Adaptive Streaming and Rendering of Large Terrains: A Generic Solution

Raphaël Lerbour
THOMSON R&D France
raphael.lerbour@thomson.net

Jean-Eudes Marvie
THOMSON R&D France
jean-eudes.marvie@thomson.net

Pascal Gautron
THOMSON R&D France
pascal.gautron@thomson.net

ABSTRACT

We describe a generic solution for remote adaptive streaming and rendering of large terrains. The challenge is to ensure a fast rendering and a rapidly improving quality with any user interaction, network capacity and rendering system performance. We adapt to these constraints so loading and rendering speeds do not depend on the size of the database. We can thus use any database with any client device. Our solution relies on a generic data structure to adaptively handle data from the server hard disk to the client rendering system. The same methods apply whatever is done with these data: only the data themselves and the rendering system vary. We base our data structure on existing solutions with good properties and add new methods to handle it more efficiently. In particular we avoid loading irrelevant or redundant data and we request the most important data first. We also avoid costly data structure operations as much as possible, in favor of "in-place" data updates and selection using sample masks.

Keywords

Planetary terrain, adaptive rendering, adaptive streaming, generic data structure, level of detail.

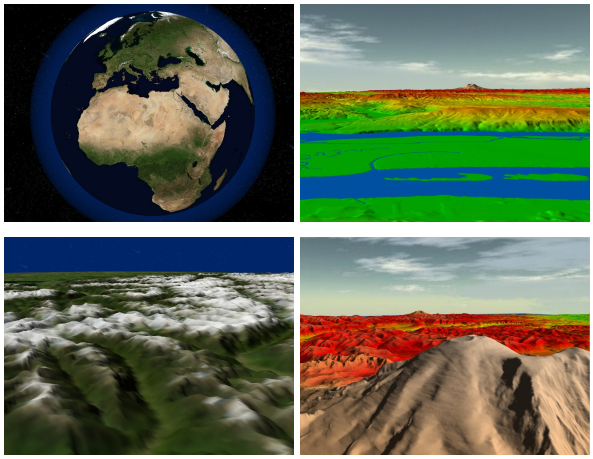
1 INTRODUCTION

Remote adaptive streaming and rendering of large terrains can be used, for instance, to visualize the Earth in 3D with great detail while loading required data over the Internet. The challenge is to ensure a fast rendering with a rapidly improving quality on any client device when a user moves freely over the terrain. Figure 1 presents the result with two example databases.

The terrain surface is uniformly discretized into 2D maps of digital samples. They are usually elevation maps used to reconstruct the relief in 3D along with color maps like photographs. Those maps are huge: a map of the Earth with a precision of 500 meters between samples is over 10 gigabytes. In most cases, such a large amount of data can neither entirely be loaded in memory nor interactively rendered. We thus need to use specifically designed data structures and algorithms.

We propose a solution that relies on a generic data structure to adaptively handle data from a server hard disk to a client rendering system as Figure 2 shows. We base this structure on well tried principles and improve its efficiency with new properties and techniques. Our method can be split into two parts:

First, we adaptively stream the data from the server to the client ("progressive loading" step in Figure 2).



(a) Earth (b) Puget Sound

Figure 1: 3D renderings at 60 frames per second. **Top:** after streaming data for 40 seconds at 1Mbps. **Bottom:** after moving towards terrain features: **a)** The Alps, **b)** Mount Rainier.

The main challenge is to continuously update a partial database within the client. Our technique avoids loading irrelevant or redundant data and adapts to the network speed. The only task of the server is to read requested data from a specifically designed file and transmit them to the client ("requests management" step).

Second, we adaptively select the data to render in the partial database of the client and the missing data to request from the server ("adaptive selection" step). A user can move the viewpoint unpredictably and any rendering system may be used. In all cases, our method adapts to the rendering speed using a measure of importance.

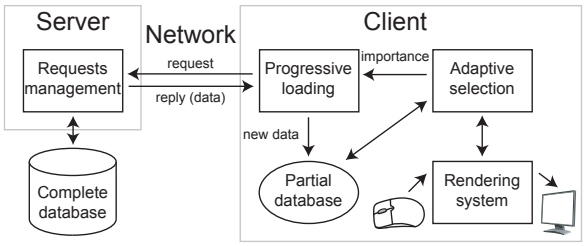


Figure 2: Architecture for adaptive streaming and rendering of terrain data. The user guides rendering on the client. Selected available data are rendered while missing data are requested from the server. The server transmits these data from its database to the client.

