TERRAIN RENDERING ALGORITHM PERFORMANCE ANALYSIS

Lukas Polok, Radek Barton, Peter Chudy, Premysl Krsek, Pavel Smrz, Dittrich Petr
Brno University of Technology, Faculty of Information Technology,
IT4Innovations Centre of Excellence, Bozetechova 2, 61266 Brno, Czech Republic

Abstract

Nowadays, the flight guidance equipment supplies practically all the information, required for aircraft navigation. Pseudo-realistic terrain visualization is undoubtedly an important part of this information. Although modern graphics processors are able to render realistic terrain at interactive frame rates, in some applications, it is beneficial to use low-power graphics hardware, perhaps from weight or power supply capacity restrictions. These low-power graphics processors typically manifest much lower computational power than conventional hardware, severely limiting the capability of terrain visualization. A novel method for caching terrain tiles is presented in this paper, enabling faster and more detailed terrain rendering, using lighter devices that consume less power. The main focus was on memory and time efficiency on common low-power graphics hardware. The terrain rendering algorithm being employed in our implementation is derived from the seamless patches algorithm. Different aspects of terrain tile swapping were examined in order to devise a simple hardware metric. An efficient tile caching approach was developed, based on this hardware metric, and its performance was evaluated.

Introduction

Present advances in mass-production low-power consumption graphics hardware allow us to build low-cost hardware-software codesign solutions suitable for small aircraft on-board instrumentation systems. Our modular 'smart' autopilot system includes primary flight display with high-performance instrument indicators and a terrain visualization rendering framework running on embedded commodity graphics hardware with OpenGL ES 2.0 support. This paper will describe impact of certain aspects and parameters of our terrain visualization algorithm on overall rendering performance and visual quality.

Pseudo-realistic terrain visualization is an important part of flight state information flow coming to pilot's perception system during instrument-guided aircraft control. Its implementation require some optimizations to be able to run on the low-power hardware. A special care has to be taken to avoid time domain artifacts which could be introduced by such optimizations and which can be particularly disturbing. Such artifacts can be caused by switching the level of detail of the terrain surface, yielding characteristic popping effects. A different kind of artifact is caused by loading terrain tiles as the observer moves across the terrain. Because the tiles are usually loaded from an external storage, the access latencies are high and can ultimately lead to stuttering.

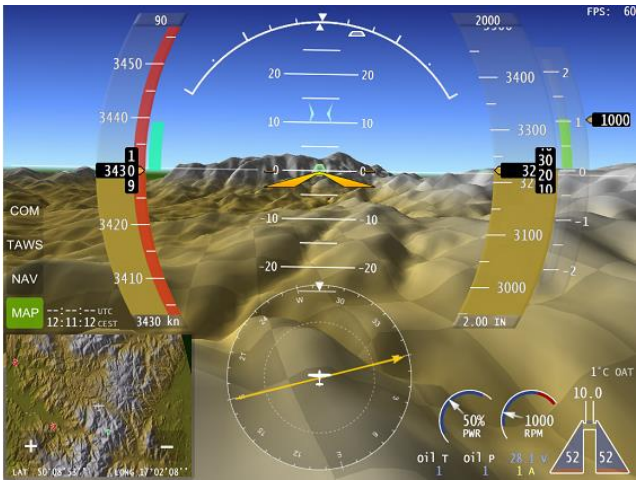

Figure 1. A screenshot from our terrain visualization, running on low-power hardware

A common characteristic of the low-power hardware is its highly optimized graphics processing unit (GPU). The main limitation of those GPUs, from the terrain rendering perspective, is the absence of unified shader architecture. That is especially advantageous for the GPU manufacturers, as vertex shaders are usually much simpler than fragment shaders. Vertex shader units therefore require smaller area on chip and draw fewer current than comparable fragment shader units. This

simplification often involves the absence of interconnect between the texturing hardware and vertex shader units. Vertex texturing is broadly supported in commodity GPUs and most of the common terrain algorithm implementations utilize this functionality at their core.

Another characteristic of the low-power low-cost devices is limited size of the main memory. Detailed large-scale height map datasets required for the practical function of the primary flight display will not fit in the main memory and therefore need to be frequently swapped from an external storage. Due to the graphic hardware limitations, some common algorithms such as clip maps are not optimal. A new algorithm is needed.

A novel method for caching terrain tiles is presented in this paper. The main focus was on memory and time efficiency on common low-power graphics hardware. The terrain rendering algorithm being employed in our implementation is derived from the seamless patches algorithm. Different aspects of terrain tile swapping were examined in order to devise a simple hardware metric. Several approaches were developed, based on this hardware metric, and their performance was compared and evaluated.

