting from the root node(s), on each node the distance to the observer will be computed. If the distance is less than the node's level of detail threshold, the nodes will be divided into their children and the process will be resumed on each of them. The process will continue until a leaf node is found or a node that has the distance to the observer higher than the threshold level of detail.

The rendering process can be optimized based on the observation that each node contains the same geometry, so all collected nodes to be rendered can be submitted to the graphics card using instanced rendering. This technique will reduce the number of draw calls to 1, but will increase the memory load, due to the necessity of submitting the transformation matrices for each node.

Another optimization can be achieved by clippping against the frustum. Due to the fact that the terrain is now divided into multiple nodes, some of them may find themselves outside the visibile frustum, so there is no need in processing/rendering them. Checking against the visibile frustum can be a difficult problem by itself, but a good aproximation is to compute the tri-dimensional bounding box using the physical position of the node with the minimum and maximum values of the height in the entire world. A better approximation suggests pre-computing the local minimum and maximum for respective patch. This can be achieved using bi-dimensional Range Minimum Queries (RMQ) – a concept described within [13].

CONTINOUS TERRAIN

Once the mesh generation stage is completed and rendered to the screen, some visual artefacts (gaps) can be seen between the terrain nodes. Those gaps appear when two siblings/cousins are found at the border of a level of detail, one of them being divided into 4 children (and increasing the quality with a factor of 2) and one of them being rendered at it's current quality. Two of the children will be placed at the border, so a T-junction is formed, meaning that an extra vertex will be placed at the middle of the edge, adding height information to that edge.

Due to their discontinuity (a node has it's own geometry), that extra piece of information will be the reason for a crack to appear (Figure 6). The crack represents a gap between patches, where the viewer can see the objects that is placed behind the terrain. In figure, the crack has the color gray because that is the clear color of the GPU (the background). Also, in the picture it can be seen that the gap has a triangular shape, due to the fact that an extra vertex is added in between two vertices with similar coordinates, so the extra vertex will have a different height value, generating a triangular shaped discontinuity.

The solution to this problem, as proposed in here, would be to use the tessellation factor to fix the gaps, by telling the graphics card to add an extra vertex to the edges that are found at the border of the level of detail.

Finding edges at the border

Edges that are found at the border of a level of detail share the property of being part in two different level of detail nodes. To find those edges, a search in the quad-tree structure can be made, so that each node will find out if it shares an edge with a neighbour that have a higher level of detail than itself.

Each non-leaf node is divided into 4 sub-nodes. It is fair to assume that every leaf and non-leaf node, except the root nodes, will have at least 2 neighbouring siblings nodes, and 2 neighbouring cousins nodes.

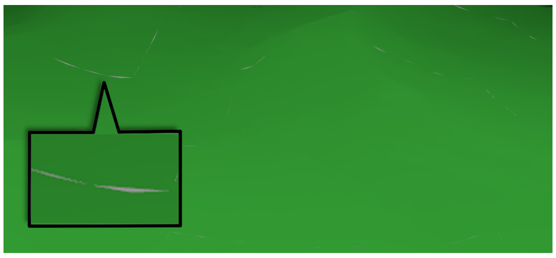

Figure 6. Example of cracks (T-junctions) when height information is applied

The sibling nodes can be queried in a constant time and the cousin nodes require a search in the tree. Each node knows its position in the bi-dimensional space (ignoring the height), so it can compute its neighbour's position, which is on the same level of detail. This neighbour does not necessarily have to exist within the tree. This information can be used to optimize the search, pruning the nodes that does not contain the theoretical computed square. From a list of 4 nodes, only one should contain the desired square, or none at all, meaning that the node cannot be found on this branch.

The search is done when a node is found having the computed position or when it is a leaf containing the geometric square. When no node is found it means that the looking-for node happens to be outside of the terrain. This case can be entirely pruned by checking the theoretical positions to be inside the terrain's border. The total search time complexity is logarithmic, so the overall searching algorithm has a logarithmic time complexity (by running 2 constant time operations and 2 logarithmic time searches).

