Practical Algorithm for Unlimited Scale Terrain Rendering

JAN VANEK, BRUNO JEZEK
Faculty of Military Health Sciences
University of Defence
Trebešska 1575, 50001 Hradec Kralove
CZECH REPUBLIC
vanek.conf@gmail.com http://www.vojenskaskola.cz/school/ud/fmhs/Pages/default.aspx

Abstract: - The paper presents an algorithm for rendering digital terrain models of unlimited scale and detail. The algorithm is straightforward and simple to implement. It integrates easily with rendering of other geographical data. It has minimal CPU requirements as it harnesses the power of modern graphics processing units. It has implicit triangle budget and provides stable frame rates.

Key-Words: Terrain Rendering, GPU, Triangulated Network, MIP-mapping, Level of Detail

1 Introduction
Terrain visualization is a task taking place in all geographic information systems. Very large datasets are often processed and visualized in real-time. Although the performance of hardware continually increases, it is necessary to optimize the form and the amount of terrain information in order to obtain sufficient frame rates during visualization. New developments in graphical hardware and its versatility and programmability bring new opportunities and needs to redesign rendering algorithms. This paper proposes an algorithm designed to take advantage of programmable graphics hardware, to be simple and easy to implement, and to handle unlimited terrain detail and scale.

2 Existing methods
The first terrain rendering algorithms, such as [1], [2] or [3], were specifically optimized for particular hardware and were not based on polygonal models. Non-polygonal approaches were used by [4], [5] or [6] even rather recently, though with a limited success. The 1990s marked a significant spread of triangle and z-buffer based rendering pipeline in practical virtual reality applications. At the same time, hardware acceleration of the pipeline by GPUs (graphics processing units) became commercially successful. Because of the limited memory resources of the GPUs and relatively low difference in performance between CPUs and GPUs very sophisticated algorithms and data structures were designed to optimize the rendered triangulated network. In the recent years, however, the GPUs have reached computational performance several orders of magnitude greater than that of CPUs, so the algorithms must mainly feed the GPU with data and use the CPU as little as possible. The triangulated network does not have to be optimal as long as it is sufficiently dense [7]. Some authors (e.g. [8]) also stress the importance of keeping rendering time budget, in which respect complex algorithms are contra productive.

In general, all algorithms attempt to solve the problem of triangle count reduction by rendering a triangulated network with continuously changing level of detail so that all triangles have a similar size in perspective projection.

The most efficient triangulated network is generated by algorithms such as [9], [10], [11] or [12], which are based on triangulated irregular networks. However, these algorithms are also the most demanding in terms of data structure and manipulation complexity, which does not correspond well with the hardware accelerated rendering paradigm.

Some improvement is introduced by algorithms proposed in [13], [14], [15] or [16]. These use triangulated regular networks and simpler data arrangements, but they still work on a per-triangle basis, which is CPU demanding and requires frequent GPU memory updates.

The methods described in [7], [17], [18], [19], [20] and [21] provide further improvement by splitting terrain in form of a triangulated regular elevation grid into tiles, thus handling batches of compact data and even allowing for out-of-core rendering.
Some authors tackle the problem using quite different approaches. [5] and [22] combine classic techniques with image based rendering. [23] renders the terrain by projecting a single triangulated grid onto it from the viewpoint.

With few exceptions, such as [8] or [24], authors do not take rendering of other geographical data into account, although they are with terrain data closely linked. Complicated methods and especially complex data structures make an integration of an algorithm to a larger system of geographical data visualization difficult, impractical and costly. Simple algorithms not only tie in better, they can actually also perform better, especially on the current generation of hardware that allows for decoupling of triangulation and terrain data.

3 Proposed Method
Most of the currently used and hardware acceleration-friendly algorithms, such as [20] or [21], display digital terrain models in form of regular elevation grid triangulation split into tiles, which change their level of detail in correspondence with the viewpoint movement.

As the viewpoint moves, the level of detail of the tiles increases in the movement direction and decreases in the other. For example, as the camera moves from the tile 13 in Fig.1 to the tile 8, the level of detail increases in tiles 3, 7 and 9 and decreases in tiles 12, 14 and 18. If the terrain data is sampled from a texture based on planar vertex coordinates, the very same effect can be achieved by translating all the vertices of all tiles by the tile height. So the terrain can be rendered using a fixed triangulated network with the following properties:

- It moves with the viewpoint
- Its connectivity is fixed, its detail decreases with increasing distance from the viewpoint
- The elevation of its vertices is modified by sampling from a texture

This premise is the starting point of the proposed method.