Previous Work

Recent algorithms rely on batched rendering of a geometry precomputed to vertex buffers. This minimizes a CPU-to-GPU data transfer bottleneck but also lowers an adaptivity of a generated shape. A main question it must be dealt with is how to optimally organize geometry clusters and the algorithms mostly differ in an used shape of the clusters and in a way how they are stitched together.

A first representative of tiled methods is Geometrical Mip-Mapping by de Boer [1]. The whole terrain is split into regular grid tiles of a same geometric size and a certain number of downsampled versions is made for each of them. A quad-tree of bounding boxes is constructed to perform a fast view-frustum culling during run-time. Leafs of the tree correspond to the tiles and point to them. A mesh continuity is achieved by rendering an interior of the tile first and then by stitching borders using triangle strips.

The Chunked LOD this algorithm [2] uses a quad-tree of tiles (chunks) too but in a different way than Geometrical Mip-Mapping [1]. A root of this tree contains a square tile with an arbitrary terrain shape simplified to some error value. Its children are four square tiles each covering a quarter of the terrain simplified with a half of the error value of the parent and so on. The tree is traversed from a top to a bottom and each time a projected error of the tile satisfies a pixel error threshold, the descending stops and the tile is rendered.

Pouderoux et al. [3] published an elaboration of a Geometrical Mip-Mapping technique [1], called Strip Masks, that more deeply explains a caching strategy for a tile streaming at different levels of detail. Another contribution is that they propose to visually mend gaps between tiles with textured planes placed under a terrain's surface projected using a view parameters.

Batched Dynamic Adaptive Meshes (BDAM) [4, 5, 6, 7] are an approach employing a triangular tile topology. A main idea exploits a property of RTINs that a triangle can be connected to another triangle from a same level, or a one level higher triangle through its hypotenuse or a one level lower triangle through one of its catheti. A terrain triangulation can be then represented as a set of triangular tiles from a binary tree hierarchy simplified to some error measure which is shared between possibly neighboring triangles from adjacent tree levels at tile borders and which is smoothly grading inside the tile.

An algorithm called Seamless Patches for GPU-Based Terrain Rendering [8] is a hybrid solution that integrate benefits of triangular tiles, which is a better local adaptivity and easier ways to avoid cracks at borders, and rectangular tiles that are more suitable for hardware optimizations. A rectangular patch is split by its diagonals into four triangular tiles. A hierarchy of this patches similar to one as in a Chunked LoD [2] algorithm is constructed but a branching factor is determined by a number of possible discrete level of details of each triangular tile to ensure a seamless connectivity. The tree is traversed from a root where an error metric is evaluated. If it is not detailed enough, procedure descends to its children and so on. If, on the other hand, the patch as a whole has a suitable resolution according to view parameters, an exact resolution for

each of its border edges is determined and an appropriate levels of detail of the triangular tiles are selected. A proper continuity is guaranteed by diagonal strips that fill holes between the triangular tiles. All, the tile and a strip geometry, is preprocessed into vertex buffers without heights which are then added using a vertex shader during rendering. This algorithm was chosen as a basis for our terrain visualization implementation.

Hardware Metrics

In order to focus the optimization of any hardware accelerated algorithm, it is necessary to devise some kind of hardware metric that could be used to highlight the bottlenecks and compare results.

In the current low-power hardware, there is mostly no option to read height data from a texture in vertex shader, the way the modern terrain rendering algorithms do. Broadcoms graphics chip on the Raspberry PI computer is a notable exception to that, but unfortunately, no samples of that hardware were available at the time of writing this paper. Therefore, all the data for terrain rendering needs to be available in vertex buffer(s) prior to actual rendering.

To render a terrain, one typically stores the necessary data in a few vertex buffers. One of the vertex buffers would hold vertices for all possible tile resolutions or configurations. There would be also a second buffer, containing indices to render the vertex data as a triangle strip or just bare triangles. And at last, there would need to be a buffer with height data per every vertex. Note that this buffer needs to contain unique data per every tile being displayed, unlike the afore-mentioned two buffers, which can be used to display practically any terrain tile. Keeping this buffer separate is a logical choice, as one only needs to update the height coordinate, and the CPU-GPU traffic is therefore reduced, compared to updating full 3D coordinates of tile vertices.