2 RELATED WORK

The first solutions for adaptive terrain rendering [LKR+96, DWS+97, RHSS98] minimize the number of geometric primitives to render an elevation map at each frame while ensuring that quality fits very strict criteria. These methods need to store the entire database in memory and require complex computations at runtime. Furthermore, as modern graphics hardware offer high rendering speed it became a better choice to compute geometry faster and use batched primitives.

The method proposed by Hoppe [Hop98] paves the terrain with blocks of polygons inside which data are ordered. These data are progressively added or removed in this order to get the desired quality. Unfortunately, this solution was not generalized to support progressive transmission and does not scale to very large terrains. Similarly, de Boer [dB00] uses a discrete set of uniform resolution levels of detail in fixed-size blocks.

Lindstrom and Pascucci [LP01] perform data fetching and rendering asynchronously, allowing for a smoother rendering. Smart data organization in a file ensures fast access to different levels of detail. However, no solution is proposed to handle distant loading and the data selection for rendering is complex.

Levenberg [Lev02] organizes the data in a hierarchical tree of blocks: it is possible to split a block in two if a better quality is desired. This solution, like [dB00], directly deals with blocks instead of samples, considerably reducing the number of objects to handle.

Cignoni et al. [CGG+03] use a similar solution and add support for progressive loading. Blocks are loaded one by one, progressively descending the tree. This allows using arbitrarily large terrains: a basic representation of the terrain is first rendered using the first level of the tree and only desired areas are refined. However, blocks representing the same terrain area at different tree levels contain and need loading redundant data. Also, the fixed-contents blocks they use require changing tree level whenever different quality is desired: costly database updates are frequent.

The clipmap [LH04] is based on the mipmap solution for texture maps. It is extended with progressive loading to support very large maps. However, the clipmap inherits mipmap drawbacks. Levels of detail are necessarily centered around the viewpoint so we cannot select the rendering quality for any terrain area based on any set of criteria. In addition, rendering a level of detail requires loading its data all around the viewpoint, although those behind the viewpoint are not rendered.

The solution proposed by Schneider and Westermann [SW06] divides blocks into several levels of detail, each of which using the previous one's data and adding its own with an "in-place" update in the graphics hardware memory. Switching between levels is done directly with masks defining which samples are used [PM05]. However, no solution is brought concerning remote loading and very large terrains are not supported.

A complete solution for terrain streaming and rendering is proposed by Gobbetti et al. [GMC+06] based on [CGG+03]. It adds wavelet data compression to manage the progressive data update. However, database updates are still frequent and even more costly because of compression, thus harming rendering adaptivity especially on slow client devices.

Livny et al. [LKES07] propose combining the tree of blocks structure with the idea of using multiple levels of detail per block. However, the tree structure is implicit and is used only to select data for rendering: no progressive loading solution is proposed.

Commercial applications for terrain streaming and rendering like the successful *Google Earth* and *NASA World Wind* have existed for a few years. They rely mostly on their user interface features and the great detail of their databases. In contrast, their terrain-related technologies do not bring significant improvements.

We note that most adaptive rendering solutions are not useable in a remote database context, and solutions that address progressive loading do it at the cost of rendering adaptivity. In this paper, we propose a generic adaptive data structure and the techniques to handle it efficiently at every step of data loading and selection.

3 OVERVIEW

Adaptively streaming and rendering huge terrain maps require using specifically adapted data structures and algorithms. After taking note of previous solutions (see Section 2), we choose to base our structure on certain existing points. We subdivide the sample map into a complete and uniform tree of blocks, then organize the samples of these blocks in a succession of levels of detail (LODs) of increasing resolution (see Section 4). We first add new properties then use new techniques to handle this structure faster and more adaptively.

Our first contribution concerning the data structure is the non-redundancy of data: successive blocks and LODs share their data instead of replacing them. In fact, the new samples of a LOD are spatially interleaved between the previous ones and we implicitly use all of

them. This minimizes the amount of data to store and load, and keeps a better coherency of data among the entire tree. The other contribution is that a block may be rendered when not all of its LODs are available. This offers the possibility to progressively load the LODs of a block. These new points are described in Section 4.2.

Once we have defined our data structure, we can use it in the different steps presented in Figure 2:

We first store the complete tree of blocks in a single file on the server's hard disk as described in Section 5. Our file organization guarantees that the data for any client request are contiguous and that we can directly obtain the position of this data chunk.

In a second step, we progressively load data to the client as explained in Section 6. A measure of importance guides the order in which data loading requests are transmitted to the server. We optimize the relevance of loaded data and prevent overloading the network by continuously updating the queue of pending requests and by transmitting only a few requests at a time.