The level of detail of a grid split into tiles does not decrease gradually and areas of the same distance from the viewpoint but different levels of detail exist in the final triangulation. In an ideal case all visible triangles should have the same screen-space size. Algorithms that optimize the level of detail based on screen-space metrics must modify vertex connectivity when the view changes, which is unsuitable for hardware accelerated rendering and creates a high latency when a user looks around. The proposed algorithm generates a triangulated network that, to a degree, maintains similar screen-space size of triangles, but it does not use screen-space metrics. The network is absolutely fixed in terms of GPU input data and thus guarantees stable frame rate even during viewpoint change. It is continuous without any cracks or T-vertices.

During rendering, the triangulated network processed by a GPU is transformed from the input data into triangles rasterized on screen. The proposed algorithm consists of two parts. The first generates the input data and is executed on CPU once before rendering loop begins. The second transforms the input data into terrain coordinates and is executed as a vertex shader on GPU for every rendered frame.

Vertices of the proposed triangulated network are positioned above concentric circles with the center in the orthogonal projection of viewpoint into the \( xy \) plane of the terrain coordinate system and their coordinates on the \( z \)-axis are sampled from an elevation grid based on the \([x, y]\) coordinates. See Fig.2.

A triangulated network is passed to a GPU as a vertex and index array. The vertex data serve as the
input to the transformation into terrain coordinates and in the proposed triangulated network vertices consist of polar coordinates: normalized radius $r$ of the circle the vertex lies on and angle $\varphi$ between the ray from the center to the vertex and the positive x-axis. Fig.3 on the left shows the network with five circles in the terrain coordinates, on the right there is the same network plotted in the polar coordinates of the input data.

With the increasing radius the number of vertices on a circle grows by 6 starting at the hexagon of the circle 0, so the number of vertices at the circle $i$ is $6i, i = 1, \ldots, n$, where $n$ is the number of all circles of the network and specifies its overall detail. Then the total number of vertices is obviously $1 + \sum_{i=1}^{n} 6i = 1 + 3n(n + 1)$.

Fig.3: Input vertex coordinates

The following pseudo code creates the vertex array for the given number of circles $n$.

```pseudo
def Create vertices (n):
    add vertex [0, 0]
    for i from 1 to n:
        for j from 0 to 6i - 1:
            $r = \frac{i}{2}$
            $\varphi = \frac{2\pi j}{6i}$
            add vertex [r; \varphi]
        end for
    end for
end procedure
```

The connectivity of the network is described by an index array, the most efficient interpretation of which is a triangle strip. The proposed triangulated network can be expressed as a single triangle strip, though at the cost of adding triangles degenerated to a line. The annulus between the circles $i$ and $i + 1$ is covered by six triangle strips, apiece connecting $i + 1$ vertices of the circle $i$ with $i + 2$ vertices of the circle $i + 1$. Each of the first five strips is appended a degenerated triangle connecting it to the next. The last strip is linked to the first strip of the next annulus by three degenerated triangles.

The following pseudo code generates an index array this way for the given number of circles $n$. The variable $m$, respectively $m_1$, denotes the first index of the currently processed annulus at the circle $i$, respectively circle $i + 1$. The variable $p$, respectively $p_1$, denotes the first index of the currently processed strip of the current annulus at the circle $i$, respectively circle $i + 1$. See Fig.4, where an example of a current annulus is in full line and of a current strip is dashed.

```pseudo
def Create indices (n):
    for i from 1 to 6:
        add index m
        add index 0
    end for
    add index i
    add index 1
    p = 1
    p_1 = 7
    for i from 2 to n - 1:
        m = p
        m_1 = p_1
        add index m_1
        for j from 0 to 5:
            for k from 0 to i - 1:
                add index p + k
                add index p_1 + k + 1
            end for
            p = p + i - 1
            p_1 = p_1 + i
        end for
        remove the last 2 indices
        add index m
        add index m_1
        add index m_1
    end for
    remove the last index
end procedure
```

