Rendering Large Terrains in Real-Time

Bojan Rupnik

Faculty of Electrical Engineering and Computer Science
University of Maribor
Slovenia

Abstract

This paper presents a method for visualisation of large terrains in real-time. The amount of data when handling large terrains can exceed video memory, the result is a low frame rate. Our method uses a quadtree-based approach to exclude unnecessary data from the rendering process and reduce the level of detail at need. Another aim is to let the GPU perform as much of processing as possible, leaving the CPU available for other tasks.

Keywords: Terrain Rendering, GPU, Quadtree, Real-Time Rendering

1 Introduction

Terrain visualisation is an important element in virtual reality, GIS, computer games, simulations, etc. When dealing with large terrains, our main concern is the amount of available memory. In real-time rendering the data is best stored in the video memory of graphic cards. Large terrains can be made of hundreds of millions of points, making them impossible to fit into the video memory, or with very large terrains, even in working memory.

Along with the memory limitations, another important factor is the graphic processing unit (GPU). Despite the high throughput of modern GPUs, it is still necessary to limit the number of drawn triangles. This is achieved by eliminating parts of the terrain from the rendering process, and by lowering the level of detail (LOD).

There are many different methods addressing the level of detail. In this section we will briefly describe some of them. Most of the methods operate on height fields.

Lindstrom et al. [1] use a two-step algorithm for terrain refinement. The terrain is represented as a height field, that first uses a coarse grained simplification, and later fine grained. At the coarse grained simplification the algorithm groups the height field in blocks of size $(2^n+1)\times(2^n+1)$, that overlap each other at the borders. At the fine grained simplification their algorithm uses edge bisection to merge several smaller triangles into fewer larger ones. The triangles are merged until a screen error tolerance is reached. The simplification and refinement of triangles are view-dependent.

Hoppe [2] uses a view-dependent refinement of progressive meshes to control the level of detail, adapted to terrain rendering. Progressive meshes use operations that split or merge vertices and edges to the triangle mesh, based on the position and orientation of the viewer. Splitting and merging the vertices is performed as long as an acceptable screen error is reached. Another interesting algorithm is ROAM [3] - Real-time Optimally Adapting Meshes. The algorithm is based on edge bisection, which is performed on triangles, instead of vertices. The algorithm uses a binary triangle tree to represent the triangles. The edge bisection is performed by traversing the tree. It uses an error metric to decide whether to simplify the terrain or not.

DeBoer's method [4], geometrical mipmapping, is one that takes the advantage of graphic hardware. The algorithm partitions the terrain into square blocks - geomipmaps. Like in Lindstrom's [1] algorithm, the simplification is performed by removing every second column and every second row. Each geomipmap is assigned an error value, which depends on the removed vertices. If the error of the geomipmap is too large, the level of detail is increased, otherwise it is decreased. Another important aspect of the algorithm is the avoidance of gaps, which is solved by a special triangulation.

In 2004, Losasso and Hoppe introduced geometry clipmaps [5]. Geometry clipmaps cache the terrain in pyramid of nested regular grids, which are centred around the viewer. The grids are stored in video memory and change as the viewer moves. Their method uses a real-time decompression and synthesis of height maps, allowing rendering of large height maps. The authors rendered a 40 GB height map, which was compressed enough to fit into memory. The method was further improved in 2005 in [6], where the authors put most of the work from the CPU to the GPU.

In the next sections we will describe an own method for dealing with large terrains. The method uses similar approaches as described in [1] and [4]. In section 2 we will describe the data structure our method uses, section 3 will cover memory management. Section 4 will describe the rendering process and level of detail selection in detail, and section 5 will reveal the results of our application.

2 Building the Quadtree

To define our terrain we use a a height field. A height field is a two dimensional array of height values. All height val-

ues are equally distanced in both coordinate direction. A simple height field consists only of height values. For rendering purposes, however, this is not sufficient. We need to add information about the exact position in the world space (the space where all objects in the scene are displayed), this is the x and z value (y being the height value) for each vertex - we create a detailed height field. The detailed height field is stored into a vertex buffer.

