Adaptive streaming and rendering of large terrains

Raphaël Lerbour

To cite this version:

Raphaël Lerbour. Adaptive streaming and rendering of large terrains. Human-Computer Interaction [cs.HC]. Université Rennes 1, 2009. English. <tel-00461667>

HAL Id: tel-00461667

https://tel.archives-ouvertes.fr/tel-00461667

Submitted on 5 Mar 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre : 3991 **ANNÉE 2009**

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Raphaël LERBOUR

préparée à l'unité de recherche 6074 IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
Composante universitaire : IFSIC
En contrat CIFRE avec Technicolor/THOMSON R&D France

Chargement progressif et rendu adaptatif de vastes terrains

Thèse soutenue à Rennes le 17 décembre 2009

devant le jury composé de :

Pascal GUITTON
Professeur, univ. Bordeaux 1 / *rapporteur*

Bernard PEROCHE
Professeur, univ. Lyon 1 / *rapporteur*

Eric MARCHAND
Professeur, univ. Rennes 1 / *examinateur*

Christian BOUVILLE
Ingénieur R&D, France Télécom / *examinateur*

Kadi BOUATOUCH
Professeur, univ. Rennes 1 / *directeur de thèse*

Jean-Eudes MARVIE
Ingénieur R&D, Thomson / *co-directeur de thèse*

Acknowledgements

I would first like to cordially thank Kadi Bouatouch for his encouragements and advices which helped me so much during the difficult times of my Ph.D. On the other side, Jean-Eudes Marvie was my source for enthusiasm and support for the rest of the time. I also thank Jean-Eudes for giving me initial research directions, and for having developed the software framework which greatly simplified my implementation work. Many thanks to Technicolor (formerly Thomson), the company that employed me for the last three years to support my Ph.D. studies through a CIFRE contract. I also want to thank Daniel Meneveaux who introduced me to Kadi Bouatouch to find my Ph.D. in Rennes.

During all my Ph.D., I have been working in a R&D project of growing size in Technicolor Rennes. Special thanks go to Pascal Gautron, who regularly helped me maintain the link between industrial work, research work and personally rewarding work. His advices were also of great value for preparing and writing scientific publications. I would also like to thank all other team members, in particular Gérard Briand, François-Louis Tariolle, Patrice Hirtzlin and Sébastien Maraux who helped me either technically or professionally. Last but not least, I want to thank the interns who shared my office six months a year: Serigne Dieng, Clémentine Havrez, Emmanuel Trivis, Benoît Saillard, Cyril Delalandre, Julien Nicolas, Thomas Piscitelli, Emilie Bosc and Dan Que Le. Their company has been very enjoyable, refreshing and motivating.

I would like to thank the members of my thesis comittee, in particular Pascal Guitton and Bernard Péroche who reviewed my manuscript, as well as Eric Marchand and Christian Bouville.

Many thanks to my friends who supported me morally during this period. In particular, Morgane accompanied me during the first year I spent in an unfamiliar context of life. Her presence and sincerity touched me at heart. I would also like to thank Gabriel, as well as all the thursday night drafters and quite unusual barflies from “Le Croix du Sud”; those guys are great and cannot know how much support they were for me.

Finally, I would like to thank my parents for their support in countless phone calls, and my brothers Benjamin and Baptiste because they are simply the best.

Contents

Contents **1**

1 Introduction **5**
- 1.1 Motivations . . . 5
- 1.2 Contributions . . . 5
  - 1.2.1 Generic adaptive data streaming and selection . . . 6
  - 1.2.2 3D rendering of large terrains and planets . . . 7
  - 1.2.3 Preprocessing huge sample maps . . . 8

2 Related work **9**
- 2.1 Main concepts of level of detail for terrain geometry . . . 9
  - 2.1.1 Triangulation constraints . . . 11
  - 2.1.2 Data structures and adaptive algorithms . . . 12
    - 2.1.2.1 Continuous frame-to-frame triangle-level simplification 13
    - 2.1.2.2 Per-block discrete level of detail selection . . . 14
    - 2.1.2.3 Multi-resolution hierarchy of blocks . . . 14
    - 2.1.2.4 Other approaches . . . 15
  - 2.1.3 Memory management and data streaming . . . 16
- 2.2 Technical issues . . . 17
  - 2.2.1 Rendering artifacts handling . . . 17
    - 2.2.1.1 Cracks . . . 18
    - 2.2.1.2 Popping . . . 19
  - 2.2.2 Parallel computing . . . 20
  - 2.2.3 View-frustum culling . . . 20
  - 2.2.4 Optimization with graphics hardware . . . 21
    - 2.2.4.1 Triangle stripping . . . 21
    - 2.2.4.2 Caching in graphics hardware . . . 22
    - 2.2.4.3 Computations on GPU . . . 23
  - 2.2.5 Preprocessing . . . 23
- 2.3 Common features . . . 23
  - 2.3.1 Texture maps . . . 24
  - 2.3.2 Planetary terrains . . . 25
  - 2.3.3 Compression . . . 26

2.3.4 Procedural detail 26
2.4 Available softwares 27
2.5 Discussion 28

3 Generic solution for adaptive data streaming and selection 31
3.1 Overview 31
3.2 Generic data structure 32
3.2.1 Advantages 33
3.2.2 New properties 34
3.2.3 Parameters 35
3.3 Server database 36
3.3.1 First approach: implicit file organization 36
3.3.2 Second approach: explicit level of detail size and position 37
3.4 Partial client database 38
3.4.1 Split and merge operations 39
3.4.2 Refine operation 40
3.5 Adaptive streaming and database update 41
3.5.1 Progressive loading 41
3.5.2 Asynchronous database updates 42
3.6 Adaptive level of detail selection for rendering 43
3.6.1 Culling blocks that are not visible 43
3.6.2 Measure of importance 44
3.6.3 Level of detail selection 45
3.6.4 Database update triggers 46
3.6.5 Masks for levels of detail extraction 48
3.7 Results 48
3.8 Conclusion 52

4 Application to 3D rendering of large terrains and planets 55
4.1 Overview 55
4.2 From elevation map to 3D geometry 56
4.2.1 Data conversion 56
4.2.2 Triangle strip masks 58
4.2.3 Caching in graphics hardware 58
4.2.4 Measure of importance 59
4.3 Fixing geometry cracks on block edges 61
4.3.1 Additional adaptive triangle strip masks 61
4.3.2 Data structure addition: neighbor blocks 63
4.3.3 Avoiding unsolvable cases 64
4.4 Texture maps 65
4.4.1 Data structure addition: bound trees 65
4.4.2 Measure of importance 66
4.4.3 Rendering 66
4.4.4 Filtering 67

4.4.5 Interpolation continuity problems on block edges 68
4.5 Planetary terrains 68
4.5.1 Gnomonic map projection on a cube 69
4.5.2 Adjustment for improved sampling 71
4.5.3 Fixing geometry cracks on cube edges 73
4.5.4 Adaptive clipping planes for improved precision and culling . . . 74
4.5.5 Discussion on other precision problems 76
4.6 Results 77
4.7 Conclusion 80

5 Preprocessing: building server files from huge input maps 81
5.1 A useful tool: dually tiled map format 82
5.2 Re-projection of a planetary map 83
5.2.1 Sample re-projection 84
5.2.2 Entire cube face map re-projection 84
5.3 Generation of the server database 86
5.4 Results 87

6 Conclusion 91
6.1 Contributions 91
6.1.1 Generic adaptive data streaming and selection 91
6.1.2 3D rendering of large terrains and planets 92
6.2 Future work 92
6.2.1 Generic solution improvements 92
6.2.2 3D rendering improvements 93
6.2.3 Further research 93

A Test databases 95

Bibliography 103

Résumé en français 107

Chapter 1

Introduction

1.1 Motivations

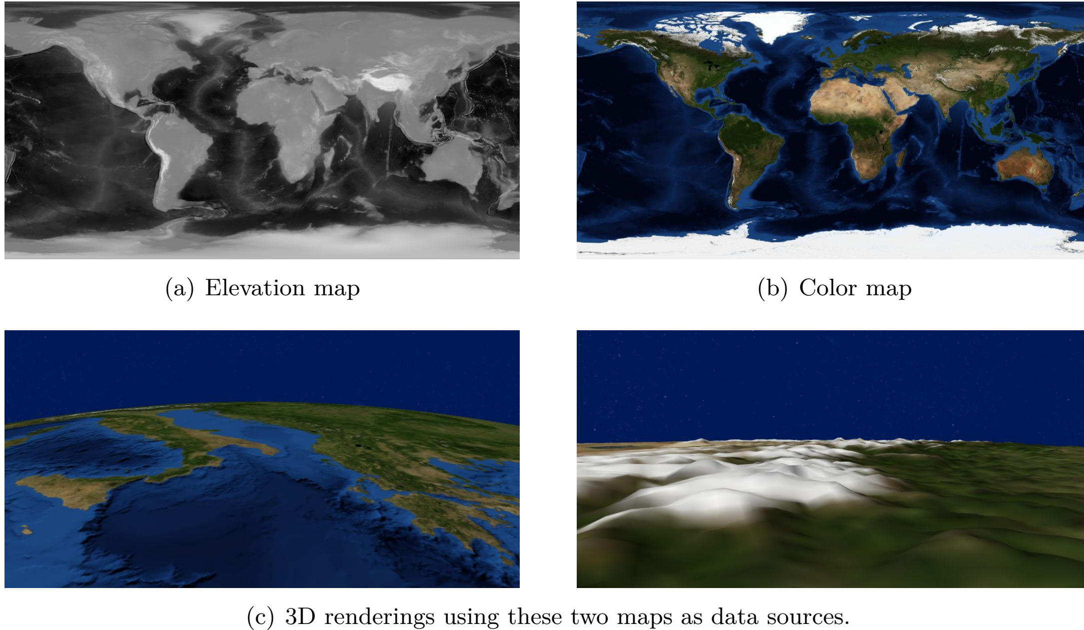
In this thesis, we propose solutions to perform adaptive streaming and rendering of arbitrary large terrains. One useful application is, for instance, to interactively visualize the Earth in 3D with great detail on any computer device while loading the required data from a huge database over the Internet. This is interesting for domains like 3D GPS (Global Positioning System) navigation devices, virtual tourism, and video games like flight simulators or multi-player games with very large and seamless playable areas. Currently, *Google Earth* and *NASA World Wind* are popular Internet-based terrain rendering softwares oriented for virtual tourism, using incredibly detailed databases.

A digital terrain is made of 2D maps of samples regularly discretizing the surface of the terrain. They are usually elevation maps used to reconstruct the terrain relief in 3D along with color maps that represent the photometric details on the surface, as shown in Figure 1.1. Such maps may be huge; for instance a global map of the Earth with a definition of only 500 meters between samples is over ten gigabytes, while the highest definition global database that we have used for our tests was over 100GB with elevation and color. Such a large amount of data can neither entirely be loaded in memory nor interactively rendered in real-time on any computer. One also probably wants to be able to visualize a terrain without having to entirely download and store a huge database on a local hard disk.

To solve these problems, we need to use data structures and techniques specifically designed to support streaming and rendering of arbitrary large terrains. These are the subject of this thesis.

1.2 Contributions

We present the most significant previous works done in the field of streaming and rendering of large terrains in Chapter 2. Unlike most of them, we chose to clearly separate generic data loading, update and selection on one side, and specific data interpretation and rendering on the other side. The contents of the original work proposed in this



(a) Elevation map

(b) Color map

(c) 3D renderings using these two maps as data sources.

Figure 1.1: Maps of elevation and color samples that represent the surface of the Earth, and example resulting 3D renderings. The Earth GEBCO database is used here. See Appendix A for a presentation of all databases we use in this thesis.

thesis are organized in three main parts which are introduced in Figure 1.2 and whose contributions are summarized hereafter.

1.2.1 Generic adaptive data streaming and selection

In the first part of this thesis, presented in Chapter 3, we introduce a generic solution relying on a generic data structure. This solution adaptively handles 2D sample maps of any size from a server hard disk all the way to a client rendering system.

We base our generic data structure on well tried principles, that consist in subdividing 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. We then add new properties and techniques to handle this structure faster and more adaptively. In particular, we minimize the amount of data to store and load by avoiding handling redundant data over different blocks and LODs. We also allow a block to be rendered when not all of its LODs are available, which offers the possibility to progressively load the LODs of each block.

Using this data structure, we adaptively stream the data from the server to the client, continuously updating a partial database on the client side. We update this database in a way that prevents many costly structure operations, in favor of in-place asynchronous data updates. In addition, we prevent overloading the network and optimize the relevance of performed operations by continuously updating a queue of loading

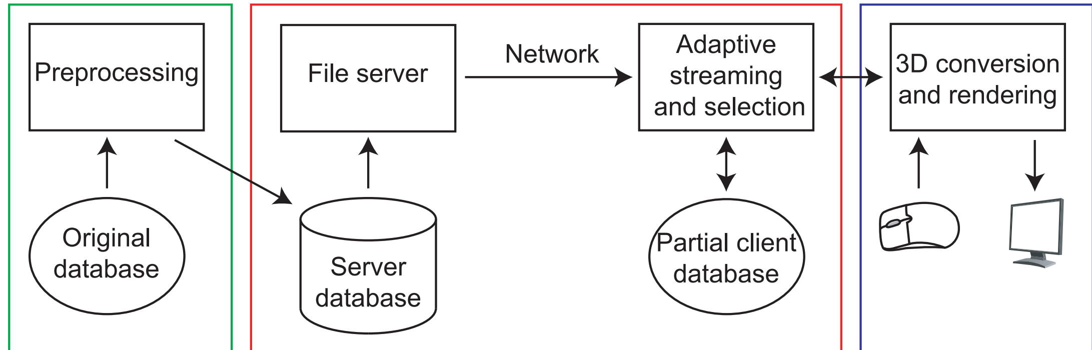


Figure 1.2: The different parts of our general solution. The green frame on the left corresponds to the preprocessing steps presented in Chapter 5. The red frame in the middle corresponds to the generic solution for data streaming and selection presented in Chapter 3. The blue frame on the right corresponds to the application of the solution to 3D rendering of large terrains presented in Chapter 4.

requests and by selecting only a few requests from this queue at a time, depending on a measure of importance. The only task of the server is to read requested data from a specifically designed file and transmit them to the client.

In a second time, we adaptively select the data to render from the partial database of the client and the missing data to load from the server. We drive this selection using a generic measure of importance. This measure depends on a general quality factor that allows adapting to a desired rendering speed or quality. This measure can also be specialized for different applications of the generic solution, depending on the nature of the data.

1.2.2 3D rendering of large terrains and planets

In the second part, presented in Chapter 4, we propose to use the generic solution for real-time 3D rendering of large terrains composed of elevation maps and texture maps. We discuss and solve specific issues and propose advanced features for this particular application of the generic solution.

We first fix the cracks that typically appear between adjacent blocks when rendering multi-resolution 3D terrains. We improve a common solution to take into account that, in our context, a block may have more than one neighbor along one edge and not all samples of a block are always available for rendering. Unlike previous solutions, we also allow adjacent blocs to use any difference in sampling definition as long as it does not produce unsolvable cracks.

We then add the possibility to handle photometry and geometry separately: we enhance the generic solution to allow rendering of textured terrain geometry by maintaining a link between a texture map and an elevation map. We also add support for sample filtering to improve the visual quality of the low resolution levels of detail.

Finally, we describe a method for rendering planetary terrains. The adaptive solu-

tion still handles elevation maps in a generic way, but we reconstruct the 3D surface of a planet. We map the planet onto the faces of a bounding cube using an improved gnomonic projection, in which the surface of the planet is sampled using regular angle steps. This projection saves much redundant data, and prevents most rendering inconsistencies compared to usual cylindrical projections. We also fix geometry cracks on the edges of the cube almost transparently. Our last contribution addresses the issue of rendering precision. A planet is a huge 3D object which cannot be directly rendered on graphics hardware without visible artifacts due to the limited precision of the depth buffer. In our method, such artifacts are reduced by adjusting the clipping planes of the view frustum in real-time. We also take advantage of those clipping planes to cull parts of the planet located behind the horizon with no additional testing.

1.2.3 Preprocessing huge sample maps

In the third and final part, presented in Chapter 5, we describe the preprocessing step required to generate a database file that can be used on the server side of the solution presented in Chapter 3, from any 2D sample map. We also describe an additional preprocessing step needed when creating a planetary database. It converts the original map into the six maps of our own planet projection. Like the rest of our methods, these steps are designed to support databases of arbitrary size.

Chapter 2

Related work

In this chapter, we present previous work done in the field of real-time adaptive terrain rendering and streaming.

We first present in Section 2.1 the main existing methods to dynamically update a simplified representation of the geometry of a terrain. This representation is used to render the terrain faster and with less memory consumption than when performing brute force rendering of the entirety of the original terrain data. Methods presented in this section are adaptive: the simple terrain representation is updated in real-time to match user-selected speed or quality requirements and according to user interactions on the rendering viewpoint. We also present solutions for out-of-core rendering and data streaming fitting the different data structures approaches.

Most adaptive terrain geometry methods lead to some visual quality problems that should be avoided, in particular the loss of geometric continuity that appears when two adjacent terrain areas get simplified independently. In addition, these methods can be enhanced on several points when implemented in order to get faster rendering. Such technical issues are discussed in Section 2.2.

Finally, some articles in literature offer other notable additions to adaptive terrain geometry rendering, which we describe in Section 2.3. In particular, important features are the support for photometric texture maps and for planetary terrains.

While many adaptive terrain rendering solutions have been proposed in the literature, a number of them leaded to publicly available and maintained software that propose interactive visualization of impressive databases. In Section 2.4 we give a short review of the most notable ones.

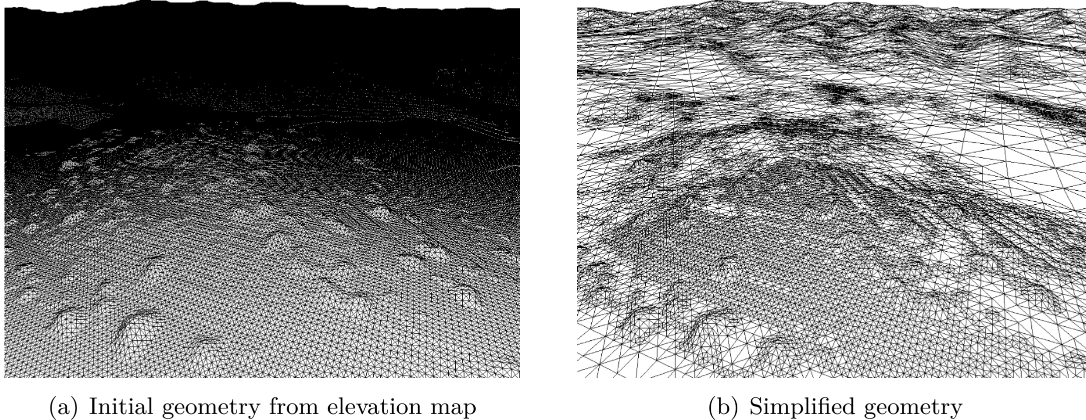
2.1 Main concepts of level of detail for terrain geometry

Terrain geometry is generally represented using rectangular and regularly sampled elevation maps (or height maps) of arbitrary large resolution and scale. Each sample has a height value that gives the elevation of the surface of the terrain on that point, relative to a reference surface like sea level. Therefore, the elevation map defines the shape of the terrain. Rendering each point of such map is in most cases impossible to

achieve in real-time, and the map cannot always be stored entirely in main memory. The goal of real-time adaptive rendering, or level of detail rendering, is to continually update a simplified geometric surface to render a large terrain map based on arbitrary constraints. These constraints can be, for instance, a minimum number of frames computed per second to control rendering speed, or a maximum screen-space error for the rendered terrain surface to control rendering quality. Screen-space error for one point of the surface is the distance in pixels between its rendered position on screen and what would be its “real” position if the full resolution geometry was rendered.

In most cases, the rendered terrain geometry is a mesh made of 3D vertices linked together with triangles, which can thus be processed with current graphics hardware. The original terrain map can be directly translated to such a mesh by using one vertex per sample and regularly triangulating the grid. Then, one can skip some vertices and adapt the triangulation to maintain a continuous terrain surface, so the mesh gets simplified and coarsened (see Figure 2.1). Conversely, when more quality is wanted for some terrain area, skipped vertices may be added again to refine the mesh. A great number of techniques based on these two operations are summarized and discussed in a recent survey [PG07], though the authors chose not to include solutions based on regular grids.

Depending on the degree of freedom allowed for coarsening and refining operations, the data structures and the algorithms that update and select data can be quite different both in complexity and rendering fidelity. In in Section 2.1.1, we present the three main families of methods concerning the constraints used for triangle-level simplification.



(a) Initial geometry from elevation map

(b) Simplified geometry

Figure 2.1: Example result of adaptive terrain rendering on one frame [LKR+96].

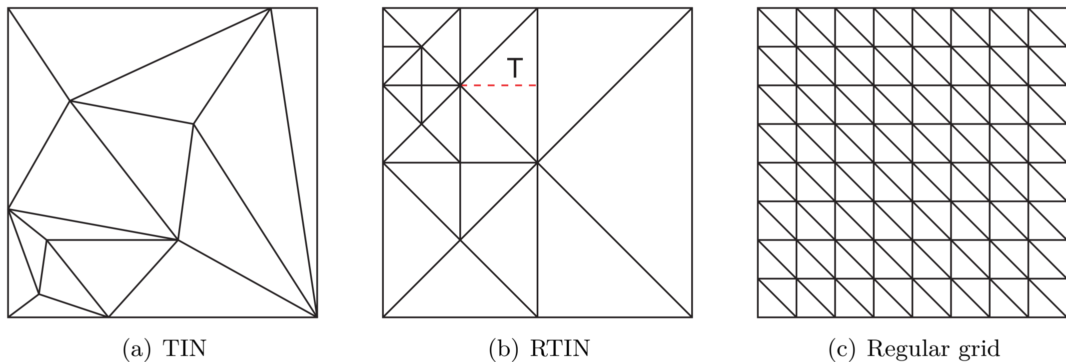
The most important considerations for efficient level of detail of terrain geometry are the choices of the data structure and of the adaptive data selection algorithms that use this structure to update a simplified terrain mesh. In Section 2.1.2, we present the main approaches proposed in the literature.

Finally, we discuss in Section 2.1.3 the possibility of achieving progressive data loading using the data structures and algorithms presented in Section 2.1.2. This point

is very important because it allows out-of-core rendering of very large terrains using a limited amount of memory, as well as data streaming over a network.

2.1.1 Triangulation constraints

In the case of 3D rendering, a terrain is a mesh made of vertices that are linked to form triangles. Initially, this mesh is a regular grid of triangles when viewed from above, with the height of each vertex computed from the elevation value of the corresponding sample in the original elevation map. The number of triangles is then reduced by adaptive level of detail algorithms, which select the ones to render. These algorithms may use some constraints for data selection in order to make the operation faster and more flexible, although this results in slightly lower visual quality for a given number of triangles. Three main types of constraints may be used, illustrated in Figure 2.2 and described in the next paragraphs.



(a) TIN (b) RTIN (c) Regular grid

Figure 2.2: Different triangulation approaches. The dotted red line shows how triangle T is split in a RTIN.

TINs (Triangulated Irregular Networks) are completely irregular structures. Vertices are represented with their three coordinates, in a pseudo-continuous space. As it has no constraint, this model gives the lowest number of triangles for a fixed screen-space error bound. However, computing a TIN is a long and complex process: every vertex of the original terrain needs to be taken into account in a bottom-up simplification algorithm, and updating the structure also requires relinking concerned vertices with updated triangles. This requires a very large number of CPU computations and is not acceptable for real-time terrain rendering. Consequently, methods that propose TINs use pre-computed update operations [Hop98] or pre-simplified sub-meshes of the terrain [CGG$^+$03a].

RTINs (Right-Triangulated Irregular Networks) is the most used model for mesh representation in adaptive terrain rendering [LKR$^+$96, DWS$^+$97, RHSS98, Paj98, Blo00, EKT01, LP01, ZZSP01, Lev02, LP02, WMD$^+$04, SW06]. The difference with TINs is that vertices are placed on the same regular grid as the original terrain map, and all the triangles are right isosceles triangles when projected onto this 2D grid. This allows

for an easier and much faster data update process based on triangle split and merge, but more triangles are needed to represent a given terrain than with TINs for the same screen-space error bound. Splitting a triangle is performed at the center of its hypothenuse (along the dotted red line in Figure 2.2), creating two new and smaller right isosceles triangles; merging a triangle is the reverse operation. A RTIN is commonly represented with a binary tree that corresponds to the recursive triangle splits starting with the very coarse mesh obtained from only the corners of the terrain map. In this way, both the selected vertices and the triangles linking them may be obtained with a top-down algorithm which can avoid taking skipped vertices into account. Using this kind of algorithm, the complexity does not depend on the size of the original terrain map.

Another, yet more constrained method is to place a vertex on every point of a regular 2D sub-grid of the original terrain map [dB00, LC03, PM05, Bro05, HDJ05, LN05, GMC+06, LKES07]. The number of vertices is defined by the size of the grid. This method has a poor ratio between screen-space error and number of triangles (due to the Nyquist-Shannon theorem about regular down-sampling), but it directly gives the simplified mesh with almost no computation needed. However, using grids that cover the whole terrain would prevent any local adaptivity. This method is thus generally combined with a tiling of the terrain into smaller square or right isosceles triangular sub-meshes (usually called “tiles”, “blocks” or “patches”) which may use different grid resolution each. Another solution is to use concentric terrain areas around the viewpoint that use grids with increasing definition [LH04, AH05]. Regular grids are particularly well suited for cases where one can render a large amount of triangles at reasonable speed on graphics hardware and wants to save processing time on the CPU. As we can see, works using this method are the most recent ones, which is logical considering the huge increase in processing power of graphics hardware in the past years.

Note that it is important not to confuse definition and resolution in the case of regular grids. The resolution of a map or grid of samples is the number of samples it contains, while its definition is the number of samples that are used to represent a unit of the surface of the terrain. Therefore, blocks with identical resolution have different definition when they cover surfaces of different size.

2.1.2 Data structures and adaptive algorithms

We can classify the majority of the existing algorithms for real-time adaptive terrain rendering into three main families, based on the compromise made between speed and local adaptivity.

Methods presented in Section 2.1.2.1 apply their algorithms using triangles or vertices as the base unit and treat the whole terrain. These methods favor rendered geometry fidelity using advanced per-triangle screen-space error metrics. Additionally, some of these methods tile the terrain into independent blocks and apply level of detail algorithms on the geometry inside these blocks. When using a technique to discard invisible blocks and/or when ordering blocks of different sizes in a hierarchical structure, this considerably reduces the number of operations.

In Section 2.1.2.2, we present methods that tile the terrain using fixed-size blocks with a set of predefined levels of detail (LOD) each. One then only needs to choose one level of detail per block and use that simplified mesh to render the block, with no need to explicitly perform simplification operations. The drawback is that geometry is updated by batches so the level of detail switches are more visually noticeable.

In Section 2.1.2.3 we present methods that apply level of detail algorithms with blocks of fixed geometry as the base unit instead of triangles, leading to a much smaller amount of processed objects and thus reducing the CPU load. The geometry of these blocks is computed when data are loaded or in a preprocessing step, but the resulting mesh is then cached for rendering and not updated until the block is split or merged. The drawback is that the geometry inside blocks is not adaptive: a compromise must be made concerning block resolution.

Some methods cannot be classified into these families because they use original approaches that make other compromises. We present them concisely in Section 2.1.2.4.

2.1.2.1 Continuous frame-to-frame triangle-level simplification

With the algorithms in [LKR+96, DWS+97, RHSS98, Hop98, Paj98, Blo00, LP01, ZZSP01, LP02], every single triangle is candidate for splitting or merging. Therefore, there can be really small and local changes in the geometry from one frame to another. This is called continuous levels of detail (CLOD).

In addition, [Paj98, ZZSP01] use fixed-size blocks that get simplified independently. In practice, the geometry of each block is not really computed continuously in [Paj98]: it is computed only once, then updated when the screen-space error exceeds a given threshold. This is a first step towards discrete levels of detail (DLOD) presented in Section 2.1.2.2.

Finally, [LKR+96, Hop98] use a hierarchical structure of terrain blocks that is top-down searched to get an initial triangulation before computing geometry inside the chosen blocks. This is a first step towards hierarchical levels of detail which are detailed in Section 2.1.2.3.

There are several approaches to obtain visual errors that guide geometry simplification. One can evaluate the screen-space error for each possible update operation at each frame [LKR+96, Paj98, ZZSP01]. Alternatively, one can use pre-computed world-space error values to speed up error computation [DWS+97, Hop98, RHSS98]. In addition, [DWS+97] uses priority queues to select the update operations to be performed from errors. Those queues are updated incrementally frame by frame, thus reducing the number of computations.

A new and more efficient method is introduced in [Blo00, LP01]. It uses a pre-computed hierarchy of nested virtual spheres, corresponding to the RTIN hierarchy of triangles. Each sphere corresponds to a possible split or merge operation. It is centered at the concerned vertex and its diameter is proportional to the corresponding error. Whenever the viewpoint enters or quits a sphere the associated split or merge operation is performed respectively. Since those spheres are organized in a hierarchy, spheres nested in another sphere are never tested until the viewpoint enters the latter.

2.1.2.2 Per-block discrete level of detail selection

Another approach to adaptive terrain rendering is to tile the terrain into fixed-size blocks and use a set of LODs with different resolution to render these blocks [dB00, LC03, PM05, Bro05, SW06]. Adaptivity is obtained by switching between LODs; this is called DLOD (discrete levels of detail). LODs may be pre-computed and stored as batches of geometry, or quickly generated on-the-fly using sample masks. Compared to continuous LOD solutions, this method saves a large number of operations but offers less precision and locality for data selection. This is convenient in the current context with high processing power in graphics hardware, as noticed when presenting regular grids in Section 2.1.1. Note that it is not possible to obtain triangles larger than the block size when using fixed-size blocks, thus sometimes forcing to render unnecessary details. We can also note that this method requires loading all blocks if one wants to render all parts of the terrain at the same time.

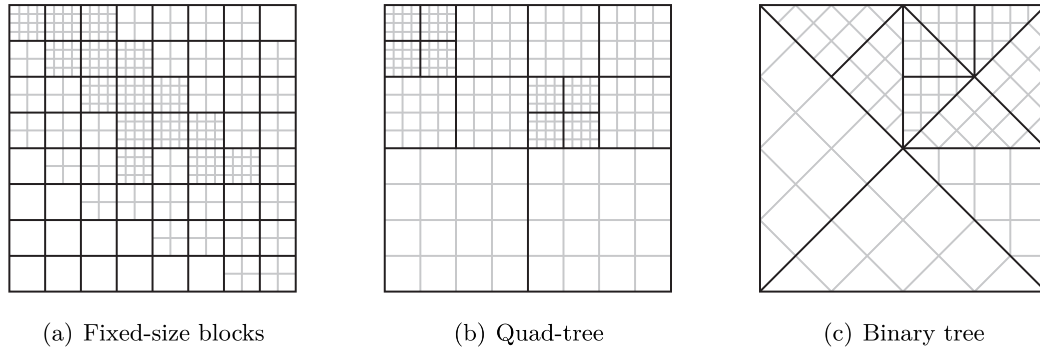
Using this method, the visual error used to select LODs is computed per block and not per vertex. This introduces many approximations to replace per-vertex screen-space error, like a pre-computed block roughness factor [PM05], a block-wide screen-space projection [SW06] or suppositions regarding the viewpoint angle to get pre-computed errors [dB00].

2.1.2.3 Multi-resolution hierarchy of blocks

Another approach of block-based terrain level of detail, which offers more adaptivity than fixed-size blocks, is to use a hierarchy of nested blocks of recursively smaller and smaller size [Lev02, CGG+03a, CGG+03b, WMD+04, HDJ05, LN05, GMC+06, LKES07]. These blocks may be rectangular blocks in a quad-tree or triangular blocks in a bin-tree as shown in Figure 2.3. Each block corresponds to a single pre-computed geometry level of detail. Like a triangle in continuous LOD solutions, a block can be refined by splitting it into a greater number of smaller blocks, or coarsened by merging it with its neighbors into a unique and larger block. All blocks get about the same number of triangles: when a block is split, more blocks and therefore more triangles are used to render the same terrain area. [LKES07] combines both the hierarchical structure and the possibility to select a LOD among multiple ones for each block.

This hierarchical approach allows to adaptively choose block sizes depending on the desired level of detail. This addresses the main drawbacks of using fixed-size blocks: only a very limited number of blocks is needed to display the whole terrain, and more blocks are loaded only when more detail is needed.

Mainly block-based errors are used to guide block split and merge operations. One can use a hierarchy of nested virtual spheres like [Blo00, LP01], but using only one sphere per block and a direct correspondence with the block hierarchy [Lev02, CGG+03a]. [HDJ05, GMC+06] update operations are selected using priority queues like [DWS+97].



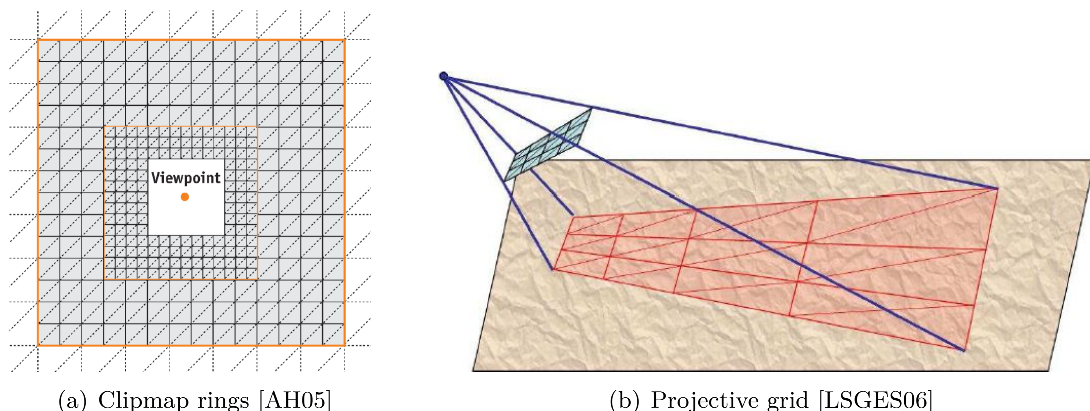
(a) Fixed-size blocks (b) Quad-tree (c) Binary tree

Figure 2.3: Subdividing an elevation map into blocks. Each block (black boundaries) renders using its own geometry (gray grids).

2.1.2.4 Other approaches

Several different approaches have been proposed in the literature, the most important of which are presented in this section.

Using the clipmap [LH04, AH05, CH06], introduced in [TMJ98] for texture maps and presented in Figure 2.4(a), one chooses successive LODs for rectangular "rings" of terrain around the viewpoint in a way similar to the mipmap for texture maps levels of detail. Each LOD represents the whole terrain but only a subset of it is rendered at a time, covering a given window that moves along with the viewpoint. LODs are selected only depending on the camera distance, thus preventing any other approach to error computation. This method is very limited in terms of flexibility and adaptivity, but has the advantage of being very fast because LODs are regular 2D grids of vertices updated incrementally and there is no vertex of block tree structure to handle.



(a) Clipmap rings [AH05] (b) Projective grid [LSGES06]

Figure 2.4: Other approaches to adaptive terrain rendering.

The projective grid method and variants [DS04, LSGES06, LRS09], presented in

Figure 2.4(b), use the same number of vertices at any time because it always uses the same grid of vertices to render the terrain. In practice, vertex coordinates are computed by projecting a regular grid from the screen to the terrain then using elevation values to transform the projected grid to a 3D model of the terrain. Therefore, although vertices are regularly ordered and triangulated, their spatial distribution is not regular at all.

[CH06] mixes both approaches by mapping a regular viewpoint-centered grid of vertices to a global rectangular clipmap, using concentric circular rings of vertices to select LODs.

2.1.3 Memory management and data streaming

Continuous level of detail solutions consider every vertex of the terrain for update operations, therefore need all the geometry at its highest LOD to be potentially available at runtime. To achieve good performance for data selection, it is desirable to have all these data in main memory. This is a problem when visualizing very large landscapes because they cannot fit in memory, thus leading to slow paging operations performed by the operating system. This also prevents progressive data loading. However, [LP01] proposes a special memory management and data organizing scheme that minimizes paging due to good data locality.

All solutions using blocks of data allow to independently load blocks from a hard disk or via a network only when needed, therefore enabling out-of-core rendering and data streaming possibilities.

However, using fixed-size blocks, all blocks need to be loaded in order to display the whole terrain, even in its coarsest form. A solution is to use a window following the viewpoint [Paj98, PM05] as illustrated in Figure 2.5. Blocks are successively loaded when this window moves over them, and may be discarded when they get outside the window. In [PM05], the size of the window progressively increases as blocks are loaded. The main drawback of this method is that far away blocks are not loaded; this creates an artificial horizon and terrain parts can suddenly appear in the distance when the viewpoint is moving.

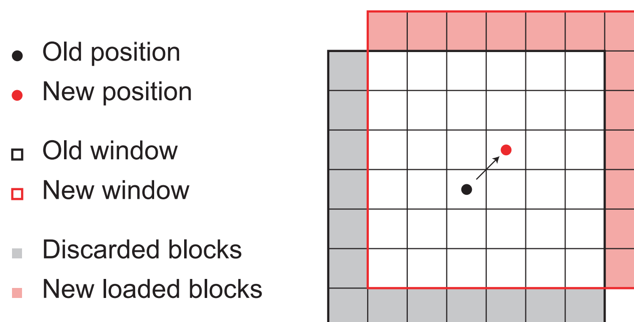


Figure 2.5: Moving window of loaded blocks, centered at the viewpoint. Discarded blocks may be cached for later use.

In the methods using a hierarchy of blocks with pre-computed geometry [CGG$^+$03a,

WMD+04, GMC+06], the whole terrain can be immediately rendered in a very coarse form using the first levels of the tree. Then, one can progressively load blocks of lower tree levels using split operations where more quality is needed.

The clipmap [LH04, AH05, CH06] is an out-of-core approach by definition since high definition LODs are not completely stored in memory. They are updated when the viewpoint moves using a moving window similar to the one presented in Figure 2.5. If a part of a high definition LOD is not loaded, a lower LOD may be used to render the terrain area.

Some additional solutions may be used to improve progressive data loading efficiency. [CGG+03b, WMD+04, SW06, GMC+06] use prefetching in order to get most of the blocks or LODs before they are needed, thus reducing loading latency in most cases. [Hop98, SW06, GMC+06] and the clipmap-based articles represent new LODs based on the previous one and a limited data complement, thus minimizing redundancy in loaded data. Finally, [Hop98] stores and loads each triangle operation independently, based on the previously loaded geometry. This is called progressive meshing and allows streaming of continuously refined geometry. However, because of the large number of operations entailed, CPU and bandwidth overhead can be high.

2.2 Technical issues

The previous section presented the main ways of performing real-time adaptive terrain rendering. However, every proposed solution must face some problems and choices to get practically useable.

For instance, removing or adding vertices to a simplified mesh without special handling can create rendering artifacts. Such issues are discussed in Section 2.2.1.

Also, every part of the computer involved in the rendering should be used efficiently and the quantity of data to handle at each frame should be minimized in order to achieve as good performance as possible. In this section, we present the most important optimizations such as parallel computing or efficient usage of graphics hardware.

2.2.1 Rendering artifacts handling

Two main artifacts are encountered when performing adaptive terrain rendering: cracking and popping.

Cracks show up when the continuity of the terrain mesh is not preserved. These are gaps in the geometry that appear between adjacent triangles that do not share exactly the same edge, or at block edges when using independent geometry blocks of different definition. Different ways to handle this problem are presented in Section 2.2.1.1.

Popping is the undesirable effect of sudden changes in the shape of the terrain when switching levels of detail, especially when using discrete LODs. Some solutions for this problem are described in Section 2.2.1.2.

2.2.1.1 Cracks

Cracks generally appear with tiled data structures, when the two corresponding edges of two adjacent blocks do not have the exact same shape because simplification is performed independently for each block. Cracks can also appear inside blocks if triangle-based simplification is not correctly managed. There are some very different families of solutions that ensure a continuous mesh and thus avoid cracks.

[LKR+96, DWS+97, RHSS98, Paj98, Blo00, LP01] use runtime dependencies systems between vertices or triangles. For instance, [LKR+96] uses flags on vertices that tell whether a split is possible, and updates those flags for every operation. In [DWS+97, Blo00], when a split is needed that would lead to a crack, splits are forced recursively in the neighborhood until there is no crack (see Figure 2.6).

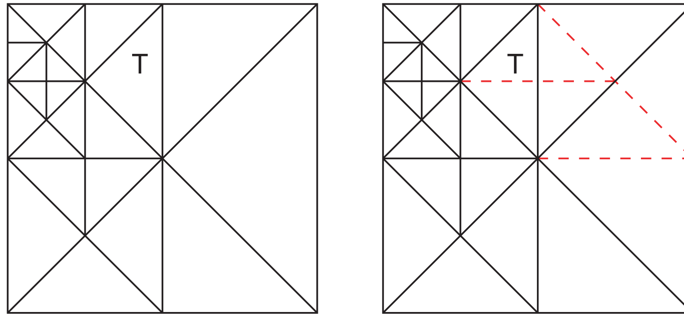


Figure 2.6: Recursive forced splits to avoid cracks. We want to split triangle T: split operations corresponding to the dotted red lines are forced.

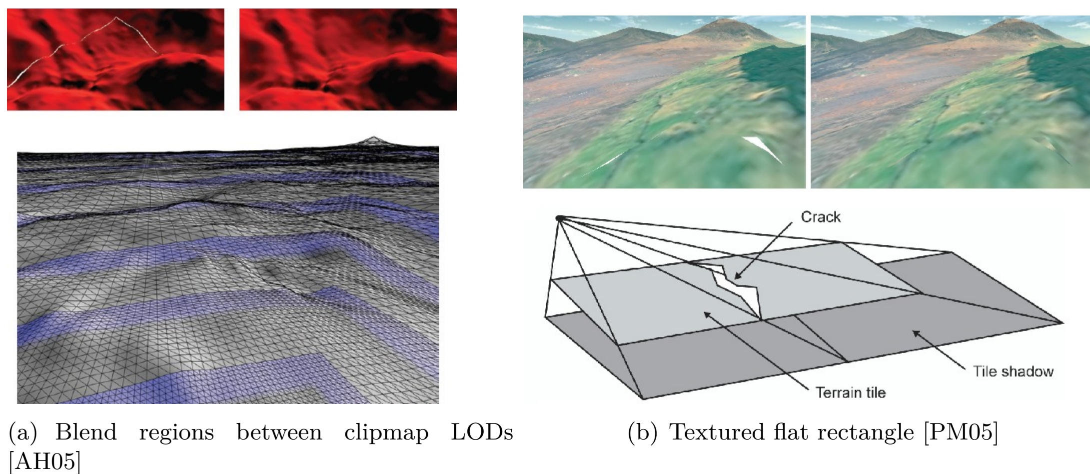
Solutions using blocks with regular LODs may skip [dB00, LC03] or add [Bro05] vertices to the edges of a block to fit with the corresponding neighbors when they do not render at the same LOD.

Algorithms using a hierarchy of triangular blocks may ensure that each block has only one neighbor per edge, for instance by forcing block splits like in Figure 2.6. Another solution to ensure this is to split triangles two by two using a “diamond” structure [Lev02, CGG+03a, HDJ05, GMC+06]. Only pairs of blocks facing each other along their hypothenuse may split, thus preventing any change of the exterior edges of the diamond. In addition to this property, one may then constrain the choice of the vertices on block edges to ensure that there are no cracks between neighbors with no need to know their respective LOD. For instance, [Lev02, GMC+06] force blocks to use a constant number of regularly spaced vertices on all their edges.

A hybrid solution is used in [LKES07]. This method does not use block-based errors to select LODs, but rather edge-based. Square blocks are then divided into as many triangular sub-blocks rendered using different LODs. In this way, the common edges of two neighbor blocks implicitly render using the same vertices. However, the four triangular sub-blocks need to stitch correctly: a solution similar to [dB00, LC03] is used internally on the diagonals of the block.

Other solutions do not avoid cracks explicitly, they rather use tricks in order to

remove or hide cracks. [WMD+04, LH04, AH05, LN05, SW06] fill cracks with vertical triangles. In addition, [LH04, AH05] use blending of consecutive LODs to make the transition unnoticeable (see Figure 2.7(a)): at the edges of clipmap LOD rings, vertices are progressively interpolated to match the next level. [WMD+04] fixed the maximum screen-space error to one pixel on block edges to ensure the vertical triangles are unnoticeable on screen. Finally, [PM05] chooses to let the cracks intact, but rather renders a flat rectangle under the geometry with the same texture as that of the terrain (see Figure 2.7(b)). This rectangle is moved and resized depending on the viewpoint position so that cracks are not visually noticed in most cases.



(a) Blend regions between clipmap LODs [AH05]

(b) Textured flat rectangle [PM05]

Figure 2.7: Various solutions for cracks.

Solutions that perform rendering based on a projective grid avoid cracks with no particular handling because a single, regularly ordered grid of vertices is rendered [DS04, CH06, LSGES06, LRS09].

2.2.1.2 Popping

Popping always happens when updating simplified geometry, especially when the level of detail algorithm is not continuous because terrain geometry suddenly changes from one LOD to another. In solutions that use strict screen-space error control, popping is limited to one pixel difference, thus hardly noticed. On the contrary, popping is quite visible in the case of discrete and regular levels of detail.

Most solutions do not address popping. However, it may be reduced using geomorphing [Hop98, RHSS98, Paj98, dB00, LP02, LC03, LN05, SW06]. This method consists in adding each new vertex at a position obtained from the previous LOD, then progressively moving it to its final position. For instance, if the new vertex splits a triangle edge in two, it is initially positioned by averaging positions of both end points of the edge. This progressive movement is obtained by interpolating the initial and final positions over time or depending on the viewpoint distance.

The blending solution used to fix cracks in [LH04, AH05] also has the effect of minimizing popping, which would otherwise be very noticeable at the edges of the clipmap LODs. As clipmapping is similar to texture mipmapping, this blending process is similar to trilinear filtering of mipmap levels.

2.2.2 Parallel computing

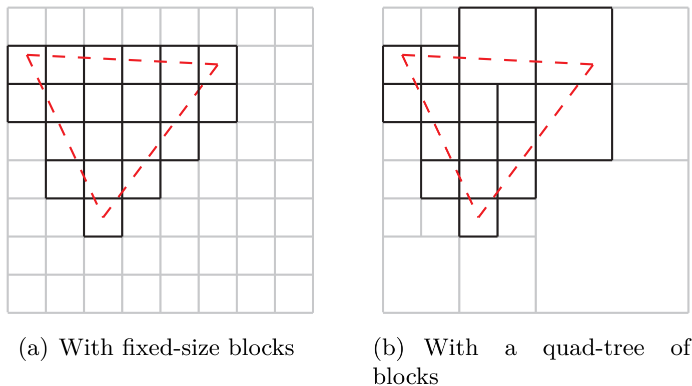
In the context of real-time rendering, parallel computing is mostly used to perform different tasks asynchronously. In particular, this smooths the frame rate when using out-of-core algorithms for data streaming: the rendering thread can continue to run even when loading new data. In addition, parallel computing allows a CPU-heavy implementation to run faster on multi-core architectures.

[LP01] proposes to desynchronize data update and rendering. The update thread adapts geometry and sends the updated result to the rendering thread when possible, while this one continually renders the available geometry. [Hop98, WMD+04, PM05, GMC+06] desynchronize data loading and selection. The selection thread also performs rendering of selected data. The data loading thread may also prefetch data based on assumed future viewpoint moves [CGG+03b, WMD+04].

2.2.3 View-frustum culling

The easiest way to safely reduce the complexity of rendered geometry is to discard parts that are not visible. Terrain parts that are located outside the 3D view frustum are not rendered on screen and may thus be culled.

Some works, especially those working on fixed-size and independent blocks, proceed by testing for each block whether it is inside the view frustum [Paj98, ZZSP01, PM05, LN05, SW06]. Knowing whether a block is inside the view frustum is generally computed quickly using bounding boxes. [ZZSP01] simplifies this test by comparing 2D coordinates without taking elevation into account (see Figure 2.8).



(a) With fixed-size blocks

(b) With a quad-tree of blocks

Figure 2.8: View-frustum culling (2D representation). Grayed blocks are not rendered.

A more efficient method to reject whole parts (typically blocks) of geometry is to organize them in a hierarchy of nested bounding volumes in a way similar to the one presented in Section 2.1.2.1. Using a top-down search of the hierarchy, any part found completely outside the view frustum is culled and the hierarchy is not further explored in this branch. This method is used in the majority of the solutions in the literature, even those that do not organize the data in a hierarchy of blocks. Note that solutions that do not rely on independent blocks cannot simply discard geometry that is outside the view frustum. Instead, this geometry is simplified to the minimum.

2.2.4 Optimization with graphics hardware

Nowadays, graphic cards with hardware accelerated 3D rendering are available on every personal computer and more and more hand-held devices. These cards render geometric scenes with dedicated processor and memory. Any modern rendering engine should take advantage of such cards using standard APIs like OpenGL in order to obtain optimal performance.

However, when a very large amount of geometric primitives are sent to the graphics hardware at every frame, the data upload bus can be a bottleneck and cause low performance. It is the case with adaptive terrain rendering because geometry is constantly updated. One way to avoid this problem is to optimize the transfers to graphics hardware using batched triangles, as explained in Section 2.2.4.1. Using batches reduces the number of individual transfers to graphics hardware. Another way is to cache geometry data in the graphics hardware memory and reduce the frequency at which this geometry is updated. Such solutions are presented in Section 2.2.4.2.

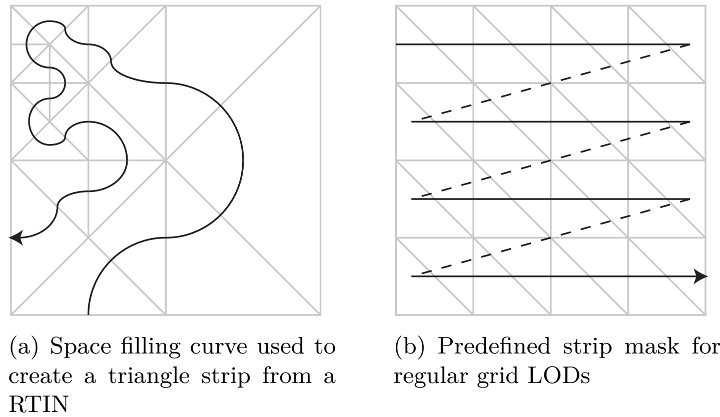
Recent graphic hardware contain programmable graphics processing units (GPUs). These processors are powerful for 3D computations and they may be used to perform some steps of the adaptive terrain rendering process. We present such techniques in Section 2.2.4.3.

2.2.4.1 Triangle stripping

A triangle strip is a common and optimized way of representing sets of contiguous triangles for rendering on graphics hardware. These triangles are encoded in a specific order that reduces vertex redundancies between triangles and allows uploading and rendering these triangles as a batch.

Most solutions use large triangle strips, typically of the size of a block. In addition, triangle strips can be preprocessed [CGG+03a, GMC+06], intelligently created at run-time using space-filling curves [LKR+96, Paj98, LP01, Lev02] (see Figure 2.9(a)), or directly obtained using predefined masks because of the regular nature of the triangulation [LC03, WMD+04, LH04, AH05, PM05, LN05, CH06, LKES07] (see Figure 2.9(b)). [LKES07] also uses triangle strip masks to fix predefined cases of cracks inside blocks.

[DWS+97, Hop98] use small triangle strips, incrementally updated at each frame according to the changes in geometry. [RHSS98, SW06] use triangle fans, a method that is well adapted to the quad-tree structure but also creates small primitives. These



(a) Space filling curve used to create a triangle strip from a RTIN

(b) Predefined strip mask for regular grid LODs

Figure 2.9: Solutions to obtain ordered triangle strips from different triangulations.

methods result to more data transfers and slower rendering because they use a high amount of small primitives. To circumvent this problem, [SW06] uploads all triangle fans of one LOD as one batch using special OpenGL extensions.

2.2.4.2 Caching in graphics hardware

Another way to optimize transfers from main memory to graphics hardware memory is to cache geometry in the latter. This method supposes that cached geometry is not updated frequently, so the cache may be used for more than one frame. It is therefore particularly well suited for methods using discrete levels of detail or blocks with fixed geometry.

[Paj98] introduced the idea of caching terrain geometry in graphics hardware. After the geometry of a block is computed, it gets uploaded to graphics hardware. The error is still re-evaluated at each frame but the geometry is not updated until the error exceeds a given threshold.

[Lev02, CGG+03a, WMD+04, GMC+06] update the geometry of a block only when this block is split or merged with its neighbors.

[LC03] has a defined number of pre-computed LODs per block and the geometry of a block is not updated until next LOD change. When the geometry of a block changes it is necessary to update adjacent blocks too, to ensure that their edges match to prevent cracks. The cache is directly manipulated by the GPU without having to upload geometry again.

[LH04, AH05, CH06] use a special incremental LOD update system called toroidal addressing. With this method it is possible to upload only the new geometry when the clipmap window moves by replacing parts that get outside the window, in a way similar to the one presented in Figure 2.5 but with LOD samples instead of blocks. This prevents re-allocating memory or moving previously cached data. Graphics hardware memory consumption is thus constant and transfers are minimized.

[SW06] uses an explicit graphics hardware memory management scheme. Memory

consumption is constant for one block, allocated once then filled as new LODs are uploaded. In addition, each LOD re-uses previously uploaded vertices so only the new vertices need to be uploaded.

2.2.4.3 Computations on GPU

Most current graphic cards come with powerful GPUs and this power can be used for some computations, thus fastening the whole rendering process.

[LC03, SW06] perform geomorphing on the GPU, allowing vertices to be modified directly in the graphics hardware memory cache instead of having to upload their new position at each modification.

[LH04, AH05] use the GPU to blend different LOD layers.

Finally, [DS04, AH05, CH06, LKES07, LRS09] use the GPU to perform displacement mapping, i.e. obtaining the final coordinates of the 3D vertices from elevation samples.

2.2.5 Preprocessing

Saving as much runtime processing as possible is desirable in order to get good real-time performance. One way to save some of this time is to defer some computations to an off-line preprocessing step when possible. The most obvious application for methods using blocks is to subdivide the original terrain map so that the data of each block may be loaded directly.

[DWS+97, RHSS98, Blo00, LP01, Lev02] pre-compute world-space errors for vertices or blocks. These values are then compared with the viewpoint distance or projected to screen-space in order to select LODs or update operations.

[Hop98, CGG+03a, WMD+04, LH04, AH05, CH06, SW06, GMC+06] pre-compute every possible LOD for the terrain. In this way, at runtime one only needs to select which ones to be loaded and rendered. However, most of these methods store and load successive LODs independently. This implies data redundancy, and therefore higher network and memory consumption at runtime.

2.3 Common features

Aside from handling plain geometry, many articles propose other features that are interesting for adaptive terrain rendering.

Section 2.3.1 is about texture mapping for terrain rendering. Texture maps are typically used as color maps to render the color of the terrain surface, or as normal maps to define how light reflects off this surface. These maps are generally larger than elevation maps, leading to the same memory problems. In consequence, one should use similar adaptive data loading and selection systems as for elevation maps.

Any terrain is located on the surface of a planet, typically the Earth. Although bounded terrains with no global curvature are enough for most applications, a great feature for terrain rendering solutions is to support entire planetary databases. In

Section 2.3.2, we present several techniques that model the surface of a planet with 2D maps.

To save disk space and more importantly network bandwidth in a data streaming context, some solutions propose to compress terrain data. They are presented in Section 2.3.3.

Finally, when an area of a terrain is rendered at the best available level of detail, we may get very flat and unrealistic rendering if, for instance, the viewpoint keeps getting closer to the surface. To avoid such situations, some articles propose to add artificially generated details. These techniques are presented in Section 2.3.4.

2.3.1 Texture maps

Texture maps are an important part of terrain rendering because they improve the visual quality for a given geometric mesh. Color maps define the photometry of the terrain surface, and normal maps help simulate geometric details on flat geometry triangles when lighting the terrain. However, most of the articles about adaptive terrain rendering do not manage texture maps in a particular way: either there is only one texture of limited size for the whole terrain, or colors and normals are stored per vertex. When stored per vertex, photometry cannot be refined independently from geometry. This may require loading unimportant geometric data just to obtain higher photometric quality, and may also cause color aliasing problems when triangles cover less than one rendered pixel in 3D rendering. On the opposite, [Blo00, CGG+03a, CGG+03b, WMD+04, LH04, AH05, HDJ05, SW06, GMC+06] treat texture maps similarly to elevation maps, encoding them in different LODs corresponding to the geometry structure.

When using a hierarchical structure for geometry simplification, the texture map is split into blocks just like the elevation map [Blo00, CGG+03a, CGG+03b, WMD+04, HDJ05, GMC+06], though limited to rectangular blocks because graphics hardware handle rectangular texture maps. In most cases, all texture blocks have the same resolution. Sampling definition gets higher for blocks lower in the hierarchy since they represent a smaller terrain area. Texture LOD is generally chosen using an error metric related to screen-space texel size, therefore it may be different from geometry LOD. However it cannot be of lower level in the tree than the corresponding geometry LOD because only one texture can be bound by geometry block when rendering on non-programmable graphics hardware. Consequently, this requires maintaining a link between the two trees of blocks.

Before being adapted to geometry level of details in [LH04, AH05], the clipmap is originally a large texture map management technique described in [TMJ98]. It is quite efficient and the drawbacks of geometry clipmaps are not very relevant in the case of textures. Texture clipmaps are used in terrain rendering softwares like *Google Earth*, geometry clipmaps and [LKES07]. In this case, texture LODs are handled and selected the same way as described for elevation maps in Section 2.1.2.4.

[PM05, Bro05] tile the texture map such that every geometry block has a corresponding texture block that is loaded at the same time. In [PM05], texture blocks

are encoded using a multi-resolution method that allows progressive texture streaming [MB03].

In all the cases where texture LODs are used, the problem of continuity between adjacent areas using different LODs rises, similarly to geometry cracks. Currently, only solutions based on the clipmap fix this problem, using standard hardware trilinear filtering since LODs are implemented as different mipmap levels of a large texture.

Another use of texture maps is normal mapping. A normal map enable high-quality shading, specifying how the light is reflected off each of its samples. In a lighted scene, this emphasizes the perceived relief of the terrain and improves the visual realism, especially when the normal map has higher resolution than the elevation map.

[WMD+04] pre-computes normal maps from the geometry and processes them at runtime the same way as textures with a quad-tree structure.

In [LH04, AH05], normal maps are updated at runtime such that they always have twice the resolution of the corresponding elevation map for each LOD. The same adaptive rendering method is used for geometry and color maps, but the clipmap levels are generated incrementally at runtime from the geometry rather than being stored explicitly.

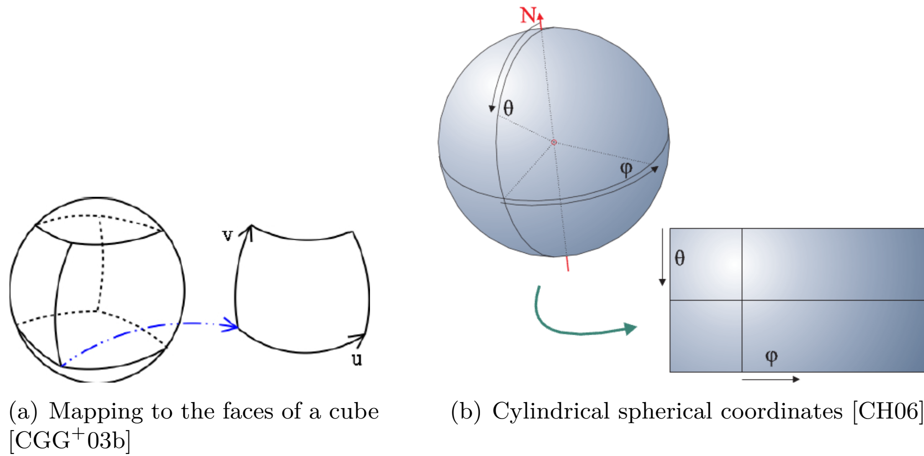
2.3.2 Planetary terrains

In this section, we summarize previous efforts to support planetary databases. Apart from handling such huge quantity of data (as discussed in Section 2.1.3), the challenge is to map the surface of the planet using an efficient and well sampled planet projection technique. Many planetary map projection methods have been invented for thousands of years [Weib], though their usage for real-time terrain rendering has rarely been discussed.

[RLIB99, CH06] and publicly available softwares (like *Google Earth* and *NASA World Wind*) propose to map the sphere using a classical cylindrical projection, parameterizing the planet with latitude and longitude angles. This obtains a planar rectangle as shown in Figure 2.10(b). However, the mapping is far from uniform on the surface. In particular, many redundant samples are stored and rendered around the poles, leading to poor performance and low visual quality.

[CGG+03b, GMC+06] project the planet surface onto a cube, with six independent square maps forming the faces of the cube (see Figure 2.10(a)). Each block has absolute geocentric characteristics for its corners, and vertex positions are obtained by barycentric interpolation of these values, with no explicit map projection.

Finally, [Aas02] proposes an adjustment of the Mercator projection that reduces visual distortion and preserves the square shape of samples on the globe as much as possible. This solution still gets singularities around the poles and have important area disparities.



(a) Mapping to the faces of a cube [CGG+03b]

(b) Cylindrical spherical coordinates [CH06]

Figure 2.10: Different planet mapping techniques.

2.3.3 Compression

While most articles use quantized encoding of samples to reduce main memory consumption per vertex, few of them make use of compression techniques for data storage and loading. Compression is performed during the preprocessing of LODs, and decompression is done in the main memory at runtime. This allows for fast out-of-core and streaming performance if decompression is performed quickly.

In [CGG+03b], pre-computed LODs are compressed using a general purpose lossless compression method that allows real-time decompression.

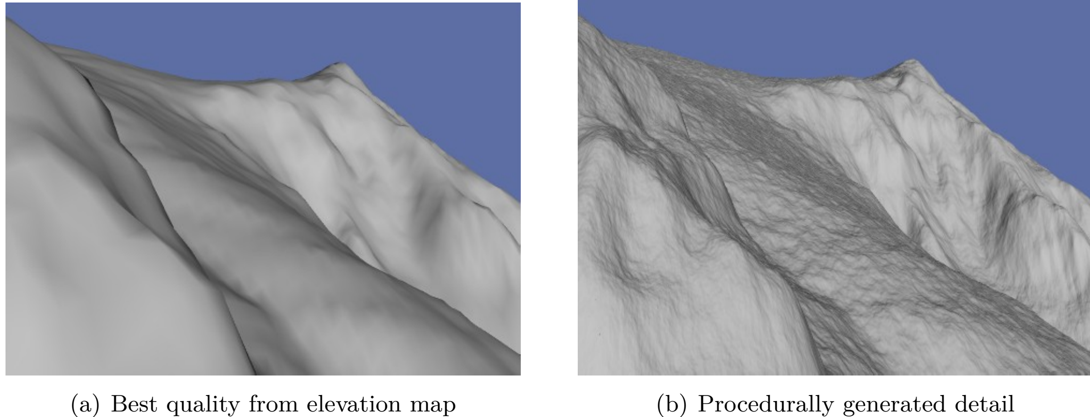
[LH04, AH05] encode any LOD as the difference between an up-sampled version of the lower LOD and the real values. Residuals are then compressed using a lossy compression method, called PTC (progressive transform coder), that allows efficient local decompression for real-time random access.

[GMC+06] uses wavelet analysis to compute low resolution blocks from their children, and stores the wavelet representation which is then used to reassemble the children by wavelet synthesis. This wavelet representation is then compressed with control over the loss in quality.

Finally, [HDJ05] notes that any image compression scheme may be used to compress independent LODs when these are regular grids of color or elevation samples.

2.3.4 Procedural detail

The highest achievable quality for a terrain is the definition of the original elevation map. When the viewpoint is very close to the ground, higher definition may be desirable to avoid rendering flat geometry due to the lack of data between two samples of the map. This can be obtained artificially by procedurally generating additional details. These details do not reflect reality; they are just some noise added into the geometry in order to make the terrain look less smooth. This effect is illustrated in Figure 2.11.



(a) Best quality from elevation map (b) Procedurally generated detail

Figure 2.11: Procedurally generated detail in addition to original data [GMC+06].

Procedural refinement consists in up-sampling data to a higher definition using sample interpolation, then using fractal noise values to deform these data and obtain artificial detail [DS04, LH04, HDJ05, LN05, GMC+06].

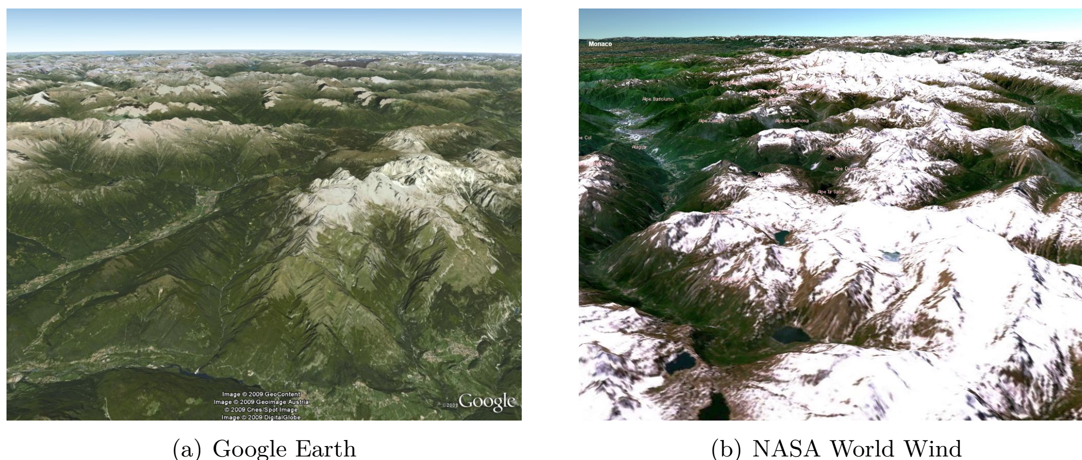
[DS04] applies procedural refinement directly on the rendered regular grid of vertices instead of the terrain map. In [AH05], the noise values are stored in a pre-computed map that is used by the GPU to modify the geometry. Finally, [GMC+06] proposes to extend the use of procedural detail in case the data for a given block are not currently available for rendering, typically when they are streamed via a slow connection. Generated details are then temporarily used until the data are received.

2.4 Available softwares

During the past decade, one has seen more and more available software products proposing adaptive streaming and 3D rendering of terrain data. Some use proprietary technologies, and some are based on articles that we presented in this chapter. The most successful ones, namely *Google Earth* and *NASA World Wind*, support streaming accross the Internet of huge 3D planetary terrains with very detailed photometry. Screen captures from these softwares are shown in Figure 2.12.

Google Earth uses the clipmapping method to manage outstandingly high definition global texture maps (up to 10cm per pixel in some areas), which are regularly updated but not free for use in other applications. However, the 3D rendering technology for elevation data seems outdated: large blocks of data are refined independently with no crack fixing, leading to visible disparities in relief definition in the distance. In addition, data distortion is clearly visible in polar areas since a cylindrical projection is used. *Google Earth* is available at `http://earth.google.com`.

NASA World Wind proposes 3D visualization of several public global databases from the NASA (National Aeronautics and Space Administration), especially SRTM for the elevation (up to 30m definition) and LandSat 7 for the photometry (15m definition).



(a) Google Earth

(b) NASA World Wind

Figure 2.12: Example 3D renderings using popular terrain applications.

It uses a solution based on a quad-tree map subdivision into blocks. When a low-quality model of the planet has reached its best definition, a new layer is rendered on top of it with higher definition data, though this leads to a sudden visual change. *World Wind* uses the same map projection as *Google Earth*. This open-source software is available at `http://worldwind.arc.nasa.gov`.

Other notable recent terrain streaming and 3D rendering applications can be cited:

- *Bing Maps 3D* or *Microsoft Virtual Earth* is a concurrent software for free global data visualization, whose methods are briefly described in [DNB06]. It is available at `http://www.bing.com/maps/`.
- *TerraExplorer* is used by the IGN (Institut Géographique National) for its Géoportail application which proposes high-definition visualization of France. It is available at `http://www.skylinesoft.com`.
- *ArcGIS Explorer* is a 3D front-end for the popular GIS (Geographic Information System) data management software *ArcGIS* which supports many kinds of planetary data. It is available at `http://resources.esri.com/arcgisexplorer/`.
- *RATMAN* is a recent open-source implementation of the solution presented in [GMC$^+$06]. It is available at `http://ratman.sourceforge.net`.
- *VWorld Terrain* (`http://www.vworld.fr`) is a planetary terrain rendering engine used in several commercial products, like the multi-player video-game *Dark and Light*.

2.5 Discussion

In this chapter we presented previous works done on adaptive terrain streaming and rendering. In this section, we summarize the main points of interest and discuss the most efficient methods and the compromises they make.

The triangulation of a simplified terrain mesh can use one of several kinds. Irregular networks (TINs) are powerful but their creation and storage are very complex, while regular grids are very easy and fast to use but need many more triangles to obtain the same visual quality as irregular networks. RTINs are a good compromise because they offer similar (although not optimal) terrain shape fidelity as TINs while being much easier to use and store because of their hierarchical structure. Nowadays, graphics hardware can render so many triangles per frame that it would slow down the rendering process to use the CPU to select triangles one by one to render. Therefore, considerations of triangle-level mesh simplification become more and more obsolete in the case of real-time rendering. It is now preferable to optimize geometry upload and caching on graphics hardware, using large batches of geometry that are updated only once in a while and that may even be pre-computed. Progressive data loading and support for huge terrains is also a new priority.

To allow high-performance out-of-core rendering and get the possibility for data streaming, tiling terrain data into blocks proved to be necessary. Blocks also offer natural methods for efficient view-frustum culling and error computation for LOD selection. In addition, when using a hierarchy of blocks of recursively smaller size, it becomes possible to get a complete representation of the terrain with few data and refine it progressively. Using blocks of fixed geometry offers one main advantage: this geometry can be pre-computed and compressed to save runtime CPU load. On the other side, blocks with multiple LODs allow adapting the geometry immediately, with no data loading or tree structure change. We note that few solutions propose to combine these two methods to obtain two-level adaptivity and get both advantages. We also note that few solutions avoid useless data storage and loading, when they are redundant in LODs or blocks representing the same terrain area at different qualities. In any case, solutions based on discrete LODs have the drawback of sudden geometry changes leading to visible popping artifacts, which may be hidden using geomorphing.

A more important type of rendering artifacts is the cracks that appear when the simplified terrain surface is not continuous. Although some methods just let cracks happen and then hide them, many solutions have been proposed to explicitly fix or avoid cracks. For instance, one can adapt the triangulation of a block to match the geometry of neighbor blocks. When using a hierarchy of blocks with pre-computed geometry, another solution is to use triangulation constraints on block edges during preprocessing, then use adapted data structure and runtime constraints to ensure that no split or merge operation that would lead to a crack is performed.

Texture maps, like elevation maps, are important data for terrain rendering and should be handled as efficiently and adaptively. Most solutions propose to place texture blocks over geometry blocks. In the case geometry blocks are used in a hierarchical data structure, textures can directly use the same scheme, using blocks with different definition. However, texture maps are not stored in the same tree structure: texture and geometry may use different LODs, although with some constraints due to hardware texture rendering limitations.

An interesting feature for terrain rendering is to handle entire planets instead of bounded rectangular elevation maps. This is performed by projecting the surface of

the planet on one or more 2D rectangular maps, then reconstructing 3D geocentric coordinates when samples are rendered. Note that using simple cylindrical projection leads to sampling inconsistencies over the surface of the planet. It is preferable to use a more uniform projection so that all the areas of the planet of a given size get almost the same number of samples and quality for a given LOD.

Unlike most previous works, we chose to clearly separate generic data loading, update and selection on one side, and specific data interpretation and rendering on the other side. The first part, presented in Chapter 3, is a generic solution relying on a generic data structure. With this solution, we adaptively handle arbitrary 2D maps of digital samples from a server hard disk all the way to a client rendering system. The second part, presented in Chapter 4, is an application of the generic solution to real-time 3D rendering of large terrains, composed of elevation maps and texture maps. We discuss and solve specific issues and propose advanced features for this particular use of the generic solution, to perform crack-free adaptive 3D rendering of textured planetary terrain maps on current graphics hardware. Finally, in Chapter 5 we present the off-line preprocessing steps we use to compute server databases in a way that minimizes runtime server activity.

Chapter 3

Generic solution for adaptive data streaming and selection

3.1 Overview

Adaptively streaming and rendering huge sample maps require using specifically adapted data structures and algorithms. After taking note of previous solutions (see Chapter 2), we chose to design our data structure by mixing 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. We then add new properties and techniques to handle this structure faster and more adaptively. We describe this data structure in Section 3.2.

Our first contribution concerning the data structure is the non-redundancy of data: successive blocks and LODs share data instead of replacing them. In fact, the new samples of a LOD are spatially interleaved with those of the previous LODs and we implicitly use all samples. This minimizes the amount of data to store and load. 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.

Once we have defined our data structure, we can use it in the different steps of our generic solution for data streaming and selection presented in Figure 3.1. The data structure and generic solution have been presented in [LMG09], applied to the context of 3D rendering of large terrains.

We first store the complete tree of blocks in a single file on the hard disk of the server as described in Section 3.3. Our file organization guarantees that the data for any client request are contiguous and that we can obtain the position of this data chunk in constant time.

On the client side, we explicitly store an incomplete tree of blocks in memory as described in Section 3.4. When a block is created, we allocate a single 2D grid 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 into previously unused grid positions. These methods reduce the number of data copies in memory as we avoid redundancy

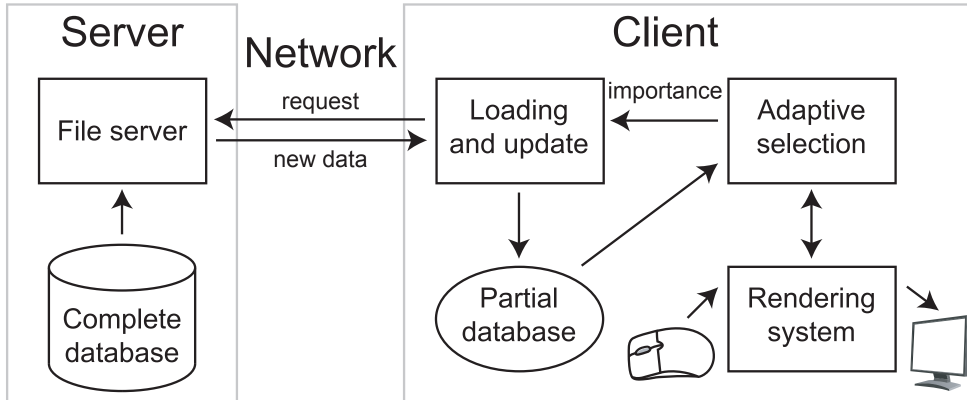


Figure 3.1: Architecture for adaptive streaming and rendering of large sample maps. The user guides rendering on the client. Selected available data are rendered while missing data are requested from the server. The server transmits these data from its database to the client, which then updates its partial database.

of data.

Adaptive update of the client database and progressive loading of data needed to fill this database are explained in Section 3.5. A measure of importance guides the order in which tree update operations are performed on the client database and in which data loading requests are transmitted to the server. We prevent overloading the network and optimize the relevance of performed operations by continuously updating the queue of requests and by selecting only a few requests from this queue at a time.

The last step is the selection of data to render on the client, described in Section 3.6. We first choose a LOD for each block using a measure of importance. This measure depends on the rendering performance and on interactive user requirements so it can adapt to the application. If a desired LOD is unavailable, we trigger a request for the corresponding update operation or data loading and we use an available one instead. Once a LOD is selected for rendering, if the block is visible, we extract the data to render using a mask that references its samples in the grid of the block.

3.2 Generic data structure

We base our solution on a generic data structure that combines two commonly used methods: the original 2D square sample map is subdivided into a complete and uniform tree of blocks, and each of these blocks has a set of levels of detail with increasing resolution. Blocks are regular 2D square grids of samples with a constant resolution. Each one is a square subset of the original map and can be rendered on its own.

The tree is a multi-resolution hierarchical subdivision of the entire original sample map. An example is shown in Figure 3.2. Each level of the tree covers the entire original map, with increasing resolution as one gets lower in the tree. Starting with a single root, the nodes of the tree are blocks of constant resolution and have a constant

number of children. The minimum number of children is four, which corresponds to a quad-tree. The areas covered by the children regularly subdivide the area of the parent.

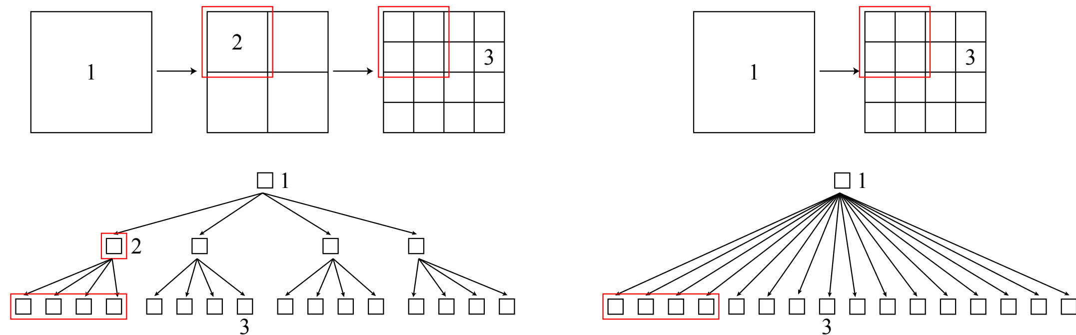


Figure 3.2: Construction of a quad-tree of blocks (**left**) and a tree with 16 children per block (**right**). **Top** — Successive regular subdivisions of the original map. Red frames cover the same area. Block 1 is the root and covers the entire original map. **Bottom** — Corresponding trees with the same numbers.

Levels of detail (LODs) are successive subsets of the sample grid of a block as shown in Figure 3.3. The last LOD uses the full resolution sample grid of the block. Note that we number LODs in reverse order, with the last LOD being number 0, for selection purposes (see Section 3.6.3 for more information). Grid subsets of the LODs are uniform over the blocks to allow using generic methods for storage, update and selection. In the rest of our solution, each LOD doubles the resolution of the grid subset in both dimensions compared to the previous one, therefore quadrupling the number of used samples. We could use other schemes which lead to less difference between successive LODs, for instance by only doubling the number of samples. While such a scheme would not be implementable for 2D image rendering with square pixels, it would allow finer adaptive grain for other applications like 3D rendering with vertices and triangles. We chose to favor genericity by keeping a regularly sampled square map form in all cases.

In Figure 3.3, we can see that we offer the option to use blocks with odd resolution. Samples in the red boxes are redundantly added so adjacent blocks get the same samples on their common edge. These additional samples are needed in the case of 3D rendering of elevation samples using triangles. See Section 4.2.1 for more information.

3.2.1 Advantages

We chose this data structure 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 areas with low quality requirements and inversely. This technique minimizes the total amount of rendered data and better distributes these data over the entire area of the map. Similarly, with the LOD structuring of blocks, we can choose a LOD

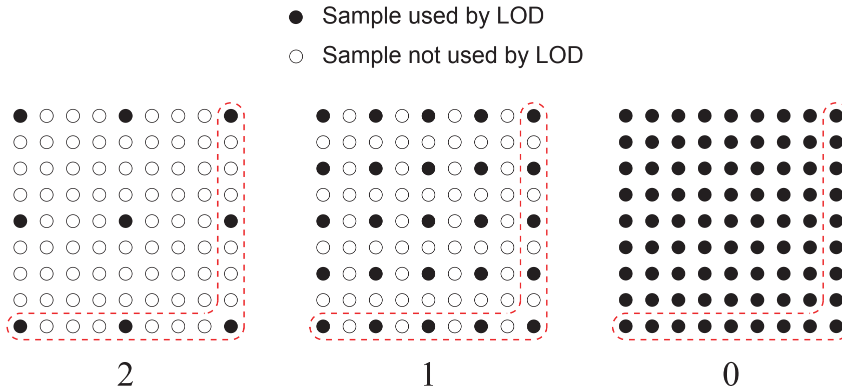


Figure 3.3: Successive LODs of a block (example of a 8×8 sample grid with three LODs). Samples in the red boxes may be added to the grid, for instance when rendering elevation samples with triangles.

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 entire map 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 area using upper tree levels. In addition, the tree structure simplifies the culling of invisible blocks using a classical depth-first tree walk-through.

3.2.2 New properties

To better adapt the data structure to our needs and improve its general efficiency, we add new properties, described hereafter.

First, a block and its children share samples because they cover the same area. This avoids loading redundant data. When a block in the partial client database splits, it gives the samples of its last LOD to its children, creating their first LODs. Conversely, it gets these samples back from the children when they are merged. Split and merge operations are described in Section 3.4.1.

Second, we allow rendering a block even if its sample grid 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 grid (with the refine operation: see Section 3.4.2) while using this grid 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.

Finally, each loaded LOD implicitly reuses the previous one and adds its new samples in the common sample grid of the block. This avoids loading redundant data. In

practice, using LODs that quadruple the number of samples from the previous one, we save 33% of loaded data compared to using fully independent and redundant LODs. However, this implies that no data filtering is applied to obtain lower quality LODs from the original sample map: only a predefined and regular subset of samples is selected. A system to allow data filtering in our solution with limited data size overhead is presented in Section 4.4.4, applied to texture maps.

3.2.3 Parameters

The first parameter that defines a database is the number of children per block, as illustrated in Figure 3.2. In practice, since children regularly subdivide the area of the parent, we work with the number of successive subdivisions by four of the parent block needed to obtain this number of children, which we call *subDepth*. The number of children per block is thus $4^{subDepth}$ ($2^{subDepth}$ in each dimension). A *subDepth* of one therefore corresponds to a quad-tree, while two gives sixteen children per block. The second parameter is the depth of the tree or maximum number of successive subdivisions into children, called *nbSubs*. Any depth may be used as long as one can subdivide the original sample map with enough blocks of valid resolution.

Considering the resolution of the original sample map in one dimension $mapWidth$, we get the one of a block $blockWidth = mapWidth/2^{subDepth \times nbSubs}$. When a block splits into a number of children, its sample grid is subdivided into as many sub-grids. Therefore, a valid $blockWidth$ must be an integer subdivisible by $2^{subDepth}$. When using odd-resolution blocks, we do not consider redundant edge samples in those characteristics: we use $blockWidth + 1$ instead of $blockWidth$ only for actual data copies and rendering.

The number of LODs per block is defined by the number of children per block. We know that the first LOD is directly obtained from the parent of the block, and that the last LOD will be given to children. Therefore, when using $2^{subDepth}$ children in one dimension, the last LOD has $2^{subDepth}$ times more samples than the first. With each LOD doubling the resolution of the previous one in one dimension, we can deduce that such blocks have $\log_2\left(2^{subDepth}\right) + 1 = subDepth + 1$ LODs. We number the LODs such that the first LOD of a block is number *subDepth*, while the last one is number 0 (see Section 3.6.3).

Using high values for *subDepth* allows to obtain more adaptivity at block level, but less at tree level. When using a given $blockWidth$, a *subDepth* of two requires as many leaf blocks as a *subDepth* of one to obtain a given sampling definition. The first difference is that blocks have one more LOD to potentially use without splitting or merging; the second difference is that the depth of the tree is halved. This means fewer costly tree update operations are needed, but these operations have a higher and less local impact on overall quality since more blocks are created or removed at once. In practice, using a *subDepth* of one (i.e. a quad-tree of blocks with two LODs) gives satisfying results and we did not feel the need for higher values. In addition, using two LODs per block means that each new LOD of any block adds a constant number of samples since the first LOD is obtained from the parent without data loading. All

new LODs thus take about the same time to load and update, and have a comparable impact on visual quality and rendering speed. This helps making adaptive methods of the solution more reliable.

Similarly, using low values for $nbSubs$ implies high $blockWidth$. This results in tree update operations with a higher and less local impact on overall quality, but also to a reduced overall number of blocks to treat in real-time during the adaptive LOD selection step presented in Section 3.6. Low $blockWidth$ is interesting for databases designed to be rendered on clients with low performance. In that case, the overall number of samples is more adjustable and the clients may select more precisely where to use more quality, though general performance is lower for a given rendering quality.

3.3 Server database

The first place where the data are located is a pre-computed file on the hard disk of the server. Many clients can connect to the server at the same time and request data corresponding to any LOD of any area of the map. Consequently, 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 construction step itself is detailed in Section 5.3.

The file first contains a header with characteristics of the database like $subDepth$, $nbSubs$, $mapWidth$ and data interpretation information for rendering. Then, the contents of the tree are flattened LOD by LOD, for instance as shown in Figure 3.4. Loadings are done one LOD at a time, hence ensuring that all the data for a single request are contiguous in the file. Note that a file may contain more than one tree (for instance, our planetary terrains presented in Section 4.5 use six trees); in that case the trees are simply stored consecutively.

Non-redundancy of data is clearly visible in Figure 3.4. 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 and send it along with initial database characteristics at the first client loading. Second, a LOD implicitly reuses the previous ones so we store only the new samples.

3.3.1 First approach: implicit file organization

In a first time, we organized the file exactly as presented in Figure 3.4. The size of each file element is known in advance because all blocks and LODs have constant resolutions and samples are stored on a constant number of bytes. Furthermore, the tree is uniform and complete and the blocks of each tree level are stored in order. Consequently, we can get the file position for any request in constant time.

The file position starting after the header for a given LOD of a given block is then:

$$pos(b) = \sum_{sub=0}^{b.sub-1} (sb \times 2^{subdepth \times sub}) + sb \times b.num + \sum_{lvl=b.lvl+1}^{subDepth-1} s(lvl) \qquad (3.1)$$

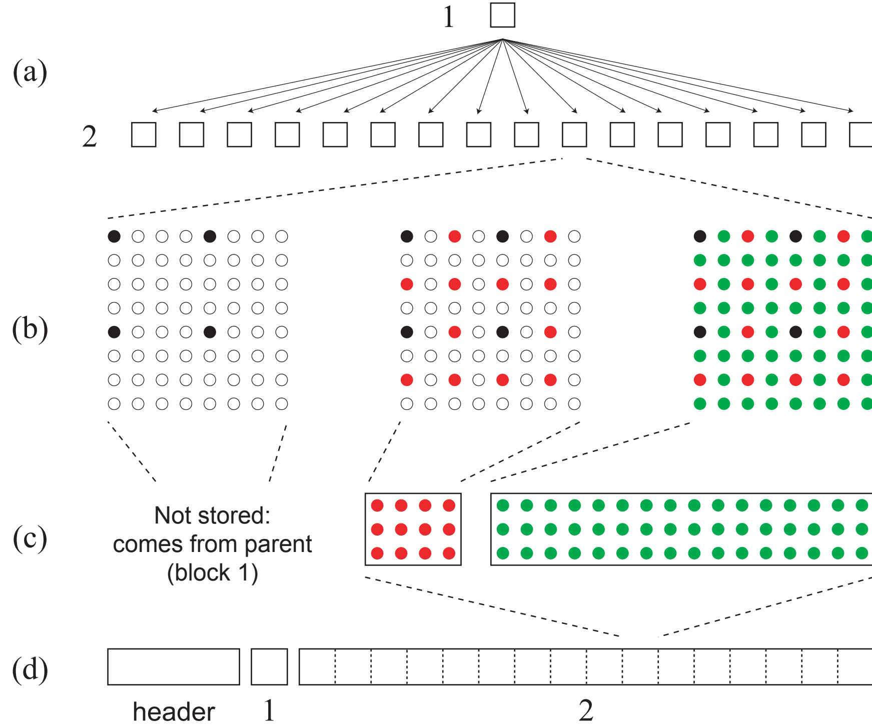


Figure 3.4: Server file organization (example of a two-levels tree of blocks with three LODs). **a)** Original tree structure. **b)** Block structuring with LODs (see Figure 3.3). **c)** Redundancy-free LOD storage. **d)** Final data order in the file with the same numbers as in (a).

where $b.sub$ is the level of the block in the tree (the root level being number 0), $b.num$ is the number of the block in that level and $b.lvl$ is the number of the desired LOD for the block. $2^{subdepth \times sub}$ is the number of blocks in the tree level number sub, sb is the constant size of the blocks and $s(lvl)$ returns the size of LOD number lvl of any block (constant for a given value of lvl). Note that we skip the first LOD the block (i.e. LOD number $subDepth$).

This approach allows for fast and easy access to any given LOD, but has some limits. First, it is not preprocessing-friendly in the case of huge input maps, as explained in Section 5.3. Second, it does not allow for variable-size LODs: $s(lvl)$ needs to be the same for all blocks, which is not possible in the context of lossless LOD data compression.

3.3.2 Second approach: explicit level of detail size and position

To circumvent the compressed LOD storage problem, we envisaged storing LODs in compressed form along with their size, then filling the remaining space with zeroes to obtain an actual file size of $s(lvl)$. Although this would allow for compressed data

transfers, the server file would not become smaller and disk accesses would not benefit from lower data sizes. Instead, we chose to use another approach for server database storage.

Using this approach, each LOD may be stored at any position in the file and may have any size. We are thus not limited to the file organization presented in Figure 3.4, though LODs themselves contain the same data. This allows us to compress LODs, to store meta-data of arbitrary size with LODs, and to build the file linearly with a bottom-up preprocessing algorithm (see Section 5.3). In our application, we implemented simple LOD compression using a general-purpose compression library.

All LODs use the same structure:

- A predefined and constant string used to check that a LOD is actually stored starting at this file position.
- The number of next LODs: if the LOD is not the last LOD of its block, this is set to 1; otherwise this is set to the number of children, or 0 if the block is a leaf of the complete tree.
- The list of next LODs: for each of them, the corresponding file position and size are stored.
- The data of the LOD themselves.

A request from the client needs to give the position and size of the LOD in the file rather than the position of the LOD in the tree (*b.sub*, *b.num* and *b.lvl*). As shown in the LOD structure, these values are obtained when the previous LOD is loaded. Consequently, the client needs to temporarily store them for each LOD. When the server receives a request, it only needs to read the given size at the given file position, which is as fast as with the first approach with constant complexity. Before reading data, the server checks that the requested file size and position are in acceptable bounds to prevent problems due to incorrect requests.

3.4 Partial client database

On the client side, the 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 database with the lowest quality. The adaptive selection step then triggers specific database update operations as explained in Section 3.6.4, progressively expanding the tree as shown in Figure 3.5. In this section, we describe these operations. As detailed in Section 3.5, we perform all the data copies in parallel with the selection and rendering so that they do not harm rendering smoothness, especially on multi-core architectures.

We store the samples of a block in a single grid of resolution equal to that of its last LOD. This sample grid is present in memory only for the leaves of the incomplete tree in order to reduce memory consumption on the client side. Conversely, all leaves of the tree have a sample grid. We thus ensure that any area of the map has a representation on the client.

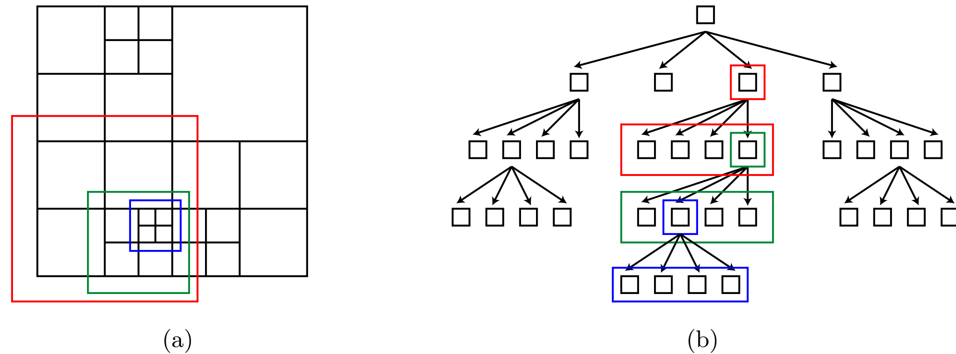


Figure 3.5: Progressive loading of a quad-tree. **a)** Successive subdivisions of the map (red, green, then blue). **b)** The corresponding incomplete tree.

3.4.1 Split and merge operations

When a fully loaded block needs to be rendered with a quality higher than it can offer, we use its children instead. If the block is currently a leaf of the incomplete tree, we have to load the children in memory: this is the split operation. The sample grid of the parent is regularly subdivided into square subsets corresponding to its children as shown in Figure 3.6. The samples of each subset are copied into the empty sample grid of the child to build its first LOD. The parent finally deletes its own sample grid. Once the split operation is done, each child can progressively load its own LODs then possibly split itself independently from the others. In this way the tree gets unbalanced, enabling us to obtain local adaptivity.

When using blocks with odd resolution (see Section 3.2), the samples on the edges of adjacent children are copied to each of these children. For instance in Figure 3.6, child 1 gets redundant samples from the grid subsets of three other children. In practice all blocks, including the parent, have one additional row and one additional column in that case, though they are not shown in the figure.

When a previously split block needs to be rendered with a quality lower than its children can offer, it gets back its data from these children: this is the merge operation. As shown in Figure 3.6, we recreate the sample grid of the block and get all of its samples from the first LOD of the children. The children are merged recursively if needed. Once the merge operation is complete, we delete the children blocks because they are no longer used for rendering.

We could easily cache blocks removed during merge operations in case they are needed again later, as long as enough memory is available. Note that our solution already offers some sort of caching by using multiple LODs per block: lower LODs may be used after a new LOD is loaded without the need to unload this one.

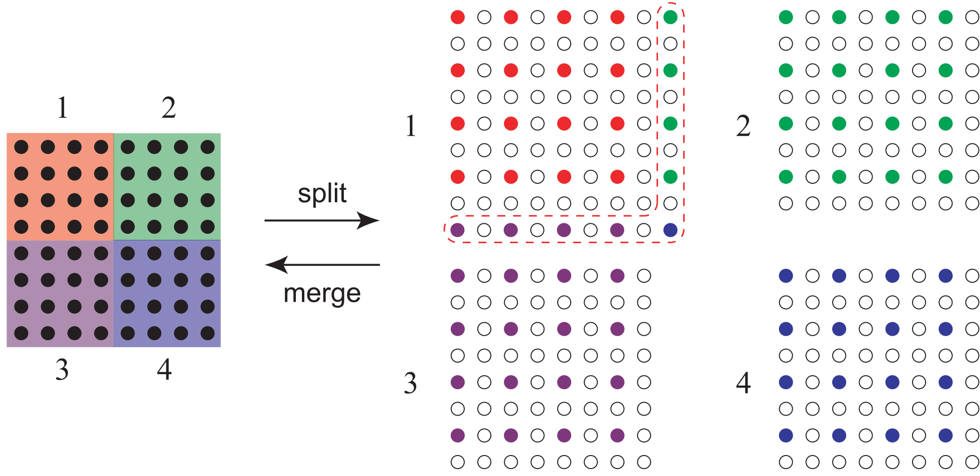


Figure 3.6: Block split and merge in a quad-tree. **Split** — **Left:** Parent block with the grid subsets of its four children. **Right:** Children blocks with their first LOD. Unfilled samples are not loaded yet. Samples in the red box are redundantly copied in case of blocks with odd resolution. **Merge** — **Right:** The four blocks to merge. **Left:** Parent block fully reconstructed from the first LODs of the children.

3.4.2 Refine operation

When a partially loaded block needs to be rendered with a quality higher than its best available LOD can offer, we load the next LOD with the refine operation. Unlike split and merge operations, refining first requires loading data from the server (see Section 3.5.1). When data are received, we add the new samples of the LOD in the corresponding unused positions of the sample grid of the block as shown in Figure 3.7.

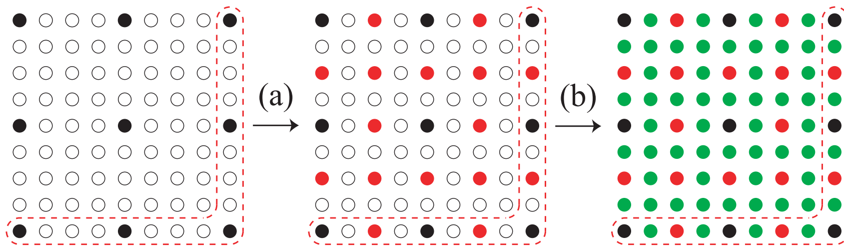


Figure 3.7: Successive block refine operations (same block as in Figure 3.3). **a)** Red samples (second LOD) are interleaved between black ones (first LOD). **b)** Green samples (third LOD) are added, the grid is now full.

A data conversion step may take place during the refine operation, still performed in parallel with rendering. It is used, for instance, to decompress the received samples, convert them from quantized elevation values to 3D vertices (see Section 4.2.1), or filter sample values using the previous LODs (see Section 4.4.4). In that case, we store samples in the client database under their converted form, but they are still organized

with the generic data structure.

3.5 Adaptive streaming and database update

In Section 3.4 we have presented the client database and the associated update operations. In this section, we present how we use these operations to adaptively and progressively fill the database according to a measure of importance. Figure 3.8 illustrates the architecture of our client-server data streaming solution as well as the different steps and threads used in the client to perform various tasks of data loading, update, selection and rendering.

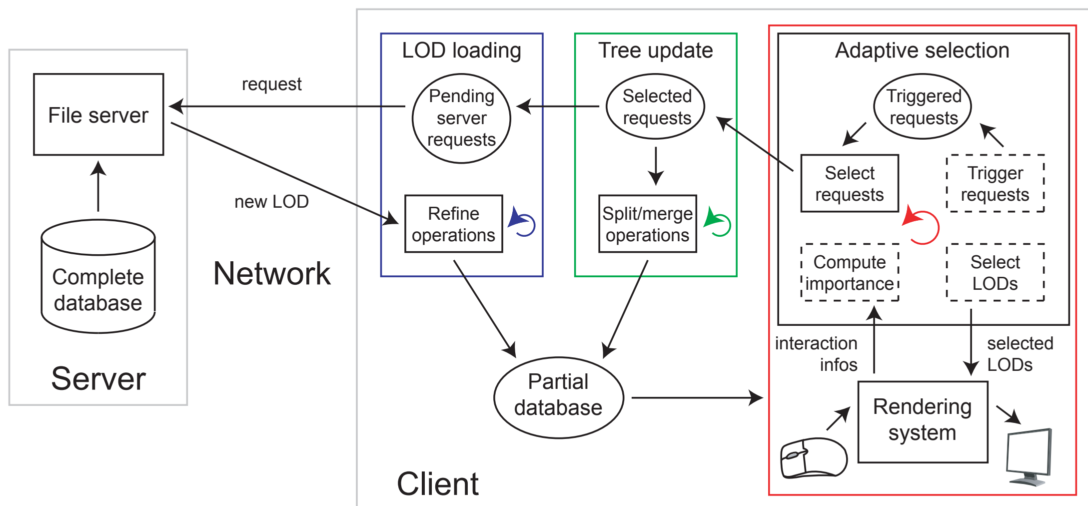


Figure 3.8: Detailed steps of the generic solution summarized in Figure 3.1. The three colored boxes with circular arrows represent the three parallel threads of the client, presented in Section 3.5.2. Steps with dashed boxes are presented in Section 3.6.

3.5.1 Progressive loading

In our solution, the server only reads and transmits data as is on demand to the clients. We manage progressive data loading on the client side. At any given time, our method streams the most important data. Therefore, we implicitly adapt to the network bandwidth and latency and the rendering quality constantly improves.

When the adaptive LOD selection step (presented in Section 3.6) needs a new LOD for a block, it triggers a loading request while rendering keeps running in parallel using the available data. When the client receives the corresponding data from the server, the partial database is updated with a refine operation and the new LOD may immediately be selected for rendering.

The server may be in a situation in which it cannot respond quickly to all requests because other clients may be asking for data. Furthermore, the network can have

any bandwidth, latency and encumbrance. Unfortunately, the longer pending time of a request, the more probable this request gets irrelevant when the data are finally received, for instance if the user has moved the viewpoint to another area of the map. To avoid overloading the network and the server with irrelevant loadings, we fix the number of pending server requests to a maximum. We choose the requests to actually transmit to the server according to a measure of importance in an additional request selection step, detailed in Section 3.5.2.

3.5.2 Asynchronous database updates

The client uses three threads running in parallel (see Figure 3.8). First, the LOD loading thread transmits the selected data loading requests to the server then adds the received data to the database using refine operations. Second, the tree update thread continuously performs selected split and merge operations. The final and main thread performs adaptive LOD selection and rendering. It also triggers database update operation requests, and selects from these requests those to be performed in priority by the other threads as noticed in Section 3.5.1. This request selection step applies for both LOD loading and tree update requests: they are handled the same way, though there is no additional latency due to the network in the latter case.

In practice, the adaptive selection step assigns an importance value to each triggered request (see Section 3.6). Aside from the list of triggered requests for each type of operation, we use a queue of selected requests with fixed and relatively small size. At each frame of rendering, if room is available in this queue, we select from the list of triggered requests the one with the highest importance and push it into the queue, until the queue gets full. Requests that are not selected are then discarded; at the next frame, the adaptive LOD selection step will trigger new requests with updated importance values.

This solution is reminiscent of the system with priority queues proposed in the literature [DWS+97, HDJ05, GMC+06], but it is more adapted for threaded operations. In particular, we can select multiple requests at once, which is convenient in case more than one operation can be performed in one frame time and/or multiple requests can be sent concurrently to the server.

In parallel, the tree update and LOD loading threads simply perform the database update operations corresponding to the selected requests in order, as long as the queue of selected requests is not empty. When an update operation is completed, we remove the corresponding request from the queue so that another request may be selected.

Split and merge operations can be performed as soon as they are selected with no latency due to data loading, so we use a higher queue size for such selected requests than for loading requests. To optimize system reactivity and resource usage, we may also adapt the total size of the queues based on an estimated number of operations that the client can perform at a time of one frame of rendering.

Since all database update operations are performed in parallel with LOD selection and rendering, their impact on rendering smoothness is limited, especially on multi-core architectures. However, some selected LODs might become unavailable and holes

might appear in the rendered map if parts of these operations are performed in parallel with rendering, like deleting the sample grid of a block that has split or deleting the children blocks that have merged. To prevent this, the tree update operations are not fully performed asynchronously. The tree update thread only performs data copies, which are the most complex parts of the operations. Subsequent tree structure changes and data deletions are performed by the main thread at the beginning of each frame before selecting the LODs to be rendered (see Algorithm 3.1).

3.6 Adaptive level of detail selection for rendering

We can render the partial database of the client at any moment, following the scheme presented in Figure 3.8 and Algorithm 3.1. At each frame of rendering, we first use a measure of importance to select a LOD to render for each block. We then trigger database update requests when unavailable LODs are selected and use available ones instead for rendering. Finally, we upload the samples of the selected LODs to the rendering system with a set of masks. In this section, we describe all these steps.

Algorithm 3.1: Operations performed on the client by the adaptive selection thread at each frame of rendering.

```
finish tree update operations performed in parallel
foreach block do
    compute visibility                      /* see Section 3.6.1 */
    compute importance                      /* see Section 3.6.2 */
    select LOD to render                    /* see Section 3.6.3 */
    if selected LOD is unavailable then
        trigger update operation request    /* see Section 3.6.4 */
        select nearest available LOD instead
select requests to be treated in parallel   /* see Section 3.5.2 */
foreach visible block do
    extract samples of selected LOD         /* see Section 3.6.5 */
    perform rendering
```

3.6.1 Culling blocks that are not visible

Only leaf blocks of the incomplete tree in the client database can be rendered because they have sample grids. However, the map is rarely entirely visible and we may thus avoid rendering blocks that are not visible. We quickly decide whether a block is visible by comparing its bounding area with the rendered map area. In the case of 3D rendering, this is called view-frustum culling and generally uses 3D bounding volumes to simplify computations. For each frame of rendering, we obtain the visibility of all blocks in a classical depth-first tree walk-through. When a block is found entirely

visible or invisible, we implicitly know that all blocks in its sub-tree have the same visibility. Invisible blocks do not get rendered and do not trigger split operations or data loading requests. Results of 2D view-frustum culling are shown in Figure 2.8.

3.6.2 Measure of importance

As introduced in the previous sections, we use a measure of importance to select LODs to render, to trigger client database update operations and to define the order in which we perform these operations. In this section, we describe this measure whose goal is to ensure 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 area that this block covers. The importance depends on a number of factors. Any set of factors may be used, based on the rendering system and the application, but we can group them in several families.

First, the importance depends on a generic quality factor common to all the blocks of a map, ensuring that the solution adapts to the required rendering speed or quality. Second, we take into account the area of the map covered by the block (by extension, its level in the tree). This gives more importance to blocks that cover larger areas and ensures that the total importance of any given area is comparable no matter how much it is subdivided in blocks. Third, we consider the user interactions (for instance, the distance between the rendering viewpoint and the block). We may then add any number of other optional factors that represent intrinsic block characteristics (for instance, an heterogeneity factor or an error value). Finally, we divide the product of all factors by the block resolution to ensure that importance values are comparable between blocks from different databases using different block resolutions. The first step of the measure of importance, representing the required quality for a block, is presented in Equation 3.2:

$$quality = \frac{qualityFactor \times blockRadius \times interactiveFactors}{blockWidth} \tag{3.2}$$

Note that all factors are 1D values. For instance, we use the radius of the area covered by the block $blockRadius$ instead of the area itself, and the width of a block $blockWidth$ instead of its 2D resolution. Note that $blockRadius/blockWidth$ represents the radius of the area covered by one sample of the full resolution block. We use 1D values because the importance itself as well as factors like the viewpoint distance are 1D. This is comparable to the principle of virtual spheres used to trigger split and merge operations used in previous works [Blo00, LP01, Lev02, CGG$^+$03a] (see Figure 3.10 for a direct analogy). In addition, an increase of any factor increases the quality value: some factors need to be inverted to get expected behavior. For instance, we want more quality for blocks with low distance from the viewpoint.

We select LODs using linearly decreasing importance thresholds (see Section 3.6.3), and we consider that quadrupling the number of samples for an area doubles the visual quality of this area. As a consequence, we first need to invert the value obtained in Equation 3.2 to obtain decreasing importance, then to get the base two logarithm to

reflect that a LOD doubles the visual quality compared to the previous one. The final generic measure of importance used in our solution is presented in Equation 3.3. See Section 4.2.4 for an application to 3D geometry blocks.

$$\begin{aligned} importance &= \log_2\left(\frac{1}{quality}\right) \\ &= -\log_2(quality) \end{aligned} \tag{3.3}$$

In case *quality* is negative, we use an arbitrary large negative value for the final importance. This happens when a factor reaches an extremum value, meaning that the block should refine and/or split as soon as possible. In particular, this is the case when the viewpoint gets inside the bounding area of the block.

3.6.3 Level of detail selection

At each frame of rendering, we compute the importance value then choose a LOD to render for each block. Each LOD has an associated importance value. When this threshold is reached, the LOD is selected if available. When it is not available, we trigger a database update request as explained in Section 3.6.4 and we use the closest available LOD instead.

Because LODs are regular grids of samples whose resolution successively doubles in both dimensions, we may use predefined importance thresholds. In practice, we simply use the number of each LOD as its importance threshold. As introduced earlier, we number LODs in reverse order. This ensures that LOD number 0 uses the full resolution of the block no matter how many LODs are in the block. This allows importance values and LOD thresholds to be comparable for differently subdivided maps.

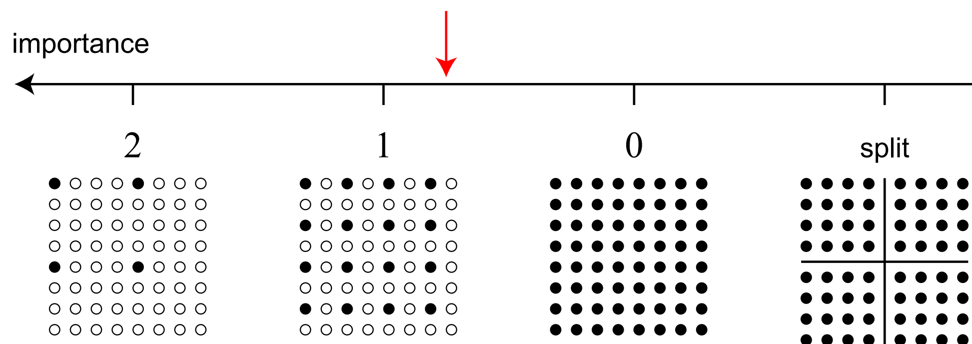


Figure 3.9: LOD selection thresholds on the scale of importance for one block (same block as in Figure 3.3). For instance, with the importance of the block located at the red arrow, LOD number 1 is selected. There is an additional threshold to trigger split operations, presented in Section 3.6.4. Since we use decreasing importance thresholds for LOD selection, a low importance value means that a high quality is desired.

Using LOD numbers as importance thresholds gives a linear scale of importance as presented in Figure 3.9 and allows selecting LODs just by rounding the importance

value to the next higher integer value. When obtaining a LOD number higher than the one of the first LOD of the block, the first LOD is always selected. In practice, this LOD does not need an explicit selection threshold. Figure 3.10 illustrates the thresholds for LOD selection in 2D using virtual circles around a block; all factors are constant except the radius of the blocks and the distance from the viewpoint. In particular, we can see that the distance between two consecutive thresholds is halved each time since we consider that the visual quality is doubled when switching LODs.

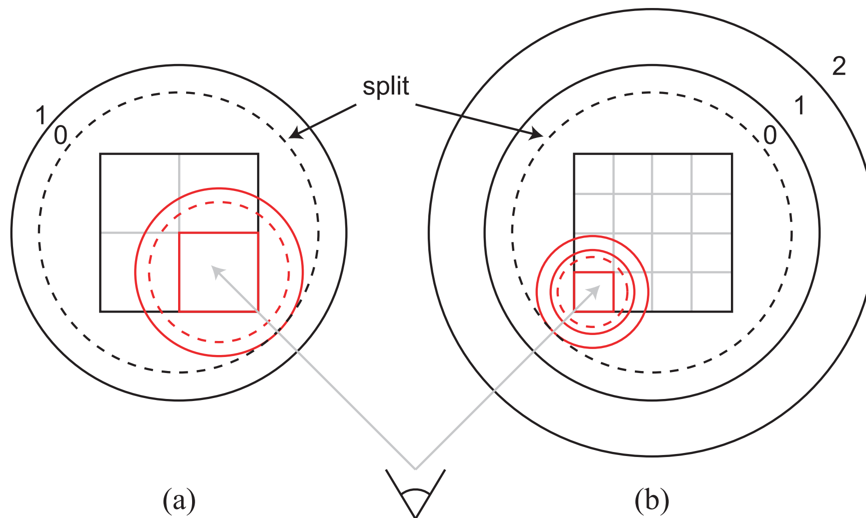


Figure 3.10: Example thresholds for block LOD selection and split trigger, depending only on the viewpoint distance to the nearest possible sample of the block (the corner). The viewpoint moves along the gray lines. **a)** Case of a quad-tree: blocks have two LODs. Note that LOD number 1 is always selected as long as the viewpoint distance does not reach the threshold for LOD number 0, materialized by the black circle. We can see that the importance of the child block (in red) is halved compared to the parent (in black) because it covers a smaller map area, as explained in Section 3.6.2. Split operation of the parent triggers when the viewpoint distance reaches the threshold for LOD number 0 of the child, which in practice corresponds to virtual LOD number -1 of the parent (see Section 3.6.4). **b)** Case of a tree using sixteen children, thus three LODs per block. We can see that the only difference in the parent block is an additional threshold to select LOD number 1 over LOD number 2.

3.6.4 Database update triggers

When a selected LOD is not available, we trigger a request for the corresponding database update operation. This request has an associated importance value that is used by the methods presented in Section 3.5.2 to select the requests to be performed in priority. Note that we want the importance of a request to increase as the need for quality grows. We also want this importance to reflect that another LOD of the block is

already available for rendering (for merge operations, we consider the last LOD of the block is available since the children use the same samples to render at their first LOD). Therefore, we obtain the importance value of a request by subtracting the importance value of the block from the threshold of the currently available and used LOD (i.e. its number).

We note that some blocks may not be rendered at the current frame of rendering (see Section 3.6.1). Only the merge operation can be triggered on such blocks because we do not need more quality for areas that do not get rendered, but we can save memory by reducing this quality. Apart from this particular situation, there are several cases where a LOD is not available, corresponding to the three types of database update operations:

First, the selected LOD may be of higher resolution than the best available one. For instance, this is the case when LOD number 1 is not loaded in Figure 3.9. In that case, we trigger a loading request which will lead to a refine operation if it gets selected.

Second, the block could have split before because more quality was once required, but now one LOD of this block is enough to render the same area. For instance, this is the case when the block in Figure 3.9 has split. In that case we trigger a merge operation, except when the selected LOD is the last one because the first LODs of the children of the block contain the same samples and we prefer to avoid data structure operations when possible. Note that unlike for the other operations, the most relevant merge requests are the ones for blocks with lowest importance value. Consequently, we get the opposite of the importance value when triggering the request.

Finally, we trigger the split of a fully loaded block when one of its children needs to load its second LOD. This is an additional threshold on the scale of importance, introduced in Figures 3.9 and 3.10. 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 by successive split and merge operations. However, children are not in the tree yet, so we need to guess the importance value of the most important child based on the one of the parent one by adjusting the block radius. A child block needs to load its second LOD (i.e. LOD number $subDepth - 1$) when its importance gets under $subDepth - 1$. Considering the radius of a child block is about the radius of the parent block divided by the number of children in one dimension, the split threshold for the parent block coincides with the threshold for virtual LOD number -1 as introduced in figure 3.10 and explained in Equation 3.4:

$$\begin{aligned} split &= \log_2\left(\frac{2^{subDepth-1}}{parentRadius/childRadius}\right) \\ &= \log_2\left(\frac{2^{subDepth-1}}{2^{subDepth}}\right) \\ &= subDepth - 1 - subDepth \\ &= -1 \end{aligned} \tag{3.4}$$

3.6.5 Masks for levels of detail extraction

Once we have selected an available LOD to render for each rendered leaf block, we upload the corresponding samples to the rendering system. We perform this by applying a mask onto the sample grid of the block. A mask is an array of indices that defines the subset of the sample grid that is used by the LOD. Since sample grid subsets of LODs are the same for all blocks, we do not have to store or compute many masks for different blocks. Instead, one mask per LOD is computed only once. See Section 4.2.2 for a concrete application to 3D geometry blocks.

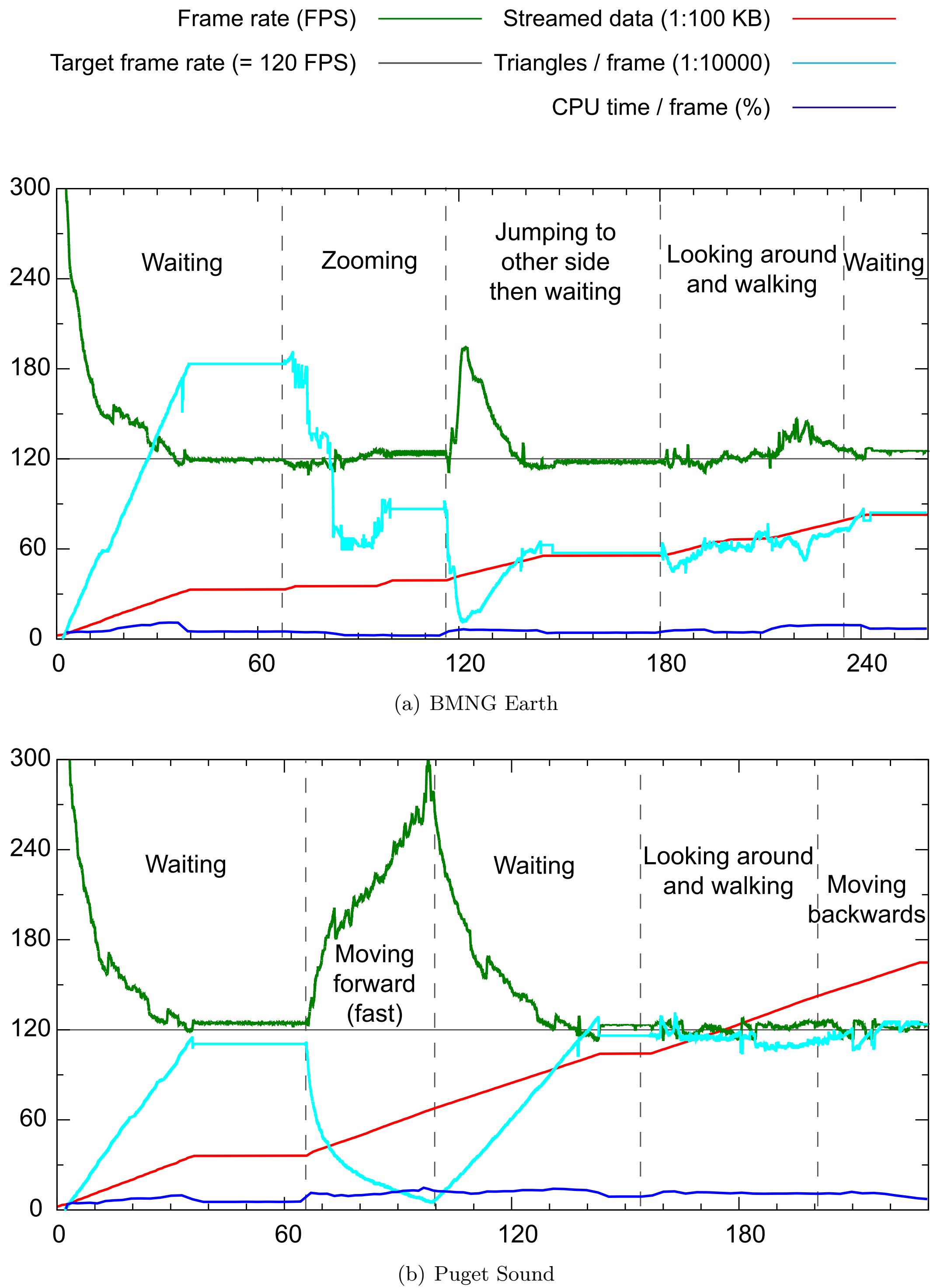
3.7 Results

The structure and methods presented in this chapter are generic: we consider maps of samples that can be rendered in multiple ways. We have tested the solution using two 3D terrain rendering applications, using maps whose samples contain both elevation and color values. The first application supports bounded terrain maps with elevation values relative to a plane; the other supports planetary maps with elevation relative to an ellipsoid. 3D vertices computed from elevation values are linked with triangles to reconstruct the 3D surface of the terrain. Issues specific to these applications are discussed in Sections 4.2 and 4.5.

We tested both applications using real-world databases and user interactions; see Appendix A for a description of the databases we used in our tests. All the test databases were preprocessed to obtain files useable by the server of our solution (see Chapter 5). We used $subDepth = 1$ and $nbSubs = 8$ as parameters (see Section 3.2.3), which led to $blockWidth = 84$ for the BMNG Earth database (made of six maps with $mapWidth = 21504 = 84 \times 2^{1\times8}$) and $blockWidth = 64$ for the Puget Sound database (one map with $mapWidth = 16384 = 64 \times 2^{1\times8}$). LODs were compressed during this preprocessing step to improve streaming efficiency (see Section 5.4 for preprocessing results). The data were streamed over an ADSL Internet connection with a bandwidth of about 40 kilobytes per second.

We first ran the client on a high performance computer equipped with a 3.2GHz Core 2 Duo processor and a GeForce 9800GTX+ graphics card, rendering with a screen resolution of 1680×1050 pixels and with hardware anti-aliasing enabled. We used a required rendering speed of 120 frames per second (FPS) to drive the quality factor of the measure of importance (see Section 4.2.4). This frame rate corresponds to around 1.2 million triangles rendered per frame in these conditions. 120 FPS is a very high value, but using lower values would not allow to test the adaptivity of the solution in regard to the rendering speed: we would always have to wait for data to stream because of the the low network bandwidth. Result 3D renderings are shown in Figure 3.13, after streaming data for selected periods of time.

In Figure 3.11(a), we present runtime results with an example trajectory using the BMNG Earth database, which is 2.76GB large. We can see that loading stops as soon as the frame rate gets below the given threshold: at that time, the quality factor converges to a constant value. No new loading is needed because data were loaded



(a) BMNG Earth

(b) Puget Sound

Figure 3.11: Statistics for interactive 3D rendering sessions with a high performance client, streaming data at 40KB/s. The horizontal axis is the time in seconds.

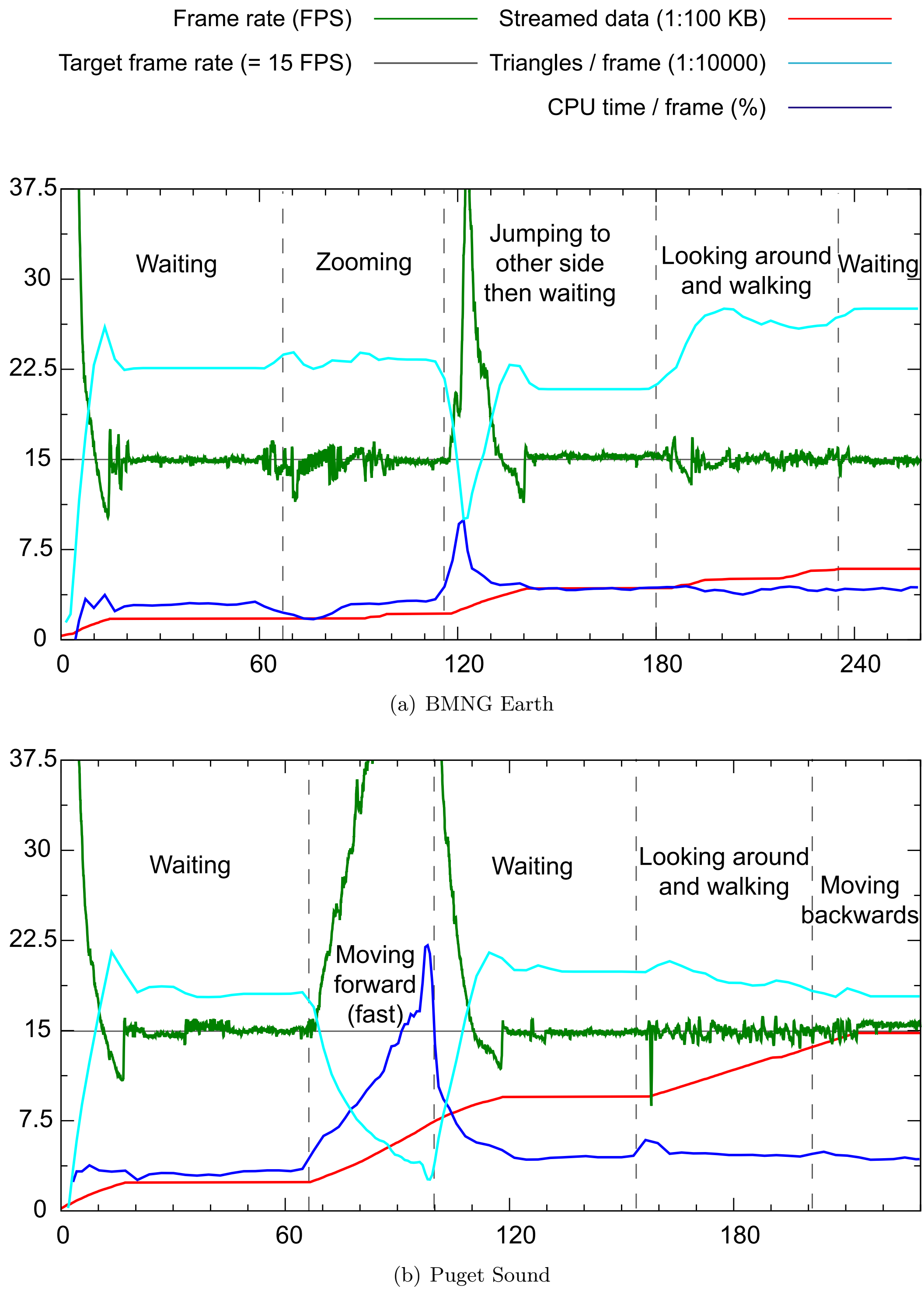
in importance order. The number of rendered samples (proportional to the number of triangles) per frame also becomes constant until a new user interaction changes a factor of the measure of importance. In standard conditions starting at second 180, the network is not always fast enough to provide maximum quality at a stable 120 FPS, but more than 95% of received data are immediately used for rendering. As we can see, importance computation and our data selection and update methods performed on the CPU take about 5 to 10% of the time for each frame. In this test, we can note that the number of rendered triangles is not identical between the different times that the target frame rate is obtained. This is certainly due to the graphics hardware performing additional optimizations like back-face culling of triangles.

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 120 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.

Results of the test with the Puget Sound database (700MB) are shown in Figure 3.11(b). When quickly moving forward, most of the available data are no longer rendered because they get behind the viewpoint. The network latency prevents us from immediately replacing them with higher quality data for visible areas, hence 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 latency: we continue to render high quality LODs even for areas getting farther until the frame rate gets too low.

We also tested our solution on a low performance computer. We used a netbook with an Atom 1.66GHz processor and a GMA450 integrated graphics chipset, using a screen resolution of 1024×600. Data were streamed over the same network as before. The target rendering speed is 15 frames per second and corresponds to around 200,000 triangles per frame in these conditions.

In Figure 3.12, we present runtime results on this low performance client, using the same databases and viewpoint trajectories as with the high performance client. We can see that the solution adapts well to these new conditions, giving similar behavior. Initial loading is faster because less data are needed to obtain the required frame rate. We can note that the frame rate is a little less stable than with the high performance client, especially when converging to the required frame rate. This is because the solution cannot always select data to correctly match the required frame rate when using large blocks. Using $blockWidth = 84$, one single block uses almost 15,000 triangles when using its last LOD, while we have about 200,000 triangles available to render the whole terrain. In Figure 3.12, we smoothed the curves that give the number of rendered triangles per frame to improve the clearness of the graph; in practice, these curves are quite jumpy at the times when the quality factor converges. As previously noted in Section 3.2.3, databases with smaller blocks are preferable on lower performance clients for the sake of stability, although they give lower general performance because there are more blocks to handle for equivalent quality.



(a) BMNG Earth

(b) Puget Sound

Figure 3.12: Statistics for interactive 3D rendering sessions with a low performance client, streaming data at 40KB/s. The horizontal axis is the time in seconds.

In Table 3.1 we present the speed of the client database update operations on both test computers. As we can see, these operations may take up to several milliseconds to complete, even on high-performance devices. This is not negligible compared to the time to render a frame and may harm rendering smoothness. This is why we perform them in separate threads. We can also see that using larger blocks results in longer operations, which is logical since more samples have to be updated. When using high values for $blockWidth$, for instance when handling color samples of a texture map (see Section 4.4), such asynchronous updates become necessary.

Database	**Refine**		**Split**		**Merge**	
	HP	LP	HP	LP	HP	LP
Puget Sound	0.3ms	1.8ms	0.2ms	1.2ms	0.1ms	0.4ms
BMNG Earth	1.2ms	8.0ms	0.3ms	1.8ms	0.2ms	0.7ms

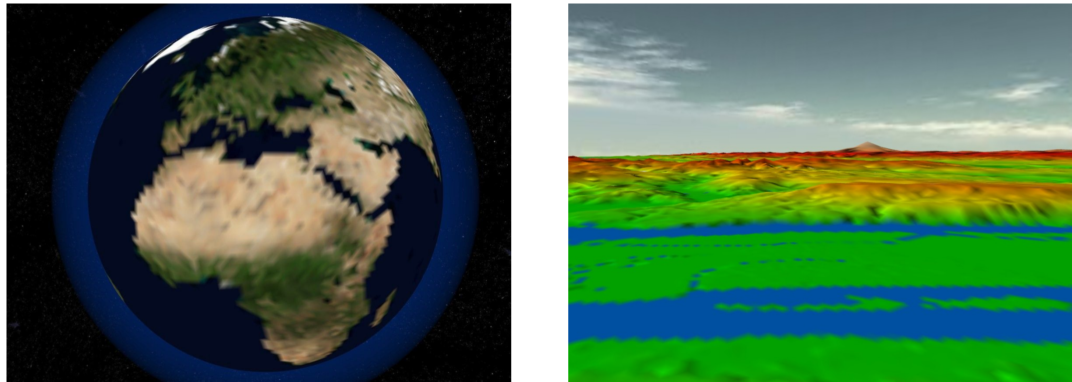
Table 3.1: Time to perform client database update operations, on a high performance computer (HP) and a low performance one (LP). Refine operations take much longer for the Earth because 3D reconstruction is more complex for planetary databases: see Section 4.5. Other differences are due to the different block sizes.

Finally, in our testing conditions, network and server limitations add a latency of about 80ms between the selection of a loading request and the start of the corresponding refine operation. This is why we run data loading in parallel with tree update operations, in addition to running both of them in parallel with data selection and rendering (see Figure 3.8). While the data loading thread is waiting for a response from the server, the tree update thread can continue performing the operations that do not require data loading.

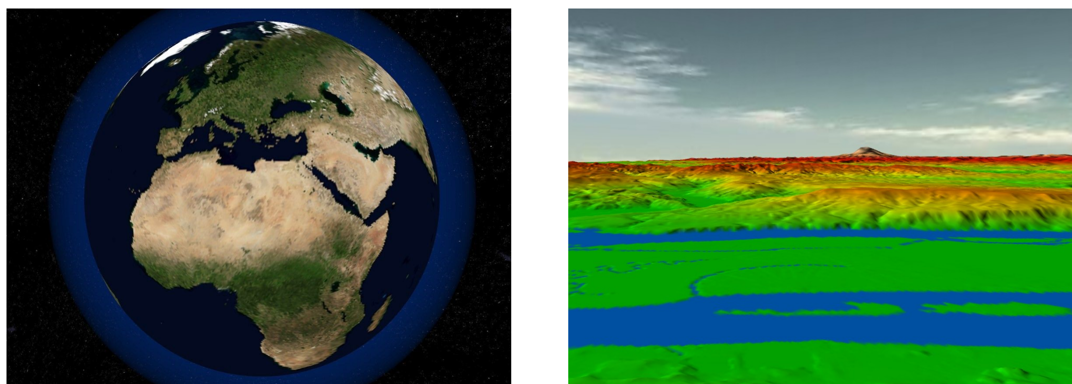
3.8 Conclusion

In this chapter, we proposed a generic solution for remote adaptive streaming and adaptive data selection for rendering of large 2D sample maps. Our methods apply whatever is done with the data; they can be used for different situations only by adapting the rendering system and the measure of importance. We can, for instance, stream a global aerial 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 routines and quality vary between these clients.

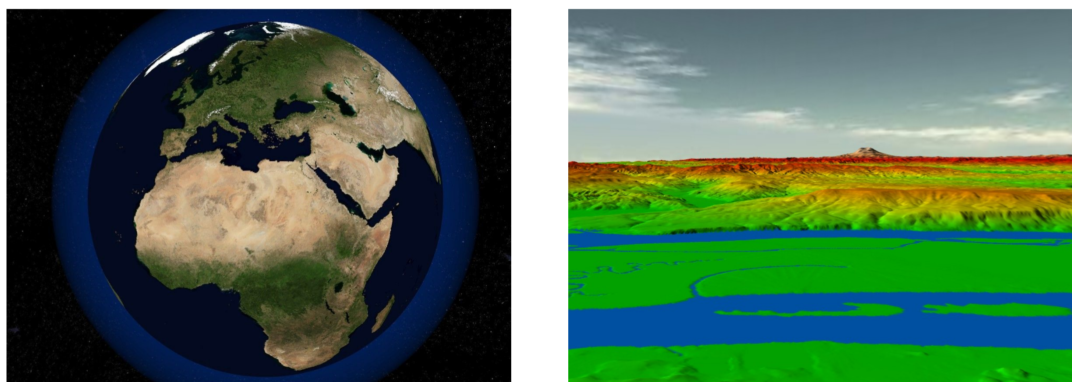
We used a data structure with good properties and added new methods to handle it more efficiently. We use this structure to manage the data from the server hard disk to the client rendering system trying to be as fast as possible. In particular we avoid loading irrelevant data, for instance by ensuring that data are not redundant between successive loadings and by always sending the most important data requests. We also



(a) Streaming for 2 seconds



(b) Streaming for 10 seconds



(c) Streaming for 40 seconds

Figure 3.13: 3D renderings of the BMNG (**left**) and Puget Sound (**right**) databases after streaming data at 40KB/s for given period.

avoid costly data structure operations as much as possible, in favor of in-place data updates and selection using sample masks.

Now that we developed this generic solution, we may base upon it to support its most challenging application: adaptive streaming and 3D rendering of large terrains, in particular entire planets. In the next chapter, we extend some parts of the solution and propose features specific to 3D rendering like texture mapping and fixing geometry artifacts. We also support the handling of planetary maps using a particular map projection and 3D reconstruction scheme.

Chapter 4

Application to 3D rendering of large terrains and planets

4.1 Overview

In this chapter, we build upon the generic solution for adaptive streaming and selection of large sample maps presented in Chapter 3 to propose a complete solution for real-time 3D rendering of large terrains based on elevation maps.

We first present how 3D rendering of elevation maps integrates with the generic solution in Section 4.2. We also propose some optimizations like triangle strip masks and data caching in graphics hardware.

Second, we fix the cracks that typically appear when rendering multi-resolution 3D terrains, in Section 4.3. We use additional sample masks for adjacent blocks so they use the same samples on their common edge. We take into account that, in our context, not all samples of a block are always available for rendering and that a block may have more than one neighbor.

We then add the possibility to refine photometry and geometry separately in Section 4.4. We enhance the generic solution to allow rendering of textured terrain geometry by maintaining a link between a texture map and an elevation map. We also improve the visual quality of the low resolution levels of detail by using sample filtering.

Finally, in Section 4.5 we describe a method for rendering planetary terrains. The adaptive solution still handles elevation maps in a generic way, but we reconstruct the 3D surface of a planet with elevation relative to a reference ellipsoid instead of a plane. We map the planet onto the faces of a bounding cube using an improved gnomonic projection, in which the surface of the planet is sampled using regular angle steps. This projection prevents storing, transferring and rendering much redundant data compared to usual cylindrical projections. We also orient and number the faces of the cube in a way that allows fixing geometry cracks on their edges almost transparently. Our last contribution addresses the issue of rendering precision: a planet is a huge 3D object which cannot be directly rendered without visible artifacts due to the limited precision of the hardware depth buffer. In our method, such artifacts are reduced by adjusting

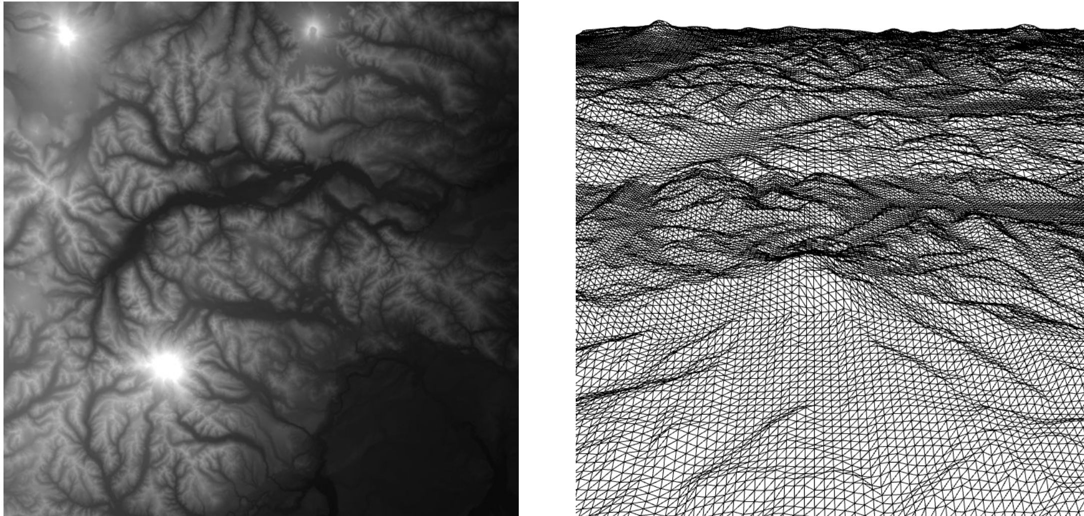
the clipping planes of the view frustum so that the rendered planet surface spreads across the entire range of depth values, hence increasing the general precision. We also take advantage of those clipping planes to cull parts of the planet located behind the horizon with no additional testing.

4.2 From elevation map to 3D geometry

The main and most interesting application of the generic solution presented in Chapter 3 is adaptive streaming and rendering of large terrains. In this section, we present how we base on this solution to handle an elevation map representing the 3D shape of a terrain and render this terrain in real-time on current graphics hardware.

4.2.1 Data conversion

Samples in elevation maps are quantized elevation values, for instance relative to a reference plane. We want to translate them into 3D floating-point coordinates in a general coordinate system before rendering. Samples are then called vertices and are connected with triangles to obtain a 3D mesh representation of the terrain that can be rendered on current graphics hardware, as shown in Figure 4.1. We can embed color values or normal vectors along with elevation values in samples and use them during rendering in order to improve the visual quality of the terrain. However, it is generally preferable to use texture maps for this kind of data (see Section 4.4).



(a) Elevation map (the brighter the higher)

(b) Corresponding 3D mesh

Figure 4.1: Hardware 3D rendering of an elevation map using triangles and vertices (subset of the Puget Sound database).

We use a data streaming and selection solution that tiles the elevation map using

independently rendered blocks of samples in order to load and render only selected parts of the data. To obtain a continuous set of triangles connecting the vertices of two adjacent blocks, one of the blocks therefore needs to access the vertices on the edge of the second one so it can connect to them. In practice, we choose to redundantly store those samples. Each block uses one additional row and one additional column in its grid of samples, respectively on the bottom and right edges, as introduced in Figure 3.3. These additional samples have the same values as the corresponding ones from the adjacent blocks at the same level of the tree. We note that discontinuities still appear when adjacent blocks are not at the same tree level, and/or do not render using the same LOD. We fix such "crack" artifacts in Section 4.3.

In the case where elevation values are relative to a reference plane (for instance, the sea level may correspond to an elevation value of zero), conversion from elevation samples to 3D vertices is simple and presented in Equation 4.1:

$$\begin{aligned} x &= u \times terrainWidth \\ y &= v \times terrainWidth \\ z &= elevation \times scale \end{aligned} \tag{4.1}$$

where *terrainWidth* is the size (in 3D units) in one dimension of the square terrain represented by the elevation map, and *scale* is the number of 3D units represented by a quantized elevation unit. These two parameters are constant over the whole terrain. On the other hand, *elevation* is the elevation value of the sample and u, v the position of the sample in the elevation map (in $[0, 1]$). The coordinate system is centered at the $0, 0$ sample of the terrain, with zero elevation.

For rendering speed reasons, we convert samples once and for all before storing them in the sample grid during refine operations (see Section 3.4.2). Only the samples in the client database change; two clients using two different rendering systems can use the same methods and stream the same data, connecting to the same server. During this data conversion step we can also, for instance, compute fake color values for vertices depending on their elevation value.

Data conversion could also be implemented with GPU shaders to transform on-the-fly a plain regular grid of vertices common to all blocks into the 3D vertices of a particular block for rendering. Such a GPU implementation would save graphics hardware memory because 3D characteristics of the samples would be stored as quantized elevation values in a texture map instead of 3D floating-point coordinates. It would also enable some nice features like real-time modification of the elevation scale. To perform this, one needs the elevation map of the block stored as a texture map, and the position and size of the block in the map. Texture coordinates of the vertices in the common regular grid represent the position of each of these vertices in the block. We can then obtain the corresponding elevation value via texture look-up, and the position of the sample in the map using the size and position of the block.

Since elevation samples are rendered in 3D, we need to take that into account when computing block visibility in order to cull those that are not visible. In practice, this is implemented by comparing the 3D bounding box of each block with the view frustum.

Bounding boxes are aligned along the axes of the 3D coordinate system to simplify this comparison.

4.2.2 Triangle strip masks

When using 3D graphics hardware to render elevation data, sample masks presented in Section 3.6.5 can be implemented using triangle strips (as introduced in Section 2.2.4.1). This standard structure defines a set of contiguous triangles to render with a succession of indices pointing to an array of 3D vertices. In our case, the triangle strip mask for one LOD points to vertices computed from the samples of the LOD as shown in Figure 4.2. In this way, the sample mask is applied directly in the graphics hardware with no additional data copy. We order our triangle strips as shown in Figure 2.9(b).

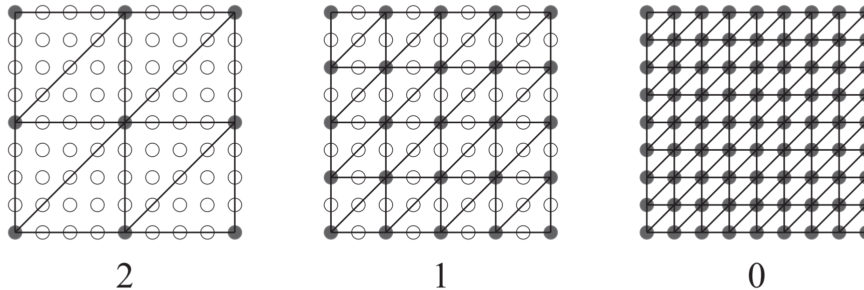


Figure 4.2: Triangle strip masks for 3D rendering of a block with elevation samples (same block as in Figure 3.3). For each LOD, the corresponding samples in the grid are connected to create triangles. Elevation values are used to compute 3D coordinates of the vertices.

This system could allow rendering a block using virtual LODs with lower definition than the first LOD of the block, by skipping even more vertices from the sample grid as long as we can subdivide block resolution enough. For instance, in the case presented in figure 4.2, one could use only the four corner samples of the block to obtain a virtual LOD number 3. However, using such LODs would cause geometry cracks to happen more frequently and we would not be able to use the solution presented in Section 4.3.3 to ensure that we can fix cracks in all cases.

4.2.3 Caching in graphics hardware

Most of the time, a given block is rendered using the same LOD for several successive frames because the importance does not vary that much or because a new LOD is requested but not available yet. In such cases, we avoid uploading the same LOD data again and again onto the graphics hardware at each frame by storing them in the dedicated memory of the graphics hardware.

We perform this on recent graphics hardware using buffer objects. Both the array of vertices and the triangle strip of the LOD are stored in this structure, which automatically caches them in the graphics hardware memory. We replace them with new

data only when another LOD is selected for the block. On recent graphics hardware able to render millions of triangles per frame at acceptable speed, this saves many data transfers from main memory to graphics hardware, as well as memory because triangle strips are shared between all the blocks. In practice, this technique improved rendering speed by up to 40% depending on circumstances.

When a new LOD is selected for a block, we entirely discard previously cached data and upload the new LOD. We could improve this technique by using incremental updates as proposed in [SW06] since each LOD re-uses the samples from the previous ones. This would require to append the samples of each new LOD to the end of the previous one as when transferred, instead of using a 2D grid of samples with spatially interleaved LODs.

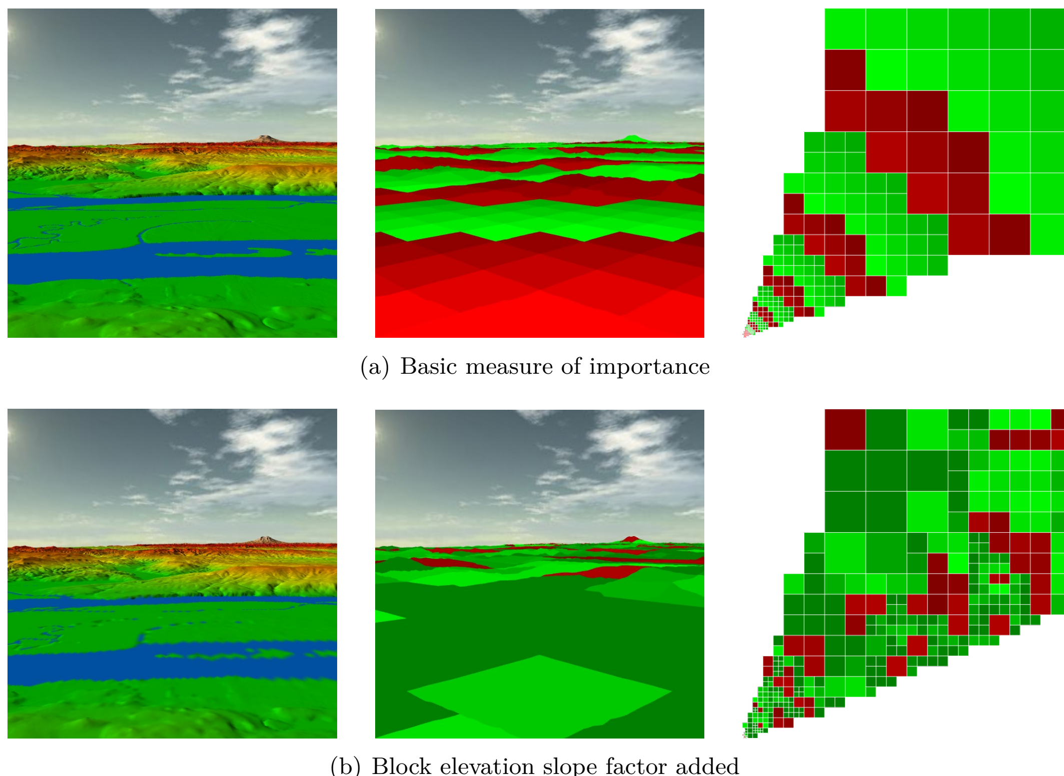
Using the GPU to compute the 3D coordinates of vertices as proposed in Section 4.2.1 would also save memory transfers and allow caching in graphics hardware. In practice, the LODs of all blocks would be implemented as constant regular grids of vertices, updated on-the-fly with referenced elevation texture maps. Such texture maps are cached in graphics hardware by definition (see Section 4.4.3).

4.2.4 Measure of importance

In this section, we specialize the factors of the generic measure of importance (presented in Section 3.6.2) to the context of 3D rendering of elevation maps.

The general quality factor is driven by the number of frames per second achieved by the rendering system. In practice, the user selects a target number of frames per second desired for rendering. We consider that rendering speed is roughly proportional to the number of rendered triangles (and therefore of rendered samples, as we can see in Figure4.2), and so is the computed quality factor. Whenever the rendering speed gets over or under the target 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. However, in Section 3.6.2 we explain that we work with 1D factors for the measure of importance. We therefore need to use the square root of the computed quality factor instead. In this case, the quality factor has a linear impact on the importance of the blocks, as desired.

Other factors introduced in Section 3.6.2 are the radius of the area covered by the block and the distance to the viewpoint. In practice, we obtain the radius from the 2D sampling definition of the block, and we compute the distance to the viewpoint from the center of the 3D bounding box of the block. To prevent a block from switching LODs and triggering update operations while the viewpoint is actually inside the bounding box of the block, we subtract the radius from the viewpoint distance. Considering that the elevation slope of the block is negligible compared to its 2D area, this gives a reasonable estimate of the distance between the viewpoint and the closest possible 3D vertex of the block. We can also consider that this distance is the same for a block and its child that is the closest to the viewpoint. Therefore, we simplified the problem to the case presented in Figure 3.10 and the split operation threshold can coincide with virtual LOD number -1 of the parent (see Section 3.6.4).



(a) Basic measure of importance

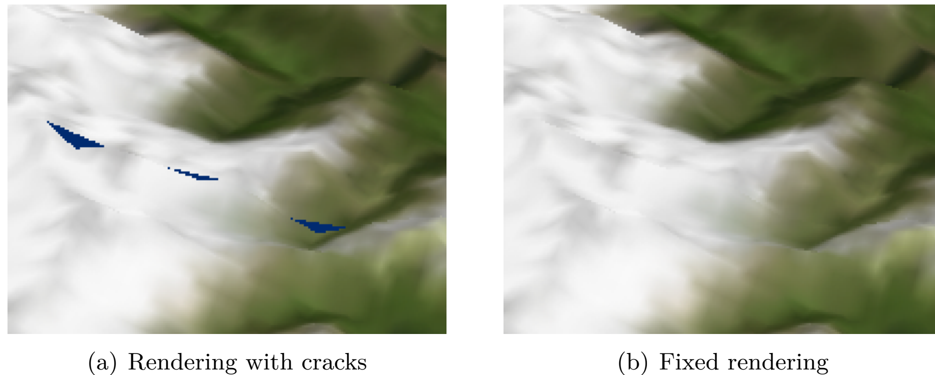
(b) Block elevation slope factor added

Figure 4.3: Impact of the measure of importance on 3D rendering, using the Puget Sound database. **Left** – 3D rendering of the terrain, with colors embedded in vertices in order to better emphasize 3D mesh quality differences. **Middle** – 3D rendering of the terrain using block importance as colors: red is more important than green and the brighter, the more important. Using two LODs per block, we select them as in the picture: green for the first LOD and red for the second. **Right** – Top view of the terrain with block bounding boxes, view-frustum culling enabled (see Section 3.6.1). The viewpoint is located on the bottom-left corner of the terrain, looking towards the top-right one. **a)** Driving the importance with the viewpoint distance creates color layers around the viewpoint corresponding to the levels of the tree: blocks split as the corresponding terrain area gets more importance, and the children get lower importance because they are smaller. **b)** When using a simple estimation of block geometry roughness, we can see mountainous areas get higher importance so mountains are rendered with higher quality. On the contrary, areas in the foreground are quite flat, so they are rendered using much less samples. This is especially visible since colors are embedded in 3D vertices. We explain in Section 4.4 how to allow rendering few 3D vertices with high photometric quality, using texture maps with a different measure of importance.

Other information can be used to get a more specific measure of importance, like the incident angle of the viewpoint and a geometry roughness factor. The impact of the measure of importance on our 3D rendering application is shown in Figure 4.3.

4.3 Fixing geometry cracks on block edges

In the context of 3D geometry rendering with triangles, subdividing a surface using blocks with LODs creates a typical artifact illustrated in Figure 4.4: vertical gaps or "cracks" appear when two contiguous blocks do not use the same samples on their common edge. With no particular handling, this happens systematically when such blocks do not render using the same definition. In this section, we present how we adapt the geometry of the blocks to avoid those artifacts.



(a) Rendering with cracks (b) Fixed rendering

Figure 4.4: Example renderings with and without crack artifacts (zoom). Note that we can unrealistically see blue sky through cracks.

4.3.1 Additional adaptive triangle strip masks

In Section 4.2.2 we showed how we use triangle strip masks to extract the samples of a LOD from the common sample grid of a block. To ensure that two adjacent blocks or "neighbors" use the same samples on their common edge, we use additional triangle strip masks for block edges as shown in Figure 4.5.

For any edge, it is always the block using the higher definition LOD that adapts, skipping samples not used by the other block. Otherwise, we would need data that have not been loaded yet. When a block needs adaptation on any of its edges, we render this block using five triangle strip masks: a center strip which does not skip any sample, and one strip per edge that stitches the center strip to the corresponding neighbor block. Note that some edges may need no adaptation but need to use an additional strip because another edge requires it. This is the case for the left edge of block (b) in Figure 4.5. As we can see, such strips do not skip samples. In addition, only one large strip mask is used for blocks that need no adaptation at all.

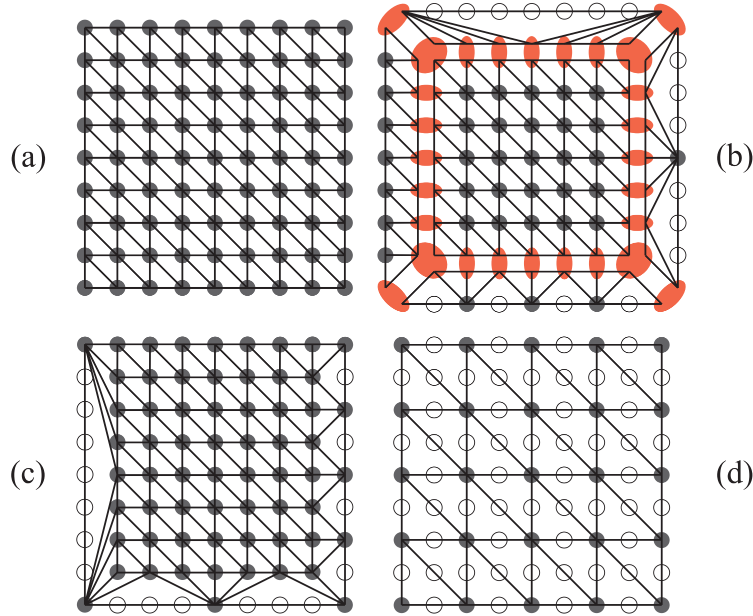


Figure 4.5: Additional triangle strip masks for block edges. **a)** Block using its last LOD. **b)** Block using its last LOD, with additional triangle strips on its edges to adapt to its neighbors based on the LOD difference. Left: no difference, bottom: one level, right: two levels, top: three levels. Red samples appear stretched here so we can see the different strips. **c)** Mirror of (b), with the strips tied as when rendered. **d)** Block using its next to last LOD.

LODs are regularly sampled so we know in advance which samples are used on the edges of any block rendering at a given LOD. Therefore, we can compute the additional edge strip to adapt to a block just by knowing the difference in definition between the LODs of the two blocks, or "LOD difference". The LOD difference $lodDiff$ of a block *this* with its neighbor block *neighbor* is obtained from the selected LOD numbers *lod* and the tree levels *sub* of both blocks: $lodDiff = neighbor.lod - this.lod + (neighbor.sub - this.sub) \times subDepth$. While the number of LODs per block is $subDepth + 1$, recall that the last LOD of a block has the same definition as the first LOD of blocks one level lower in the tree. We therefore multiply the tree level difference by *subDepth* only.

For instance, in Figure 4.5, block (b) adapts to neighbor block (d) which has twice lower definition on their common edge (one LOD difference). Block (b) therefore uses an additional triangle strip that uses only one sample in two from its selected LOD on this edge. In practice, we pre-compute all possible additional edge triangle strip masks. We can thus choose which strip mask to apply on any edge based on the selected LOD, the edge orientation and the LOD difference with the neighbor block.

4.3.2 Data structure addition: neighbor blocks

Deciding whether a block needs to adapt on its edges require knowledge of its neighbor blocks. We therefore maintain a table of four neighbors per block: one per edge. The neighbors may be on different tree levels because the tree is unbalanced. However, the table is not updated when a neighbor that is on the same tree level as the block splits because this would lead to more than one neighbor per edge. In that case, we know that the block has a definition lower or equal to that of the children of the neighbor, thus the block will not adapt on the corresponding edge until these children are merged.

During a split operation, newly created children get their neighbors amongst the other children and the neighbors of the parent as explained in Algorithm 4.1. We first compute the coordinates of the neighbor in the child grid of the parent, which identifies the corresponding child. However, when the requested neighbor is out of the child grid, we need to get it from across the edge of the parent. Note that if the neighbor of the parent has split, we know that it is on the same tree level. We can therefore obtain the child on the common edge which is the neighbor of the newly created block. Once we have found the neighbor of a block, we update the table of neighbors of the new block, but also that of the neighbor when both blocks are on the same tree level. The children of the new neighbor, if any, are also updated recursively.

Algorithm 4.1: Find and update the neighbor of a newly created block along its left edge (other edges use the same algorithm, with corresponding directions).

Data: *block*: a newly created block with coordinates u, v inside the child grid of its parent block *parent*.
if $u - 1 \geq 0$ **then**
    `// neighbor is a "brother" of block`
    neighbor ← child of *parent* with coordinates $u - 1, v$
else
    neighbor ← neighbor of *parent* on left edge
    if *neighbor is not a leaf* **then**
        neighbor ← child of *neighbor* with coordinates $2^{subDepth}, v$
        `//` $2^{subDepth}$ `is the maximum value for child coordinates`
left neighbor of *block* ← *neighbor*
if *neighbor and block are on the same tree level* **then**
    right neighbor of *neighbor* ← *block*   `/* recursive update */`

During merge operations, the neighbors of a removed block are updated similarly. For each of them, if the neighbor on the common edge is the removed block, it is updated as the parent of the removed block instead.

When rendering a block using a given LOD, for each edge we check whether the neighbor is rendered. A negative answer means that the neighbor has split or is not visible, consequently the current block does not adapt on this edge. If the neighbor is rendered, we then compute the LOD difference. The current block adapts on the

common edge only if the LOD difference $lodDiff$ is positive, i.e. it uses a higher definition than the neighbor.

4.3.3 Avoiding unsolvable cases

Most existing solutions force neighbor blocks to be either on the same tree level or to have one level difference. They may even prevent blocks from using internal LODs to reduce the number of cases where cracks appear. This ensures that neighbors either do not need adaptation or adapt with only one LOD difference. We chose to allow more neighbor LOD differences to get more freedom in LOD selection, and to avoid having to split a high number of neighbor blocks just because one of them needs to be split. However, there are cases where we cannot stitch the common edge of two neighbor blocks if they have a high LOD difference. This happens when the lowest-definition LOD of the neighbor on the lower tree level has a definition higher than the highest-definition LOD of the other block, as illustrated in Figure 4.6. This occurs especially often when using a $blockWidth$ not of 2^n form.

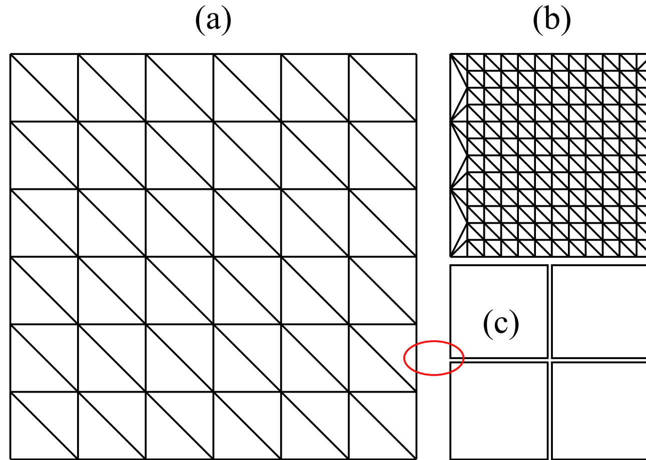


Figure 4.6: Example of neighbor blocks that cannot be stitched. **a)** A 13×13 block ($blockWidth = 12$) using its first LOD (7×7 sample grid subset). **b)** A neighbor block located one lower tree level, using its last LOD. It adapts correctly on its left edge. **c)** Neighbor blocks located two tree levels lower. These blocks cannot adapt to block (a) whatever LOD they use because one of their corner samples (circled in red) is not present in block (a).

To ensure zero crack, we set a safe maximum tree level difference between neighbors. It is the number of times we can divide the resolution of the first LOD (in one dimension) by two times the number of LODs per block. In other words, it is the largest integer x such that $blockWidth$ is divisible by $2^{subDepth \times (x+1)}$.

Ensuring a maximum tree level difference requires using new constraints for split and merge operations. First, a block cannot merge if it is the neighbor of a block that is on the maximum allowed tree level in regard with the tree level of this block. This check

needs to search the sub-trees of all neighbors. Second, a block cannot split if it has a neighbor that is on the minimum allowed tree level. However, the tree structure may change between a trigger and the actual operation. Consequently, these constraints are checked just before performing corresponding operations, not when triggering.

The second constraint could prevent some very important blocks from splitting, for instance when a block is just in front of the viewpoint but has an invisible neighbor just behind the viewpoint, which therefore will not refine or split. To circumvent this problem, we check the split constraint a first time before triggering the operation. If it is not met, we instead force the neighbors of the current block to trigger a refine and/or split operations, using the importance of the current block and ignoring their visibility. The current block will be able to split when the constraint is finally met.

4.4 Texture maps

Terrain texture maps are usually larger than elevation maps, and they have different visual meaning for the user. In practice, they represent photometric properties for the corresponding terrain surface area. Texture maps can be either color maps, or normal maps that improve the shading of the terrain surface. We want to be able to manage texture levels of detail apart from the elevation. As a proof-of-concept, we chose to support texture maps using the generic solution presented in Chapter 3. As for elevation, we use a tree of blocks with LODs to represent the original texture map.

4.4.1 Data structure addition: bound trees

A texture block does not render itself, it is used by one or more elevation blocks when they are rendered. Texture and elevation maps use distinct trees. Consequently, we need to maintain a link between the two incomplete trees on the client so that elevation blocks can know which texture blocks to use. In practice, each block stores a pointer to the block of the other tree which covers the same terrain area, if any.

Classical texture rendering requires that an elevation block rendered with one triangle strip uses only one texture block. In practice it uses the bound texture block if present, otherwise it recursively asks its parents until a bound texture block is found. Conversely, a texture block cannot be rendered if it has no bound elevation block. As a consequence, we need to use texture blocks with a resolution higher than that of elevation blocks if we want to be able to get significantly higher definition for texture than for geometry.

We need to use some constraints on split and merge operations to ensure that we never obtain texture blocks that cannot be rendered. First, a texture block cannot split if the bound elevation block is a leaf of the tree. Second, an elevation block cannot merge if the bound texture block is not a leaf. However, split and merge operations are performed asynchronously in the generic solution. We therefore need to check the constraints first when triggering the operations, but also when starting to perform them because the structure of the trees may have changed in between.

There are several solutions to allow one elevation block to use more than one texture block for rendering, and therefore to avoid such constraints. For instance, we could use multi-texturing with specially crafted textures, but this would severely harm the rendering speed because of runtime texture generation. We could also render elevation blocks using one triangle strip per texture block, but this would cause much more rendering calls and require additional constraints for fixing cracks. Finally, we could use a GPU shader to correctly map different textures on the corresponding parts of the geometry based on texture coordinates.

4.4.2 Measure of importance

The main advantage of using our solution to manage texture maps is that we can independently select a LOD for any terrain area. In practice, we use the measure of importance with factors presented in Section 4.2.4 while offering the option to add factors designed for texture maps, though we did not implement such additional factors in our application. For instance, we may use the incident viewpoint angle and rendering screen resolution in a way similar to hardware mipmapping LOD selection. This would allow to give importance to areas covering a high part of the screen, as well as to avoid texture aliasing problems. We may also use an heterogeneity factor to give importance to areas with many visible details that can be refined. Finally, we may avoid slow runtime paging of graphics hardware memory by capping the quality factor using the available amount of texture memory, hence forbidding further refinement when all the graphics hardware memory is used.

However, in any case importance values should be comparable between maps of different nature, either elevation or texture, in order to get a consistent behavior of the request selection scheme presented in Section 3.5.2. We therefore need to ensure for any additional factor that the values amongst all blocks of the map average to 1, unless one deliberately wants to give more importance to maps of a particular nature.

4.4.3 Rendering

When culling blocks that are not visible to avoid them triggering refine and split operations (see Section 3.6.1), we noticed that the visibility of a texture block and the bound elevation block are the same because they cover the same terrain area. We thus compute visibility of elevation blocks first, then re-use the result for the bound texture blocks.

Elevation blocks need texture coordinates to correctly map texture blocks on their vertices. We store one single mask for the texture coordinates of all the elevation blocks. When the rendered texture block covers a larger area than the elevation block (i.e. it is on a higher tree level), we use the texture matrix of the graphics hardware to automatically zoom in and shift the texture coordinates and obtain the corresponding sub-area.

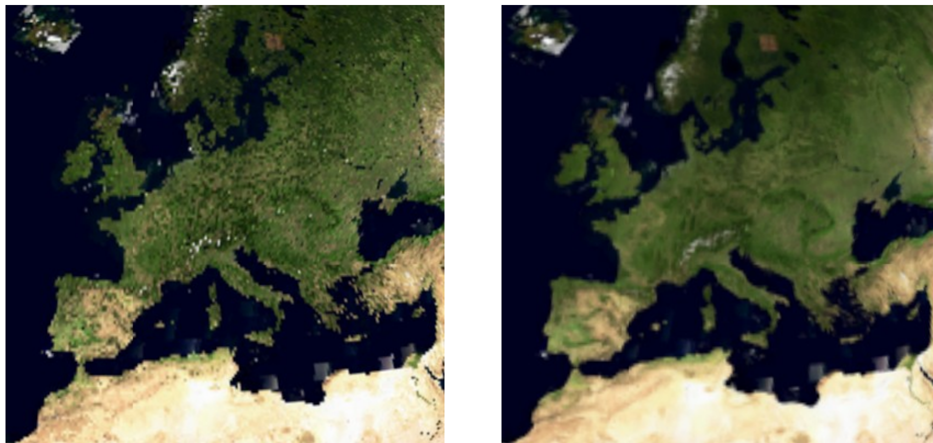
Finally, texture LODs are implemented as mipmaps for 3D rendering. When we load a new LOD, we add a mipmap to the texture. The graphics hardware can then

independently select which LOD to use for rendering. To avoid preventing mipmapping when only the first LOD of a block is available, we also create lower resolution mipmaps during split operations; they correspond to virtual lower definition LODs. Unlike geometry, texture maps and their mipmaps are automatically cached in graphics hardware to achieve best rendering performance.

4.4.4 Filtering

Although regular point down-sampling is often acceptable for elevation maps, this is not the case for texture maps. We therefore choose to use filtering when building the texture tree during the creation of the server file. We may use any image filtering method as long as it produces regularly sampled LODs.

We implemented the *Progressive Texture Map* (PTM) texture filtering method [MB03]. It uses simple and fast bilinear interpolation, and stores small delta values within each LOD to losslessly reconstruct samples based on 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 loading. Figure 4.7 offers a visual comparison. We choose this method because it is very fast to decode so it can be used with low performance clients.



(a) Point sampling (b) PTM filtering

Figure 4.7: Example renderings of the Earth TrueMarble database with and without data filtering. With no filtering Spain and Morocco look tied, ground is grainy, coastlines and surface features are not continuous. With filtering, everything looks much smoother but most topological details are better conserved.

When using data filtering, different LODs of a block use different values for a given sample. This prevents us from using a single sample grid for each block. We therefore store each LOD of the block in a separate grid. This implies some data redundancy: with each LOD quadrupling the number of samples compared to the previous one, the memory overhead tends towards 33% compared to redundancy-free storage. Anyway,

using mipmaps for texture map rendering causes the same overhead in graphics hardware memory: using data filtering better takes advantage of this system. During split operations, we split all LODs instead of re-creating lower resolution mipmaps. After several successive splits, one LOD may become unable to split because its resolution is not subdivisible enough. In that case, we cache the LOD on the parent block and stop using it until the block merges.

4.4.5 Interpolation continuity problems on block edges

Similarly to geometry cracks, using texture blocks may cause continuity problems on block edges. This occurs when using bilinear filtering for texture rendering, which is a feature one wants to use on any recent graphics hardware to improve visual quality. No interpolation may take place on block edges because it is also the edge of the texture itself: graphics hardware cannot know which texture block is on the other side of the edge. Therefore, a seam is clearly visible on edges, as shown in Figure 4.8.

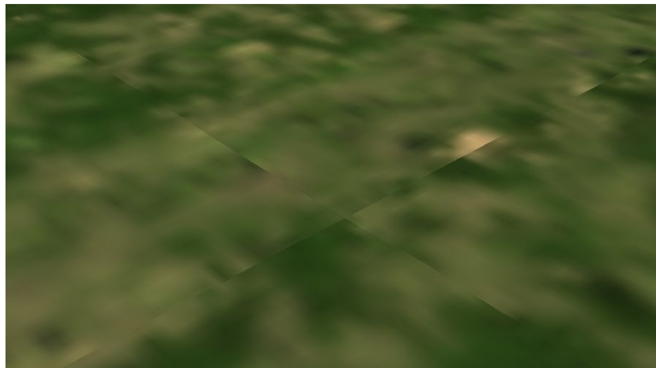


Figure 4.8: Example of independent texture blocks rendered using bilinear interpolation. Discontinuities are clearly visible at block edges.

A solution is available to circumvent such texture tiling problems in graphics hardware, called texture borders. This consists in storing additional rows and columns along the edges of the texture, which should contain the same samples as the adjacent texture and are used for bilinear interpolation. However, in our solution any block can switch LODs at any time and therefore use a different number of edge samples with different values. Implementing texture borders would then require obtaining the edge samples of adjacent textures at each frame of rendering, in a way similar to how we choose additional triangle strips masks to fix geometry cracks (see Section 4.3.1). This would imply very frequent texture updates and severely harm rendering speed.

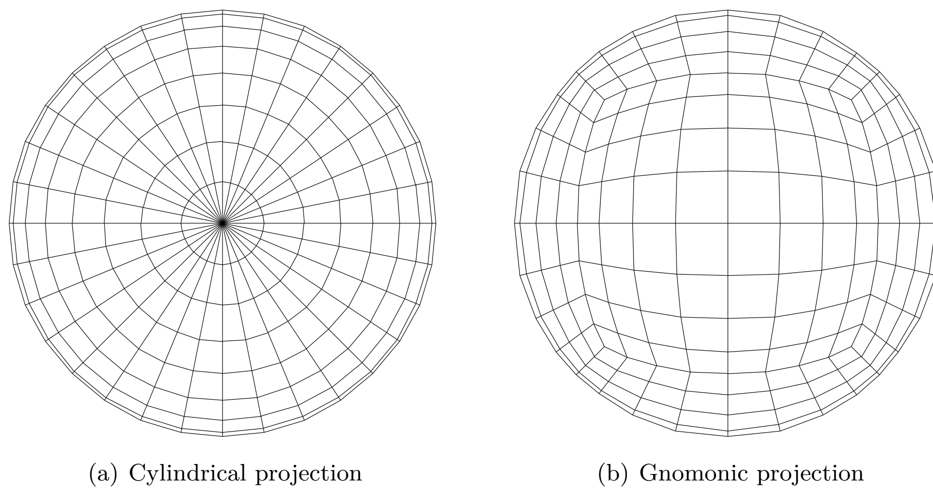
4.5 Planetary terrains

The generic solution for adaptive streaming and rendering of terrains presented in Chapter 3 can support any size of database. In this section, we present how we take

advantage of this solution to handle huge planets as well as the planet-specific routines we add to render such particular type of terrains.

4.5.1 Gnomonic map projection on a cube

To represent a planetary terrain using maps of samples, one needs to perform a 2D projection of a 3D surface. Planets are generally modeled using a reference sphere then mapped to a rectangle with a cylindrical projection. However, such projections induce some huge distortion in terrain areas located around the poles as can be seen in Figure 4.9(a). Those areas get far more samples than those around the equator, and square blocks of samples get a triangular shape when reconstructed in 3D.



(a) Cylindrical projection (b) Gnomonic projection

Figure 4.9: Comparison between map projections: isometric views of reconstructed 3D planets centered at the north pole, with blocks subdividing the 2D map into square sub-areas. We can see that blocks around the pole become very small and stretched in the cylindrical projection, although they use the same resolution as other blocks. Gnomonic projection offers less distortion and sampling disparities.

Instead, we chose to map the planet onto the six faces of a bounding cube, thus obtaining six square maps. We use the gnomonic projection [Weia] to project points from the surface of the reference sphere to the nearest tangent face of the cube as shown in Figure 4.10. We chose this projection because it offers a more uniform sampling than simple cylindrical projection as seen in Figure 4.9. This saves most redundant data storage, resulting in 25% smaller databases (see Section 5.4 for preprocessing results) and less rendering distortion. This projection has also the advantage of offering fast 3D reconstruction since it only requires one 3D vector normalization.

Elevation values are relative not to the surface of the cube but to the reference surface of the planet defined by a given datum. A datum defines a geographic coordinate system that allows to handle spherical coordinates but reconstruct a more accurate ellipsoidal model of the planet. For instance, the WGS84 standard is used for the

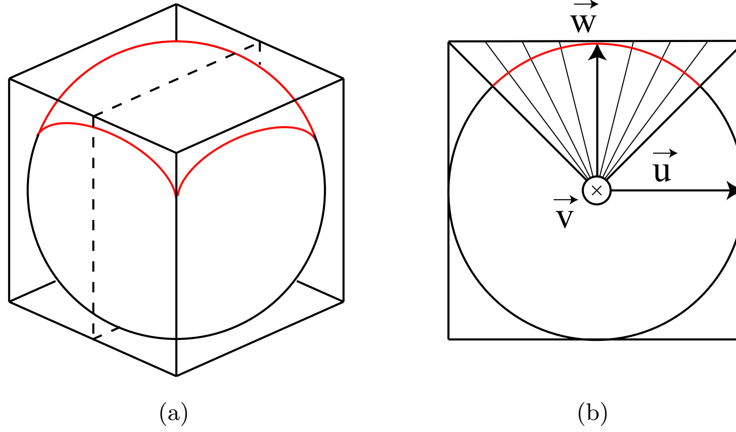


Figure 4.10: Mapping a spherical planet onto a cube with the gnomonic projection. **a)** Side view of the cube. The top part of the sphere (in red) projects onto the top face of the cube. **b)** 2D side cut of the cube along the dotted plane from (a). The gnomonic projection maps points from the surface of the sphere to the plane of projection along their direction from the center of the sphere. The cube face is regular sampled.

Earth. To render the planet correctly we need to project the samples back, keeping the proportions and curve of the planet. For each sample of a block, we first 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 and add the de-quantized elevation value of the sample to get its 3D coordinates in a planet-centric cartesian coordinate system, with axes oriented along the edges of the cube. Those coordinates define a 3D vertex which is used to render the sample. The whole computation for one sample is presented in Equations 4.2.

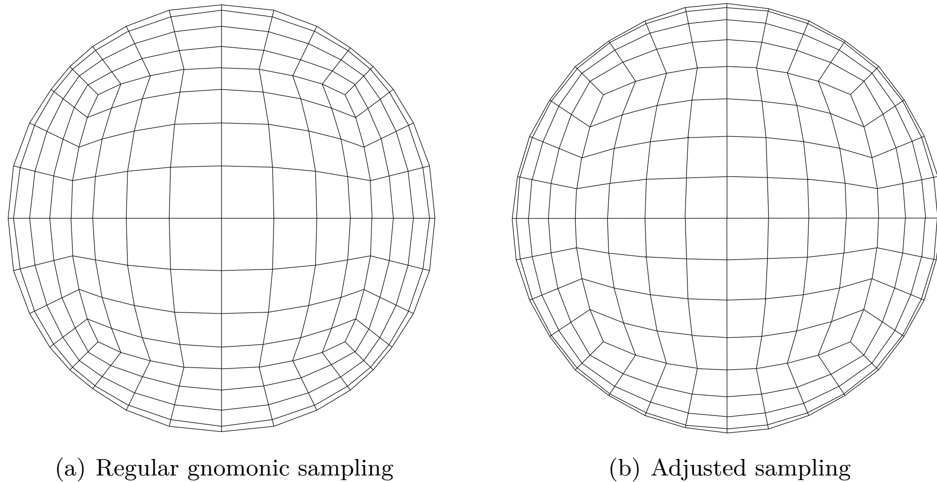
$$\begin{aligned} N &= \frac{r}{\sqrt{1 - ecc \times w^2}} \\ h &= elevation \times scale \\ x &= u \times (N + h) \\ y &= v \times (N + h) \\ z &= w \times (N \times (1 - ecc) + h) \end{aligned} \tag{4.2}$$

where r and ecc are the equatorial radius and the eccentricity of the planet defined by the datum and $scale$ is the number of 3D units represented by a quantized elevation unit. These three parameters are constant over the whole planet. On the other hand, $elevation$ is the elevation value of the sample and u, v, w the unit vector obtained by normalizing the coordinates of the sample on the gnomonic plane of projection (with $u, v \in [-1, 1]$ and $w = 1$ before normalization, as can be seen in Figure 4.10) then rotating this vector around the center of the planet to obtain the desired cube face. Finally, x, y, z are the coordinates of the resulting 3D vertex.

As introduced in Section 4.2.1, this transformation is performed during the refine operation when the data are received and could as well be implemented on GPU. We just need to apply Equation 4.2 instead of elevating vertices from the plane.

4.5.2 Adjustment for improved sampling

Projecting a planet onto a cube using the gnomonic projection offers more uniform sampling than simple cylindrical projection of the whole planet. However, like any planetary map projection there are still sampling inconsistencies: samples get closer on the surface of the sphere as we get farther from the center of projection. As a consequence, the area covered by a sample is approximately 75% smaller around the corners of the cube than around the center of the faces, as we can see in Figure 4.11(a). We limit this problem by applying a simple adjustment to the projection.



(a) Regular gnomonic sampling (b) Adjusted sampling

Figure 4.11: Comparison between gnomonic map samplings: isometric views of 3D reconstructed planets centered at the north pole. We can see some disparities in the areas of the blocks when using regular sampling of the gnomonic plane of projection: blocks located around the corners of the cube are smaller than those at the pole. There are much less of these disparities with our sampling adjustment. In particular, this is visible at the equator (exterior bound of the planet in these pictures) which now gets regularly subdivided like in figure 4.9(a) although it is split into four maps.

Instead of regularly sampling the plane of projection, we choose to sample the map directly on the surface of the sphere with steps expressed as angles, as presented in Figure 4.12. Since we use a planet-centric cartesian coordinate system whose axes are aligned with the edges of the cube, we can define two leading axes $\vec{u}, \vec{v}$ for each face of the cube, with the third one $\vec{w}$ passing through the center of projection of this face. We can then perform independent 2D rotations around both of these 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. This sampling is more uniformly distributed on

the surface of the planet as seen in Figure 4.11. The area covered by a sample is only 33% smaller around the corners of the face than around the center of projection.

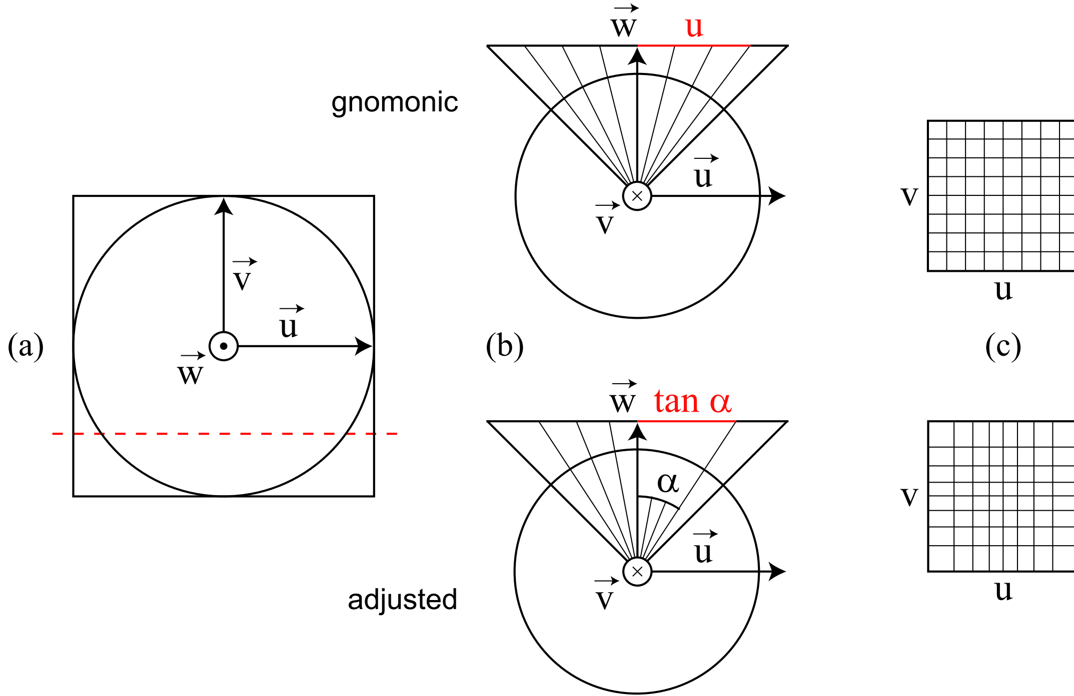


Figure 4.12: Sampling adjustment for the gnomonic projection. **a)** Top view of the cube. We use vectors $\vec{u}$ and $\vec{v}$ to parameterize the top face. **b)** 2D side cuts of the cube along the dotted red line from (a). Gnomonic sampling (**top**): the line of projection is regularly sampled by translating along $\vec{u}$. Adjusted sampling (**bottom**): the surface of the circle is regularly sampled by rotating around $\vec{v}$. **c)** Top views of the face with projected samples. Gnomonic sampling: the face is a regular 2D map. Adjusted sampling: the coordinates of a sample in the face are the tangent values of its angular coordinates.

When projected onto a face of a cube using the gnomonic projection, our samples do not map regularly on the plane of projection. 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 4.12(b)). We can therefore store sample values in a regular 2D map, then obtain the coordinates of any sample on the gnomonic plane in $[-1, 1]$ by computing the tangents of the two angles.

We note that we could incrementally compute the coordinates of all samples of a block in the gnomonic plane of projection without using any runtime tangent computation. This would require to pre-compute the tangent value of every possible angle step and to store the coordinates of the center of each block in the gnomonic plane of

projection. The trigonometric identity $\tan(\alpha+\beta) = \frac{\tan\alpha+\tan\beta}{1+\tan\alpha\tan\beta}$ could then give the coordinates of any sample of the block. However, rounding errors would incrementally cumulate as we use limited precision floating point values. This would lead to unacceptable visual problems like the vertices on the common edge of two neighbor blocks not exactly coinciding.

4.5.3 Fixing geometry cracks on cube edges

The faces of the cube used to project the planet are different maps, but they all represent a single 3D terrain. We should therefore correctly stitch the edges of those maps to avoid cracks in 3D rendering. In this section, we implicitly handle this peculiar case so the solution proposed in Section 4.3 is fully applicable with no data structure addition.

We number and orient the faces of the cube as presented in Figure 4.13 to facilitate neighborhood management. In particular, the axes of adjacent maps along their common edge point in the same direction to ensure that blocks along the edge have the same order on both sides. That way, we may use the same system as for neighbors that are part of the same map. However, adjacent 2D maps cannot have the same axis on their common edge in all cases because they are the faces of a closed 3D object. We thus want, for given face and edge, to directly identify the neighbor face and know which of its edges is concerned. Table 4.1 gives the neighborhood correspondences between the faces of the cube.

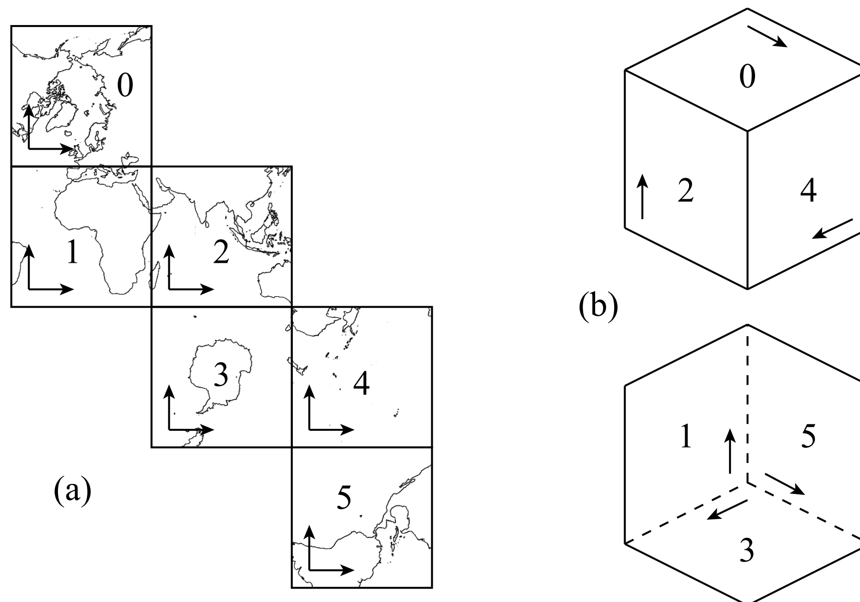


Figure 4.13: Orientation and numbering of the faces of the cube. **a)** Patron of the cube with continents of the Earth for reference. Horizontal axes: $\vec{u}$, vertical axes: $\vec{v}$. **b)** Side view of the cube (top: visible side, bottom: hidden side). $\vec{v}$ axes are represented near the "left" edge of the face ($u = -1$). All $\vec{w}$ axes are oriented towards the exterior of the cube to form cartesian coordinate systems.

<table>
<tr><th>Face #</th><th>Edge</th><th>Neighbor #</th><th>Neighbor edge</th></tr>
<tr><td rowspan="4">Even</td><td>Left</td><td>+5</td><td rowspan="2">Right</td></tr>
<tr><td>Top</td><td>+4</td></tr>
<tr><td>Right</td><td>+2</td><td rowspan="2">Top</td></tr>
<tr><td>Bottom</td><td>+1</td></tr>
<tr><td rowspan="4">Odd</td><td>Left</td><td>+4</td><td rowspan="2">Bottom</td></tr>
<tr><td>Top</td><td>+5</td></tr>
<tr><td>Right</td><td>+1</td><td rowspan="2">Left</td></tr>
<tr><td>Bottom</td><td>+2</td></tr>
</table>

Table 4.1: Neighborhood correspondences between the faces of the cube. Left, top, right and bottom refer to edges where $u = -1$, $v = 1$, $u = 1$ and $v = -1$ in respective order. Neighbor face number is obtained by adding the given value to the current face number, modulo six. We know that both faces use the same direction for the axis on their common edge.

4.5.4 Adaptive clipping planes for improved precision and culling

Graphics hardware use $near$ and far clipping planes to bound the depth of the rendered geometry. 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 that no visible part of the planet (or even the whole planet) is abusively ignored. However, using an arbitrary large $near - far$ difference decreases the rendering precision and causes flickering problems due to the limited precision of the hardware depth buffer. Moreover, when placing the far plane behind the planet, any surface that is visually culled by the planet itself (the grayed area in Figure 4.14) is uselessly processed by the graphics hardware. To prevent these problems, we adapt both planes according to the position of the viewpoint. Resulting renderings are presented in Figure 4.15.

Clipping planes are normal to the direction of the viewpoint and are defined by their distance from 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.

We place the far distance at the horizon; it is computed as explained in Figure 4.14. Using this method, the rendered surface gets smaller as the viewpoint gets closer to the planet. The adaptive solution can then use more data to improve the quality of visible and important areas. Equations 4.3 show how we compute the desired distance $\|VK\|$ from the known values $\|OT\|$, $\|OM\|$, $\|VO\|$ and $\alpha = \widehat{KVO}$. $\|OT\|$ and $\|OM\|$ are the minimum and maximum values for the radius of the planet, computed from the datum (which defines the equatorial and polar radiuses) and the minimum and maximum elevation values in the database. $\|VT\|$ and $\|TM\|$ are obtained with the pythagorean theorem. In practice, the difference between $\|OT\|$ and $\|OM\|$ is quite small, thus $\|TM\|$ is smaller than shown in Figure 4.14. For instance, using the WGS84 datum

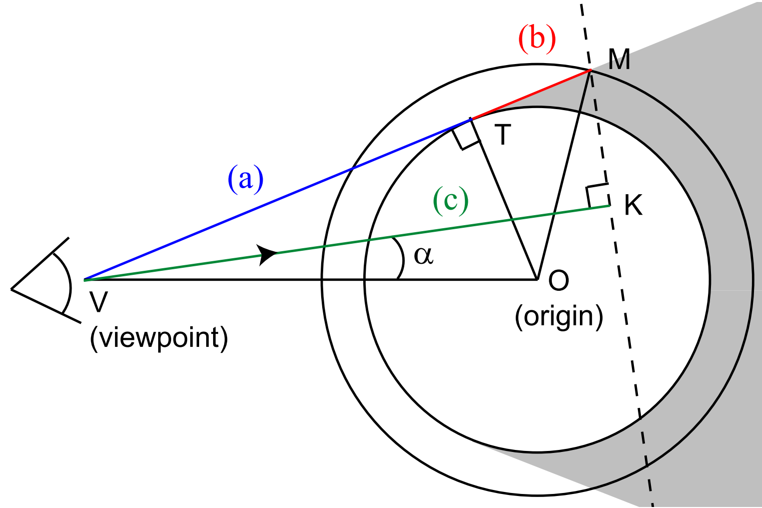


Figure 4.14: Horizon culling: we compute the distance $\|VK\|$ to the far plane. The dotted line KM is the 2D projection of this plane, anything behind it is not rendered. The grayed area is visually culled by the planet itself. **a)** We first compute $\|VT\|$, based on the minimum planet radius $\|OT\|$ 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 planet radius and the maximum one $\|OM\|$. **c)** Finally, we project $\overrightarrow{VM}$ onto the viewpoint direction vector to get $\|VK\|$.

and the SRTM elevation database for the Earth, the difference is of less than 0.5% and $\|TM\| = 626$km.

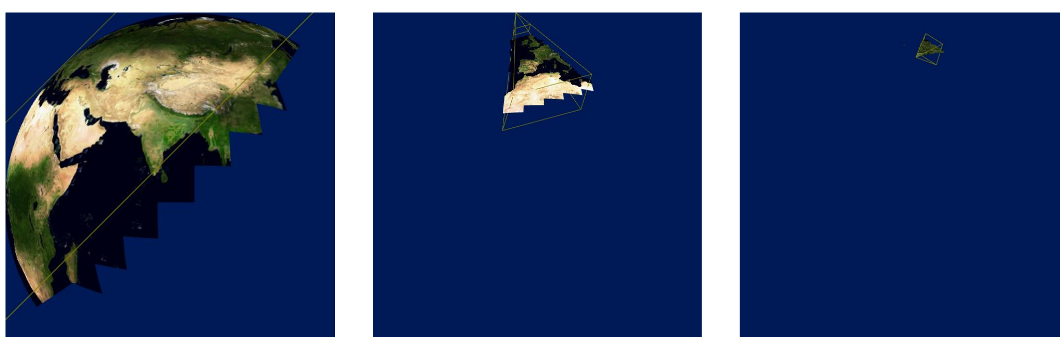
$$\begin{aligned} \|VK\| &= \|VM\| \times \cos \widehat{MVK} \\ &= \|VM\| \times \cos(\widehat{TVO} - \alpha) \\ &= (\|VT\| + \|TM\|) \times (\frac{\|VT\|}{\|VO\|} \cos \alpha + \frac{\|OT\|}{\|VO\|} \sin \alpha) \end{aligned} \quad (4.3)$$

In practice, $\|VK\|$ may become smaller than the distance between the viewpoint and the surface of the planet $\|VO\| - \|OT\|$, or even become negative. This is true especially when the user looks away from the planet. In such cases, we avoid culling the whole planet by bounding the far distance to $\|VO\| - \|OT\|$.

We compute the *near* distance in a much simpler manner: we use the minimum distance to the planet $\|VO\| - \|OM\|$. We bound this distance to an arbitrary value when it becomes negative, meaning the viewpoint is under the maximum planet radius. We finally adjust the *near* distance in respect with the field of view to avoid culling parts of the planet on the corners of the screen.

One can note that adapting the clipping planes leads to abusive culling of objects not on the surface of the planet, for instance satellites or other planets. To circumvent this problem, we could use a specific view frustum for each distant object and render

(a) 3D renderings of the Earth from different distances



(b) Corresponding rendered areas at planetary scale, with view frustum

Figure 4.15: Adaptive clipping planes for horizon culling and improved rendering precision. Parts of the planet in front of the viewpoint but behind the horizon are automatically culled, thus allowing visible areas to be rendered with higher definition LODs and higher depth precision. There is no other geometry comparison than between block bounding boxes and view frustum planes.

them in back-to-front order, clearing the hardware depth buffer between them to avoid rendering inconsistencies.

4.5.5 Discussion on other precision problems

When rendering a planet on graphics hardware using 32-bit floating-point precision, 3D coordinates of the vertices on the surface get very limited precision. For instance, the Earth has a radius of about 6.5 million meters, so we obtain 0.5 to 1m precision on its surface with a global coordinate system. We may use higher precision for intermediate computations, like 3D reconstruction of vertices from elevation values which requires using normalized direction vectors, but rendering itself is always limited by graphics hardware. This becomes problematic when rendering very detailed databases.

Since we subdivide the terrain into blocks of constant resolution but covering smaller and smaller terrain areas, the precision we want for a block is inversely proportional to the covered terrain area. Consequently, we propose to use a local coordinate system for each block, centered on it and with precision depending on the size of the block. Graphics hardware offer the possibility to define a different coordinate system for each

rendered object by defining a matrix that converts vertices into a viewpoint-centered coordinate system. Consequently, the local precision is preserved when the viewpoint is located near a block no matter how far it is from the center of the planet.

However, this solution requires ensuring that conversions between coordinate systems are reversible to avoid cumulating rounding errors during block split and merge operations. The easiest way to achieve this is to maintain original elevation values in blocks instead of 3D vertex coordinates, then convert to floating-point precision in a local coordinate system only when uploading vertices to the graphics hardware or using a GPU shader. Another solution is to maintain fixed-point precision coordinates. Unlike on-the-fly elevation to 3D conversion, this allows for asynchronous computation of vertex coordinates during refine operations and direct sample copies during split and merge operations, while still obtaining high precision values. In practice, using 32-bit fixed-point values quantized into the bounds of the planet radius, we get a precision of 3mm on every point of the global coordinate system and are free of rounding errors when translating the center of the coordinate system.

4.6 Results

We have tested the proposed methods using blocks with two LODs to get uniform block update times: one refine operation occurs per block and all those operations add the same number of samples. Once we have built the files on the server (see Chapter 5), we connected a client equipped with a 3.2GHz Core 2 Duo processor and a GeForce 9800GTX+ graphics card for a real-time 3D interactive walk-through using a screen resolution of 1680×1050 pixels, then we tested rendering performance. Several screen captures are shown in Figure 4.16. To be able to directly compare speed in different conditions, we asked the adaptive rendering solution for a fixed number of 2 million triangles per frame instead of a target frame rate. We obtained the results presented in Table 4.2.

We first verify that using different block resolutions gives different behaviors as predicted in Section 3.2.3. Using texture blocks costs more CPU time compared to embedded colors because such blocks need to be managed by the solution. The impact is less visible with the Puget Sound database because texture blocks are larger. Texture mapping also has an impact on hardware rendering speed of about 15% (up to 35% when compared to rendering without any color map and using small elevation blocks). On the contrary, fixing cracks actually improved performance in cases where large elevation blocks are used. We suppose that this is due to hardware performance limits with large triangle strips, although we have seen that using larger strips usually yields higher frame rates. Anyway, we can see that crack fixing has a little impact on rendering speed.

We have also tested the speed of the block refine operations that occur when a new LOD is received. With texture filtering, creating a LOD with $blockWidth = 168$ takes 0.55ms, compared to 0.45ms without filtering. This represents almost a 25% increase, i.e. the cost of copying and updating samples from the previous level. Using our planet

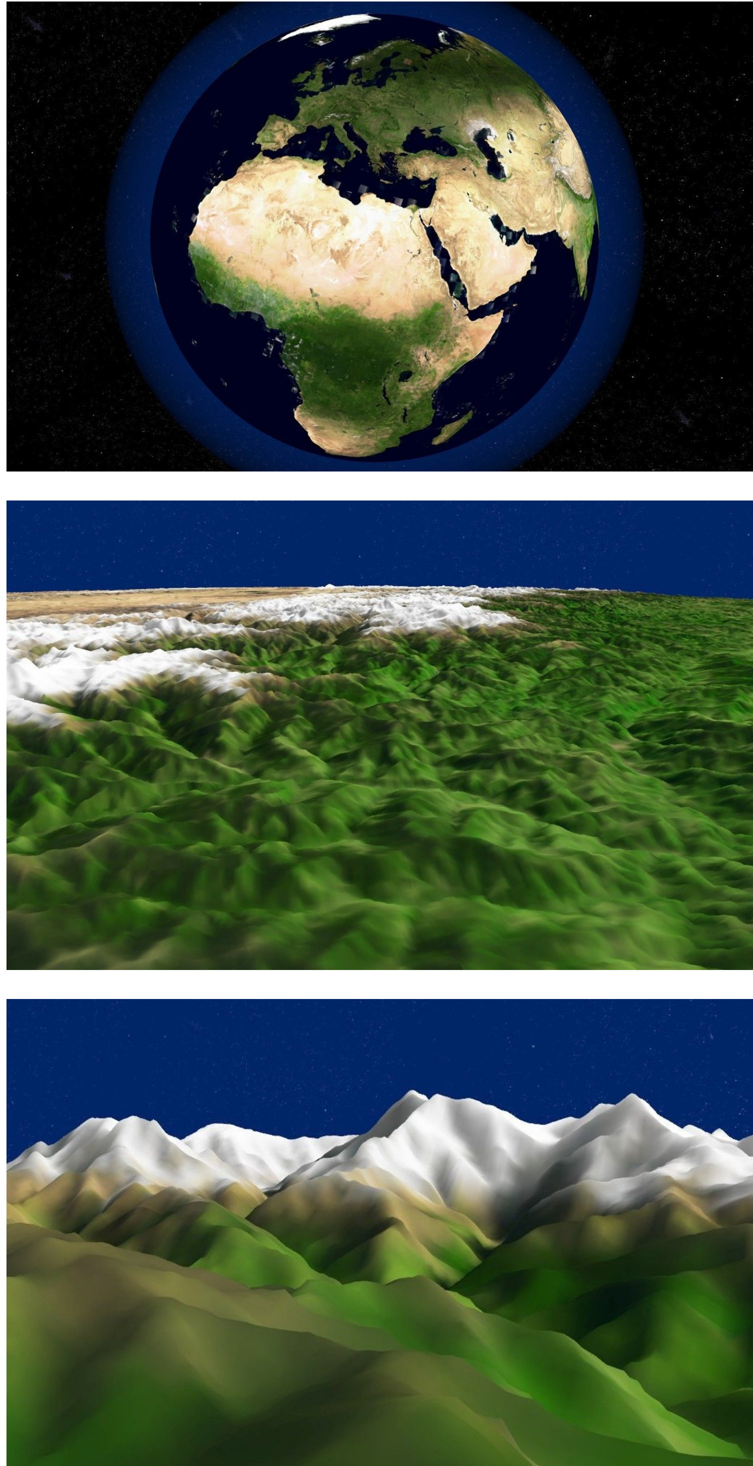


Figure 4.16: Screen captures taken during real-time 3D terrain walk-through using our application with the SRTM and TrueMarble Earth databases.

Database	*blockWidth*	**FPS**	**CPU per frame**		**With cracks**	
			Time	Proportion	FPS	Impact
Puget Sound	64	133	0.9ms	12%	141	+6%
+ texture	64, 256	113	1.05ms	12%	121	+7%
BMNG	84	164	0.45ms	7%	161	-2%
+ texture	84, 168	141	0.8ms	11%	135	-4%
SRTM	52	101	1.7ms	17%	109	+8%
+ TrueMarble	52, 168	65	2.35ms	15%	70	+7%
SRTM	104	185	0.2ms	4%	172	-7%
+ TrueMarble	104, 168	162	0.35ms	6%	160	-1%

Table 4.2: Results of real-time 3D rendering on a high-performance client system at about 2 million triangles per frame, with and without texturing and cracks fixing. CPU per frame is the time used by our software data selection methods at each frame; the rest is used by 3D rendering itself. The last column gives the rendering speed obtained when not fixing cracks, using the same partial client database state as with cracks fixing. When rendering with no texture, we embedded color values in samples along with elevation values for the Puget Sound and BMNG databases, but no color is available for the SRTM database. All databases use *subDepth* = 1 (two LODs per block).

projection adjustment, the reconstruction of the new 3D vertices from elevation samples of the second LOD of a block with $blockWidth = 52$ takes 0.44ms, compared to 0.25ms with the standard 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 the network latency.

Next, we have tested the solution on a low-performance client system. We used a netbook with an Atom 1.66GHz processor and a GMA450 integrated graphics chipset, using a screen resolution of 1024×600. Because of graphics hardware limitations, we were forced to disable caching of geometry data in graphics hardware and to use texture blocks with power of two width. When asking the adaptive rendering solution for 100,000 triangles per frame, we obtained 26 FPS on the Puget Sound database, and 35 FPS on the BMNG and SRTM + TrueMarble Earth databases. When using large blocks with $blockWidth = 104$, we obtained 44 FPS, but using 90,000 triangles per frame. When asking for more quality, the number of triangles directly jumped over 110,000.

The CPU time for each frame was about 0.3ms in all cases, less than 1% of the frame time. This is because the bottleneck for 3D rendering on this netbook is the graphics hardware, not the CPU. Using textures costs an average of 15% rendering speed, and disabling crack fixing improved the rendering speed by about 3%. Finally, we have noted that refining the second LOD of a block with $blockWidth = 52$ costs 2.9ms using the adjusted projection.

4.7 Conclusion

In this chapter, we proposed methods for important features of high-quality interactive 3D terrain rendering, built on top of the generic solution presented in Chapter 3.

We first extended the generic solution to the context of 3D hardware rendering with triangles. We then fixed geometry cracks between adjacent blocks with different levels of detail, adapting a common solution to the context of data streaming. We also extended the solution to support rendering of textured terrain and added sample filtering for improved quality.

Next, we mapped planetary terrains onto a cube, using an adjusted version of the gnomonic projection to get a much more uniform sampling of the planetary surface than with classical representations. This prevents storing, transferring and rendering redundant data and avoids rendering inconsistencies around the poles. Finally, we used adaptive clipping planes to improve the rendering precision of planets on commodity graphics hardware and automatically cull most of the planet surface located behind the horizon.

So far, we have proposed a complete runtime solution for streaming and 3D rendering of large terrains. However, data streamed over the network are initially stored in a file on the hard disk of the server with a particular data organization designed to minimize the server activity, which needs to be pre-computed. In addition, we have developed an original sampling for the gnomonic projection to store planetary maps, but this requires an additional preprocessing step since most available planetary databases use another projection. In the next chapter, we describe both the re-projection and the server file creation preprocessing steps, which are designed to support input maps of arbitrary size.

Chapter 5

Preprocessing: building server files from huge input maps

In this chapter, we describe the two preprocessing steps we use to obtain a server database useable in our solution from any input database. These steps are presented in Figure 5.1.

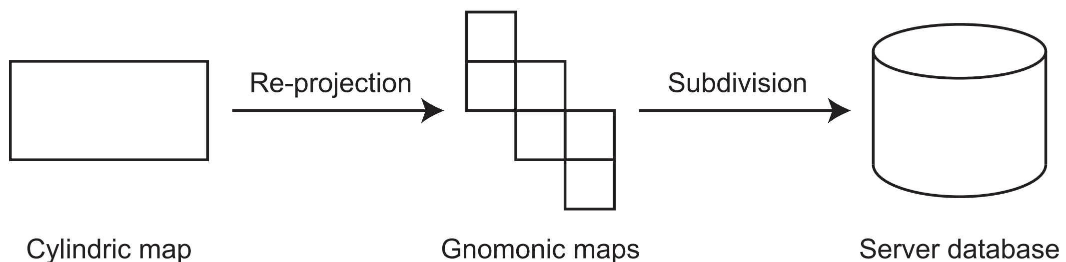


Figure 5.1: Preprocessing steps. The re-projection step, presented in Section 5.2 and used only for planetary databases, converts a cylindrical planetary map to the six maps we use in our solution. The subdivision step, presented in Section 5.3, generates a database file that can be used by our runtime server application from either a non-planetary map or six maps obtained from the re-projection step.

In Section 4.5, we proposed to use a customized gnomonic projection on the faces of a bounding cube to represent the surface of a planet using six square maps of samples. However, most planetary databases available use the equirectangular cylindrical projection to represent the planet using one global rectangular map. As a consequence, a preprocessing step is required before building a planetary database for our solution, to re-project the input map using our projection. This step is presented in Section 5.2.

In Chapter 3, we presented a generic solution to progressively and adaptively load and render huge maps of samples in real-time. To maximize performance, this solution relies on a specifically crafted data structure used in all the places where the data are stored, including the database on the server hard disk as explained in Section 3.3. We

build this file in an off-line preprocessing step, which we describe in Section 5.3.

The algorithms used in both preprocessing steps are designed to support maps of arbitrary size. They consist in subdividing the maps into smaller areas that may be loaded entirely in memory. To simplify the input and output of such huge sample maps, we developed a dedicated tool which we present in Section 5.1.

5.1 A useful tool: dually tiled map format

In order to support maps of arbitrary size in the two preprocessing steps presented in this chapter, we need to avoid loading the entire maps in memory at once, either for input or output: both algorithms work on subsets of these maps. In addition, we noted that most of the available maps are so huge that the authors use smaller rectangular tiles covering sub-areas of the map, all tiles being stored as independent image files. To address these two points, we developed a special map format that implicitly reconstructs the full map from the tiles and allows loading only a rectangular and arbitrary window of the full map. This tool is summarized in Figure 5.2.

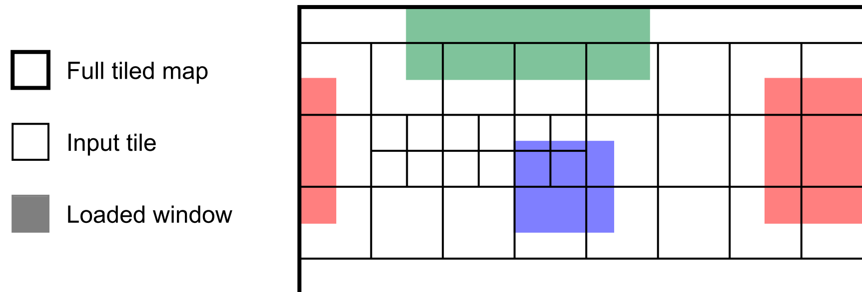


Figure 5.2: A dually tiled map. The map is a set of tiles of any size. We can load arbitrary windows of the map, even across the edges.

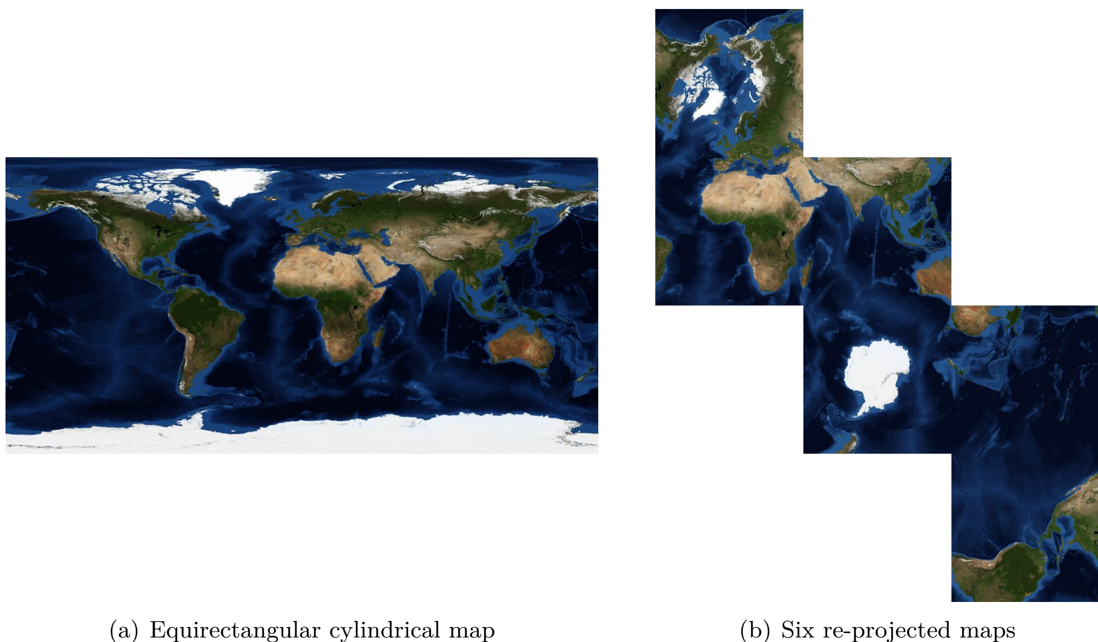
The first feature of our dually tiled map format is that we allow the map to be made of any number of rectangular tiles. These tiles are defined by their position and size in the map. When accessing a sample of the map, we find the corresponding tile and adjust image coordinates to obtain the corresponding value.

The second feature is that we allow to load only one given rectangular window of the map in memory, defined by its position and size. When a window is requested, we find the corresponding tiles and adjust the window for each tile, then we read the corresponding samples from the image files. Those samples may then be accessed as explained before. The input image files are raw maps of samples with no compression, so loading a window is straight forward.

Finally, we added a simple feature to simplify planet re-projection: the ability to load a window that is located across an edge of the map. For instance, with a cylindrical projection, the opposite east-west edges are adjacent on the surface of the planet. In practice, loading such a window can be implemented by loading two separate windows.

5.2 Re-projection of a planetary map

In this section, we explain how we compute the six square maps used in our planet rendering solution by re-projecting samples from a global map that uses the equirectangular cylindrical projection (see Figure 5.3). We first give the formulas used for the re-projection of one point in Section 5.2.1, then propose a re-projection algorithm that can handle maps of arbitrary size in Section 5.2.2.



(a) Equirectangular cylindrical map (b) Six re-projected maps

Figure 5.3: Example of re-projection using the adjusted gnomonic projection (Earth GEBCO database). The six output maps are tied to form the patron of the cube bounding the planet.

As explained in Sections 4.5.1 and 4.5.2, the sampling of the surface of the planet is not the same depending on the projection. However, most of the available databases using the equirectangular cylindrical projection fully take advantage of the definition of the map at the equator but contain more and more redundant data as we get closer to the poles. For the maps computed using our projection, we use the same equatorial definition. In practice, the width of each output map $mapWidth$ is one quarter of the width of the input map $inputWidth$. In this way, the total size of the six output maps is $\frac{3}{8} \times inputWidth^2$, while the input map has a size of $\frac{1}{2} \times inputWidth^2$. We save 33% of mostly redundant data but still get the same overall definition as the input map. In practice, we accept a small difference in definition to be able to obtain maps that can be subdivided into enough blocks for our adaptive streaming and rendering solution.

5.2.1 Sample re-projection

For each sample of an output map, we first compute its re-projected coordinates in the input map, then obtain the value of the corresponding sample from this map. We use Equations 5.1 to convert a point on the surface of a planet from gnomonic coordinates $u, v \in [-1, 1]$ (presented in Section 4.5.1) to cylindrical coordinates $\phi' \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ and $\lambda' \in [-\pi, \pi]$ (classical latitude and longitude). ϕ, λ are the coordinates of the center of projection used for the gnomonic projection in the input map; in our case, it is the center of the corresponding cube face. We used the formula from [Weia] and simplified it using trigonometric identities to improve preprocessing speed. Before applying Equations 5.1, we rotate gnomonic coordinates to match $\vec{u}, \vec{v}$ axes specified in Section 4.5.3, based on the cube face.

$$\begin{aligned} \lambda' &= \arctan\left(\frac{u}{\cos\phi - v\sin\phi}\right) + \lambda \qquad (5.1) \\ \phi' &= \arcsin\left(\frac{\sin\phi + v\cos\phi}{\sqrt{u^2 + v^2 + 1}}\right) \end{aligned}$$

To apply our sampling adjustment, we use $u = \tan u'$ and $v = \tan v'$ where $u', v' \in [\frac{-\pi}{4}, \frac{\pi}{4}]$ are the coordinates of the point using our angular sampling of the planet surface.

5.2.2 Entire cube face map re-projection

We re-project the six maps corresponding to the six faces of the cube independently, using the same algorithm multiple times. This step, presented in Algorithm 5.1, consists in recursively subdividing the output map. The map area is regularly subdivided into four sub-areas until we can load the corresponding window of the input map in memory. We then re-project and get the values of all individual samples of each area. Both the input and output maps use the dually tiled map format presented in Section 5.1.

This algorithm ensures that we do not load an input map window using more memory than a given value *maxmem*. However, the algorithm requires the ability to compute the area (or window) in the input map corresponding to a given area of the output map.

To simplify the computations and avoid loading too large input map windows around the poles of the planet (i.e. the top and bottom edges of the input map), we force the top and bottom faces of the cube to use the poles as their center of projection. In practice, poles themselves are either not stored in input maps or represented by the entire top and bottom rows of the map, respectively. When re-projecting an elevation map with odd width, the center sample of the map gets re-projected to the center of projection of the map. We may thus directly use $\phi' = \phi$ and $\lambda' = \lambda$ for the poles instead of applying Equations 5.1. When subdividing such a map into four smaller areas, one of the sub-areas of the map obtains one additional row and one additional column because of the odd width: this area contains the pole. However, the rest of the area may re-project far from the arbitrary λ' longitude of the pole. This would require loading a large window of the input map just for this point. In practice, we ensure that the sub-area containing the pole is the one whose bounding input map

Algorithm 5.1: AreaReproj – Planetary map re-projection. One iteration of this algorithm obtains the samples of the given area of the output map from the input map: initially, call with on area covering the whole output map. *maxmem* is the maximum memory size that can be loaded at once. The output map area is recursively subdivided until the corresponding input map window is small enough to be loaded entirely.

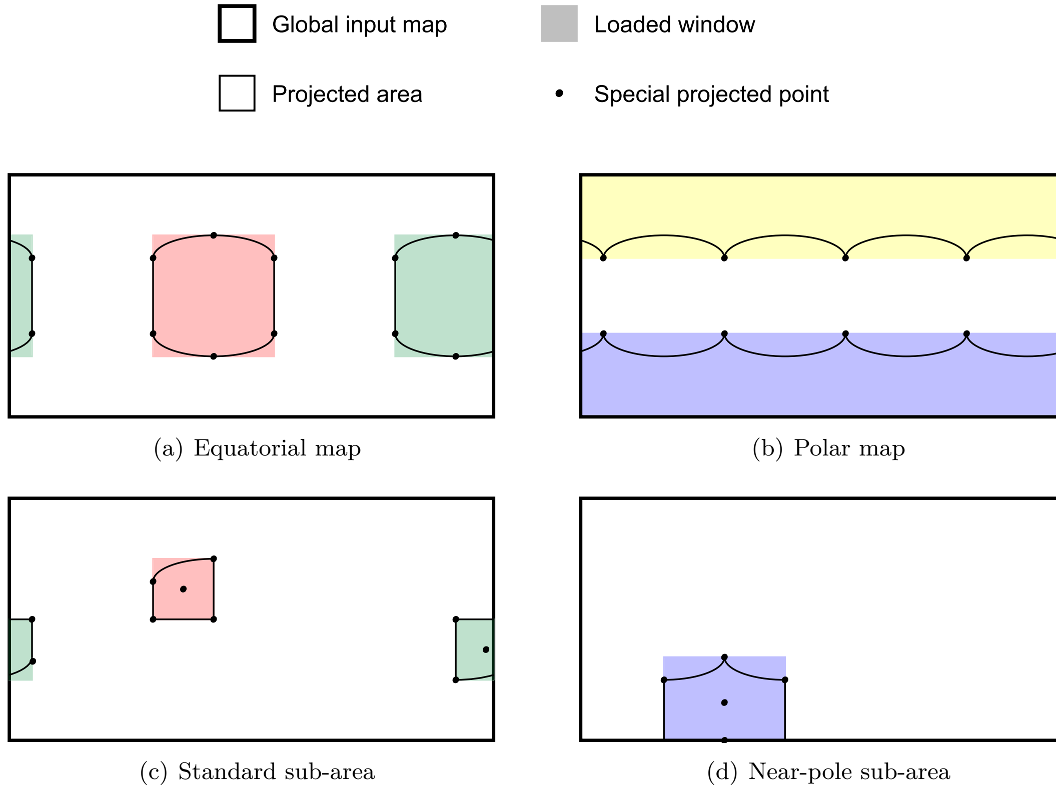
```
Data: x, y, w, h: coordinates and size of the area in the output map; λ, φ:
      coordinates of the center of projection of the output map; input, output:
      input and output maps.
x', y', w', h' ← area x, y, w, h projected using λ, φ      /* see Figure 5.4 */
if memory size of area x', y', w', h' from input > maxmem then
    // further subdivide the area
    AreaReproj(x, y, w/2, h/2)
    AreaReproj(x + w/2, y, w/2, h/2)
    AreaReproj(x, y + h/2, w/2, h/2)
    AreaReproj(x + w/2, y + h/2, w/2, h/2)
else
    // load and re-project the area
    load area x', y', w', h' from input
    create tile x, y, w, h in output
    foreach sample u, v in area x, y, w, h do
        λ', φ' ← sample u, v projected using λ, φ    /* see Section 5.2.1 */
        copy the value of sample λ', φ' from input to output tile
    unload data from input
```

window is centered at this latitude. The pole is located at the corner of this sub-area with odd width: when the area gets recursively subdivided, its own sub-area containing this corner is the one that gets an odd width.

In these conditions, we can re-project specifically selected points from the output map area and get the bounding input map window. Possible cases are presented in Figure 5.4.

Since top and bottom faces are centered at the poles of the planet, other faces are centered at points located on the equator. When re-projected, the entire map of an equatorial face covers areas as presented in Figure 5.4(a). We may thus re-project only the corners of the map and the center of the top and bottom edges. The maps of the polar faces have all their corners projected to the same latitude. However, we know that such maps include the corresponding pole, so we expand the bounding window to the edge of the input map as shown in Figure 5.4(b).

In most cases, when using huge input maps the window corresponding to an output map cannot be loaded entirely in memory. The algorithm then regularly subdivides the area in four equal parts. In this case, re-projecting the corners of the area is enough to



(a) Equatorial map
(b) Polar map
(c) Standard sub-area
(d) Near-pole sub-area

Figure 5.4: Specific points of output map areas to re-project in order to obtain bounding input map window. There are some different cases based on whether the output map is polar or equatorial, and whether the re-projected area is an entire map or a sub-area.

obtain the bounding window, as can be seen in Figures 5.4(c) and 5.4(d). However, we also need to re-project the center of the area in order to detect the windows that are located across the edges of the input map.

5.3 Generation of the server database

The final preprocessing step consists in creating the server database file from one or more square 2D maps of samples. This step takes maps using our dually tiled map format as input (typically, the output from the re-projection preprocessing step) and may be used on multiple maps to create a single file, for instance a planetary terrain with six maps. This step only performs regular subdivision and de-interleaving of the samples from the input map, with no change in sampling definition.

Using the first server database approach presented in Section 3.3.1, building the file can be done linearly but consecutive samples are located on distant parts of the original input map. This requires either loading the entire input map into memory, which is not

possible for huge maps, or performing a file seek for each accessed input sample, which is very slow. To circumvent this problem, we chose to use a smarter bottom-up tree building algorithm, presented in this section. However, using this algorithm prevented us from linearly writing the file, therefore slowing down the process. We could have used a different file organization scheme allowing linear writing, and subsequently updated the LOD localization formula, but we preferred to completely switch to the second approach presented in Section 3.3.2 which does not require using any particular LOD ordering. When building a LOD, this method requires knowing the file position and size of the next LOD, either in the same block or for all the children blocks. This is suited for bottom-up construction of the tree. We present this final preprocessing step in Algorithm 5.2.

We subdivide the map into blocks using a depth-first walk-through of the virtual tree defined by parameters *nbSubs* and *subDepth* presented in Section 3.2.3. When we reach a leaf block of the tree, we obtain the corresponding samples from the input map, compute the LODs of the block, then return the samples of its first LOD to the parent. Non-leaf blocks recursively apply the algorithm on their children, then when all the children have been entirely processed, it uses the returned samples to compute its own LODs and return the first one to its own parent.

We may load the input map area corresponding to a block when it is small enough to be loaded entirely in memory before actually subdividing it, instead of independently loading data for each leaf of the tree. The *loaded* flag tells whether the corresponding area from *input* is currently loaded. In addition, the $justloaded$ flag allows remembering if the area was loaded in the current iteration of the algorithm, in order to release the input data when all the sub-tree has finished processing. Unlike with re-projection, computing the coordinates of the area x, y, w, h of the input map corresponding to a block sub, u, v is trivial since we perform only regular subdivisions of the map: $w = h = blockWidth \times 2^{subDepth \times (nbSubs - sub)}$, and $x = u \times w$, $y = v \times h$.

5.4 Results

We tested the re-projection preprocessing step on several planetary maps that use the equirectangular cylindrical projection, then we built the files for our runtime application using the second preprocessing step on these re-projected maps as well as the Puget Sound database. We performed the tests on a computer equipped with a 3GHz Xeon 5160 processor with 3GB main memory, but specifying 1GB maximum memory usage because of operating system limitations. Input and output files are both on a RAID mirror of two hard drives running at 10,000 rotations per minute and with a 3Gbps interface. The results for both preprocessing steps are shown in Table 5.1.

The SRTM database is actually 58.4GB large but does not provide data for polar areas ($\phi > 60$ï"$\frac{1}{2}$ and $\phi < -60$ï"$\frac{1}{2}$) nor for tiles covering only oceans. We filled those areas using the BMNG data, implicitly up-sampled to the same definition as SRTM using our dually tiled map format. This leads to a virtual but global 174GB input map.

As we can see, both preprocessing steps successfully handle very large input maps.

Algorithm 5.2: BlockSubdiv – Bottom-up subdivision of a 2D sample map to build a server database file. One iteration of this algorithm computes the LODs of the given block and all its sub-tree: initially, call on the root block ($sub = 0$, $u = 0$, $v = 0$). *maxmem* is the maximum memory size that can be loaded at once. Subdivision parameters *nbSubs* and *subDepth* are presented in Section 3.2.3.

Data: *sub*: level of the block in the tree; u, v: coordinates of the block in the map at this tree level; *input*: input map; *output*: subdivided server file; *loaded*: boolean telling whether the area of the input map corresponding to this block is loaded.

Result: *map*: samples of the first LOD of the block (shared with its parent); $pos, size$: position and size of the second LOD of the block in the file.

$map \leftarrow \emptyset$ `/* grid of samples of the block */`
$pos, size \leftarrow \emptyset$ `/* list of positions and sizes of next LODs */`
$justloaded \leftarrow false$ `/* whether we loaded data in this iteration */`
if $sub = nbSubs$ **then** `/* the block is a leaf */`
    if $loaded = false$ **then**
        `// we need to load the area`
        load area sub, u, v from *input*
        $loaded, justloaded \leftarrow true$
    $map \leftarrow$ samples of area sub, u, v from *input*
else
    if $loaded = false$ **and** *memory size of area* sub, u, v *from input* $\leq maxmem$ **then**
        `// we can load the area`
        load area sub, u, v from *input*
        $loaded, justloaded \leftarrow true$
    foreach *child block* $sub + 1, u', v'$ **do** `/*` $4^{subDepth}$ `children */`
        `// recursive call`
        BlockSubdiv($sub + 1, u', v', input, output, loaded$)
        merge results into $map, pos, size$
`// at this point,` *map* `contains all the samples of the block`
`// and` $pos, size$ `are those of the children (or` $\emptyset$ `for leaf blocks)`
for $lod \leftarrow$ *last LOD* **to** *second LOD* **do** `/* de-interleave LODs to file */`
    write $pos, size$ to *output* `/* those of the next LOD (or children) */`
    samples $\leftarrow$ samples of *map* in LOD *lod* but not in previous LOD
    remove *samples* from *map*
    process *samples* `/* filtering and/or compression */`
    copy *samples* to *output*
    $pos, size \leftarrow$ just written *output* file chunk `/* those of` *lod* `*/`
`// at this point,` *map* `contains the samples of the first LOD`
`// and` $pos, size$ `are those of the second (last treated) LOD`
if $justloaded = true$ **then**
    unload data from *input*

Input map			Re-projection		File creation	
Name	Type	Size	Time	Size	Time	Size
Puget Sound	elev.	512M	N/A	N/A	50s	280M
	color	768M			1m25	458M
SRTM	elev.	174G	8h55	126G	4h50	14.9G
TrueMarble	color	41.7G	1h35	31.0G	36m35	6.84G
BMNG	elev.	6.95G	27m20	5.16G	6m45	879M
	color	10.4G	33m40	7.75G	9m25	1.89G
GEBCO	elev.	445M	1m25	330M	30s	198M
	color	667M	1m35	496M	45s	206M
Mars MOLA	elev.	1.93G	6m05	1.41G	2m40	744M
Mars MOC	color	3.86G	24m20	2.96G	9m20	1.31G

Table 5.1: Preprocessing results. Re-projection uses about the same equator definition as the input maps.

The re-projected databases are about 25% smaller than the input maps due to our projection on a cube that removes most of the redundant data.

To create the files for our application, we used the subdivision parameters presented in Table 5.2. Results in Table 5.1 were obtained using sample filtering for color databases. For comparison, we obtained a 5.96GB file in 30 minutes for TrueMarble without sample filtering. In all cases, the server files are even smaller than the re-projected maps due to LOD compression. When using no compression, since the file generation step only re-organizes the order of samples, such files have a similar size as that of the corresponding unprocessed maps. The difference due to the LOD header data, and redundant block edge samples for elevation maps, is negligible.

Database	**Type**	$mapWidth$	*nbSubs*	$blockWidth$
Puget Sound	elev.	16384	8	64
	color		6	256
SRTM	elev.	106496	11	52
TrueMarble	color	43008	8	168
BMNG	elev.	21504	8	84
	color		7	168
GEBCO	elev.	5376	6	84
	color		5	168
Mars MOLA	elev.	11264	7	88
Mars MOC	color	23040	7	180

Table 5.2: Subdivision parameters. All databases use *subDepth* $= 1$ (two LODs per block). We chose to use $50 < blockWidth \leq 100$ for elevation maps and $128 < blockWidth \leq 256$ for texture maps. *nbSubs* results from the other parameters. Note that planetary databases (all but Puget Sound) actually use six maps each.

Chapter 6

Conclusion

In this thesis, we presented our work on real-time adaptive streaming and 3D rendering of large terrains. In this chapter, we conclude our work by summarizing our contributions and discussing potential directions for future work.

6.1 Contributions

6.1.1 Generic adaptive data streaming and selection

We first proposed a generic solution for remote adaptive streaming and rendering of arbitrarily large 2D sample maps. Our methods 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. In addition, our methods may be used with any kind of the data.

We based this solution on a generic data structure, which mixes two solutions with good properties and adds new methods to improve the efficiency of streaming and rendering. We used this data structure to manage the data from the server hard disk all the way to the client rendering system, focusing on speed. In particular we avoided loading irrelevant data by ensuring that data are not redundant between successive loadings and by always sending the most important data requests first. We also avoided costly data structure operations on the client as much as possible, in favor of in-place asynchronous data updates and extraction of levels of detail using sample masks.

We used a generic measure of importance to drive adaptive streaming and selection of levels of detail as well as to trigger client database update operations. It depends on a general quality factor that allows adapting to a user-specified rendering speed or quality, by impacting the importance of all areas of the map in the same way. Other factors may be changed when applying the solution to a specific kind of data, so as to reflect their nature and improve adaptivity.

On the server, we used a particular file organization that ensures that only a single file access is required to handle a loading request. We described how to generate a server database using a preprocessing step that supports input maps of any size.

6.1.2 3D rendering of large terrains and planets

We based upon the generic solution to support fast 3D hardware rendering of large terrains. We specialized the measure of importance, and we took advantage of optimized data techniques by extending the concept of sample masks using triangle strips and by caching levels of detail on graphics hardware.

We fixed geometry cracks between adjacent terrain areas with different levels of detail, improving a common solution to adapt it to the context of data streaming. Unlike previous solutions for fixing cracks, we allowed such adjacent areas to use any difference in sampling definition as long as it does not produce unsolvable cracks.

We also supported textured terrains by applying the generic solution to texture maps, then binding such maps with corresponding elevation maps for rendering. We improved rendering quality of texture maps by extending the generic solution to take advantage of hardware mipmapping and adding support for sample filtering, instead of the regular down-sampling implied by our data structure.

In another part we mapped 3D planets onto a cube, using an adjusted version of the gnomonic projection to get a much more uniform sampling of the surface than with classical representations. This prevents storing, transferring and rendering redundant data and avoids rendering inconsistencies around the poles. Compared to the gnomonic projection, our adjustment allows better distribution of the samples on the plane of projection, therefore lesser disparities in the areas covered by samples. Since most of the available data are distributed with cylindrical projection, we developed a re-projection preprocessing step which we designed to support input and output maps of any size.

Finally, we used adaptive clipping planes to improve the rendering precision of planets on graphics hardware and automatically cull most of the planet surface located behind the horizon.

6.2 Future work

6.2.1 Generic solution improvements

At first, our data structure could allow more adaptivity: one may use levels of detail that double instead of quadrupling the number of samples compared to the previous one. While this would require some adjustment of the measure of importance and sample masks, it would also allow for finer data selection with no tree structure update as well as more progressive loading. In addition, our solution to fix geometry cracks in 3D rendering applications, presented in Section 4.3, can easily adapt to new levels of detail, especially when no samples are added on the edges of blocks.

Although we already offer the possibility to render a level of detail while other ones with higher and/or lower definition are also available for the same block, the tree update operations still remove levels of detail from the client database. Some may be reconstructed during merge operations, but the others need to be loaded again if they are later selected because blocks are completely removed from the partial client tree

during split operations. An explicit block caching method would improve the reactivity of the generic solution and avoid some redundant data loadings.