There are several aspects of terrain rendering on low-power GPU, under above mentioned conditions. These include rendering from more than one vertex buffer, rendering from more than one index buffer, updating shader uniform parameters for individual tile transformation, setting constant vertex attribute for individual tile transformation, setting a different vertex attribute pointer to source data from arbitrary portion of vertex buffer, and last but not least, the size of the batch in which the tiles are updated. All these are described in subsequent paragraphs.

Rendering from more than one vertex buffer may come in handy when having multiple sets of vertex data. The scenarios may include having separate vertex buffers for different tile resolution or topology, or having a tile allocation scheme, requiring the use of separate buffers. Also, one might want to consider sourcing data from multiple vertex buffers to reduce the number of index buffers required (the index buffers for tiles can be typically shared among the tiles of the same resolution, and since the indices cover the same range of vertices, typically starting with zero).

Rendering from more than one index buffer is typically not a strong requirement, as the rendering API functions enable simple selection of range of data in the buffer, being used as indices in order to render some geometry. In particular, the glDrawElements() functions enables selection of byte offset in the buffer, and of the number of indices being read. On the other hand, it might be an architectural requirement of the terrain rendering subsystem to keep the buffers separate.

Because of sharing position data (except the height) among the tiles (or at least among the tiles of the same resolution), the tile transformation needs to be somehow passed to the GPU. This may include tile position, rotation and possibly scale. One of the ways to pass this information is to employ uniform shader parameters, that would be read and interpreted by the vertex shader.

The other way of accomplishing the same goal might be to use constant vertex attributes. Vertex attributes are usually sourced from arrays, stored in vertex buffer objects. On the other hand, it is possible to set the value of any vertex attribute to constant value for the use in the next draw calls. The only difference between constant vertex attribute and uniform shader parameter is that the vertex attribute is a global property and uniform is a property associated with a program object.

In order to reuse index data, it might be necessary to change the physical address from where the vertex data are sourced, inside a vertex buffer object. This is done by a call to the glVertexAttribPointer() function, entering a format of the vertex attribute, and offset in the buffer. This is a

complementary method to using multiple vertex buffers.

Finally, due to severe limitations of available memory, so typical for low-power devices (such devices can share e.g. 256 MB of memory between GPU, the application and indeed the operating system), there needs to be an efficient caching scheme for replacing the tiles in memory as they are needed. Since the GPU and the CPU needs to communicate the data, be it using a bus and direct memory access in high end systems, or possibly a plain memory copy in low-power systems, there is always some latency overhead before the transfer is initiated. Therefore, it might be a good strategy to transfer tile data in bigger batches, rather than transferring individual tiles separately.

To obtain the above mentioned hardware metrics, a simple benchmark was devised. Since most of the times can be expected to be of the magnitude of tenths of milliseconds, it is not possible to use a direct approach of measuring the time. Instead, a minimalist real-world terrain rendering loop was devised, initially rendering the terrain by a single draw call, all the tiles being stored in one long vertex buffer and all the indices in one long index buffer. This gives baseline frame time, which is averaged over many frames. In the actual benchmark, the rendering loop ran for 10 seconds while counting the frames, and as the time budget was exceeded, the resulting frame time is simply a ratio of time elapsed to number of rendered times. It is important to make sure that the frame time is limited by the vertical sync to 1/60th of a second, by issuing sufficient workload to keep the GPU busy for longer time. The results were also validated as not being rasterizer-bound by setting the viewport from its original size (800 x 600) to 2 x 2 pixels, and comparing the results. Then, to measure any of the above-mentioned metrics, the rendering loop is modified so as to incorporate a method of implementing an operation, associated with such metric, and the resulting average frame time is subtracted from the baseline frame time. For example, to measure time required to set shader uniform parameter, one would modify the loop to only use the first tile vertex data, and set translation parameter prior to rendering each tile. As the differences are quite small, it is also important to establish a significance level to make sure that the measurement is actually not only noise, caused by frame rate fluctuation. To do that, the baseline benchmark is ran many times, and the standard deviation is used as a level below the measurements are considered insignificant. This scheme is very simple to implement, gives results that are easy to interpret in a real-world situation, and is also quite robust. The results, obtained for the NVIDIA Tegra 2 GPU, running on a module with ARM4I CPU with 256 MB of memory, can be seen in table 1. The terrain for the benchmark consisted of 1024 tiles, arranged in a 32 x 32 matrix, each tile having 64 x 64 vertices.