On the client side, we explicitly store an incomplete tree of blocks in memory as Section 7 describes. When a block is created, we allocate a single 2D array of samples in memory and initialize it with partial data from its parent. We then progressively load new LODs and copy their samples "in-place" in previously unused array positions. These methods, enabled by the non-redundancy of data, reduce the number of data copies in memory.

The last step is the selection of data to render on the client, described in Section 8. We first cull invisible blocks, then choose a LOD for each visible block using a measure of importance. This measure, presented in Section 9, depends on the rendering performance and on interactive user requirements so it can adapt to the application. If a desired LOD is unavailable, we request it from the progressive loading step and we use one with lower quality instead. Once a LOD is selected for rendering, we extract the data to render using a mask that references its samples in the array of the block.

Section 10 introduces results on two example applications: 3D rendering of planetary and non-planetary terrains. Note that only the rendering system differs: no specific methods are used in the main solution.

4 GENERIC DATA STRUCTURE

We base our solution on a generic data structure that combines two commonly used methods: the terrain map is subdivided into a complete and uniform tree of blocks [Lev02], and each of these blocks has a set of levels of detail with increasing resolution [dB00].

Blocks are uniform 2D arrays of samples with a constant resolution. Each one represents a specific square area of the terrain and can be rendered on its own.

The tree is a multi-resolution hierarchical structuring of the terrain map. Each level of the tree covers the entire terrain, with increasing resolution and quality as one gets lower in the tree. Starting with a single root node, the nodes of the tree are blocks with a constant number of children – the minimum is four and corresponds to a quad-tree. The children's covered terrain areas uniformly subdivide the parent's one as shown in Figure 3. The tree can have any depth, as long as one can subdivide the terrain map with enough blocks.

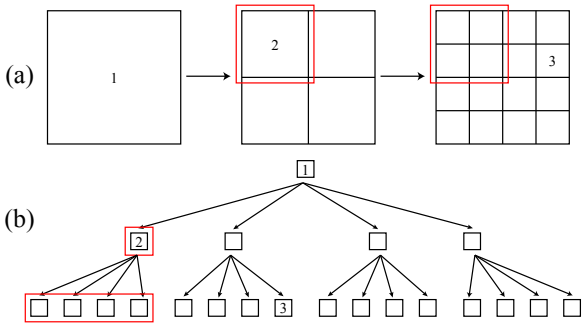


Figure 3: Construction of a three-level quad-tree of blocks. **a)** Successive uniform subdivisions of the terrain map. Red frames cover the same terrain area. Block 1 is the root and covers the entire terrain. **b)** Corresponding tree with the same numbers.

Levels of detail (LODs) are successive subsets of the sample array of a block as shown in Figure 4. Each LOD contains the previous one and adds new samples; the last LOD uses the full sample array of the block. Array subsets of the LODs are uniform over the blocks to allow using generic methods for storage, update and selection. In our application, each LOD doubles the resolution of the array subset in both dimensions compared to the previous one – minus one row and one column when using odd-resolution blocks: see Section 8.2. The number of children per block is thus defined by the number of LODs: for instance when using three LODs, the last one has sixteen times more samples than the first, hence the block has sixteen children.

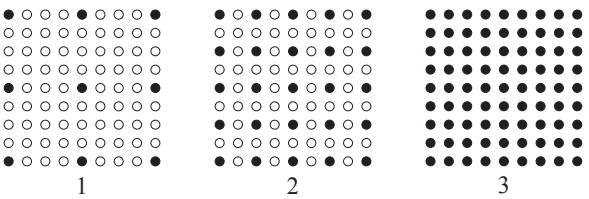


Figure 4: Successive LODs of a block (example of a 9×9 sample array with three LODs). Full black samples are used, unfilled ones are not.

4.1 Advantages

The data structure was chosen for its properties in adaptive streaming and rendering, described hereafter.

The tree structure allows us to use only a few blocks in the upper tree levels to render terrain areas with low quality requirements, and inversely. This technique minimizes the total amount of rendered data and better

distributes these data over the terrain. Similarly, with the LOD structuring of blocks, we can choose a LOD to render for each block based on the desired quality. This makes the data selection adaptive not only in the tree of blocks but also within these blocks, thus lowering the number of costly data structure update operations.

Another property is that the terrain can always be entirely rendered at a minimum quality even if not all the tree levels are loaded. We may thus progressively load the tree, starting with the root block then descending where needed. When a block is loading, we continue rendering the terrain area using upper tree levels. In addition, the tree structure simplifies the culling of invisible blocks using a classical depth-first walk-through.

4.2 New Properties

To better adapt the structure to our needs and improve its general efficiency, we add two new properties:

First, a block and its children share samples because they cover the same terrain area: this avoids loading redundant data. When a block in the partial client tree splits, it gives the samples of its last LOD to its children, creating their first LODs. Reciprocally, it gets these samples back from the children when they are merged. Split and merge operations are described in Section 7.1.

Second, we allow rendering a block even if its sample array is not fully loaded: only the samples of the selected LOD need to be available. Consequently, we can progressively load the LODs of a block into a common sample array (with the refine operation: see Section 7.2) while using this array to render previous LODs.

Using these properties, we can get one level down in the tree with no need to load all the data of the new blocks. They get their first LOD from the parent and only those who need more quality start loading their next LODs.

5 SERVER DATABASE

The first place where the data are located is a precomputed file on the server's hard disk. Many clients can connect to the server at the same time and request data corresponding to any terrain area: hard disk accesses are random and very frequent. In order to minimize disk activity, we optimize the organization of the data in the file when constructing it.

The file first contains a header with characteristics of the database like the resolution of the blocks and their number of LODs. Then, the contents of the tree are "flattened" as shown in Figure 5, LOD by LOD. Loadings are done one LOD at a time, hence ensuring that all the data for a single request are contiguous in the file.

The size of each file element is known in advance: all blocks and LODs have uniform resolutions and samples are stored on a constant number of bytes. Furthermore, the tree is uniform and complete and the blocks of each level are stored in order. Consequently, we can get the file position for any request in constant time.

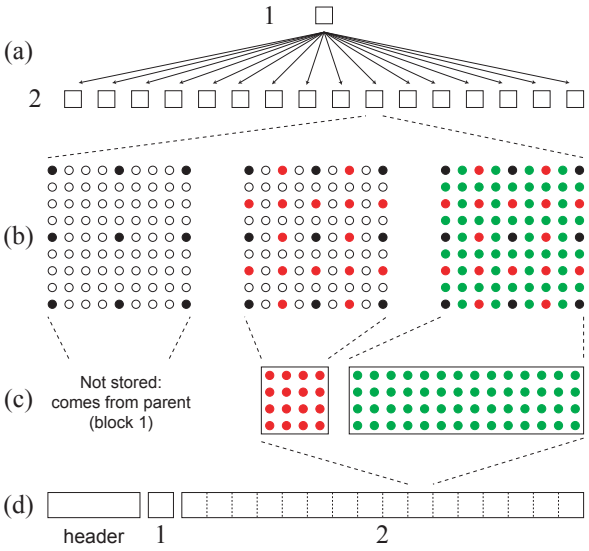


Figure 5: File organization (example of a two-levels tree with three-LODs blocks). **a)** Original tree structure. **b)** Block structuring with LODs (see Figure 4). **c)** Redundancy-free LOD storage. **d)** Final data order in the file with the same numbers as in (a).

Non-redundancy of data is clearly visible in Figure 5. First, we do not store the first LOD of any block because it comes from its parent. The root block does not have a parent, we thus store its first LOD in the file header. Second, a LOD implicitly reuses the previous ones so we store only the new samples.

6 PROGRESSIVE LOADING

In our solution, the server only reads and transmits data "as is" to clients on demand. We manage progressive data loading on the client side as Figure 2 shows. At any given time, our method streams the most important data: it implicitly adapts to the network speed and the rendering quality constantly improves.

When the adaptive selection step (presented in Section 8) needs a new LOD for a block, it sends a loading request to the progressive loading step while rendering continues in parallel with the available data.

The server usually may not respond quickly to all requests because the network can have any speed and latency and there may be other clients asking for data. Unfortunately, the longer a request is pending the more it is probable that this request is no longer relevant at data reception, for instance if the user viewpoint has moved. To avoid overloading the network and the server with irrelevant loadings, we restrict the number of pending server requests using a fixed-size queue on the client. When the client receives data from the server, we update the partial database with a refine operation and we remove the request from the queue.

To choose the requests to queue, the adaptive selection step gives an importance value along with each request (see Section 9) and continuously updates this

value. When room is available in the queue, the progressive loading step selects the request with the highest importance, adds it into the queue, and transmits it to the server. The adaptive selection step then sends again requests that were not selected if they are still relevant.