6.2.2 3D rendering improvements

As already introduced in Section 4.2.1, 3D reconstruction of elevation samples can be implemented on the GPU. This would save memory, improve the speed of refine operations and allow to change 3D reconstruction parameters interactively.

In Section 4.5.5, we discussed about further improvements of the 3D rendering precision, in particular for planetary terrains. The solution we propose uses local coordinate systems for blocks and avoids floating-point computations except for final rendering.

Future work could also add other features presented in Chapter 2, like the generation of procedural details over best available definition, the fixing of popping artifacts with geomorphing, the support for more advanced compression and/or filtering techniques like wavelets, etc.

6.2.3 Further research

Future works may also base new research topics upon our terrain streaming and rendering solution, in particular by taking advantage of the hierarchical structuring of the database and the regular sampling of the levels of detail. For instance, in a 3D rendering application, this should simplify the detection of collisions, the self-shadowing of the terrain from sunlight, and the automatic positioning of 2D vectorial roads or borders on the 3D surface of the terrain.

We also think that it would be interesting to use a projective grid 3D rendering method to obtain a constant triangle count (see Section 2.1.2.4), in conjunction with our generic solution to handle data streaming and database updates instead of requiring to store the whole elevation map in the memory of the client.

Appendix A

Test databases

In this appendix, we give the origin and characteristics of the databases we used in this thesis. All databases are available for free on the Internet. Given database sizes correspond to their uncompressed raw data form. For planetary databases, given definition is true only at the equator though it may be considered global because redundant data are clearly visible in non-equatorial areas.

Elevation databases use 16 bits per sample, as signed values with one meter per unit, except Puget Sound which has unsigned values with 0.1 meter per unit. Color databases use 24 bits per sample (RGB color with 8 bits per component), except Mars MOC which has grayscale color with 8 bits per sample. Planetary databases use the equirectangular cylindrical (or “Plate Carrée”) map projection, and Earth databases conform to the WGS84 standard datum.

Puget Sound

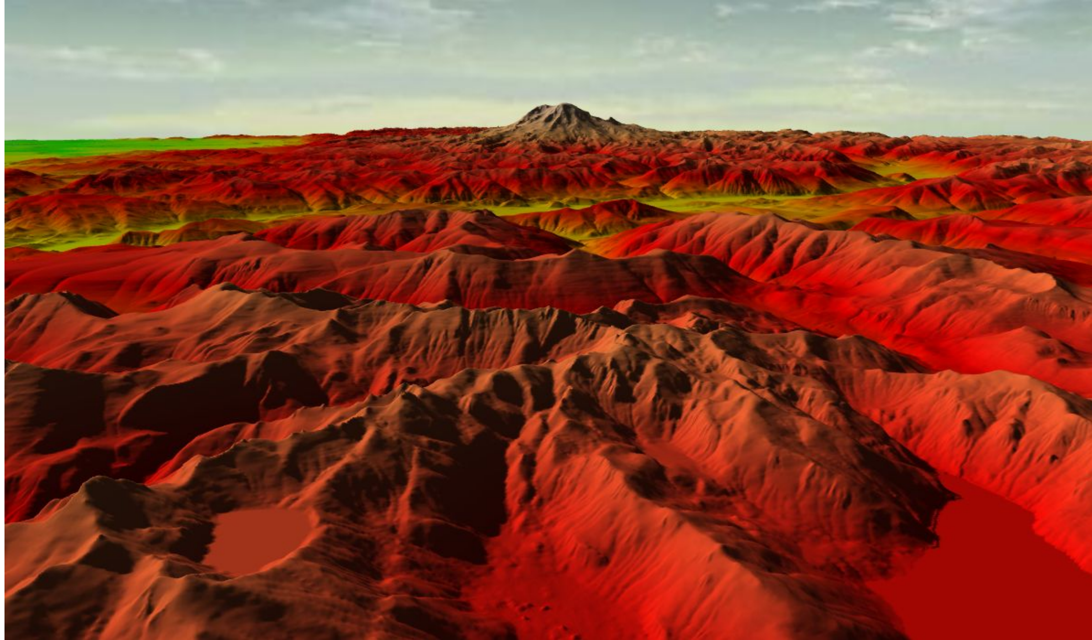


Figure A.1: Rendering using the Puget sound database: view of Mount Rainier, WA, USA.

Provider	USGS (United States Geological Survey)
URL	`http://www.cc.gatech.edu/projects/large_models/ps.html`
Definition	10m
Resolution	16384 × 16384
Size	512MB elevation, 768MB color

Table A.1: Characteristics of the Puget Sound database.

This non-planetary database was first used in [LP01]. It is a digital model of the Puget Sound region in Washington state, USA. In particular, mounts Rainier and St. Helens are included in the database. Colors were computed depending on elevation and an arbitrary sun position, they do not represent reality.

Blue-Marble Next Generation (BMNG)

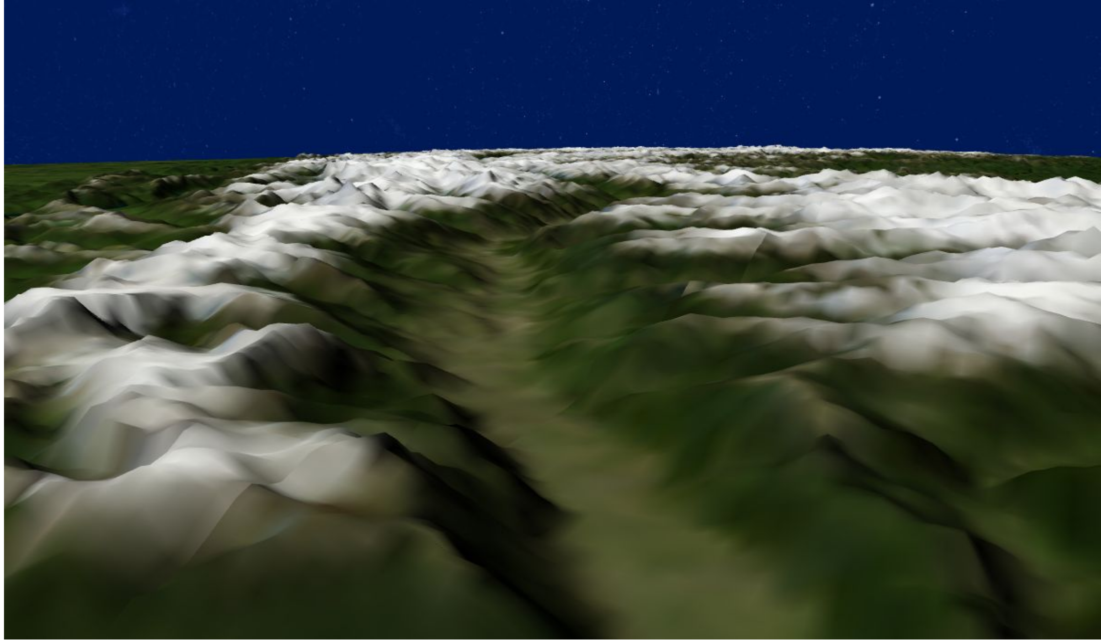


Figure A.2: Rendering using the BMNG database: view of the Swiss Alps.

Provider	NASA
URL	`http://earthobservatory.nasa.gov/Features/BlueMarble/`
Definition	500m
Resolution	86400×43200
Size	6.95GB elevation, 10.4GB color

Table A.2: Characteristics of the BMNG database.

This planetary database represents the Earth. Provided by NASA's Earth Observatory [SVS+05], it proposes monthly color maps for the year 2004; we used month June. The elevation database is actually made of several other databases, including SRTM (presented in the next section) though with much lower definition.

Shuttle Radar Topographic Mission (SRTM)

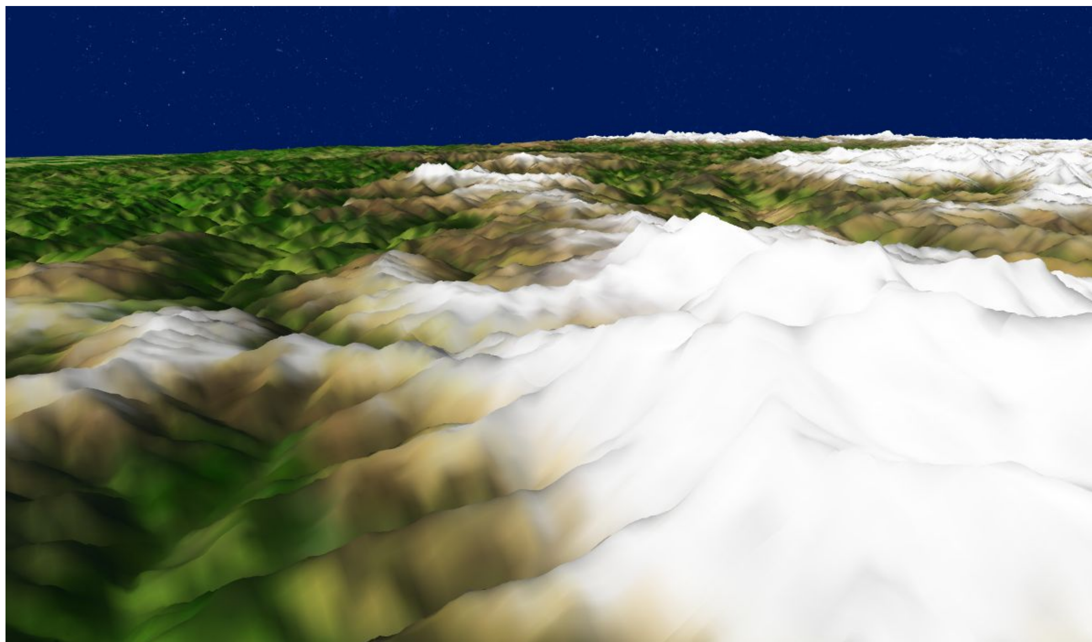


Figure A.3: Rendering using the SRTM and TrueMarble databases: view of the Himalayas, Nepal.

Provider	NASA and CGIAR-CSI
URL	`http://srtm.csi.cgiar.org/`
Definition	90m
Resolution	432000×144000 (432000×216000 with poles)
Size	58.4GB

Table A.3: Characteristics of the SRTM database.

This planetary database represents the Earth. It only contains elevation data: we used the TrueMarble database (presented in the next page) to map colors onto it. SRTM data available from the NASA are incomplete, so we used the SRTM version 4 database from the CGIAR Consortium for Spatial Information. This improved version fills gaps in the original SRTM data to cover the Earth surface up to 60 degrees north and south. We used lower definition polar data from the BMNG database to obtain a global database (see Section 5.4).

SRTM used to be the highest definition available elevation database of the Earth surface. But, recently the ASTER elevation database was released by the NASA and Japan's METI, with 30m definition up to 83 degrees north and south. We were unable to test our solutions with this database because of its late release. It is available at `http://asterweb.jpl.nasa.gov/gdem.asp`.

Unearthed Outdoors TrueMarble^{TM}

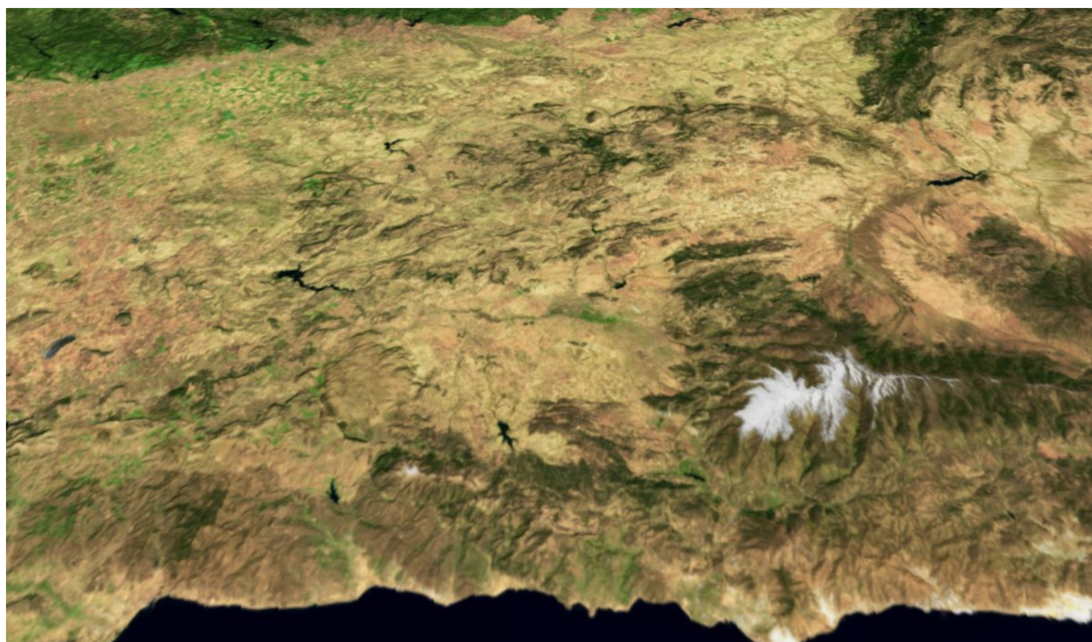


Figure A.4: Rendering using the SRTM and TrueMarble databases: view of Andalusia, Spain.

Provider	NASA and Unearthed Outdoors
URL	`http://www.unearthedoutdoors.net/global_data/true_marble/`
Definition	250m
Resolution	172800 × 86400
Size	41.7GB

Table A.4: Characteristics of the TrueMarble database.

This planetary database represents the Earth. It only contains color data: we used the SRTM database (presented in the previous page) for elevation. The TrueMarble database is based on the raw multi-band Landsat 7 ETM+ global database from the NASA, much improved to obtain real-life visible color and transformed to a classical planet projection. However, high definition versions (up to 15m) of this database are not available for free; we were limited to 250m definition.

General Bathymetric Chart of the Oceans (GEBCO)

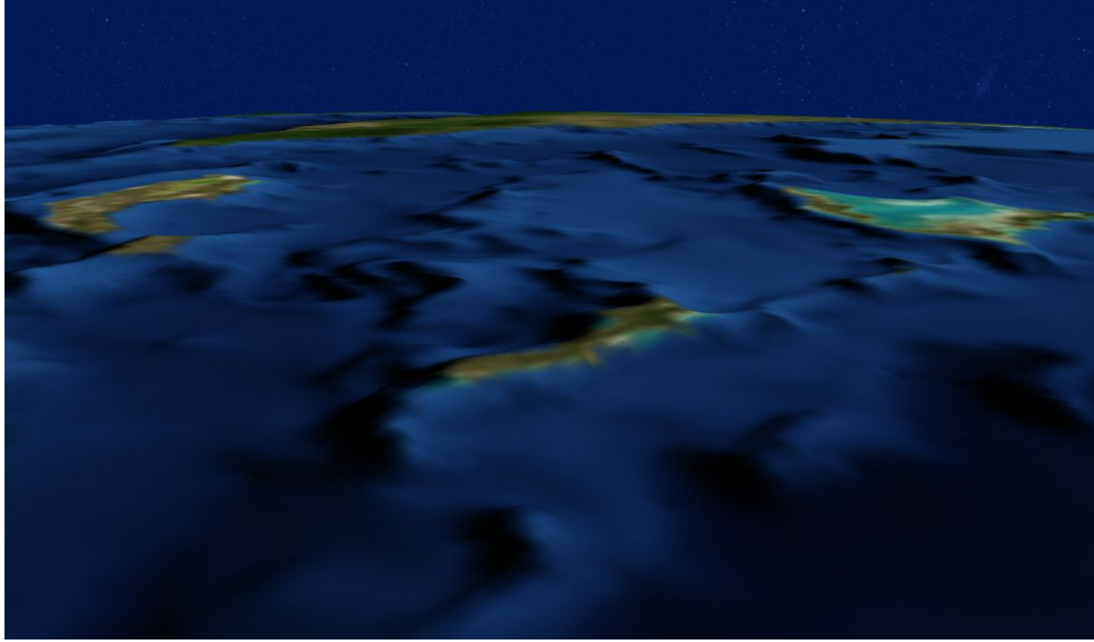


Figure A.5: Rendering using the GEBCO database: view of the Caribbeans.

Provider	BODC (British Oceanic Data Centre) and NASA
URL	`http://www.bodc.ac.uk/data/online_delivery/gebco/`
Definition	2km
Resolution	21600×10800
Size	445MB elevation, 667MB color

Table A.5: Characteristics of the GEBCO database.

This planetary database represents the Earth, including bathymetric data. While the original database contains only elevation data, an improved version is available as part of the BMNG release, along with artificially computed colors for the oceans based on the bathymetry. A higher definition version of this database is in the works, and partially available. A concurrent database with same definition, sharper but with more noise, is ETOPO1 from the NOAA NGDC. It is available at `http://www.ngdc.noaa.gov/mgg/global/global.html`.

Mars Orbiter Laser Altimeter (MOLA) and Mars Orbiter Camera (MOC)

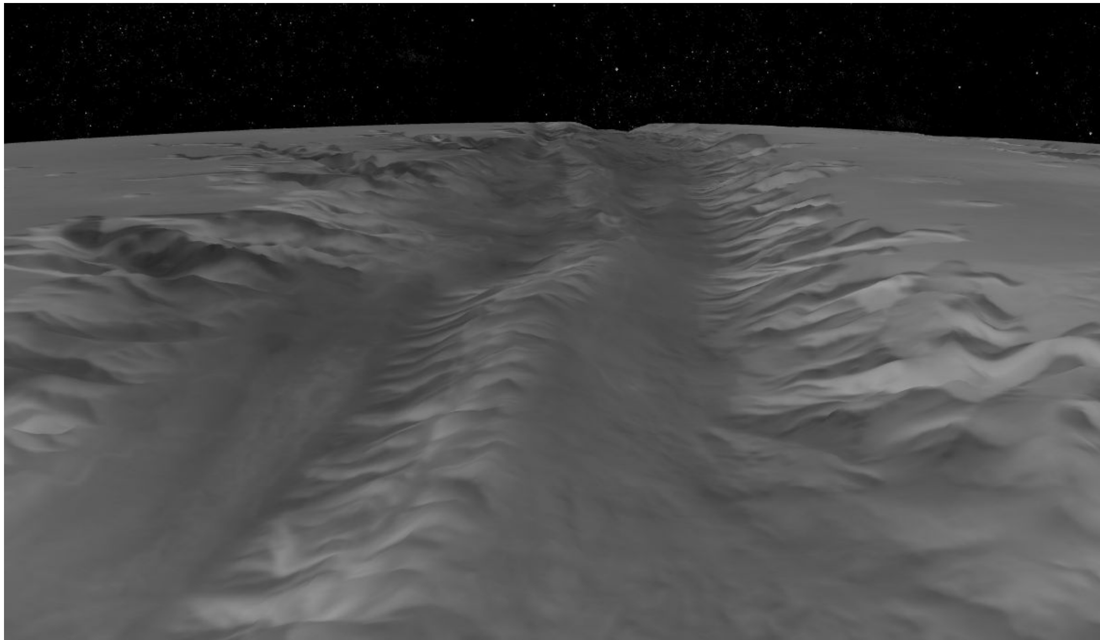


Figure A.6: Rendering using the MOLA and MOC databases: view of Valles Marineris.

Provider	NASA	
URL	`http://pds-geosciences.wustl.edu/missions/mgs/megdr.html` `http://www.msss.com/mgcwg/mgm/`	
Definition	MOLA	500m
	MOC	250m
Resolution	MOLA	46080 × 23040
	MOC	92160 × 46080
Size	MOLA	1.93GB
	MOC	3.86GB

Table A.6: Characteristics of the MOLA and MOC databases.

These two planetary databases represent the planet Mars. Both are from the Mars Global Surveyor (MGS) program. MOLA is for the elevation, and MOC for the photometry although it has no color. Colored photometric maps of Mars exist, but with much lower definition.

Bibliography

[Aas02] Rune Aasgaard. Projecting a regular grid onto a sphere or ellipsoid. In *Advances in Spatial Data Handling*, pages 339–350. Springer-Verlag, 2002.

[AH05] Arul Asirvatham and Hugues Hoppe. Terrain rensering using GPU-based geometry clipmaps. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[Blo00] Jonathan Blow. Terrain rendering at high levels of detail, 2000.

[Bro05] Anders Brodersen. Real-time visualization of large textured terrains. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 439–442, New York, NY, USA, 2005. ACM Press.

[CGG+03a] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum*, 22(3):505–514, September 2003. Proc. Eurographics 2003.

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

[CH06] Malte Clasen and Hans-Christian Hege. Terrain rendering using spherical clipmaps. In *EuroVis 2006 - Eurographics / IEEE VGTC Symposium on Visualization*, pages 91–98, 2006.

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

[DNB06] Soumyajit Deb, P. J. Narayanan, and Shiben Bhattacharjee. Streaming terrain rendering. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 181, New York, NY, USA, 2006. ACM.

[DS04] Carsten Dachsbacher and Marc Stamminger. Rendering procedural terrain by geometry image warping. In *Proceedings of the 15th Eurographics Workshop on Rendering Techniques, Norköping, Sweden, June 21-23, 2004*, pages 103–110. Eurographics Association, 2004.

[DWS+97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[EKT01] William S. Evans, David G. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica*, 30(2):264–286, 2001.

[GMC+06] Enrico Gobbetti, Fabio Marton, Paolo Cignoni, Marco Di Benedetto, and Fabio Ganovelli. C-BDAM – compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum*, 25(3), September 2006. Proc. Eurographics 2006.

[HDJ05] Lok M. Hwa, Mark A. Duchaineau, and Kenneth I. Joy. 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.

[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, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[LC03] Bent Salgaard Larsen and Niels Jørgen Christensen. Real-time terrain rendering using smooth hardware optimized level of detail. In *WSCG*, 2003.

[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. IEEE Computer Society, 2002.

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

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

[LKR+96] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118, New York, NY, USA, 1996. ACM Press.

[LMG09] Raphaël Lerbour, Jean-Eudes Marvie, and Pascal Gautron. Adaptive streaming and rendering of large terrains: a generic solution. In *WSCG*, 2009.

[LN05] Thomas Lauritsen and Steen Lund Nielsen. Rendering very large, very detailed terrains, 2005.

[LP01] Peter Lindstrom and Valerio Pascucci. Visualization of large terrains made easy. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 363–371, Washington, DC, USA, 2001. IEEE Computer Society.

[LP02] Peter Lindstrom and Valerio Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002.

[LRS09] Falko Löffler, Stefan Rybacki, and Heidrun Schumann. Error-bounded gpu-based terrain visualisation. In *WSCG*, 2009.

[LSGES06] Yotam Livny, Neta Sokolovsky, Tal Grinshpoun, and Jihad El-Sana. Persistent grid mapping: A GPU-based framework for interactive terrain rendering. In *Pacific Graphics*, 2006.

[MB03] Jean Eudes Marvie and Kadi Bouatouch. Remote rendering of massively textured 3D scenes through progressive texture maps. In *The 3rd IASTED conference on Visualisation, Imaging and Image Processing*, volume 2, pages 756–761. ACTA Press, 2003.

[Paj98] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 19–26, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[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 *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 299–306, New York, NY, USA, 2005. ACM Press.

[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.

[RLIB99] Martin Reddy, Yvan Leclerc, Lee Iverson, and Nat Bletter. Terravision II: Visualizing massive terrain databases in VRML. *IEEE Comput. Graph. Appl.*, 19(2):30–38, 1999.

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

[TMJ98] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158. ACM Press, 1998.

[Weia] Eric W. Weisstein. Gnomonic projection. From MathWorld – A Wolfram Web Resource. `http://mathworld.wolfram.com/GnomonicProjection.html`.

[Weib] Eric W. Weisstein. Map projections. From MathWorld – A Wolfram Web Resource. `http://mathworld.wolfram.com/topics/MapProjections.html`.

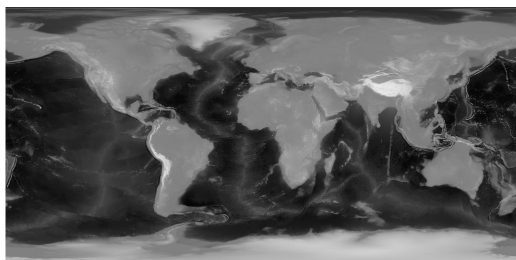
[WMD+04] Roland Wahl, Manuel Massing, Patrick Degener, Michael Guthe, and Reinhard Klein. Scalable compression and rendering of textured terrain data. In *WSCG*, pages 521–528, 2004.

[ZZSP01] Youbing Zhao, Ji Zhou, Jiaoying Shi, and Zhigeng Pan. A fast algorithm for large scale terrain walkthrough. In *International Conference on CAD&Graphics*, 2001.

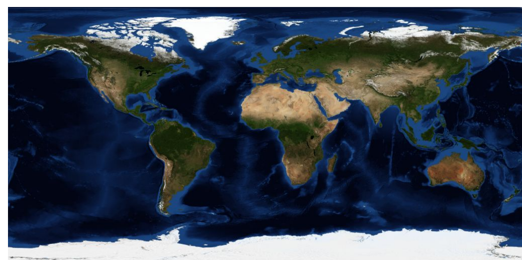
Résumé en français

Introduction

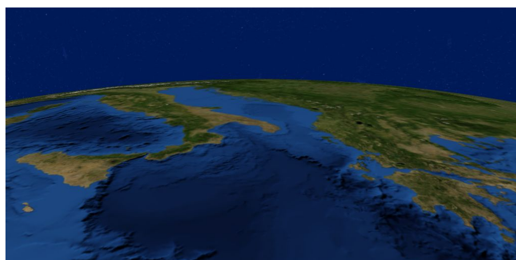
Le chargement progressif et le rendu adaptatif de vastes terrains peut servir, par exemple, à visualiser la Terre hautement détaillée en 3D sur n'importe quel type d'ordinateur pendant que l'on charge les données nécessaires à ce rendu depuis Internet. Ceci est intéressant pour des domaines tels que les outils de navigation GPS (Global Positioning System) en 3D, le tourisme virtuel, et les jeux vidéo tels que des simulateurs de vol ou des jeux multi-joueurs comprenant de très vastes aires de jeu. Actuellement, *Google Earth* et *NASA World Wind* sont des logiciels populaires de rendu de terrain en 3D, proposant des bases de données très détaillées.

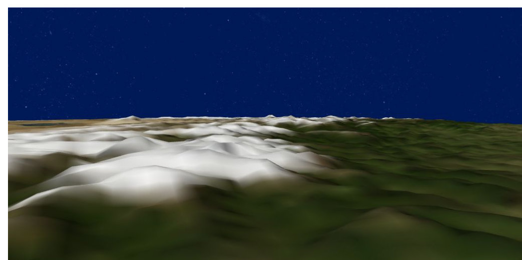


(a) Carte d'élévations



(b) Carte de couleurs





(c) Rendus 3D utilisant des données issues de ces deux cartes.

Figure 1 – Cartes d'échantillons d'élévation et de couleur représentant la surface de la Terre, et des exemples de rendus 3D issus de ces cartes. La base de données GEBCO est utilisée. Voir l'Appendice A pour une présentation des bases de données utilisées dans cette thèse.

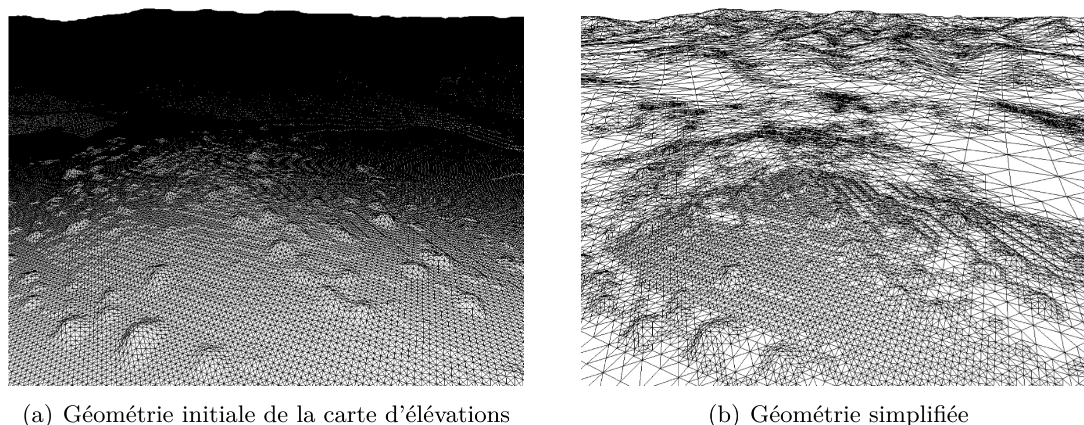
La représentation numérique d'un terrain est un ensemble de cartes d'échantillons en 2D, qui discrétisent régulièrement la surface du terrain. Ce sont généralement des cartes d'élévations afin de reconstruire le relief du terrain en 3D accompagnées de cartes de couleurs qui représentent les détails photométriques de la surface, comme le montre la Figure 1. De telles cartes peuvent être immenses ; par exemple, une carte de la Terre avec une définition de seulement 500 mètres entre les échantillons fait plus de dix giga-octets, tandis que la base de données avec la plus haute définition parmi celles que nous avons utilisé fait plus de 100Go, comprenant élévation et couleur. Une telle quantité de données ne peut pas être chargée entièrement en mémoire ni entièrement rendue en temps réel, quel que soit l'ordinateur utilisé. D'autre part, on souhaite généralement pouvoir visualiser un terrain sans avoir à télécharger ou stocker entièrement une immense base de données localement.

Afin de résoudre ces problèmes, nous devons utiliser des structures de données et des techniques spécifiquement conçues pour supporter le chargement progressif et le rendu de terrains de taille quelconque. C'est le sujet de cette thèse.

Travaux existants

Principaux concepts pour le rendu adaptatif de terrain

La géométrie d'un terrain est généralement représentée par des cartes d'élévations. Chaque échantillon contient une valeur qui donne l'élévation de la surface du terrain en ce point, relativement à une surface de référence tel que le niveau de la mer. L'objectif du rendu adaptatif de terrain est de continuellement mettre à jour une surface géométrique simplifiée représentant le terrain, en fonction de quelques contraintes. Ces contraintes peuvent permettre, par exemple, de contrôler la vitesse ou la qualité du rendu.



(a) Géométrie initiale de la carte d'élévations

(b) Géométrie simplifiée

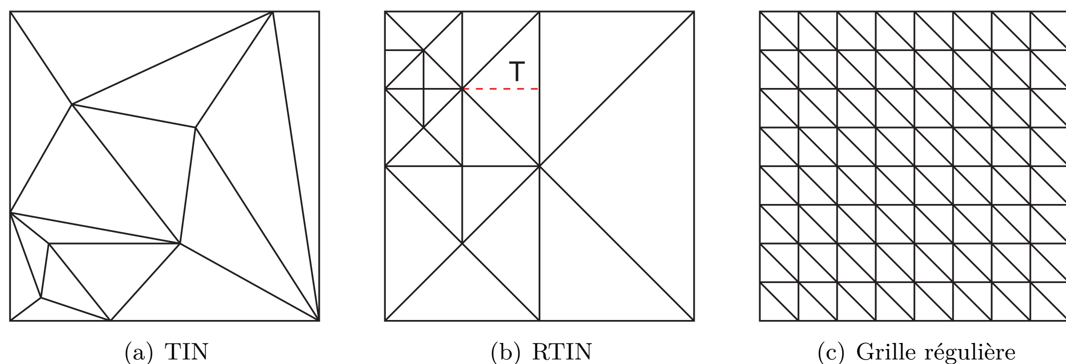
Figure 2 – Exemple de simplification adaptatif de la géométrie d'un terrain [LKR+96].

Dans la plupart des cas, la géométrie d'un terrain est une surface faite de sommets 3D liés entre eux par des triangles qui peuvent être manipulés par les cartes graphiques

actuelles afin d'en effectuer le rendu. Initialement, la carte d'élévations peut être directement convertie en une telle structure en utilisant un sommet par échantillon puis en triangulant régulièrement la grille de sommets. Ensuite, on peut retirer certains sommets et adapter la triangulation afin de maintenir une surface continue tout en simplifiant la géométrie (voir la Figure 2). Réciproquement, quand plus de qualité est demandée pour une partie du terrain, on peut alors rajouter à nouveau les sommets qui avaient été retirés pour raffiner la géométrie. Une grande quantité de techniques basées sur ces deux opérations sont résumées et discutées dans [PG07].

Contraintes de triangulation

Les algorithmes utilisés pour simplifier la géométrie d'un terrain consistent à sélectionner les sommets ou triangles à retirer ou ajouter. Ces algorithmes peuvent utiliser certaines contraintes lors de cette sélection afin d'accélérer et simplifier le processus, bien que ceci implique une qualité de rendu inférieure pour un nombre de triangles donné. Trois types de contraintes peuvent être utilisées, illustrées sur la Figure 3 et décrites dans cette section.



(a) TIN (b) RTIN (c) Grille régulière

Figure 3 – Différentes approches pour la triangulation. La ligne rouge pointillée indique comment le triangle T est découpé dans un RTIN.

Les TINs (Triangulated Irregular Networks) sont des structures complètement irrégulières. Puisqu'il n'a aucune contrainte, ce modèle offre le plus haut rapport entre qualité de rendu et nombre de triangles affichés. Cependant, calculer un TIN simplifié est un processus long et complexe. En pratique, il nécessite une très grande quantité de calculs sur le processeur, ce qui n'est pas acceptable dans le contexte du rendu de terrain en temps réel. En conséquence, les méthodes qui utilisent des TINs le font pour pré-calculer des sous-ensembles de géométrie ou des opérations de mise à jour qui sont ensuite utilisés lors du rendu [Hop98, CGG+03a].

Les RTINs (Right-Triangulated Irregular Networks) sont le modèle le plus utilisé pour représenter la géométrie dans les algorithmes de rendu adaptatif de terrain [LKR+96, DWS+97, RHSS98, Paj98, Blo00, EKT01, LP01, ZZSP01, Lev02, LP02, WMD+04, SW06]. Contrairement aux TINs, tous les triangles sont des triangles iso-

cèles rectangles au vu de la carte d'échantillons. Ceci permet des opérations de mise à jour bien plus simples et rapides, basées sur la découpe et la fusion de triangles, mais implique que plus de triangles doivent être utilisés pour afficher un terrain donné à une qualité donnée, comparé aux TINs. La découpe d'un triangle s'effectue au centre de son hypoténuse (le long de la ligne rouge pointillée sur la Figure 3), créant ainsi deux nouveaux triangles isocèles rectangles deux fois plus petits ; la fusion de triangles est l'opération inverse. Un RTIN est généralement représenté par un arbre binaire qui correspond aux découpes successives des triangles. Ainsi, la simplification de la géométrie s'effectue par un parcours en profondeur de cet arbre incomplet, dont la complexité ne dépend pas de la taille de la carte d'origine.

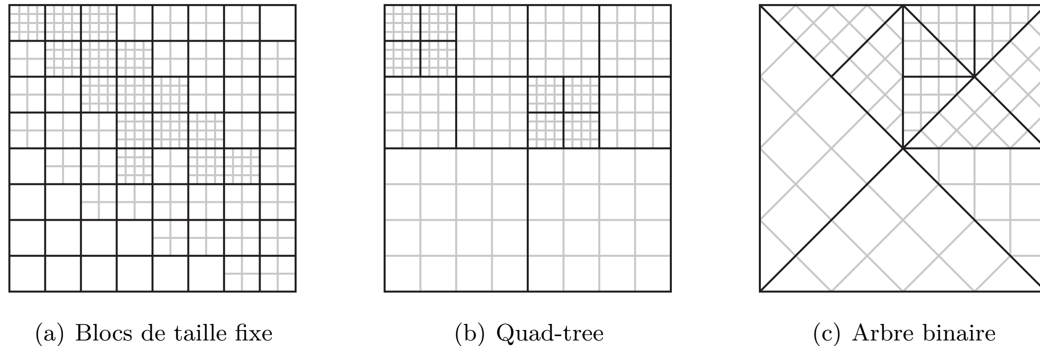
Une autre méthode consiste à positionner un sommet sur chaque point d'une grille régulière constituant un sous-ensemble de la carte d'origine [dB00, LC03, LH04, AH05, PM05, Bro05, HDJ05, LN05, GMC+06, LKES07]. Cette méthode a un assez bas rapport entre qualité de rendu et nombre de triangles affichés, mais elle permet d'obtenir une géométrie simplifiée sans un seul calcul explicite. Ce système est généralement combiné avec un pavage du terrain en plusieurs sous-ensembles carrés ou triangulaires appelés blocs, qui peuvent chacun utiliser une résolution de grille différente. L'utilisation de grilles régulières est adaptée aux cas où l'on peut effectuer le rendu d'une grande quantité de triangles à une vitesse raisonnable sur un matériel dédié tout en souhaitant économiser du temps de calcul sur le processeur central. Nous pouvons remarquer que les travaux utilisant cette méthode sont les plus récents, ce qui est logique étant donnée l'amélioration constante de la puissance de calcul des cartes graphiques.

Structures de données et algorithmes

Nous pouvons classifier la majorité des méthodes existantes pour le rendu adaptatif de terrain selon trois familles, en fonction du compromis effectué entre la vitesse et la capacité d'adaptation.

Certaines méthodes appliquent leurs algorithmes en utilisant les triangles ou sommets comme l'unité de base [LKR+96, DWS+97, RHSS98, Hop98, Paj98, Blo00, LP01, ZZSP01, LP02]. Par conséquent, les changements dans la géométrie peuvent être très petits d'une image à l'autre : la mise à jour est continue. Ces méthodes favorisent la fidélité de la géométrie rendue par rapport au terrain d'origine. Toutefois, elles nécessitent d'avoir toute la carte d'échantillons de potentiellement disponible lors des calculs, autrement dit de la stocker en mémoire vive si l'on souhaite de bonnes performances. Ceci est problématique lors du rendu de très vastes terrains car ils ne logent alors pas entièrement en mémoire, et pose également problème si l'on souhaite effectuer un chargement progressif des données via un réseau.