If a node has all neighbours at the same or higher level of detail, it is rendered using the same tessellation level on all 4 edges. If a node has a neighbour with lower level of detail, it will multiply the tessellation factor on the shared edge with a factor of 2 (Figure 7).

This method reduces the maximum supported tessellation level by the graphics card with a factor of 2. It allows the developer to compute dynamically the tessellation factor on each node, because of the search returning the exact neighbour. A simpler approach, when all nodes use the same, constant, tessellation factor, can be achieved by mathematically computing each neighbour's level of detail, therefore avoiding the tree search.

Shading

To add lighting in the scene, normals have to be computed on each vertex. This is expensive and it should happen in a pre-processing stage. Due to the property of having the same vertex position in the tri-dimensional space (by reading the height value from a height map instead of generating it), the pre-processing can happen only once and store the results in a digital image, called a normal map. This information can be later mapped the same way height is mapped on each vertex. In order to compute normals a Sobel filter [14] can be used, sampling 8 neighbours (from the grid) for each vertex and using their height information. If a node is located at the border of the terrain's grid, will assume his "outside" neighbours as having the height value equal to a constant (usually 0).

This process is highly independent from one vertex to another, so it can be parallelized using a compute shader program. The height map can be passed as a texture to the compute shader and for each pixel the Sobel filter can be applied, generating the normals (Figure 8).

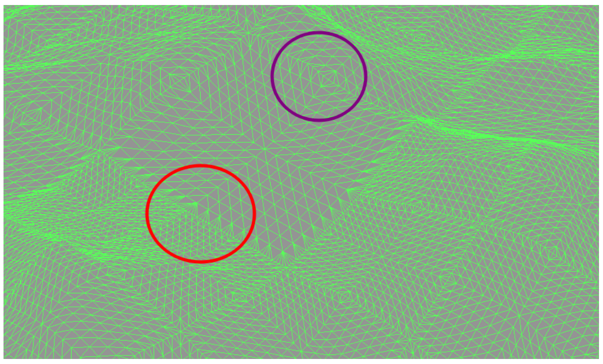

Figure 7. Continuous terrain using tessellation factor

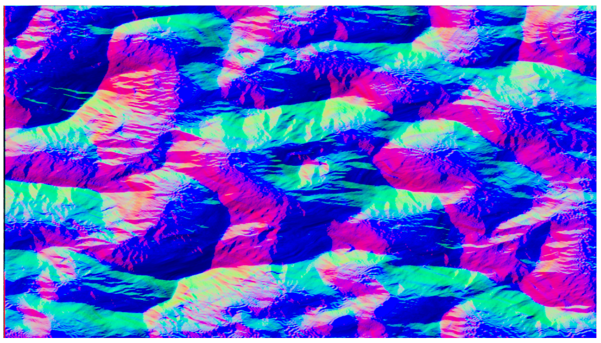

Figure 8. The normal map resulted from the height map in figure 2

Multi-texturing

To decorate the terrain, multiple textures should be combined. Decorating using a single texture is simple, but not realistic. Pre-processing a multiple textures in a single texture in a digital image processor software is hard and not scalable. The solution left is to texture the terrain while generating using a set of textures.

The solution proposed in this paper suggests using a splat map (also named blend map) to define where to apply a texture on the terrain plane. The textures must be blended together and the splat map must be interpolated due to the tessellation step, so the only available choise is to use a floating point digital image. Each image can cover a bi-dimensional area (matching the terrain) and can store for each pixel at most 4 float channels. Each channel can be used to define the percentage of importance of a single texture, so a simple splat map can be used to combine a maximum of 4 textures. Depending on the developer's preferences, those can be either enough or not. In case they are not enough, a texture atlas can be generated. This is a digital image in which multiple smaller digital images are stored in a known grid format. For example, creating a 2x2 grid will allow storing 4 digital images with 4 channels each, so it can define up to 16 textures. Each increment of the grid will decrease the quality of the images (usually with a factor of 2, but it is not a rule) and increase the number of digital images it can store.