7 CLIENT DATABASE

The client database is an incomplete and unbalanced tree of blocks. The very first data loading contains the first LOD of the root block: we can immediately start rendering the terrain with the lowest quality. The adaptive selection step then triggers specific database update operations as explained in Section 8.1, progressively expanding the tree as Figure 6 shows. All data copies are performed in parallel with selection and rendering so they do not harm rendering smoothness, especially on multi-core architectures. Only the tree structure changes and data deletion are protected.

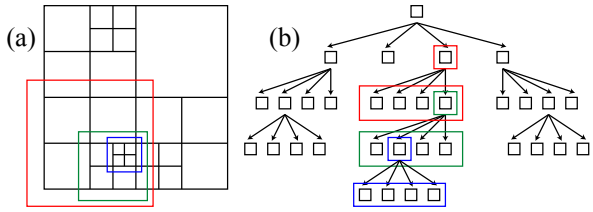


Figure 6: Progressive loading of a quad-tree. **a)** Successive subdivisions of the terrain map (red, green, then blue). **b)** The corresponding incomplete tree.

We store the samples of a block in a single array with the resolution of its last LOD. This array is present in memory only for the leaves of the tree in order to reduce memory consumption on the client. Reciprocally, all leaves of the tree have a sample array: we thus ensure that any terrain area has a representation on the client.

7.1 Split and Merge Operations

When a fully loaded block needs to be rendered with higher quality than it can offer, we use its children instead. If the block is a leaf, we have to load the children in memory with the split operation. The sample array of the parent is uniformly subdivided into square subsets corresponding to its children as Figure 7 shows. Samples of each subset are copied into the child's previously empty sample array to build its first LOD. The parent finally deletes its own sample array. Once the split operation is done, each child can progressively load its own LODs and eventually split itself independently from the others: that is how the tree gets unbalanced, and thus how we get local adaptivity.

When a previously split block needs to be rendered with lower quality than its children can offer, it gets back its data from these children with the merge operation. As Figure 7 shows, we recreate the block's sample array and get all of its samples from the first LOD of its children. The children are merged recursively when

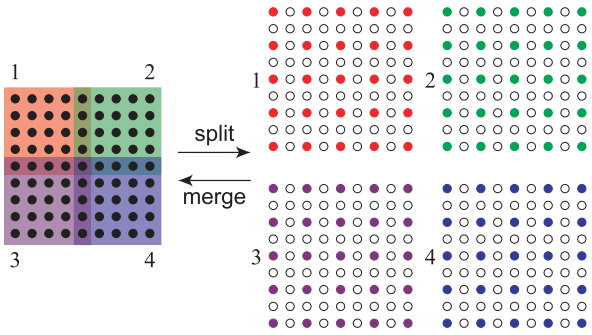


Figure 7: Block split and merge in a quad-tree. **Split — Left:** Parent block with the array subsets of its four children. **Right:** Children blocks with their first LOD. Unfilled samples are not loaded yet. **Merge — Right:** The four blocks to merge. **Left:** Parent block fully reconstructed from the children's first LODs.

needed. Once the merge operation is done, we delete the children blocks because they are no longer used for rendering. One can note that we could cache those blocks in case they are needed again some time later, as long as enough memory is available. We plan to implement this in the future, although our solution already offers some caching by using multiple LODs per block.

7.2 Refine Operation

When a partially loaded block needs to be rendered with higher quality than its maximum available LOD can offer, we load the next LOD with the refine operation. Refining first requires loading data from the server: see Section 6. At data reception, we add the new samples of the LOD in the corresponding unused positions of the sample array of the block as Figure 8 shows.

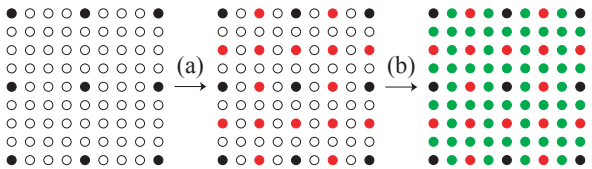


Figure 8: Block refinements (same example as Figure 4). **a)** Red samples (second LOD) are interleaved between black ones (first LOD). **b)** Green samples (third LOD) are added, the array is now full.

For rendering speed reasons, it is possible that samples received from the server need transforming their format or mode of representation once for all before being stored in the sample array. For instance, in the case of 3D rendering of planetary data, quantified elevation values relative to a reference ellipsoid may be translated into 3D floating-point coordinates in a global coordinate system. As for other data update operations, this is done in parallel with rendering. Note that only the samples in the client database change: two clients using two different rendering systems can use the same methods and data, connecting to the same server.