The number of indices in an index array created this way is $13 + \sum_{i=2}^{n} (12i + 2) = 6n^2 - 4n - 3$, and the number of triangles defined by it is two less, as is always the case with a triangle strip.
Vertex coordinates in the terrain coordinate system are derived from the \([r; \varphi]\) input data in a vertex shader. The \(x\) and \(y\) coordinates calculated from a parametric equation of a circle, the \(z\) coordinate is sampled from an elevation grid stored in a texture using the calculated \(x\) and \(y\) coordinates. The circle radii \(r\) are normalized in the input data to values equidistantly ranging from 0 for the central vertex to 1 for outer-most circle. In the vertex shader they are transformed to the interval \((0; r_{\text{max}})\), where \(r_{\text{max}}\) is the parameter that defines visibility range. If the radii were equidistant in the interval too, the level of detail would be the same throughout the triangulated network. The proposed algorithm redistributes the radii so that the circles are equidistant when projected on a spherical cap corresponding to the viewpoint and visibility range. Half of the central angle of the cone above the spherical cap is \(\alpha_{\text{max}} = \arctan \frac{r_{\text{max}}}{P_z}\), where \(P\) is the viewpoint and \(P_z\) is its \(z\) coordinate in the terrain coordinate system. The radius \(r_i\) of the circle \(i\), above which is the currently shaded vertex, is then given as \(r_i = P_z \tan(\alpha_{\text{max}})\), see Fig.5. Finally, the \(x\) and \(y\) coordinates of the vertex in the terrain coordinate system are \(x = P_x + r_i \cos \varphi\) and \(y = P_y + r_i \sin \varphi\).

Fig.4: First indices of an annulus and a strip

Fig.5: Radius computation in vertex shader

Fig.6: Circle distribution for different viewpoints

Fig.7: Terrain undulation with viewpoint change

The proposed algorithm mitigates the problem to an acceptable level by additional sampling of the elevation texture per vertex and by the use of MIP-mapping.

For the vertex at the circle \(i\) of radius \(r_i\) the elevation is sampled between the radii \(r_{i-1}\) and \(r_{i+1}\). The number of samples \(s\) is calculated so that it represents how many times the sampled range is larger than the radius of the first circle. Therefore \(s = \left\lceil \frac{r_{i+1} - r_{i-1}}{P_z \tan(\alpha_{\text{max}})} \right\rceil\). The number of samples is...
further capped by a predefined limit so that the computational load on the GPU remains under control. The final vertex z coordinate is an average of the samples.

MIP-mapping is a texturing technique used in hardware accelerated rendering for anti-aliasing and computational optimizations. A texture is extended with a pyramid of textures, each of which is averaged from the previous at half its size. During sampling the pyramid level can be specified by a coefficient \( u \) from 0 for the base texture in full resolution to \( k = \log_2 \rho_{\text{max}} \), where \( \rho_{\text{max}} \) is the texture resolution, i.e. the maximum of width and height. If \( u \) is not an integer the GPU calculates a weighted average of the two closest levels. The algorithm determines the coefficient \( u \) so that the number of texture samples per the interval between the circle radii \( r_1 \) and \( r_{l-1} \) is the same as per the radius of the first circle, that is

\[
u = \log_2 \frac{r_{l-1}}{P_z \tan\left(\frac{\pi}{\rho_{\text{max}}}\right)}.
\]

Because of the additional sampling and MIP-mapping, the level of detail decreases rapidly with the increasing distance from the viewpoint. The algorithm is further modified not to excessively accent the detail close to the viewpoint at the expense of the distant one by using \( f(P_z) \) instead of \( P_z \) in the calculations. By the choice of the function \( f \) it is possible to reach an adequate compromise between the level of detail in the proximity of the viewpoint and the rate of its decline, as can be seen in Fig.8, where the upper image shows a terrain with \( f(P_z) = P_z \) and the lower one with \( f(P_z) = \max\{5P_z, 50\} \).

![Fig.8: Level of detail decrease rate](image)

4 Conclusion

The proposed algorithm has been implemented and tested both on artificial terrain models and on real datasets. The artificial terrains were rendered with detail and scale limited only by the floating point precision of viewpoint coordinates and z-buffer. Scale-to-detail ratios of six orders of magnitude were achieved easily. The algorithm was also applied to spherical terrain models with only a single additional transformation in vertex shader. The simplicity of the algorithm allowed for its effortless integration with different texturing techniques to render real terrains of hundreds of thousands square kilometers in scale and square meters in detail. The biggest drawback of the algorithm is the terrain undulation with viewpoint motion, which can, however, be managed.

References:


