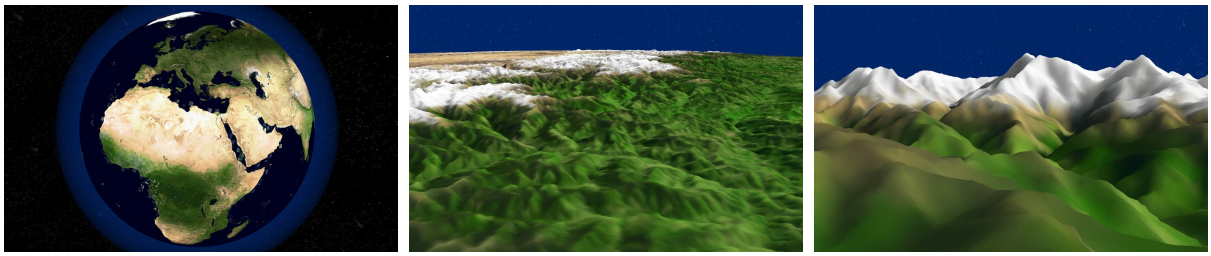
Adaptive Real-Time Rendering of Planetary Terrains

Raphaël Lerbour Jean-Eudes Marvie Pascal Gautron
{raphael.lerbour, jean-eudes.marvie, pascal.gautron}@thomson.net
Thomson Corporate Research
1, avenue de Belle Fontaine, CS 17616
65576 Cesson-Sevigne, France



ABSTRACT

As virtual worlds applications get more and more demanding in terms of world complexity and rendering quality, rendering virtual terrains and planets in real-time introduces many challenges. In this paper, we provide a full-featured solution for rendering of arbitrary large terrain datasets in the context of client-server streaming. Our solution automatically adapts to arbitrary network bandwidths and client capabilities, ranging from high-end computers to mobile devices. We address the problem of rendering highly complex terrain databases comprising several hundred gigabytes of data, which therefore cannot be entirely loaded in memory nor rendered in real-time. The contributions of this paper solve important issues for high quality terrain rendering: adaptive texture mapping, inexpensive removal of geometry cracks and support of planetary terrains.

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

1 INTRODUCTION

Virtual 3D worlds become more and more omnipresent in many applications, ranging from video games to cinema and virtual training. As time gets by, the virtual worlds applications get more and more demanding in terms of world complexity and rendering quality. Rendering virtual terrains and planets in real-time introduces many challenges, in particular for data representation, streaming and high quality rendering. We address the problem of rendering highly complex terrain databases comprising several hundred gigabytes of data, which therefore cannot be entirely loaded in memory nor rendered in real-time. Building upon a generic solution for terrain streaming, the contributions of this paper solve important issues for high quality terrain rendering: adaptive texture mapping, removal of geometry cracks and support of planetary terrains.

A virtual terrain is first described by geometry, which provides the relief information. However, the photometry embeds essential clues to the terrain structure and contents: a high quality terrain texture provides information on the ground type, vegetation, climate... We define a dual decomposition of the photometry information that matches the representation of the terrain and allows for high-performance, independent refinement of the geometry and photometry. Consecutive levels of details are filtered for further quality improvement.

While the use of multi-resolution eases the streaming and rendering of large terrains, artifacts or *cracks* tend to appear at the edge of adjacent terrain patches which use different levels of detail. We introduce a simple and secure method for stitching together adjacent patches regardless of their current level of detail.

Virtual terrains are generally considered planar, yielding a straightforward management of elevation data. However, our method supports arbitrarily large terrain datasets, and is hence able to render an entire planet with high detail. We address the representation of planetary terrains in which the elevation is relative to a reference ellipsoid instead of a plane. Based on a gnomonic projection scheme, we devise a crack-free uniform-angle sampling method for high quality projected elevation data. Our approach reduces the size of the dataset while preserving geometry and photometry

details.

Current graphics hardware rely on depth buffering for determining the polygons visible through each pixel of the image. At the planetary scale, the precision of the depth buffer may be challenged, yielding flickering and crawling artifacts. We solve this issue by adjusting the clipping planes of the view frustum so that the rendered planet surface. This increases the depth precision by using the entire depth representation range, while clipping parts of the planet located behind the horizon.

This paper is organized as follows: Section 2 describes relevant previous work in the field of adaptive rendering of large terrains. Section 3 presents the generic adaptive streaming solution in which our contributions are integrated. The next sections introduce our contributions: Section 4 presents our method for adaptively mapping high quality texture maps. In Section 5, we devise our method for patch stitching across different levels of details. Section 6 addresses the rendering of planetary terrains and adaptive clipping.

2 RELATED WORK

Adaptively streaming and rendering huge terrain maps requires using specifically adapted data structures and algorithms. An extensive survey of techniques for rendering large terrains have been recently published by Pajarola *et al.* [PG07]. This section describes previous work particularly related to this paper.

To manage those maps all the way from a server hard disk to a client rendering system, we use an existing generic solution based on a hierarchy of square blocks with levels of detail [LMG09]. Section 3 briefly describes this solution.

The *Clipmap* [TMJ98] is a powerful solution for huge texture support in 3D hardware rendering with no constraint due to geometry. *Geometry Clipmaps* [LH04] extend this concept to geometry and photometry information. While this method provides high performance and quality, the level of detail selection is performed in circular zones all around the viewer, without accounting for the actual visual importance of each terrain block. Furthermore, such circular refinement scheme may introduce a performance hit when streaming the terrain over a low-bandwidth network.

However, the generic solution we use for adaptive streaming and rendering can handle any kind of map: like many other works [CGG$^+$03, WMD$^+$04, Hwa05], we link two identical hierarchical structures respectively representing textures and elevation.

To avoid cracks between adjacent blocks when rendering a hierarchy of geometric terrain blocks, one can use triangular blocks with fixed resolution on their edges [Lev02, CGG$^+$03, Hwa05]. This needs no explicit handling but prevents using levels of detail in blocks. *Geomipmaps* [dB00] use square blocks with levels of detail like us, and fix cracks by skipping samples on the edges of high definition blocks when they are not used by their neighbors. There is no multi-resolution block hierarchy so neighborhood management is simple, but this is not adapted for large terrains. Finally, one can solve cracks in a hierarchy of square blocks with no neighborhood knowledge by selecting LODs at block edges instead of centers [LKES07]. Block centers are then rendered using additional sample masks to internally stitch the LODs of the edges. Unfortunately, this requires all the data of the terrain to be available so that both sides of an edge render at the selected LOD.

Planets in *Google Earth*, *NASA World Wind* and other solutions [CH06] use the equirectangular cylindrical or *plate carrée* projection, the traditional standard for digital representations of planets. This projection causes large sampling distortions for non-equatorial regions: in particular, polar areas look very stretched when rendered and much redundant data are stored and rendered. Using a cube to represent the planet like [CGG$^+$03] offers a significant improvement. However this solution does not use regularly sampled blocks: all samples need barycentric coordinates to compute their 3D position from those of the corners of the block. We prefer using a regular map projection and directly obtain positions with no additional information or interpolation.

3 GENERIC STREAMING SOLUTION

The generic solution underlying our work adapts to the speed of the network and the rendering performance of the client so any database can be used in any conditions [LMG09] (Figure 1). This section recalls the base data structure and the algorithms of [LMG09].

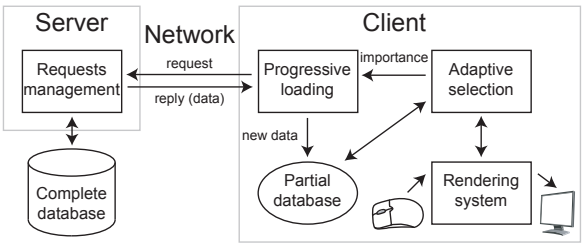


Figure 1: Architecture for adaptive streaming and rendering of terrain data. The user guides rendering on the client. Selected available data are rendered while missing data are requested and fetched from the server.

3.1 Data structure

The data structure, presented in Figure 2, consists in subdividing the sample map into a uniform tree of blocks, then organizing the samples of these blocks into a succession of levels of detail (LODs) of increasing resolution. Successive blocks and LODs share their data to avoid redundancy: new samples are spatially interleaved between the previous ones. In addition, a block only needs the data of its first LOD to be ren-

dered, allowing the other LODs to be loaded progressively.

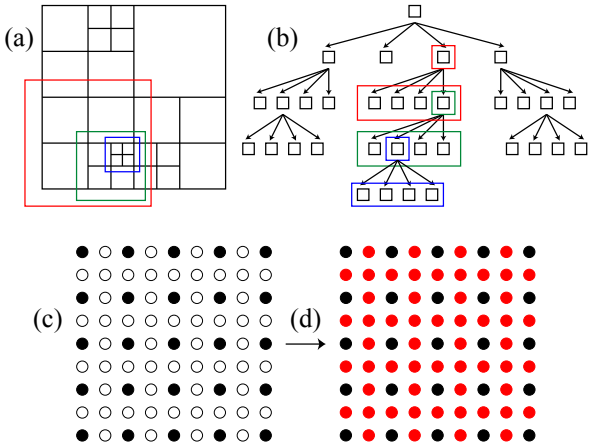


Figure 2: Tree of blocks with levels of detail. **a)** Successive subdivisions of the terrain map on the client side (red, green, blue). **b)** The corresponding incomplete tree. **c)** First LOD of a 9×9 block, only solid black samples are used **d)** Red samples (second LOD) are interleaved between black ones (first LOD).