Along with the detailed height field, we also need to define a rule how to triangulate the height field. We create an index buffer in which each vertex is presented by an index. The index buffer is an array of indices, which specify which vertices in the vertex buffer make triangles.

The vertex and index buffers are used to store the data directly into the video memory. This way the graphic card does not need to wait for the data to be transferred, and the GPU has all the data ready to be rendered.

The problem with large terrains is the memory limit. The video memory is not sufficient to store the whole terrain. Also, graphic cards have a limited size for a single vertex buffer. We need to divide our terrain into smaller segments. Our method uses the quadtree structure to subdivide the terrain.

The quadtree is a data structure where each node has up to four children. Before the subdivision a simple height field is stored in the root. We use simple height fields to minimise memory usage during preprocessing. To create detailed height fields, each node carries information about its position and size in the world space.

With all the necessary data in the root node, we subdivide the height field into four equally-sized segments. Height values that are on the borders of the segments are multiplied and assigned to the new segments (see figure 1). Each new segment becomes a child, which is also given the information about its position and size. The subdivision continues, until we have reasonably-sized height fields, our desired size is under 2^{16} vertices per node. After the subdivision is complete, the height fields exist only in the leaves of the quadtree.

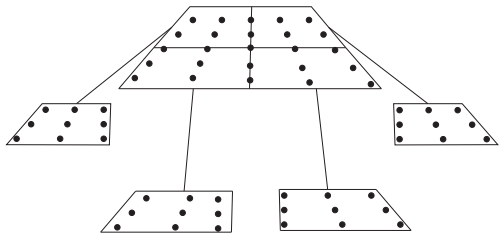

Figure 1: Terrain data subdivided into four equally-sized segments.

Having the position and the size of the nodes, we can calculate the bounding box of each node. The bounding box of the root covers the whole terrain, each child only a quarter of its parent. The leaves cover the smallest height fields.

After completing the subdivision, each leaf contains a simple height field. However, we need a detailed height field to render the nodes. We can calculate the position of each vertex in the height field using the position and size of the node. For lighting purposes we also need to compute normals for each vertex, as well as texture coordinates for texturing. With normals needing 3 values and texture coordinates 2 more, we need up to 8 values to describe each vertex. Using the float data type we need 32 bytes per vertex for a detailed height field. This means eightfold memory consumption compared to the simple height field.

3 Memory management

We want to put as much of the data into the video memory as possible to ensure fast rendering. Using 32 bytes per vertex we can reach the limit of the video memory rather quickly. Table 1 shows memory consumption for different terrain sizes (terrain size is the number of vertices in our height field) . A terrain consisting of about one million points does not pose a problem, but doubling the dimensions we already hit the memory limit of most graphic cards in use today, doubling the dimensions again we exceed it. A terrain consisting of 8192×8192 vertices would exceed the memory an application can address under Windows XP.

1024×1024	32 MB
2048×2048	128 MB
4096×4096	512 MB
8192×8192	2 GB

Table 1: Terrain size and memory consumption.

Working with large terrains, we cannot put all the data into the video memory. Consequently we cannot render the whole terrain in real-time. Instead we choose which nodes can be rendered and which not. Those to be rendered are stored in the video memory as vertex buffers, the rest remains in RAM. If there isn't sufficient RAM available, nodes can also be stored on the hard drive and their height fields removed from memory.

In our memory hierarchy (see figure 2) the quadtree nodes move between video memory and RAM, and if necessary between RAM and the hard drive. Which nodes are loaded/unloaded depends on the camera movement (see figure 3). The camera is positioned in the centre of the inner square. The nodes that are the closest to the camera remain in the video memory (innermost layer) and are rendered. The nodes in the vicinity of the camera (just outside the view) are stored in RAM as detailed height fields or vertex buffers (second layer). The rest of the nodes are stored in RAM as simple height fields (third layer) and if necessary on the hard disk (outer layer).