8 ADAPTIVE RENDERING

We can render the partial database of the client at any moment. According to a measure of importance, we first trigger database updates and select a LOD to render for each visible block. We then send the samples of those LODs to the rendering system with a set of masks.

8.1 Data Selection

To select data to render in the partial database of the client, we first compute an importance value for all blocks as explained in Section 9. Second, we select the blocks to render. Only leaves of the tree can be rendered because they have sample arrays. However, the view frustum usually does not include the entire terrain and some blocks are thus invisible: we cull them using a classical depth-first tree walk-through. Finally, for each block we choose the LOD to render. Each LOD has an associated importance value: when this threshold is reached, the LOD is selected if available.

When a required LOD is not available, we trigger the corresponding database update operation. First, this LOD may be of higher resolution than the maximum available one. In that case, we send a request the progressive loading step. Second, the block may have been split before because more quality was once required, but now one LOD of this block is enough to render the same terrain area. In that case we trigger a merge operation, except when the desired LOD is the last one: the first LODs of the block's children contain the same samples and we prefer to avoid data structure operations when possible. Merge operations are triggered the same way for invisible blocks in order to save memory.

Unlike the other operations, we trigger the split of a fully loaded block when one of its children needs to load its second LOD. Children are not in the tree yet, so we guess their importance based on the parent's one. We use this guard because splitting blocks as soon as they are fully loaded could lead to tree structure instabilities when the importance of a block varies slightly, uselessly harming the overall performance.

8.2 Masks for Level of Detail Rendering

Once we selected an available LOD to render for each visible block, we send the corresponding samples to the rendering system. We do this by applying a mask on the sample array of the block; it defines the subset used by the LOD. There is one mask per LOD computed only once, common for all the blocks because the sample array subsets are uniform. We thus do not have to store or compute many masks for different blocks.

When using 3D graphics hardware to render elevation data, masks can be implemented using triangle strips [PM05]. This standard structure defines a set of contiguous polygons to render with a succession of indices pointing on an array of 3D vertices. In our case, each triangle strip mask points on vertices computed from the samples of the desired LOD as Figure 9 shows. This way, the sample mask is applied directly in the graphics hardware with no additional data copy.

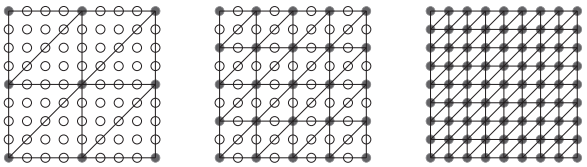


Figure 9: Triangle strips for 3D rendering of a block with elevation samples (same example as Figure 4). For each LOD, the corresponding samples in the array are linked to create triangles. Elevation values can be used to compute 3D vertices coordinates.

Note that we use odd-resolution blocks with common boundary samples so that adjacent triangles in neighbor blocks stitch together. However, discontinuities appear when those blocks are not rendered at the same LOD. This paper does not present the details of this method, but we can avoid such gaps using additional triangle strip masks on block boundaries [LKES07].

9 MEASURE OF IMPORTANCE

We use a measure of importance to select the LOD to render for visible blocks, to trigger updates of the partial database, and to define the order in which we transmit requests to the server. It ensures good adaptivity for both loading and rendering. We can get an importance value for any block at any time; it represents the quality desired for the terrain area that this block covers.

Any measure of importance can be used, based on the rendering system and the application. However, it always depends on a generic quality factor ensuring that the solution adapts to the rendering speed. In practice, the user selects the number of frames per second he or she wants for rendering. Whenever the rendering speed gets over or under this value – given or taken a variation tolerance threshold to minimize instability –, the quality factor respectively increases or decreases in proportion until the target frame rate is obtained.

We know that the quality increases as one descends the tree, so we give lower importance to blocks with a small radius. In most cases we also want to give higher importance to blocks close to the viewpoint. Other information can be used to get a more specific measure of importance, like the viewpoint's incident angle and the block's geometry roughness. Equation 1 is an example measure of importance, and Figure 10 shows its impact on our 3D rendering application.

$$importance = log_2\left(\frac{qualityFactor \times radius}{viewpointDistance}\right) \quad (1)$$

We select LODs using their numbers as importance thresholds: the log_2 function reflects that a LOD has twice the samples of the previous one in both dimensions. In addition, to get the importance of a loading

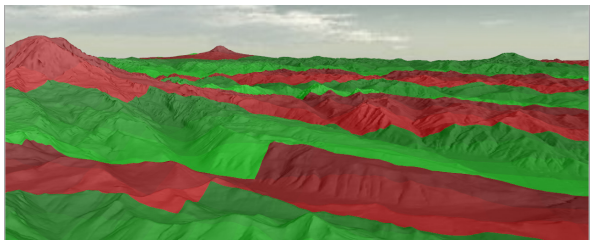


Figure 10: Impact of the measure of importance on 3D rendering. Importance values define the colors: red is more important than green and the brighter, the more important. Using two LODs per block, we can select them as in the picture: green for the first LOD and red for the second. Color layers around the viewpoint correspond to the levels of the tree: blocks split as they get close, the terrain areas covered by their children are smaller so they have lower importance.

request, we also want to take into account that a lower LOD of the block is already available. We thus subtract the number of this LOD from the computed importance.

10 APPLICATIONS AND RESULTS

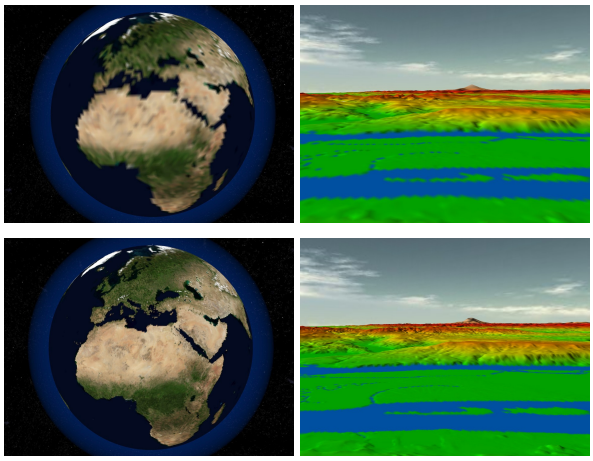
The structure and methods presented in this paper are generic: we deal with maps of samples, but those may be rendered in multiple ways. We have implemented two 3D rendering applications: one handles bounded terrains with elevation relative to a plane; the other handles planets mapped onto a cube with elevation relative to a reference ellipsoid.

We tested both applications using real-world databases and user interactions. The planetary terrain is the Earth (Figures 1a, 11a) and the local terrain is the Puget Sound (Figures 1b, 11b). Samples are made of 2 bytes for elevation and 3 bytes for RGB color. The client runs on a computer with a 2.4GHz Core 2 Duo CPU, 2GB of RAM and a GeForce 8800 GTS graphics card. Data are streamed uncompressed over a 1Mbps ADSL connection and rendered on a window of 1600×900 pixels. The target rendering speed is 60 frames per second (FPS) and corresponds to around one million polygons per frame with our rendering system and hardware.

10.1 Earth

The Earth database is a 13GB file built from the NASA BMNG [SVS+05], a set of color and elevation maps with 500m precision at the equator. We use six ten-level quad-trees of 43×43 sample blocks with two LODs.

Figure 12 presents runtime results with an example trajectory. We can see that loading stops as soon as the frame rate gets below the given threshold: the quality factor adapts so the importance of the blocks decreases. No new request is added in the fixed-size queue and, because data were loaded in importance order, no more loading is needed to better distribute samples over the terrain surface. In standard conditions starting at second 180, the network is not fast enough to provide maximum quality at a stable 60 FPS, but 99.5% of received data are immediately relevant for rendering. View-frustum culling and importance computation take less than 5% of the time for each frame.



(a) Earth (b) Puget Sound

Figure 11: 3D renderings after streaming data for 2 seconds (**top**) and 10s (**bottom**) at 1Mbps. See Figure 1 for desired 60 FPS renderings after 40s.

We also ran the same test with the client located on the same computer as the server. In these conditions, the initial loading until achieving the target frame rate takes less than five seconds. The rendering speed then stays stable at 60 FPS for the remaining of the test. This configuration can be used, for instance, to compute and broadcast in real-time a terrain walk-through video.

10.2 Puget Sound

The Puget Sound database is a 16385×16385 samples map with 10m precision and false color, for a total of 1.27GB. We use a nine-level quad-tree of 65×65 sample blocks with two LODs.

Results of the test are shown in Figure 12. When quickly moving forward, most of the available data are no longer rendered because they get behind the viewpoint. The network delay prevents us from immediately replacing them with higher quality data for visible areas: this explains the large frame rate increase. However in standard conditions starting at second 155, this does not apply and the frame rate is stable while loading because less data are required at once. When moving backwards, the adaptive quality factor compensates for the network delay: we continue to render areas getting farther in high quality until the frame rate gets too low.