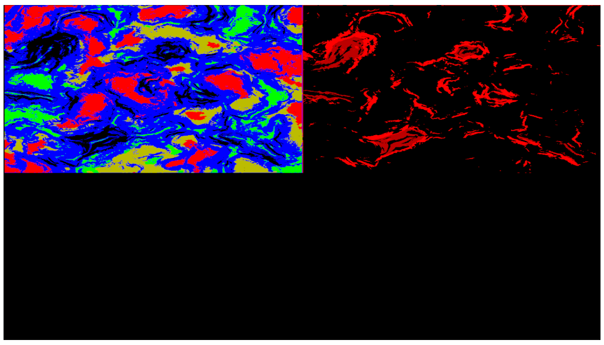

Figure 9. The splat map generated from the height map in figure 2 and normal map in figure 7

Using this approach, the system can use compute shaders to generate based on the height and the slope of the map in a given point the textures that applies in the respective position. The compute shader will input the number of textures, the height map and the normal map and will compute the size of the grid of the atlas (e.g.: using only power of 2 sizes for the grid, for 5 textures the grid size will be 2 x 2 = 4 digital images => 4 x 4 (channels each) textures = 16 textures) using the inputted number of textures. It will read the values from both height map and normal map for a pixel and compute which textures apply in the respective area (which texture is complient with both the height and the slope values). Figure 9 shows the output of this processed, using a set of 5 textures. It can be seen in the figure that it only contains colors in the top half. This is because the digital image has been splitted into a grid of 4 smaller digital images. The example contains only 5 textures, so only 5 color channels are used. In the top right digital image, all 4 color channels are used, each representing the coresponding texture (channel red for texture #1, channel green for texture #2, channel blue for texture #3, channel alpha for exture #4). There's only 5 textures, so in the top right digital image will contain only the red channel, and the bottom left and bottom right parts will contain no information, so will be completely dark.

RESULTS

Combining all the previously described techniques will generate an efficient and ready to use terrain. This terrain can be scaled and improved in quality, based on the developer's preferences, by tweaking some values, such as number of root nodes for the quad-tree structure, tessellation factor, terrain size (scales on XZ-axis or Y-axis), textures and their conditions to apply on the map. The implementation can be scaled based on user's computer configuration, to result in a good performance on low-end devices.

Figure 10 illustrates an example rendering of a terrain sample generated by using the approach described within this paper.

To test the performance of the implementation, some benchmarking was done. The computer on which the tests ran had the following configuration:

- CPU: Intel® Core™ i7-7700HQ CPU @ 2.80GHz x 8
- RAM: 16 GB
- GPU: NVIDIA GeForce GTX 1050/PCIe/SSE2 (Mobile)
- VRAM: 4 GB
- OpenGL 4.1.0, on NVIDIA 465.31
- OS: Ubuntu 20.04
- Compiler: GNU G++ 9.3.0

Table 1 shows the test results. The columns have the following meaning:

- Root nodes: the number of individual patches that are combined to form the final terrain;
- Processed vertices (CPU): the average number of vertices that are processed by the CPU at each frame; the number of vertices in the total scene can be larger, but due to frustum culling, some of them are cut out of the processing stage
- Level of detail: specifies if the level of detail is enabled on the respective tests. If enabled, the CPU handles the update method on each frame, for each node. This carries a O(log) complexity. If disabled, the update method has a constant time complexity;
- Tessellation Level (GPU): specifies the constant level of tessellation used by the GPU to add extra vertices to the patches;
- Frames per second: the resulted metric used to compare the performance of the entire system;

The tests were divided into 4 categories, depending on the root nodes (with 4, 16, 32 and 64) root nodes, each containing 3 individual tests (except the category with 64 root nodes, which contains 4). With these tests, we wanted to demonstrate the performance impact in scenarios when the GPU receives a higher workload (when the tessellation level increases) or when the CPU receives a higher workload (when the level of detail is increased). In the last category, an additional test was made, in which we wanted to describe the huge impact on GPU performance, even with a low tessellation level.

In each category, the first test (and the second in the last category) proves that the performance decreases 3 to 4 times due to an exponential increas in the number of vertices processed on the GPU. The second test (respectively the last) proves that the performance decreaseses 4.5 to 5.5 times with an exponential increase in the number of vertices processed on the CPU and an addition of an extra O(log) method running each frame for each drawn node.

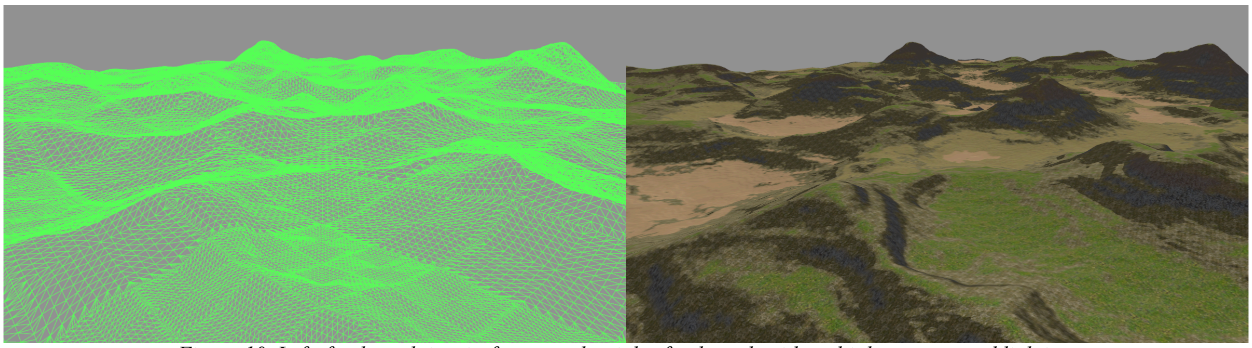

Figure 10. Left: final result in wireframe mode, right: final result with multiple textures and lighting

- *Table 8. Results obtained for the terrain generation system benchmarking.*

Root Nodes	Processed vertices (CPU)	Level of detail	Tessellation Level (GPU)	Frames per second
4	256	Disabled	1	5720.53
4	256	Disabled	32	2351.13
4	4194304	Enabled	32	447.27
16	1024	Disabled	1	5211.73
16	1024	Disabled	32	1368.60
16	16777216	Enabled	32	294.07
32	2048	Disabled	1	5246.13
32	2048	Disabled	32	1369.6
32	33554432	Enabled	32	254.93
64	4096	Disabled	1	4450.53
64	4096	Disabled	32	742.80
64	67108864	Enabled	32	152.13

The tests were made with V-sync disabled, forcing both the CPU and the GPU to run at 100% usage, generating the maximum number of frames.

In real use-cases, it is reasonable to assume that only high quality terrain will be used, so filtering the entries with level of detail enabled and plotting them will return the following plot (Figure 11).

It can be seen (Figure 11) that the number of frames per second decreases linearly with a linear increase in the number of vertices processed. The reason behind their inverse proportionality is that because more vertices have to be processed, each consuming some execution time. The entire frame costs more execution time so the overall performance is affected, returning a decreased number of frames per second.

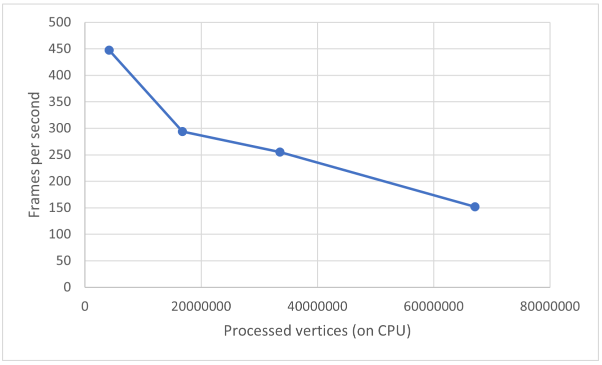

Figure 11. Tabular data plotted, on horizontal axis – processed vertices on CPU side, on vertical axis – frames per second

The value 32 for the tessellation level have been chosen because it is the maximum value the algorithm can use. The maximum value the GPU allows for the tessellation level is 64, but due to the fac that the algorithm requires multiplying the tessellation level on some edges with 2, the maximum allowed value is decreased to 32.