Table 1. Hardware metrics benchmark results

Operation		**Time [mS]**
Vertex buffer swap		3.52
Index buffer swap		4.01
Shader reconfig		2.90
Vertex attrib reconfig		7.54
Vertex attrib pointer reconfig		23.38
Tile upload (batch size)	1	185.78
	2	152.96
	4	138.12
	8	159.77
	16	167.38
	32	167.63
	64	156.29
	128	155.50
	256	154.46
	512	153.44
	1024	150.62
Standard deviation		0.314375

As can be seen, all the operations are well above the noise level, given by standard deviation, and can be considered valid. Also note that the results are per rendering 1024 batches (tiles), and are therefore scaled thousand times (and should be in nS instead of mS). But in this context, the relative magnitudes are important, rather than absolute values.

The first striking fact is that the tile upload is by far the most expensive operation (25% of tiles were updated each frame, in batches of size given in the middle column in table 1). Also, the upload can be made faster by batching data for more tiles in a single transfer. This shall be the first priority in the proposed algorithm.

Also very interesting is that setting shader uniform parameter is actually faster than setting the

value of constant vertex attribute. This is very easy to incorporate in any algorithm, without the need for greater changes.

Another fact is that the rendering pipeline is stalled by changing the source of vertex attribute data, and it is much faster to swap index buffer instead, in case there is one with indices pointing to the right place in the vertex buffer. It would seem it might be faster to keep the data in multiple vertex buffer as time for vertex buffer swap is much smaller, but after swapping the vertex buffer, vertex attribute pointers need to be reconfigured as well (the time for that is not included), making it even slower. Also, it is impossible to load batches of data to multiple vertex buffers at once.

System architecture

The results from the benchmarks led to a simple decision about the optimal solution to the tile caching problem. The rendering system was written in the C++ language and comprises of several layers. The bottom-most retrieval layer takes care about reading the tiles from the ASTER GDEM dataset [9], stored on a flash drive. Above that, there is the cache layer. The caching algorithm is plugged as a template parameter, and can be easily exchanged for a different one. The cache layer can interpolate and crop the height data to produce any rectangle of any level-of-detail for uploading to the GPU and display.

If the tiles were uploaded individually, the performance wouldn't be optimal. That's why there is one more layer between the cache and the GPU. It is called the contiguous buffer allocator, and it is a simple memory allocation policy, priorizing contiguous allocation of in-frame data in order to be able to transfer it in one piece to the GPU. The allocation of the data works in a single linear memory space, and can be actually easily mirrored to the GPU buffers as well (the relative address allocated in CPU memory space is used in GPU memory space as well).

The allocating algorithm works in two passes. The allocator keeps a list of geometry generators, each of which gives a description of how much data in what format it needs to transfer to the GPU. It also keeps track of allocated blocks, and their assignment to the generators. In the first pass, these geometry generators are polled to see how much memory needs to be allocated. Then, the associated blocks (if any) are deallocated, and the process of looking for sufficiently long contiguous space begins. All the blocks are sorted by format (so as to avoid changing vertex attribute pointers, if possible), and are allocated as a single contiguous block of memory, at the lowest address where it doesn't collide with other blocks that remain allocated. The block can also be broken to several smaller blocks if it exceeds sufficient size, beyond which the benefit of transferring the block in a single memory transaction diminishes. Once the addresses of the blocks are resolved, the second pass begins, where the geometry generators obtain the addresses of blocks in memory and start filling it with data. At the end, the block is copied to the GPU.

Results

The described allocation policy is best described by the algorithmic complexities involved. The allocation table of blocks is represented by a tree, where the address of the first byte of the block is used as the key of the tree node. The unused space is also being represented by blocks, which are flagged as unused.

When the blocks are reallocated which were allocated already, they need to be first deleted, merging the free space in unused blocks. This operation takes linear time in the number of blocks, because the blocks are arranged in the tree by their address, and it only takes O(1) lookup to the neighbor residing at lower address to merge the unused blocks incrementally. After that, the allocated blocks need to be sorted by vertex format. That takes O(n log n) time, but in some systems where all the vertices have the same format, this step can be omitted. Finally, a contiguous free space, large enough to hold the blocks need to be found. This takes worst-case linear time in number of blocks, allocated and unused, and O(log n) at average, since the search stops at the first usable block, rarely reaching the end of the list. Therefore, the complexity of allocation of N blocks with M blocks already allocated is:

$$O(N) + [O(N \log N)] + O(\log M) \quad (1)$$

The second logarithmic term can be further diminished by hierarchic representation of the used blocks, since these are typically allocated many in one block, and can be represented as such. This

would highly reduce M, leading to nearly linear allocation time in number of blocks (or constant allocation time if seen from the perspective of allocating a single block), which is a very good result.

The other aspect of the proposed allocator is the amount of wasted space it allows. To be able to formally prove it, let's assume, without the loss of generality, that the blocks being allocated are all of equal size. It is easy to see that the most memory-thrashing pattern is to allocate all the blocks, and then reallocate all but the one before the last one. That then serves as a pivot that holds the free space left from it. The next allocation again loses one block, making the blocks small enough to fit left from the pivot. This process can continue recurrently until all the blocks are deposited. It is easy to see that the worst-case space needed to hold k blocks of size S is:

$$(2k - 2)S \qquad (2)$$

Although this is not a very good result, it is paid for by the allocator speed. In practice, the results are typically much better. To calculate average space needed, a benchmark was devised where 1000 blocks were allocated at random, for many steps, until the average results converged. The results are in table 2.

Table 2. Allocator workspace

Block size distribution	Space (average)
Constant (S)	1542.31 S
Gaussian with center at S	1866.08 S
Uniform between 0 and S	825.73 S

The ideal result for storing 1000 blocks of size S would be 1000 S. Due to the inevitable memory fragmentation, that is unfortunately not possible. The gaussian results are somewhat worse than constant size blocks, most likely because constant size blocks tend to align nicely even when reallocated, causing less fragmentation.

Note that although the last result seems to be incorrect (since 1000 blocks is stored and average space is smaller than that), it is caused by the distribution of the block sizes, where the majority of blocks have smaller size than S, due to the uniform distribution.

Conclusions, future work

A novel memory allocation technique was proposed in this paper, enabling fast contiguous allocation of data to be transferred to the GPU (or any other target, separated by a bus with latency). The algorithm is well suited for efficient terrain tile caching. It was implemented in a primary flight display system for fast realistic terrain rendering on low-power hardware.

The technique need not be limited to the rendering of terrain, but can be extended for rendering of all the graphical primitives in the said primary flight display, including the indicators and other user interface elements. It would be interesting to connect the technique with statistic prediction of block reallocation in order to stabilize the frequently allocated blocks at one place in the storage, and to be able to minimize memory fragmentation in general and further increase performance.

References

[1] de Boer, W. H., "Fast Terrain Rendering Using Geometrical Mipmapping", Unpublished paper, available at http://www.flipcode.com/articles/article_geomipmaps.pdf, 2000, pp. 1-7

[2] Ulrich, T., "Rendering Massive Terrains Using Chunked Level of Detail Control", SIGGRAPH Course Notes 3, 5, 2002

[3] Pouderoux J. and Marvie J., "Adaptive Streaming and Rendering of Large Terrains Using Strip Masks", Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia, 2005, pp. 306

[4] Cignoni P., Ganovelli F., Gobbetti E., Marton F., Ponchio F., and Scopigno R., "BDAM - Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization", Computer Graphics Forum 22, 3, 2003, pp. 505-514

[5] Montani C., Cignoni P., Rocchini C., and Scopigno R., "External Memory Management and Simplification of Huge Meshes", IEEE Transactions on Visualization and Computer Graphics 9, 4, 2003, pp. 525-537

[6] Gobbetti E., Cignoni P., Ganovelli F., Marton F., Ponchio F., and Scopigno R, "Interactive Out-of-Core Visualisation of Very Large Landscapes on

Commodity Graphics Platform", Lecture notes in computer science, 2003, pp. 21-29

[7] Gobbetti E., Marton F., Cignoni P., Di Benedetto M. and Ganovelli F., "C-BDAM - Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering", Computer Graphics Forum 25, 3, 2006, pp. 333-342

[8] Livny Y., Kogan Z. and El-Sana J., "Seamless Patches for GPU-based Terrain Rendering", The Visual Computer 25, 3, 2008, pp. 197-208

[9] NASA Land Processes Distributed Active Archive Center, "ASTER Global Digital Elevation Model", available online at http://www.gdem.aster.ersdac.or.jp/.

Acknowledgements

This work was supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/ 02.0070), by Technology Agency of the Czech Republic research project “Smart Autopilot” (TACR TA01010678) and by the Advanced Recognition and Presentation of Multimedia Data project (FIT-S-11-2).

Email Addresses

The addresses of the authors are the following: {ipolok, ibarton, chudyp, krsek, smrz, idittrich}@fit.vutbr.cz.