Memory management is used when the camera leaves a region (a region being space covered by a quadtree leaf). When the camera leaves one region and enters a new one,

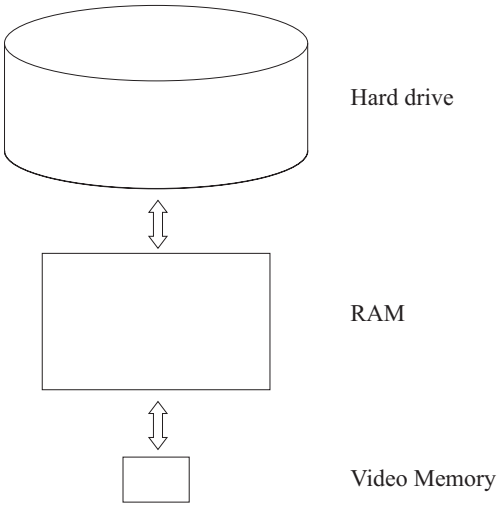

Figure 2: The memory hierarchy.

vertex buffers that are now outside the visible field of the camera (see the innermost layer in figure 3) are released from video memory, but remain in RAM. The free space is then occupied by vertex buffers that are inside the camera view.

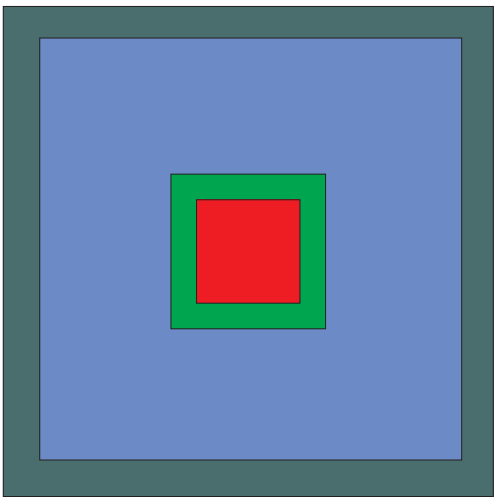

Figure 3: Data stored in the video memory (inner square), RAM (second and third layer), and on the hard disk (outer layer).

4 Rendering and Level of Detail

With the quadtree built, we have created the vertex buffers, however, the vertex buffers must be associated with index buffers. Our subdivision method ensures that each vertex buffer is of the same size. In this way we can use the same index buffer for all vertex buffers. As index buffers are also stored in video memory, we have gained additional space.

Having all the data in the video memory, we can proceed with the rendering process. However the memory is not the only bottleneck. The GPU can only render a certain amount of triangles per frame. We need to keep the number of rendered triangles as low as possible, while the quality of the rendered terrain remains as high as possible. We use different levels of detail, to limit the number of rendered triangles.

Level of Detail

The index buffer defines the triangulation of our height field. Our method uses different index buffers to change level of detail of rendered quadtree nodes. We create 5 different index buffers at 5 levels of detail. The highest level displays a node at full resolution. Each next level displays a node at a quarter resolution of the previous level.

Our level of detail is a small integer (between 0 and 4). We use this number to calculate the step with which the neighbouring vertices are defined: $step = 2^{LOD}$. With LOD set at 0, the step equals 1, resulting in a mesh, where all the vertices are connected with their neighbours. With a LOD of 1 and the step of 2, we ignore every second row an column of the height field, reducing the number of vertices. With each increase of LOD, we reduce the resolution by three quarters. Figure 4 shows the basic idea of changing the levels of detail.

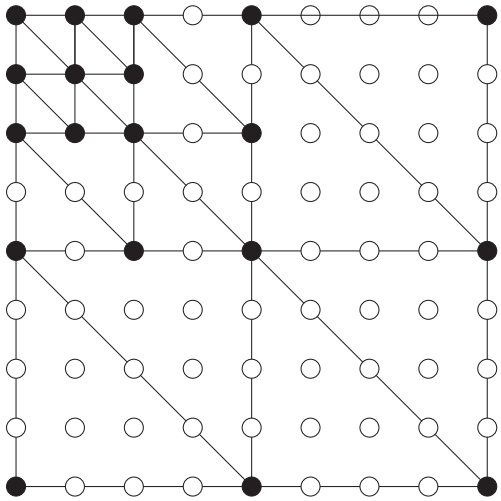

Figure 4: Parts of the terrain at different levels of detail - black vertices form triangles, white vertices are excluded.

Excluding vertices can cause gaps on the borders where terrain parts at different levels of detail meet. To avoid gaps we use a method that triangulates the borders of the height field at the highest level of detail, the rest is triangulated at the desired level of detail. This way we get some additional triangles at the borders, but the decrease of vertices *inside* of the vertex buffer is still significant. Figure 5 shows an example triangulation at the third level of detail. Additional triangles are needed to connect the border triangulation and the LOD triangulation.

The vertex buffers (height fields) do not change in the

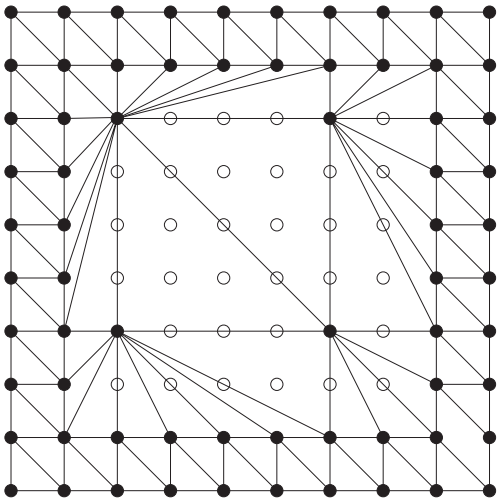

Figure 5: Triangulation of a vertex buffer at third level of detail.

process. We only operate on index buffers to change the level of detail. So we only use a different index buffer to create another triangulation.

As mentioned before, we can use a single index buffer to render all vertex buffers. Now that we are using level of detail, we use up to 5 different index buffers, all of which are created during preprocessing.

Figures 6 and 7 show the same terrain rendered at the highest level of detail (0) and at a lower level (2). The original terrain is the size of 1024×1024. At LOD of 0 the height field is formed from all 1048576 vertices, at LOD of 2, however, only about 70000 vertices are used. Despite the radical decrease in the number of triangles, the quality of the terrain remains relatively good, as figure 7 shows. In table 2 we can see how the level of detail reduces the number of vertices.

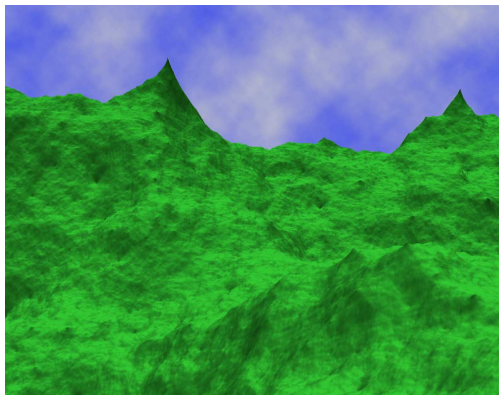

Figure 6: Terrain of the size 1024×1024 vertices, rendered at full resolution.

We choose the level of detail depending on the position of the camera. The further a node is positioned from the camera, the lower its level of detail is. The node in which the camera is positioned is always rendered at full resolution, as are its neighbours. The outer nodes are rendered at a lower level of detail. Nodes rendered at different levels of detail form *rings* around the camera (see figure 8). Rings that are further away from the camera are *thicker* than those close by, the outer ring is rendered at the lowest level of detail and covers all outer nodes. In the figure 9 we can see the quadtree nodes (which can be recognised by thick borders) rendered at different levels of detail, with the camera in the centre of the terrain.

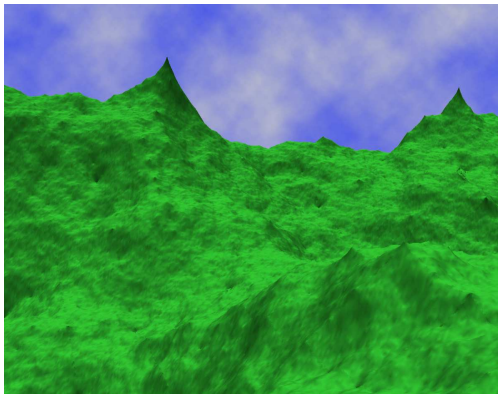

Figure 7: Terrain of the size 1024×1024 vertices, rendered at $1/16$ of the original resolution.

LOD	Resolution
0	100%
1	25%
2	6.25 %
3	1.56%.
4	0.4%.

Table 2: Approximate reduction of the original resolution at different levels of detail.

Frustum Culling

As the viewer (camera) has a limited viewing angle, he can never see the whole terrain at once. He can only see the part that is in his viewing frustum. Therefore, we can exclude the nodes that are outside the viewing frustum from rendering. In average we never need to render more than half of the quadtree. Figure 10 shows visible nodes (orange) and those not rendered (white).

Frustum culling is being processed from the root downwards. Each node is tested, whether it is inside the frustum, or outside. If the node is inside the frustum, all of its children need to be tested as well. As soon as a node appears outside the frustum, it is no longer necessary to test its children, as they are definitely outside the frustum as well. Frustum culling must be performed every time the camera moves or rotates.

We use our quadtree structure to combine level of detail selection and frustum culling. We only need to set the level of detail for the visible nodes, ignoring the rest of the quadtree. Also, we only change the level of detail after the camera has moved from one node to another.

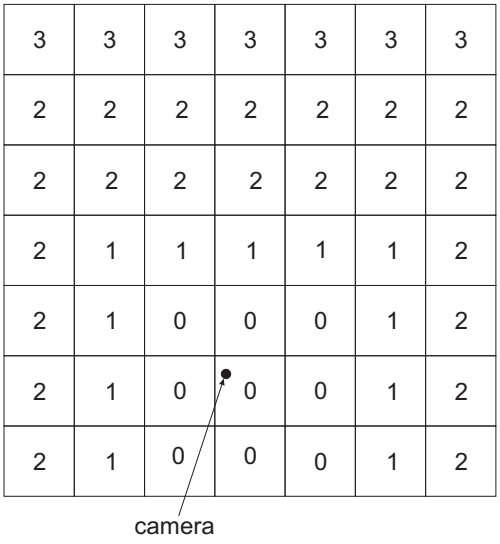

Figure 8: Levels of detail - lower number means higher level.

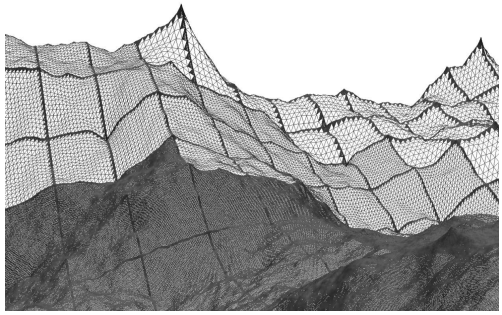

Figure 9: Terrain grid at different levels of detail.

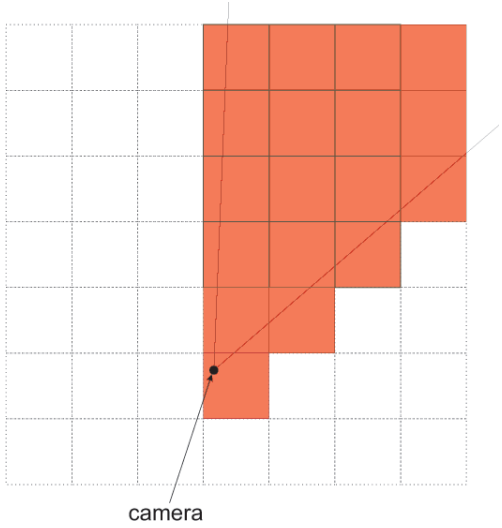

Figure 10: Frustum culling - orange nodes are rendered, the others are excluded.

Collision Detection

To ensure that the camera is always positioned above or on the terrain, we use a simple collision detection method. Knowing the plane coordinates of the camera (x and y), we check whether the z coordinate is above the triangle with the same planar coordinates. We can identify the triangle by traversing the quadtree. We start in the root and check in which child the point (x,y) coordinates are included. We continue the process for each child, until we reach a leaf which contains a detailed height field.

With the terrain information contained in the node, we can calculate which triangle is around the point. When the triangle is identified, we calculate the exact height value z at its (x,y) coordinates. If the camera height is below z, we simply set it at the new value. Depending on the desired functionality, the camera can either fly above the terrain, which means we only correct the height when the camera moves below the terrain, or the camera "walks" over the terrain, in which case we correct the height every time the camera moves.

5 Results

The testing was performed in the following environment:

- Procesor: Intel Core 2 Duo E6700 at 2.66 GHz
- Memory: 2048 MB
- Graphic card: NVidia GeForce 7950 GX2 1024 MB
- Operating system: Windows XP SP2
- Programming language: C++ with DirectX

Lowering the level of detail can play a significant role in improving performance. Table 3 shows frame rates achieved with and without changing the level of detail.

Terrain size	LOD	no LOD
256×256	1300 FPS	795 FPS
512×512	1006 FPS	302 FPS
1024×1024	779 FPS	90 FPS
2048×2048	78 FPS	48 FPS

Table 3: Frame rates of rendering different sized terrains, with and without LOD.

Frustum culling was disabled during the testing, in order to demonstrate the influence of the LOD algorithm. As frustum culling depends entirely on the camera position, it could boost the frame rate to the maximum in the best case (entire terrain outside the viewing frustum). The worst case (entire terrain inside the frustum) would be the same as seen in the table.

Loading larger terrains than 2048×2048 vertices results in the same frame rates, as our method removes terrain parts from the video memory and the rendering process.

With large terrains the frame rate decreases slightly when the camera leaves a region. This is because new height fields must be computed and loaded into video memory. Figures 11 and 12 show the output of our application.

6 Summary and future work

Our implementation takes advantage of the modern hardware, especially the graphic card. By transferring most of the work to the GPU, we leave CPU relatively unburdened and ready to perform other needed tasks (artificial intelligence, physics, etc.).

Unlike many others, our algorithm is not limited to dimensions of form 2^n or $2^n + 1$, it can deal with any dimensions. It deals with large terrains by simply rendering only the needed part of the terrain, and storing the rest in the memory when not needed.

For now, its disadvantage is the visible change of the geometry during camera movement. When the camera movement causes a change of the level of detail, we get pop-up effects, meaning that parts of the terrain suddenly become more detailed or less detailed than they were before. In future this will be solved with geomorphing using vertex shaders. Another feature will be the possibility of adding objects to the terrain, such as trees, rivers, houses, etc.

References

[1] 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", 1996. Proceedings of the ACM SIGGRAPH Conference on Computer Graphics, pages 109–117, 1996.

[2] Hugues Hoppe, "View-dependent refinement of progressive meshes," in Proceedings of SIGGRAPH 97, Turner Whitted, Ed., Los Angeles, California, Aug. 1997, Computer Graphics Proceedings, Annual Conference Series, pp. 189-198, ACM Press.

[3] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: realtime optimally adapting meshes. In IEEE Visualization, pages 81-88, 1997.

[4] Willem H. de Boer, E-mersion Project, http://www.connectii.net/emersion, World Wide Web, October 2000.

[5] Losasso, Frank, and Hugues Hoppe. 2004. Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)23(3), pp. 769-776

[6] A. Asirvatham and Hugues Hoppe. 2005. Terrain Rendering Using GPU-Based Geometry Clipmaps. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional.

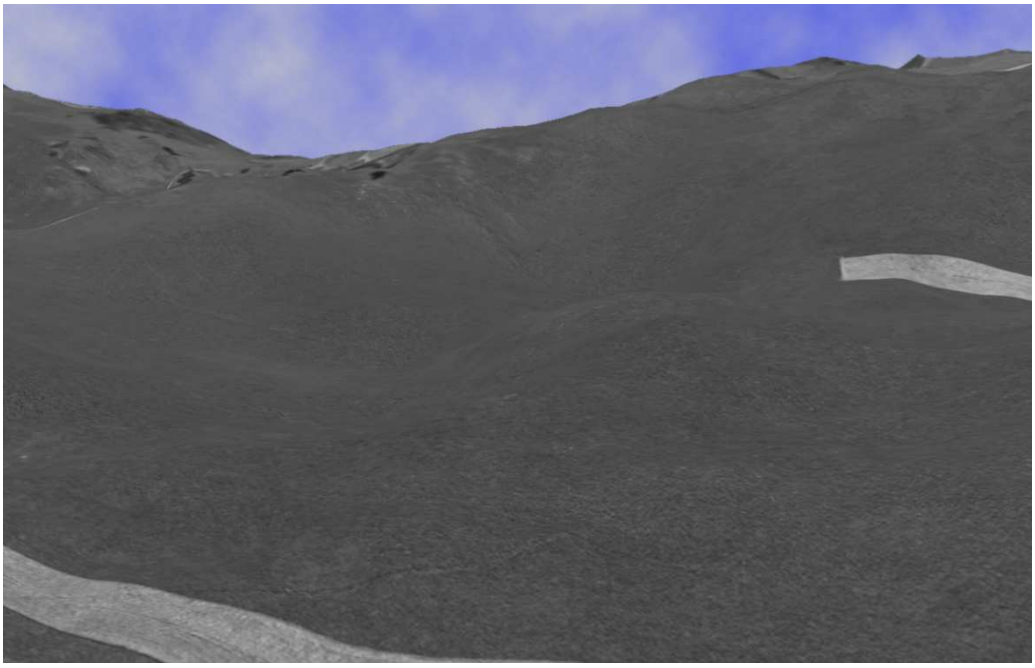

Figure 11: Terrain of the size 1000×1000 vertices, rendered with a texture.

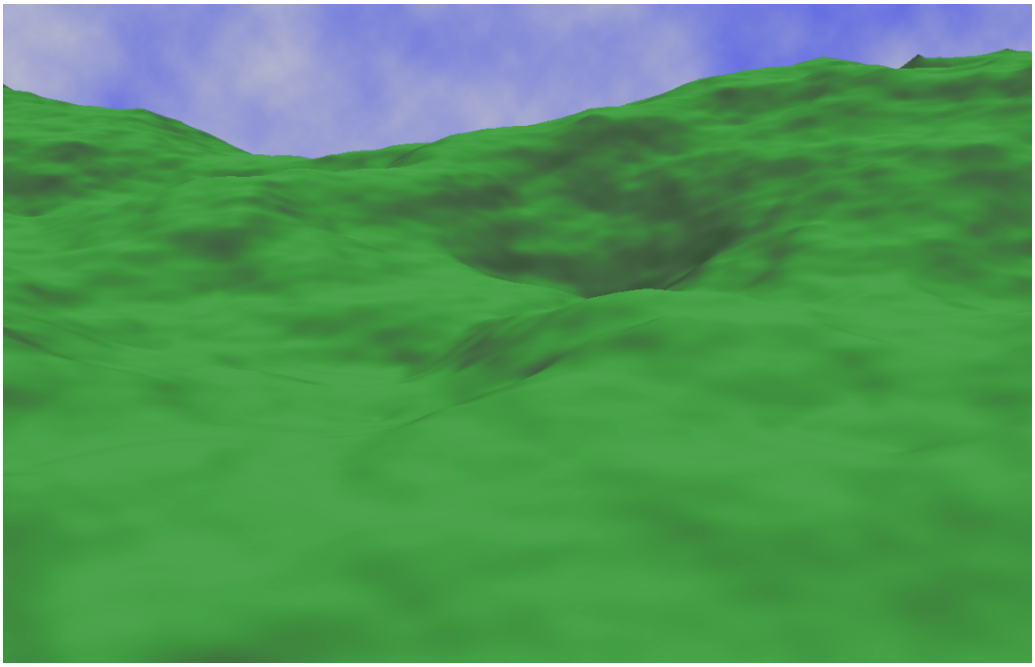

Figure 12: Terrain of the size 1000×1000 vertices, rendered with smooth lighting.