The value 64 for the root nodes has been chosen as the maximum to perform the test, due to the fact that using 128 (the next value) will already cover the entire height map used for this development process, so each level of detail will require to compute the height values by linear interpolating, reaching the maximum precision values (8-bit per channel), generating the so called "step artifacts".

CONCLUSION

In this paper we have described a method that was used to generate the terrain for a custom-built game engine. The terrain generation technique is capable of producing dynamic terrain efficintly, with features such as varying level of detail, shading and multi-texturing. This method is efficient due to its ability to be entirely pre-computed, its cost being reduced to only a tree search, with an overall complexity of O($\log_4$). Using tessellation, the workload is split on both CPU and GPU and the communication between them is reduced to a single draw call.

The results from the tests show that the proposed implementation is capable of producing a upwards of 150 frames per second for geometries of up to 67 million vertices. In our opinion, this indicates a pretty robust implementation, that could potentially see usage even on devices with more modest computing capabilities.

REFERENCES

[1] T. Archer, "Procedurally Generating Terrain," Morningside College, Sioux City, USA, 2011.

[2] D. Maung, Y. Shi and R. Crawfis, "Procedural textures using tilings with Perlin Noise," *Proceedings of the 2012 17th International Conference on Computer Games: AI, Animation, Mobile, Interactive Multimedia, Educational & Serious Games (CGAMES),* p. 60–65, 2012.

[3] A. Mangra, A. Sabou and D. Gorgan, "TSCH algorithm - Terrain synthesis from crude heightmaps," *Revista Romana de Interactiune Om-Calculator,* vol. 9, no. 2, pp. 119-144, 2016.

[4] J. R. Woodward, "The explicit quad tree as a structure for computer graphics," *The Computer Journal,* vol. 25, no. 2, pp. 235-238, 1982.

[5] J. Génevaux, G. E., A. Peytavie, E. Guérin, C. Briquet, F. Grosbellet and B. Benes, "Terrain Modeling from Feature Primitives," *Computer Graphics Forum,* vol. 34, no. 6, 2015.

[6] J. Pelikán and J. Horáček, "Geometry and tessellation," 2012.

[7] A. K. Jakobsen, "Tessellation based terrain rendering, Master thesis," Norwegian University of Science and Technology, Department of Computer and Information Science, 2012.

[8] A. Frasson, T. Engel and C. Pozzer, "Improving Terrain Visualization Through Procedural Generation and Hardware Tessellation," *Proceedings of SBGames 2016,* 2016.

[9] H. Kang, H. Jang, C.-S. Cho and J. Han, "Multi-resolution terrain rendering with GPU tessellation," *The Visual Computer,* vol. 31, no. 4, 2014.

[10] L. Zhang, J. She, J. Tan, B. Wang and Y. Sun, "A Multilevel Terrain Rendering Method Based on Dynamic Stitching Strips," *International Journal of Geo-Information,* vol. 8, no. 6, 2019.

[11] S. Kalem and A. Kourgli, "Irregular Morphing for Real-Time Rendering of Large Terrain," *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences,* pp. 755-762, 2016.

[12] P. Thévenaz, T. Blu and M. Unser, "Image Interpolation and Resampling," in *Handbook of Medical Imaging: Processing and Analysis*, I. N. Bankman., Ed., Academic Press, 2000.

[13] A. Amir, J. Fischer and M. Lewenstein, "Two-Dimensional Range Minimum Queries," in *Proceedings of the Combinatorial Pattern Matching, 18th Annual Symposium, CPM 2007*, 2007.

[14] I. Sobel and G. Feldman., "An Isotropic 3×3 image gradient operator," 1990.