11 CONCLUSION

We proposed a generic solution for remote adaptive streaming and rendering of large terrains. Our methods apply whatever is done with the data: only the data themselves, the rendering system and the measure of importance vary. We can, for instance, stream an aerial

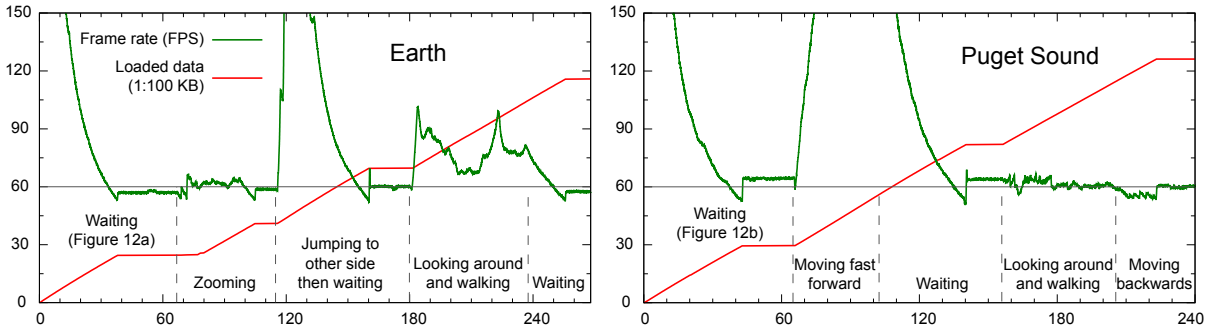


Figure 12: Statistics for interactive 3D rendering of the Earth and Puget Sound databases, streaming data over a 1Mbps connection. The horizontal axis is the time in seconds. The gray line is the target frame rate of 60FPS.

photography for 2D rendering with zoom or render the Earth in 3D for a geo-positioning application. We adapt to the loading and rendering speeds so they do not depend on the size of the database. We can thus use a single database on a single server with any kind of client, like a smartphone with 3G connection or a desktop computer with 3D graphics hardware and broadband connection: only the rendering quality varies.

We based our data structure on existing solutions with good properties and added new methods to handle it more efficiently. We use this structure to manage the data from the server hard disk to the client rendering system trying to be as fast as possible. In particular we avoid loading irrelevant data, for instance by ensuring that data are not redundant between successive loadings and by always sending the most important data requests. We also avoid costly data structure operations as much as possible, in favor of "in-place" data updates and selection using sample masks.

In the future, we plan to produce results with low performance devices and present specific features based on the generic solution, like our method to render planetary terrains. In addition, we are working to handle texture and elevation maps concurrently, fix geometry gaps and avoid down-sampling artifacts.

REFERENCES

[CGG+03] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum*, 22(3):505–514, 2003.

[dB00] W. H. de Boer. Fast terrain rendering using geometrical mipmapping. Unpublished, available at: `http://www.flipcode.com/articles/article_geomipmaps.pdf`, 2000.

[DWS+97] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the conference on Visualization '97*, pages 81–88, 1997.

[GMC+06] E. Gobbetti, F. Marton, P. Cignoni, M. Di Benedetto, and F. Ganovelli. C-BDAM – compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum*, 25(3), 2006.

[Hop98] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 35–42, 1998.

[Lev02] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 259–266, 2002.

[LH04] F. Losasso and H. Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics*, 23(3):769–776, 2004.

[LKES07] Y. Livny, Z. Kogan, and J. El-Sana. Seamless patches for GPU-based terrain rendering. In *Proceedings of WSCG '07*, pages 201–208, 2007.

[LKR+96] 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. In *Proceedings of ACM SIGGRAPH 96*, pages 109–118, 1996.

[LP01] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 363–371, 2001.

[PM05] J. Pouderoux and J.-E. Marvie. Adaptive streaming and rendering of large terrains using strip masks. In *Proceedings of GRAPHITE 05*, pages 299–306, 2005.

[RHSS98] S. Roettger, W. Heidrich, P. Slusallek, and H. Seidel. Real-time generation of continuous levels of detail for height fields. In *Proceedings of WSCG '98*, pages 315–322, 1998.

[SVS+05] R. Stockli, E. Vermote, N. Saleous, R. Simmon, and D. Herring. The Blue Marble Next Generation - a true color earth dataset including seasonal dynamics from MODIS. NASA Earth Observatory, 2005.

[SW06] J. Schneider and R. Westermann. GPU-friendly high-quality terrain rendering. *Journal of WSCG*, 14(1-3):49–56, 2006.