3.2 Adaptive streaming

The complete tree of blocks is first stored in a single file on the server's hard disk. The data for any client request are contiguous and their position is obtained directly.

The client then progressively loads data from the server. A measure of importance guides the order in which data requests are transmitted. To optimize the relevance of loaded data and prevent overloading the network, the request queue is continuously updated.

An incomplete tree of blocks is explicitly stored on the client and constantly updated with asynchronous split and merge operations. When a block is created during a split, it allocates a single 2D array of samples in memory and obtains its first LOD from its parent. It can then load and interleave new LODs in previously unused array positions.

3.3 Adaptive rendering

The last part of the solution is the selection of data to render on the client. This part first culls invisible blocks, then chooses an LOD for each block using a measure of importance. This measure depends on a general quality factor that adapts to the rendering speed. When a desired LOD is unavailable, this triggers a new data request or tree update operation and an available LOD is used instead. Once an LOD is chosen for rendering, a precomputed mask extracts its samples from the array of the block as in [PM05].

4 ADAPTIVE PHOTOMETRY

Texture maps convey much detail about the terrain such as the local climate, ground type and vegetation. Such information tends to contain high frequencies, and therefore the texture maps are usually larger than elevation maps. Just like the terrain geometry, this potentially huge amount of data requires an adaptive management of the levels of detail. We propose to extend the geometry management scheme presented in Section 3 by introducing a dual multi-resolution structure for texture management. Geometry and photometry are kept coherent by linking their respective LOD trees.

4.1 Linked trees

Textures are mapped onto the terrain geometry to provide extra details on the appearance of the ground. As the texture and elevation LOD trees may have different size and depth, we use distinct trees and maintain a link between the two incomplete trees on the client so that each elevation block uses an adapted texture block. In practice, each block stores a link to the block of the other tree which covers the same terrain area, if any.

Classical texture rendering usually requires mapping one texture onto each elevation block. Our approach relaxes this constraint: when rendering an elevation block, we use its link to the texture tree to find the exact texture block correspondence. In available, the texture block is used directly. Otherwise, we recursively query the parents of the elevation block until a valid linked texture block is found.

We also introduce two constraints on split and merge operations to avoid creating texture blocks that cannot be used due to the lack of corresponding geometry data. First, we avoid splitting a texture block if the corresponding elevation block does not have any children on which the refined texture could be mapped. That is, a texture block cannot split if the linked elevation block is a leaf of the tree.

The second constraint prevents an elevation block from merging if its linked texture block has children. That is, an elevation block cannot merge if the linked texture block is not a leaf.

Let us recall that our solution aims at streaming terrain data through heterogeneous networks. Therefore, as described in [LMG09], the split and merge requests are queued and performed asynchronously with a potentially important delay due to network latency. Consequently, we enforce the constraints not only when triggering such operations, but also just before the actual execution of the operations.

4.2 Implementation Details

This section details specific implementation aspects of our work for further performance.

The split and merge operations are performed according to a measure of importance which depends on the on-screen coverage of the block. Therefore, we evaluate the importance for geometry blocks, and reuse it to refine their linked texture blocks.

A terrain block is defined by a triangle mesh, which requires texture coordinates for texture mapping. In our approach, the elevation tree may be deeper than the texture tree: a texture block may be mapped onto many terrain blocks. In this case the texture coordinates must be adapted to obtain a coherent appearance. Such adaptation can be performed inexpensively by transforming a default texture coordinate set using OpenGL texture matrices.

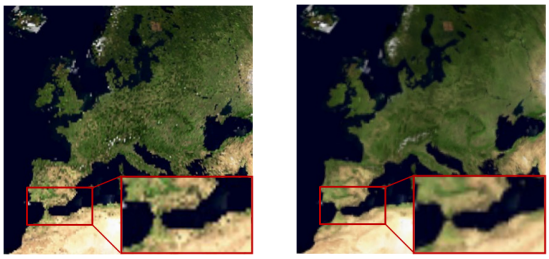
Texture LODs are implemented as mipmaps for 3D rendering: when we load a new LOD, we add a mipmap to the texture to increase quality. To enforce mipmapping even when only the first LOD of a block is available, we also create lower resolution mipmaps during splits.

We also take advantage of programmable graphics hardware to loosen the two constraints of Section 4.1. In those constraints, the refinement level of the texture tree is limited to avoid mapping several texture blocks onto one geometry block. Using the support of multiple textures in the fragment shader, a single geometry block can be rendered using several texture blocks.

4.3 Filtering

Although nearest-point subsampling is often acceptable for elevation maps, aliasing artifacts arise using this technique for texture maps. We therefore choose to use filtering when building higher levels of the texture tree during server file creation. We may use any image filtering method as long as it produces regularly sampled LODs. Wavelets come into mind, but we prefer using a scheme that is very fast to decode so it can be used with low performance clients.

We integrated the *Progressive Texture Map* (PTM) texture filtering method [MB03] in our solution (Figure 3). This multi-resolution scheme perfectly matches the LOD data structure used in our method. Furthermore, it uses simple and fast bilinear interpolation, and stores small delta values within each LOD to losslessly reconstruct samples using the previous LOD instead of redundantly transmitting all the samples. This adds a 8.3% overhead in LOD size compared to point sampling, but still saves 18.7% compared to redundant LOD streaming.



(a) Point sampling (b) PTM filtering

Figure 3: Visual impact of texture filtering.

When using texture filtering, different LODs of a block use different values for a given sample. This prevents us from using a single sample array for each block. We therefore store each LOD of the block in a separate array. During split operations, we split all of them instead of creating lower resolution mipmaps.

5 FIXING GEOMETRY CRACKS

Rendering multi-resolution terrains leads to the use of several levels of detail depending on the importance of each block. In terms of geometry, adjacent blocks may have different resolutions and hence cannot be perfectly stitched, yielding vertical gaps or "cracks" (Figure 4).

This section details a simple yet efficient method for avoiding such artifacts using *edge strip masks* to stitch together a set of geometry blocks with heterogeneous refinement levels.

(a) Rendering with cracks (b) Block stitching

Figure 4: Our method avoids cracks by stitching blocks with different resolutions.

5.1 Edge Strip Masks

As explained in Section 3.3, we use triangle strip masks to extract the samples of an LOD from the common sample array of a block. As adjacent blocks may have different resolutions, we aim at adjusting the triangle strip masks so that the edges of adjacent blocks share the same set of vertices.

We first propose to subdivide a block into five parts: inner block samples and four sets of edge samples. We keep the base idea of strip masks [PM05] in the inner block for managing the main level of detail of the block. On the block edges we create triangle strips called *edge strip masks*, which stitch the block to the level used in the adjacent block (Figure 5).

For each edge, *edge strip masks* systematically reduce the resolution of the block with the higher definition LOD, skipping vertices not used by the other block. Otherwise, we could need data that are not loaded yet. Note that only one large strip is used for blocks that do not need stitching.

Note that the *edge strip masks* undergo the same process as regular strip masks: each strip mask is cached within the graphics memory and reused at need. Therefore our crack removal technique does not introduce any processing nor rendering overhead.

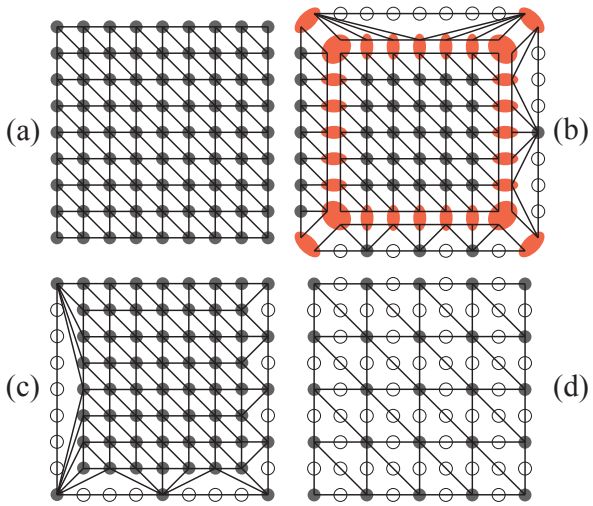


Figure 5: *Edge strip masks* for block edges. **a)** Block using its last LOD. **b)** Block using its last LOD, with *edge strip masks* on its edges to adapt to its neighbors based on the LOD difference. Left: no difference, bottom: one level, right: two levels, top: three levels. Red samples appear stretched for legibility. **c)** Mirror of (b), with the strips stitched as when rendered. **d)** Block using its next to last LOD.

5.2 Neighborhood management

Deciding which *strip edge* to use on the edges of a block requires knowledge of the resolution of its neighbors. To this end, we maintain a table of one neighbor per block edge, possibly with a different resolution. To ensure coherency and robustness, we do not update the neighbors list when a neighbor that is on the same tree level splits: this would lead to more than one neighbor per edge. In that case, we know that this neighbor block has equal or higher resolution. As the purpose of *edge strip masks* is the reduction of resolution across edges, we simply use the higher resolution *edge strip mask*. Potential reduction will be performed by the higher resolution neighbor blocks.

Note that our aim is the removal of cracks at the edges of the blocks. If an edge is not visible, we simply use the default *strip mask* as no adaptation is needed.

5.3 Split and merge constraints

In some cases, we cannot stitch the common edge of two neighbor blocks. This happens when the lowest-definition LOD of the block on the lower tree level has a higher definition than the current LOD of the other block (see Figure 6). This occurs especially often when using a block resolution not in $2^n + 1$ form. To enforce the robustness and reliability of the method, we add constraints of the split and merge operations to limit the maximum difference of tree levels between neighbors based on their resolution.

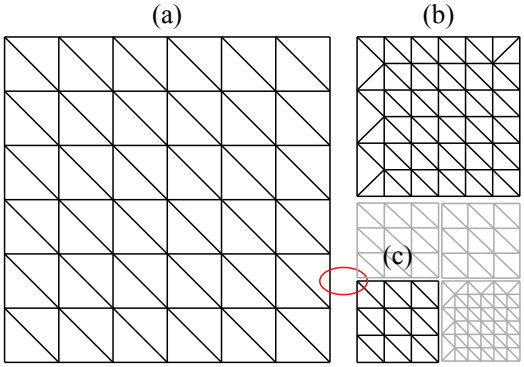


Figure 6: Non-stitchable blocks: **a)** A 7×7 block using its last LOD. **b)** A neighbor block located one tree level lower: it adapts on its left edge. **c)** A neighbor block located two tree levels lower. This block cannot adapt to block (a) as its upper-left sample (in red) is not shared by this neighbor.

6 PLANETARY TERRAINS

The previous Sections introduce our method for streaming and rendering highly detailed terrains. In this Section we propose an extension to planets by mapping a virtual terrain onto a planet-shaped ellipsoid.

6.1 Map projection

The representation of planetary information such as the 3D relief onto a 2D elevation map requires a step of projection, which can be done in numerous ways. Planets are generally modeled using a sphere mapped onto a rectangle using a cylindrical projection. However, this projection induces sampling issues: areas around the poles get far more samples than those around the equator.

Instead, we map the sphere onto the faces of a bounding cube. We use the gnomonic projection to project points from the surface of the sphere onto the corresponding tangent face (Figure 7). The main reason behind this choice is a more uniform sampling than simple cylindrical projection as well as fast 3D reconstruction. After this projection we build a tree of blocks for each side of the cube, its root block covering the entire side.

Elevation values are relative not to the surface of the cube but to the reference surface of the planet defined by a given datum (for instance, the WGS84 standard is used for the Earth). Using such a datum allows to handle spherical coordinates but reconstruct a more accurate ellipsoidal model of the planet. To render the planet correctly we first project the samples back into spherical coordinates. Then, for each sample of a block, we use the gnomonic projection to get the normalized direction of the corresponding point on the surface from the center of the planet. We then apply the datum formula and add the de-quantified elevation value of the sample to get its 3D coordinates in a planet-centric coordinate system for rendering.

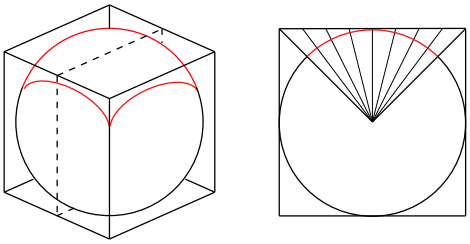


Figure 7: Mapping of a spherical planet onto a cube with the gnomonic projection. **a)** The top part of the sphere (in red) projects onto the top side of the cube. **b)** 2D cut of the cube along the dotted plane in (a). The gnomonic projection uniformly samples the cube side by mapping points from the surface of the sphere onto the plane of projection based on their direction from the sphere center.

6.2 Adjusted Gnomonic Sampling

Projecting the planet onto a cube using the gnomonic projection offers a more uniform sampling than simple cylindrical projection. However, like any map projection there are still sampling inconsistencies: the solid angle covered by a sample is approximately 75% smaller around the corners of the face than around the center of projection. To overcome this problem, we introduce a simple adjustment to the gnomonic projection for improved sampling quality.

Instead of sampling the plane of projection, we propose to sample the map directly in spherical coordinates with steps expressed in terms of angles (Figure 8). Using a planet-centric cartesian coordinate system whose axes are aligned with the edges of the cube, we define two leading axes for each face of the cube, with the third one passing through the center of projection of this face. We can then perform independent 2D rotations around both leading axes in $[\frac{-\pi}{4}, \frac{\pi}{4}]$ using a single angle step to obtain a 2D sampling of the surface of the planet that can be projected onto the face of the cube. The samples are then more evenly distributed on the surface of the planet: the solid angle covered by a sample is only 33% smaller around the corners of the face than around the center of projection, yielding a gain both in terms of compactness and quality.

When projected using the gnomonic projection, our samples do not map evenly onto the projection plane. We note that, in regard to a given leading axis, the 1D coordinate of the projected sample is the tangent value of the angle formed by the sample, the center of the planet and the center of projection (see Figure 8(b)). Therefore our solution allows us to store sample values within a simple uniform 2D map. The coordinates of any sample on the gnomonic plane in $[-1,1]$ can then be obtained by computing the tangent of the two angles. Another advantage of this method is its straightforward integration within our crack removal solution.

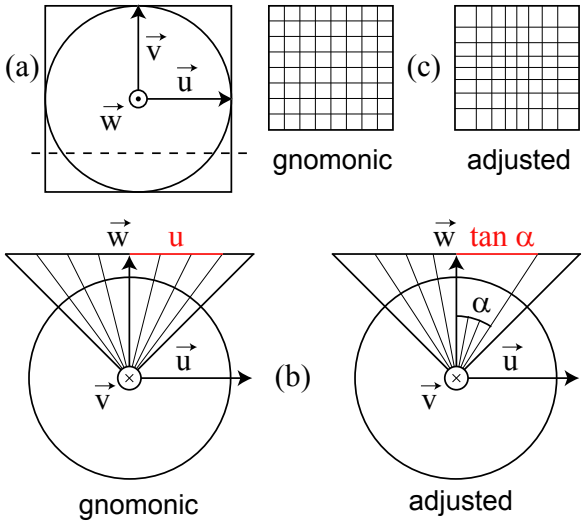


Figure 8: Adjusted gnomonic sampling. **a)** Top view of the cube. We use vectors $\vec{u}$ and $\vec{v}$ to parameterize the top face. **b)** 2D side cuts of the cube along the dotted line from (a). Gnomonic projection: the line of projection is uniformly sampled by translating along $\vec{u}$. Adjusted projection: the surface of the circle is uniformly sampled by rotating around $\vec{v}$. **c)** Top views of the face with projected samples. Gnomonic: the face is a regular 2D map. Adjusted: we use the tangent of the angles to get the coordinates of a sample in the plane of projection.

6.3 Geometry cracks between maps

Our adjusted gnomonic sampling scheme allows for high quality terrain sampling of each side of the cube. However, as the sides of the cube correspond to different maps, classical crack artifacts may appear when using heterogeneous resolutions across edges. This section explains how our projection can benefit from our crack removal solution (Section 5).

The core of the method is based on numbering and orienting the faces of the cube in a way that facilitates neighborhood management: that is, we want to ensure that blocks are ordered the same way on both sides of a given edge. This would allow us to use the same system as for neighbors that are part of the same map. However, neighbor maps cannot have the same $\vec{u}$ or $\vec{v}$ axis on their common edge in all cases because they are the sides of a cube. Figure 9 presents how we organize the sides of the cube to match the cube edge vectors, and Table 1 gives the neighborhood correspondences between sides.

6.4 Rendering and culling improvement

Graphics hardware use *near* and *far* clipping planes to bound the depth of rendered geometry, and we also take these planes into account when performing view frustum culling. In the case of a planet, which is a very large body, we want the *far* plane to be far enough to ensure

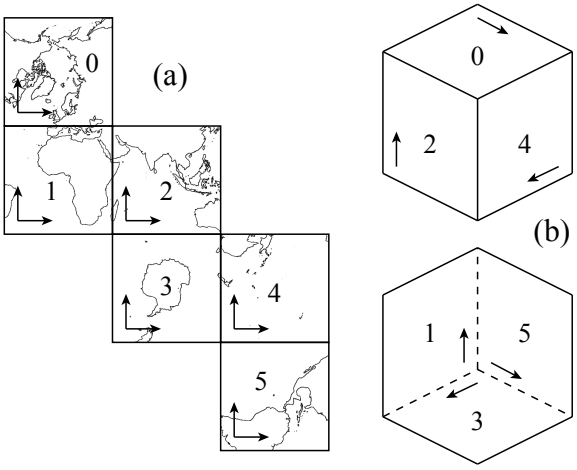


Figure 9: Orientation and numbering of the faces of the cube. **a)** Pattern of the cube with continents of the Earth for reference. Horizontal axes: $\vec{u}$, vertical axes: $\vec{v}$. **b)** Side view of the cube (top: visible side, bottom: hidden side). $\vec{v}$ axes are placed near the "left" edge of the face ($u = 0$).

Face #	Edge	Neighbor #	Neighbor edge
Even	Left	+5	Right
	Top	+4	
	Right	+2	Top
	Bottom	+1	
Odd	Left	+4	Bottom
	Top	+5	
	Right	+1	Left
	Bottom	+2	

Table 1: Neighborhood correspondences between the faces of the cube. Left, top, right and bottom respectively refer to edges where $u = 0$, $v = 1$, $u = 1$ and $v = 0$. Neighbor face number is obtained by adding the given value to the current face number, modulo six.

that no visible part of the planet is abusively ignored. However, using an arbitrary large *near-far* difference make poor use of the depth buffer rendering precision and causes flickering problems. Moreover, when placing the *far* plane behind the planet, all the surface that is visually culled by the planet itself (the grayed area in Figure 10) is uselessly rendered. We avoid those issues by adapting both planes to the position of the viewpoint. Clipping planes are orthogonal to the viewing direction and are defined by their distance to the viewpoint. Since we manipulate only distances and the planet can be approximated using a sphere, we may work in the plane defined by the center of the planet, the viewpoint position and the viewpoint direction vector.

The *far* distance corresponds to the horizon and is computed as Figure 10 explains. Equations 1 present how we compute the desired distance $\|VK\|$. Using this method, the rendered surface gets smaller as the viewpoint gets closer to the planet: the adaptive solution can

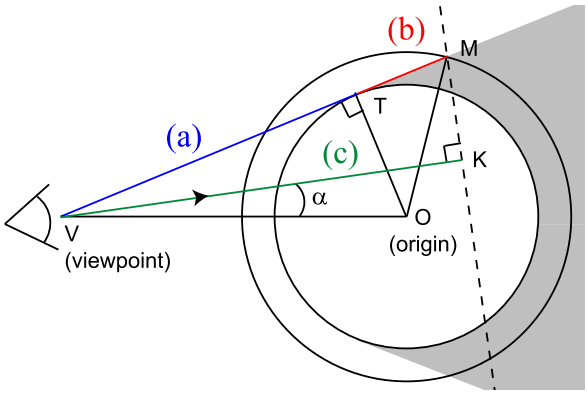


Figure 10: Horizon culling: we compute the distance $\|VK\|$ to the far plane. KM is the 2D projection of this plane, anything behind it is not rendered. The grayed area is visually occluded by the planet itself. **a)** We first compute $\|VT\|$, based on the minimum planet radius $\|OT\|$ and the distance $\|VO\|$ between the viewpoint V and the center of the planet O. **b)** We then add the constant $\|TM\|$ based on the minimum and maximum planet radii $\|OT\|$ and $\|OM\|$. **c)** We then project $\overrightarrow{VM}$ onto the viewing direction to obtain $\|VK\|$.

then use more data to improve the quality of visible and important areas.

$$\begin{aligned} \|VK\| &= \|VM\| \times \cos \widehat{MVK} \qquad (1) \\ &= \|VM\| \times \cos(\widehat{TVO} - \alpha) \\ &= (\|VT\| + \|TM\|) \times \\ &\quad \left(\frac{\|VT\|}{\|VO\|}\cos\alpha + \frac{\|OT\|}{\|VO\|}\sin\alpha\right) \qquad (2) \end{aligned}$$

We compute the *near* distance in a much simpler manner: the minimum distance to the planet is $\|VO\| - \|OM\|$. We tune this value using the field of view to avoid culling parts of the planet on the screen corners.

7 RESULTS

We tested the proposed methods on large global datasets available online: CGIAR-CSI SRTM for elevation with 90m definition, and Unearthed Outdoors TrueMarble^{TM}for color with 250m definition. Those maps use the *plate carrée* projection: we reprojected the data before creating our terrain files (Table 2).

The re-projected datasets are 25% smaller than the source due to the more uniform sampling that filters out most of the redundant data. In addition, we extended the server file format to support variable size LODs and added simple and fast LOD compression using Zlib to save disk and network bandwidth, leaving decompression on the client side.

Once we built the files on the server, we connected a client equipped with a 3.2GHz Core 2 Duo and a GeForce 9800GTX for a real-time 3D interactive Earth walkthrough using a screen resolution of 1680×1050

Input dataset		Projection		File creation	
Name	Size	Time	Size	Time	Size
SRTM	174G	9h	127G	4h50	15G
TrueMarble	42G	1h40	31G	40m	7G

Table 2: Preprocessing results. Re-projection uses about the same definition as the source maps. Without texture filtering, we obtained a 6GB file in 30 minutes for TrueMarble. The SRTM dataset is actually 58GB large but provides incomplete data which we filled with zeroes, yielding a global virtual 174GB input. Computed files contain a 11-level quad-tree of 53×53 blocks for each cube side for SRTM, and a 8-level quad-tree of 168×168 blocks per face for TrueMarble. We use two LODs per block to get uniform block update times in performance tests.

pixels and a network bandwidth of 1Mbps as shown on first page. With a 2 million polygons budget per frame, the terrain renders at 39.3 frames per second with fixed cracks. This is a small 4% decrease from rendering with cracks due to the additional triangle strips. About 5 to 10% of the rendering time is consumed by the generic solution, the rest being used for rendering.

We also tested the speed of the block update operations that occur when a new LOD is received. With texture filtering, creating a 168×168 LOD takes 0.89ms: this is only 6% more than without filtering. Using our planet projection adjustment, reconstruction of the new 3D vertices of a 53×53 LOD from elevation samples takes 0.57ms, compared to 0.33ms with plain gnomonic projection and 0.12ms when simply elevating samples from a plane. This is an important increase but those times are still negligible compared to network latency, even on lower performance clients. In addition, update operations do not directly interfere with rendering smoothness because they run in a separate thread.

8 CONCLUSION

This paper provides a full-featured solution for real-time rendering of arbitrarily large terrain datasets within a client-server context over the internet. We proposed solutions to critical issues of high quality terrain rendering: adaptive texture mapping, removal of geometry cracks and support of planetary terrains.

Our adaptive photometry scheme introduces a dual multi-resolution data structure for high definition texture representation. Our solution yields high performance storage, streaming and rendering.

Artifacts known as "cracks" tend to appear in many multi-resolution terrain rendering methods. We introduce *edge strip masks*, an inexpensive and robust method for cracks removal based on data masks.

As our technique is able to manage fully-detailed, planet-sized terrains, we propose an adjusted gnomonic sampling scheme for storing and rendering planetary terrains accounting for the actual planet shape. We also propose specific improvements for rendering virtual planets on graphics hardware.

REFERENCES

[CGG+03] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet-sized batched dynamic adaptive meshes (P-BDAM). In *Proceedings of IEEE VIS*, pages 147–155, 2003.

[CH06] Malte Clasen and Hans-Christian Hege. Terrain rendering using spherical clipmaps. In *Proceedings of EuroVis*, pages 91–98, 2006.

[dB00] Willem H. de Boer. Fast terrain rendering using geometrical mipmapping. In *Unpublished and only available at http://www.flipcode.com/articles/article_geomipmaps.pdf*, 2000.

[Hwa05] Lok M. Hwa. Real-time optimal adaptation for planetary geometry and texture: 4-8 tile hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):355–368, 2005.

[Lev02] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings of VIS*, pages 259–266, 2002.

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

[LKES07] Yotam Livny, Zvi Kogan, and Jihad El-Sana. Seamless patches for GPU-based terrain rendering. In *Proceeding of WSCG*, pages 201–208, 2007.

[LMG09] Raphael Lerbour, Jean-Eudes Marvie, and pascal Gautron. Adaptive streaming and rendering of large terrains: A generic solution. In *Proceedings of WSCG*, 2009.

[MB03] Jean Eudes Marvie and Kadi Bouatouch. Remote rendering of massively textured 3D scenes through progressive texture maps. In *Proceedings of IASTED*, pages 756–761, 2003.

[PG07] Renato Pajarola and Enrico Gobbetti. Survey on semi-regular multiresolution models for interactive terrain rendering. *The Visual Computer*, 2007.

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

[TMJ98] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *Proceedings of SIGGRAPH*, pages 151–158, 1998.

[WMD+04] Roland Wahl, Manuel Massing, Patrick Degener, Michael Guthe, and Reinhard Klein. Scalable compression and rendering of textured terrain data. In *Proceedings of WSCG*, pages 521–528, 2004.