Une autre approche consiste à paver le terrain en utilisant des blocs de taille fixe, chacun avec un ensemble de niveaux de détail (LOD, pour « level of detail » en anglais) [dB00, LC03, PM05, Bro05, SW06]. Les niveaux de détail sont des sous-ensembles simplifiés de la géométrie du bloc. Il suffit alors de choisir un niveau de détail par bloc et d'utiliser cette géométrie simplifiée pour en effectuer le rendu, sans avoir besoin d'effectuer des opérations de simplification explicites. Comparée à une mise à jour



(a) Blocs de taille fixe (b) Quad-tree (c) Arbre binaire

Figure 4 – Subdivision d'une carte d'élévations en blocs. Chaque bloc (aux arêtes noires) est rendu en utilisant sa propre géométrie (grilles grises).

continue de la géométrie, cette méthode évite un grand nombre d'opérations mais offre une moins bonne précision locale pour la sélection des données à afficher. Comme la triangulation en grille régulière, cette méthode est adaptée au contexte actuel proposant de hautes performances de rendu. D'autre part, utiliser des blocs permet de les charger indépendamment depuis un réseau à la demande. Cependant, tous les blocs doivent être chargés si l'on souhaite afficher l'ensemble du terrain.

Une troisième solution est d'appliquer les algorithmes de mise à jour continue de la géométrie en utilisant des blocs à géométrie fixe à la place des triangles [Lev02, CGG+03a, CGG+03b, WMD+04, HDJ05, LN05, GMC+06, LKES07]. Cette méthode organise la géométrie du terrain en un arbre de blocs qui représentent des parties du terrain de plus en plus petites au fur et à mesure que l'on descend dans l'arbre. Ainsi, beaucoup moins d'objets sont manipulés par les algorithmes de sélection et de mise à jour, ce qui réduit considérablement la charge du processeur. Ceci évite les principaux inconvénients de la méthode précédente : seul un nombre réduit de blocs est nécessaire pour afficher tout le terrain, et plus de blocs sont chargés seulement là où l'on souhaite une plus haute qualité (voir la Figure 4). Nous remarquons cependant que peu de solutions évitent de charger inutilement les données redondantes parmi les blocs qui représentent une même partie du terrain.

Il existe d'autres méthodes ne pouvant pas être classées dans ces trois familles car elles se basent sur d'autres compromis, telles la clipmap [TMJ98, LH04, AH05, CH06] ou la grille de projection [DS04, LSGES06, LRS09], que nous ne décrirons pas ici.

Traitement des artefacts de rendu

Les « fissures » sont des artefacts qui se produisent quand la continuité de la géométrie du terrain n'est pas préservée. Ce sont des trous dans la géométrie qui apparaissent entre deux triangles adjacents lorsque ceux-ci ne partagent pas exactement la même arête. Ceci se produit tout particulièrement lorsque l'on utilise des blocs, car la géométrie est alors simplifiée indépendamment pour chaque bloc.

[LKR+96, DWS+97, RHSS98, Paj98, Blo00, LP01] utilisent des systèmes de dépendance entre sommets ou triangles, et peuvent forcer des découpes récursives dans le voisinage d'un triangle jusqu'à ce qu'il n'y ait pas de fissure (voir la Figure 5). Les solutions qui utilisent des LODs réguliers de résolutions multiples peuvent ajouter ou retirer des sommets sur l'arête d'un bloc pour qu'elle corresponde avec celle de son voisin [dB00, LC03, Bro05, LKES07]. Enfin, les solutions qui utilisent une hiérarchie de blocs triangulaires peuvent assurer que chaque bloc a exactement un voisin sur chacune de ses arêtes, par exemple en forçant des découpes de blocs comme dans la Figure 5 ou bien en utilisant des contraintes particulières pour les opérations de mise à jour [Lev02, CGG+03a, HDJ05, GMC+06]. Avec un seul voisin par arête, on peut alors restreindre le choix des sommets sur ces arêtes pour assurer qu'il n'y a aucune fissure entre voisins sans avoir besoin de connaître leur géométrie respective.

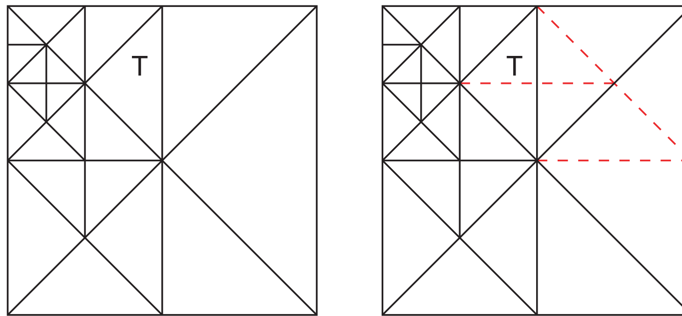


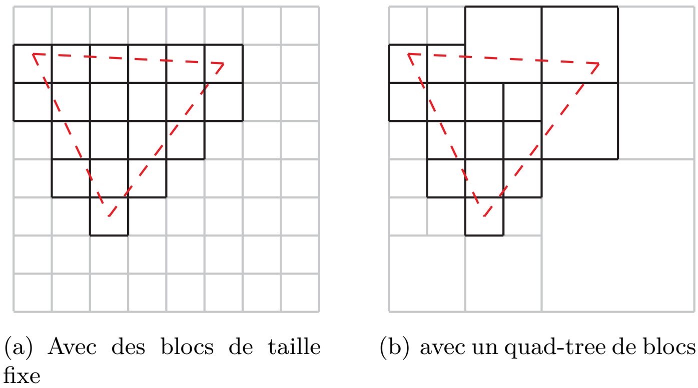
Figure 5 – Découpes de triangles récursives pour éviter les fissures. Pour découper le triangle T, les découpes correspondant aux lignes rouges pointillées sont forcées.

Optimisations

Afin d'obtenir les meilleures performances de rendu possibles, chaque élément de l'ordinateur impliqué dans le rendu doit être utilisée de façon efficace et la quantité de données à traiter pour chaque image doit être minimisée.

Dans le contexte du rendu temps réel, il est possible d'utiliser le calcul parallèle pour effectuer plusieurs tâches de façon désynchronisée. Ceci permet au système d'être plus rapide sur une architecture multi-processeurs et évite des saccades lors du rendu lors des chargements : le rendu peut continuer alors que de nouvelles données sont chargées en parallèle [Hop98, LP01, WMD+04, PM05, GMC+06].

Une façon simple de réduire la complexité de la géométrie à traiter lors du rendu est de ne pas prendre en compte les parties qui ne sont pas visibles étant données les caractéristiques actuelles du point de vue du rendu. Une méthode efficace pour arriver à cela est de paver le terrain avec des blocs et de tester rapidement pour chaque bloc s'il est visible ou non grâce à une boîte englobante [Paj98, ZZSP01, PM05, LN05, SW06] (voir la Figure 6). Cette méthode peut être améliorée lorsque les blocs sont organisés en un arbre : si un bloc est entièrement visible ou invisible, on ne descend plus l'arbre sur la branche correspondante.

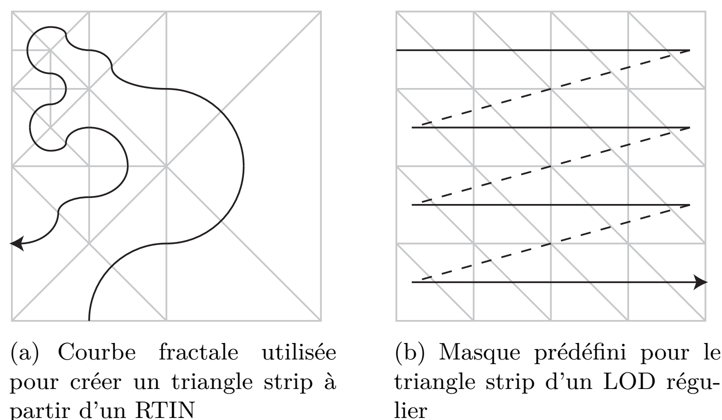


(a) Avec des blocs de taille fixe

(b) avec un quad-tree de blocs

Figure 6 – Elimination de blocs en fonction de la pyramide du point de vue. Les blocs grisés ne sont pas traités lors du rendu.

De nos jours, le rendu 3D est principalement effectué sur un processeur et une mémoire dédiés, ce qui offre de hautes performances. Toutefois, lorsqu'une très grande quantité de triangles sont envoyés à ce matériel dédié à chaque étape de rendu, comme c'est le cas pour le rendu de terrain, le bus utilisé pour l'envoi des données peut limiter la vitesse de rendu. Une méthode pour éviter ce problème est de réduire le nombre de transferts en regroupant les triangles en lots. Dans la plupart des cas, ceci est implémenté avec une structure appelée « triangle strip » qui encode un ensemble de triangles continus de façon optimisée. Les triangle strips peuvent avoir jusqu'à la taille d'un bloc ; ils peuvent de plus être pré-calculés [CGG+03a, GMC+06], calculés grâce à des courbes fractales particulières [LKR+96, Paj98, LP01, Lev02], ou directement obtenus en utilisant des masques prédéfinis grâce à la nature régulière de la triangulation [LC03, WMD+04, LH04, AH05, PM05, LN05, CH06, LKES07] (voir la Figure 7).



(a) Courbe fractale utilisée pour créer un triangle strip à partir d'un RTIN

(b) Masque prédéfini pour le triangle strip d'un LOD régulier

Figure 7 – Triangle strips obtenus à partir de différents types de triangulation.

Une autre solution pour optimiser les transferts vers la mémoire graphique est de

conserver la géométrie dans cette dernière et de réduire la fréquence à laquelle cette géométrie est mise à jour. Cette solution est particulièrement adaptée aux méthodes qui utilisent des blocs à géométrie fixe ou avec un nombre réduit de LODs [Paj98, Lev02, CGG+03a, WMD+04, GMC+06, LC03]. D'autre part, [LH04, AH05, CH06, SW06] utilisent une méthode de mise à jour incrémentale pour ses LODs, permettant d'éviter des envois redondants lors des mises à jour.

Enfin, il est également possible d'effectuer certains calculs une fois pour toutes dans une étape de pré-calcul afin d'obtenir de meilleures performances lors du rendu. Les méthodes qui utilisent des blocs et/ou LODs de géométrie prédéfinie peuvent, par exemple, pré-découper la carte d'échantillons pour que cette géométrie puisse être chargée d'un seul tenant [WMD+04, LH04, AH05, CH06, GMC+06], voire dans certains cas pré-simplifier cette géométrie [Hop98, CGG+03a, SW06].

Cartes de texture

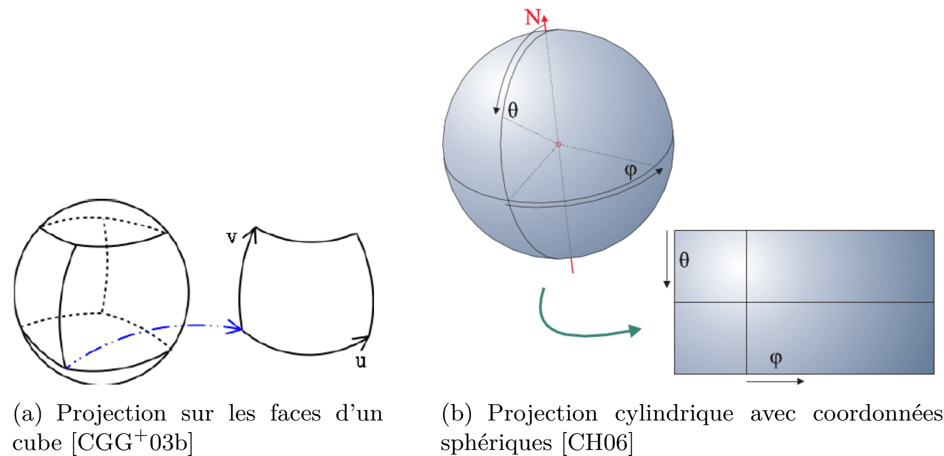
Les cartes de texture sont importantes pour le rendu de terrain parce qu'elles améliorent la qualité visuelle lors du rendu de la géométrie correspondante, en définissant les propriétés photométriques de la surface du terrain telles que sa couleur. La plupart des solutions existantes ne gèrent pas les cartes de textures d'une façon particulière, ce qui offre des possibilités très limitées. A l'inverse, [Blo00, CGG+03a, CGG+03b, WMD+04, LH04, AH05, HDJ05, SW06, GMC+06] traitent les cartes de texture de la même façon que les cartes d'élévation.

Quand un arbre de blocs est utilisé pour simplifier la géométrie, la carte de texture est également découpée en blocs [Blo00, CGG+03a, CGG+03b, WMD+04, HDJ05, GMC+06]. Idéalement, la qualité de la texture est choisi indépendamment de celui de la géométrie, cependant des contraintes matérielles empêchent généralement un bloc de texture d'être plus bas dans l'arbre que le bloc de géométrie associé. Il est alors nécessaire de maintenir un lien entre les deux arbres de blocs.

Terrains planétaires

Bien que les terrains de taille limitée et sans courbure planétaire sont suffisants pour la plupart des applications, le support des terrains planétaires est une fonctionnalité importante pour un système de rendu de terrain. Les planètes, en plus de représenter une très grande quantité de données, sont des terrains sphériques qui doivent alors être projetés sur une carte 2D. De nombreuses méthodes de projection ont été inventées depuis des milliers d'années pour la cartographie [Weib], mais leur utilisation dans le contexte du rendu de terrain a rarement été discuté.

[RLIB99, CH06] et des logiciels tels que *Google Earth* et *NASA World Wind* proposent de projeter la sphère avec une projection cylindrique classique, en paramétrant la surface de cette sphère avec des angles de latitude et de longitude pour créer une carte rectangulaire (voir la Figure 8(b)). Cependant, l'échantillonnage de la surface n'est pas du tout uniforme. En particulier, de nombreux échantillons redondants sont stockés et affichés autour des pôles, ce qui implique une baisse de performances et/ou



(a) Projection sur les faces d'un cube [CGG+03b]

(b) Projection cylindrique avec coordonnées sphériques [CH06]

Figure 8 – Plusieurs techniques de projection de planète.

de qualité. Une autre solution est de projeter la surface de la planète sur un cube, en utilisant six cartes indépendantes qui forment les faces du cube [CGG+03b, GMC+06] (voir la Figure 8(a)).

Aperçu de notre approche

Contrairement à la plupart des travaux existants, nous avons choisi de séparer clairement d'un côté une solution générique et adaptative pour le chargement progressif, la mise à jour et la sélection des données à afficher, et de l'autre côté les méthodes propres à l'interprétation et au rendu de ces données. Une troisième partie contient les méthodes de pré-calcul. Dans la Figure 9, nous introduisons ces trois parties dans lesquelles se partagent nos travaux.

Dans la première partie de cette thèse, nous introduisons une solution générique qui repose sur une structure de données générique. Cette solution manipule des cartes d'échantillons de taille quelconque, depuis le disque dur d'un serveur jusqu'à un système de rendu client.

Pour créer notre structure de données, nous combinons deux principes éprouvés consistant à subdiviser la carte d'échantillons en un arbre de blocs, puis à organiser les données de ces blocs en une succession de niveaux de détail (LODs). Nous ajoutons ensuite de nouvelles propriétés et techniques pour manipuler cette structure de façon plus rapide et adaptative. En particulier, nous minimisons la quantité de données à stocker et à charger en évitant de prendre en compte les échantillons qui sont redondants entre les différents blocs et/ou LODs. Nous permettons aussi d'effectuer le rendu d'un bloc lorsque tous ses LODs ne sont pas disponibles, ce qui permet de charger progressivement les LODs de chaque bloc.

En utilisant cette structure de données, nous chargeons progressivement les données depuis le serveur vers le client, mettant continuellement à jour une base de données par-

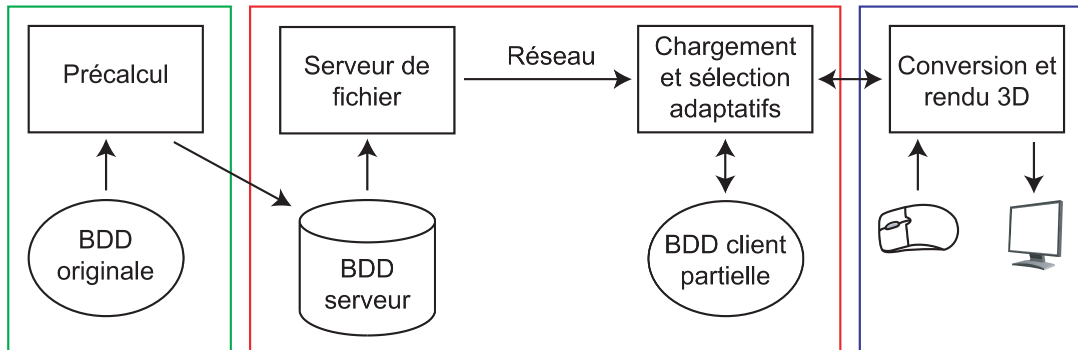


Figure 9 – Les différentes parties de cette thèse. Le cadre rouge central correspond à la solution générique pour le chargement et la sélection de données. Le cadre bleu de droite correspond à l'application de cette solution pour le rendu 3D de vastes terrains. Le cadre vert de gauche correspond aux étapes de pré-calcul.

tielle chez le client de façon adaptative. Nous mettons à jour cette base de données d'une façon qui évite de nombreuses opérations structurelles coûteuses, auxquelles nous préférons des mises à jours effectuées sur place et de façon asynchrone. De plus, nous évitons de surcharger le réseau et optimisons la pertinence des opérations effectuées en mettant continuellement à jour une file de requêtes et en sélectionnant seulement quelques-unes de ces requêtes à la fois, en fonction d'une mesure d'importance. L'unique tâche du serveur est de lire les données demandées depuis un fichier conçu spécifiquement pour lui et de les transmettre au client.

Dans un second temps, nous sélectionnons de façon adaptative les données à afficher depuis la base de données partielle du client et les données manquantes à charger depuis le serveur. Nous guidons cette sélection grâce à une mesure d'importance générique. Cette mesure dépend d'un facteur de qualité qui permet à la solution de s'adapter à une vitesse ou une qualité de rendu désirée. Cette mesure peut également être spécialisée pour différentes applications de la solution générique en fonction de la nature des données manipulées et du système de rendu.

Dans la deuxième partie de cette thèse, nous proposons d'utiliser la solution générique pour effectuer le rendu 3D temps réel de vastes terrains composés de cartes d'élévations et de textures. Nous résolvons les problèmes spécifiques à cette application et proposons des fonctionnalités avancées additionnelles.

Nous corrigeons dans un premier temps les fissures qui apparaissent sur l'arête commune de deux blocs adjacents. Nous améliorons une solution existante afin de prendre en compte que, dans notre contexte, un bloc peut avoir plus d'un voisin le long d'une unique arête et que tous les échantillons d'un bloc ne sont pas forcément toujours disponibles. Contrairement aux solutions existantes, nous permettons également à deux blocs adjacents d'utiliser n'importe quelle définition, tant que ceci ne produit pas de fissures qui ne peuvent pas être corrigées.

Nous ajoutons ensuite la possibilité de manipuler la photométrie et la géométrie du

terrain séparément : nous améliorons la solution générique pour permettre le rendu de géométrie texturée en entretenant un lien entre une carte de texture et une carte d'élévations. Nous supportons également des méthodes de filtrage de données afin d'améliorer la qualité visuelle des niveaux de détail de basse résolution.

Enfin, nous décrivons une méthode pour effectuer le rendu de terrain planétaires. La solution adaptative manipule toujours des cartes d'élévation de façon générique, mais nous reconstruisons la surface de la planète en 3D. Nous projetons la surface de la planète sur les faces d'un cube grâce à une amélioration de la projection gnomonique permettant d'échantillonner cette surface avec un pas angulaire régulier. Cette projection éviter de stocker une grande quantité de données redondantes, et ne souffre pas de la plupart des incohérences de rendu propres aux projections cylindriques habituelles. Nous corrigeons aussi de façon transparente les fissures dans la géométrie au niveau des arêtes du cube. Notre dernière contribution sur ce sujet corrige en partie le problème de la précision de rendu, dont les limitations matérielles créent des artefacts lors du rendu d'un immense objet 3D tel qu'une planète. Avec notre méthode, ces artefacts sont réduits en ajustant de façon adaptative les plans de profondeur minimum et maximum de la pyramide de rendu en temps réel. Nous utilisons également ces plans pour éliminer les parties de la planète situées au-delà de l'horizon sans test supplémentaire.

Dans la troisième et dernière partie de cette thèse, nous décrivons l'étape de pré-calcul qui crée un fichier de base de données utilisé par le serveur dans notre solution générique à partir de n'importe quelle carte 2D. Nous décrivons aussi une autre étape de pré-calcul nécessaire pour la création d'une base de données planétaire. Cette étape re-projette la carte d'origine afin d'obtenir les six cartes correspondant aux six faces de notre cube. Comme le reste de nos méthodes, ces deux étapes de pré-calcul sont conçues de façon à pouvoir supporter des bases de données de n'importe quelle taille.

Chargement et sélection adaptatifs de données génériques

Structure de données générique

Nous basons notre solution de chargement et sélection adaptatifs de cartes d'échantillons sur une structure de données générique qui combine deux méthodes éprouvées : la carte d'échantillons carrée est subdivisée en un arbre de blocs complet et équilibré, et chacun de ces blocs est constitué d'une succession de niveaux de détail de résolution croissante. Les blocs sont des grilles d'échantillons de résolution constante. Chacun est un sous-ensemble carré de la carte d'origine et peut être affiché tel quel.

L'arbre est une subdivision hiérarchique et donc multi-résolution de la carte d'origine. Un exemple est montré sur la Figure 10. Chaque niveau de l'arbre couvre l'ensemble de la carte, avec une résolution croissante lorsque l'on descend l'arbre. A commencer par une seule racine, les nœuds de l'arbre sont des blocs ayant une résolution et un nombre de fils constants. Le nombre minimum de fils est de quatre, ce qui correspond à un quad-tree. Les parties de la carte couvertes par les fils subdivisent uniformément celle couverte par le père.

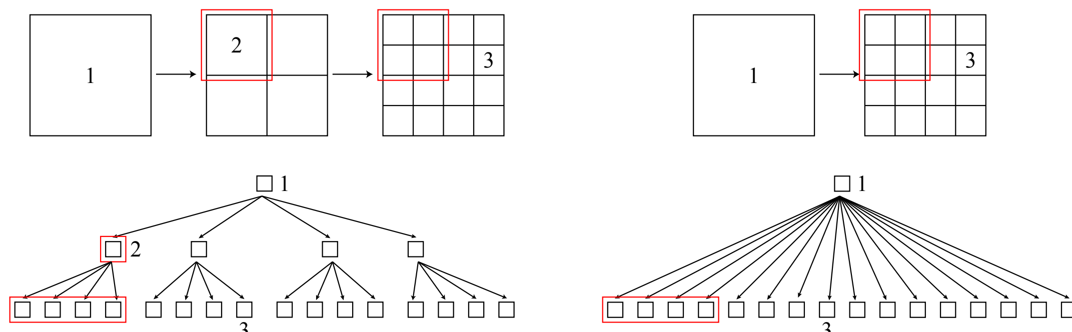


Figure 10 – Construction d'un quad-tree de blocs (**à gauche**) et d'un arbre avec 16 fils par bloc (**à droite**). **En haut** — Subdivisions uniformes successives de la carte d'origine. Les cadres rouges couvrent la même partie de la carte. Le bloc 1 est la racine et couvre l'ensemble de la carte. **En bas** — Les arbres correspondants avec les mêmes numéros de blocs.

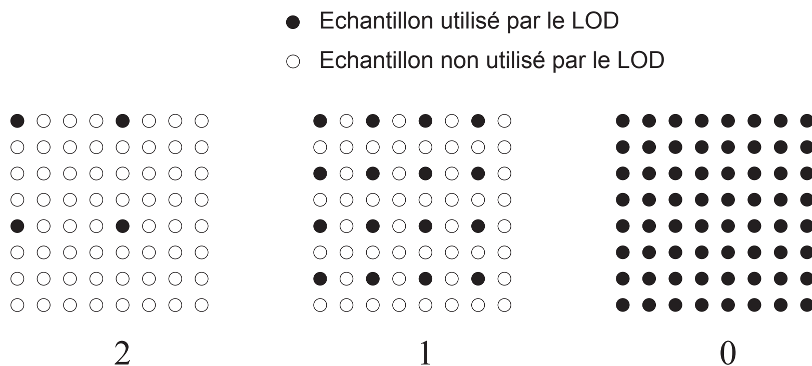


Figure 11 – Trois LODs successifs d'un bloc de 8×8 échantillons.

Les niveaux de détail (LODs) sont des sous-ensembles successifs de la grille d'échantillons d'un bloc comme le montre la Figure 11. Le dernier LOD (numéro 0) utilise la grille entière du bloc. Les sous-ensembles de la grille constituant les LODs sont les mêmes d'un bloc à l'autre afin de permettre l'utilisation de méthodes génériques pour leur stockage, leur mise à jour et leur sélection. Dans le reste de la solution, chaque LOD double la résolution de la grille dans les deux sens comparé au précédent, ce qui quadruple le nombre d'échantillons.

La structure arborescente nous permet d'utiliser seulement quelques blocs des hauts niveaux de l'arbre pour effectuer le rendu des parties de la carte qui ont besoin de peu de qualité, et inversement. Cette technique minimise la quantité totale de données rendues et permet de mieux distribuer ces données sur l'ensemble de la carte. Avec la structuration des blocs en LODs, nous pouvons aussi choisir un LOD à afficher pour chaque bloc en fonction de la qualité désirée. Ceci rend la solution adaptative non seulement au niveau de l'arbre de blocs mais aussi à l'intérieur de ces blocs, ce

qui réduit le nombre d'opérations de structure coûteuses. Une autre propriété est que l'ensemble de la carte peut toujours être affiché avec une qualité minimum même si tous les niveaux de l'arbre ne sont pas chargés. Nous pouvons donc charger progressivement l'arbre, en commençant par la racine puis en descendant là où le besoin se fait sentir par découpe de blocs. Quand un bloc est en cours de chargement, nous continuons à afficher la partie correspondante de la carte avec les niveaux supérieurs de l'arbre.

Le premier paramètre qui définit nos bases de données est le nombre de fils par bloc. En pratique, nous utilisons le nombre de subdivisions par quatre du père nécessaires pour obtenir ce nombre de fils, que nous appelons $subDepth$. Le nombre de LODs par bloc est alors $subDepth - 1$. Le deuxième paramètre est la profondeur de l'arbre, soit le nombre de découpes de blocs successives possibles, que nous appelons $nbSubs$. A partir de la résolution de la carte d'échantillons d'origine selon une dimension $mapWidth$, nous obtenons celle d'un bloc $blockWidth = mapWidth/2^{subDepth \times nbSubs}$.

Pour mieux adapter la structure de données à nos besoins et améliorer son efficacité générale, nous y ajoutons de nouvelles propriétés. Tout d'abord, un bloc et ses fils partagent des échantillons parce qu'ils couvrent la même partie de la carte, qui sont alors recopiés sans chargement lors des opérations de découpe et de fusion. Ceci évite de charger des données redondantes. Ensuite, nous permettons d'effectuer le rendu d'un bloc même si sa grille d'échantillons n'est pas entièrement chargée. Seuls les échantillons du LOD sélectionné pour le rendu doivent être présents. En conséquence, nous pouvons charger progressivement les LODs d'un bloc pour les stocker dans une grille d'échantillons commune tout en utilisant cette grille pour afficher les LODs précédents. Grâce à ces propriétés, nous pouvons découper un bloc sans avoir besoin de charger les données de tous ses fils. Ils obtiennent leur premier LOD depuis le père et seuls ceux qui ont besoin de plus de qualité de rendu commencent à charger leurs LODs suivants.

Enfin, chaque LOD réutilise implicitement le précédent et ajoute ses propres nouveaux échantillons dans la grille du bloc. Ceci évite aussi de charger des données redondantes. Toutefois, ceci implique que les échantillons partagés ne peuvent pas changer de valeur d'un LOD à l'autre, ce qui empêche d'appliquer des méthodes de filtrage d'échantillons. Nous décrivons comment utiliser ces méthodes dans une autre section.

Base de données du serveur

Le premier endroit où sont situées les données est un fichier pré-calculé sur le disque dur du serveur. De nombreux clients peuvent se connecter au serveur simultanément et demander les données de n'importe quel LOD n'importe où sur la carte. Par conséquent, les accès au disque dur sont aléatoires et fréquents. Afin de minimiser l'activité du disque dur, nous optimisons l'organisation des données dans le fichier lors de sa construction par une étape de pré-calcul décrite à la fin de cette thèse.

Le fichier contient d'abord une en-tête comprenant les caractéristiques de la base de données et des informations nécessaires pour en effectuer le rendu. Ensuite, le contenu de l'arbre est aplati LOD par LOD, par exemple comme indiqué sur la Figure 12. Les chargements se font LOD par LOD, assurant ainsi que les données pour toute requête d'un client sont contiguës dans le fichier. La non-redondance des données est

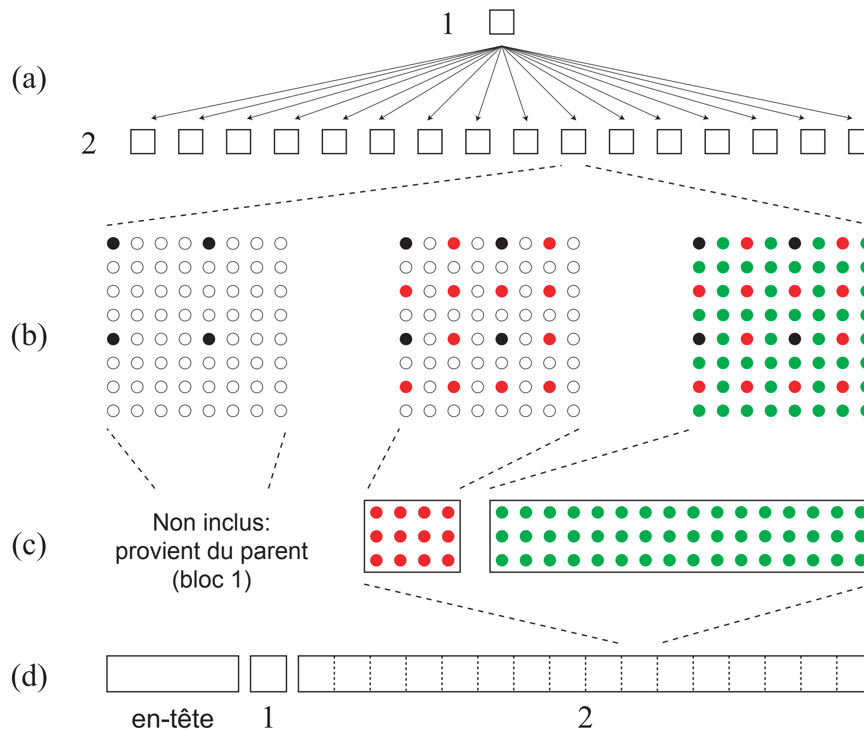


Figure 12 – Organisation du fichier serveur (exemple d'un arbre à deux niveaux avec trois LODs par bloc). **a)** Arbre d'un bloc. **b)** LODs d'un bloc. **c)** Stockage de LOD sans redondance. **d)** Aplatissement de l'arbre dans le fichier.

mise en évidence sur la Figure 12. Premièrement, nous ne stockons pas le premier LOD d'un bloc parce qu'il le partage avec son père. Deuxièmement, chaque LOD réutilise implicitement les précédents donc nous ne stockons que les nouveaux échantillons.

Une requête de chargement de données envoyée par un client doit fournir la position et la taille du LOD dans le fichier. Ces valeurs sont stockées dans les données du précédent LOD ; par conséquent, le client doit les garder temporairement pour chaque LOD. Quand le serveur reçoit une requête, il lui suffit de lire les données correspondantes avec une complexité constante. Avec cette approche, chaque LOD peut être stocké à n'importe quelle position dans le fichier et peut avoir une taille quelconque. Nous ne sommes donc pas limités à l'organisation du fichier présentée dans la Figure 12. Ceci nous permet notamment de compresser les LODs et de construire le fichier linéairement avec l'algorithme de pré-calcul présenté à la fin de cette thèse.

Base de données partielle du client

Du côté du client, la base de données est un sous-ensemble incomplet et généralement déséquilibré de l'arbre de blocs. Le tout premier chargement contient le premier LOD du

bloc racine ; nous pouvons alors immédiatement commencer à afficher la base de données en très basse qualité. L'étape de sélection adaptative de données déclenche alors des opérations de mise à jour de la base de données, ayant pour effet de développer l'arbre progressivement comme le montre la Figure 13.

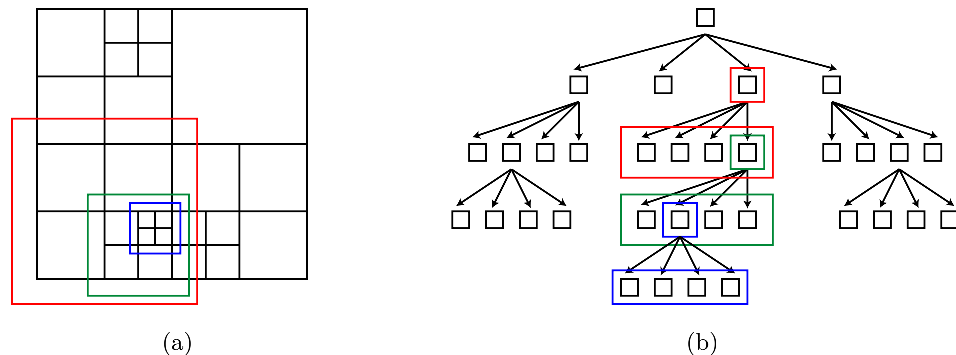


Figure 13 – Chargement progressif d'un quad-tree. **a)** Subdivisions successives de la carte (rouge, puis vert, puis bleu). **b)** L'arbre incomplet correspondant.

Nous stockons les échantillons d'un bloc dans une unique grille ayant la résolution du dernier LOD du bloc. Cette grille est présente en mémoire seulement pour les feuilles de l'arbre incomplet afin de réduire l'utilisation mémoire chez le client. Réciproquement, toutes les feuilles de l'arbre ont une grille d'échantillons. Nous assurons donc que toutes les parties de la carte ont une représentation affichable chez le client.

Si un bloc est actuellement une feuille de l'arbre incomplet et que tous ses LODs sont chargés, nous pouvons utiliser l'opération de découpe pour charger ses fils en mémoire. La grille d'échantillons du père est subdivisée uniformément en sous-ensembles carrés correspondant aux fils. Les échantillons de chaque sous-ensemble sont copiés dans la grille vide du fils afin de créer son premier LOD. Chaque fils peut alors charger de nouvelles données indépendamment des autres. Inversement, nous pouvons utiliser l'opération de fusion pour nous recréer la grille d'échantillons du bloc et récupérer ses données depuis les fils. Les fils peuvent être fusionnés récursivement si besoin. Nous supprimons alors les fils de l'arbre incomplet car ils ne sont plus utilisés pour le rendu.

Enfin, nous pouvons charger un nouveau LOD pour un bloc qui n'est pas entièrement chargé avec l'opération de raffinement. Contrairement aux opération de découpe et de fusion, ceci nécessite un chargement de données depuis le serveur. Quand les données sont reçues, nous ajoutons les nouveaux échantillons dans les emplacements vides de la grille du bloc correspondant au sous-ensemble du LOD comme indiqué sur la Figure 11. Une étape de conversion de données peut avoir lieu lors de l'opération de raffinement. Elle peut servir, par exemple, à décompresser les échantillons reçus et/ou à convertir leurs valeurs pour une représentation mieux adaptée au système rendu.

Chargement et mise à jour adaptatifs de la base de données

Dans cette section, nous présentons comment nous utilisons les opérations présentées dans la section précédente pour remplir progressivement la base de données du client en fonction d'une mesure d'importance. La Figure 14 illustre l'architecture de notre solution client-serveur de chargement progressif de données ainsi que les différentes étapes et processus légers utilisés chez le client pour effectuer les diverses tâches de chargement, mise à jour, sélection et rendu des données.

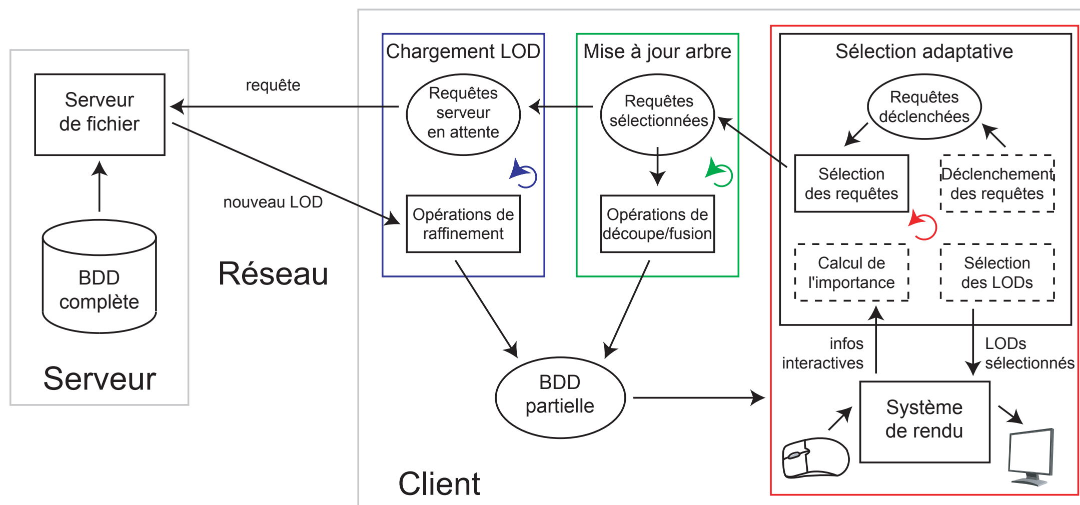


Figure 14 – Etapes détaillées de la solution générique. Les trois cadres colorés avec des flèches circulaires représentent les processus légers s'exécutant en parallèle chez le client. Les étapes dans les cadres noirs sont présentés dans cette section, tandis que ceux avec les cadres pointillés le sont dans la section suivante.

Chargement progressif

Dans notre solution, le serveur se contente de lire et transmettre des données aux clients à la demande. Nous gérons le chargement progressif de ces données du côté client. Quand l'étape de sélection de LOD (présentée dans une autre section) a besoin d'un nouveau LOD pour un bloc, elle déclenche une requête de chargement pendant que le rendu continue en parallèle avec les données disponibles. Quand le client reçoit les données correspondantes, la base de données partielle est mise à jour avec une opération de raffinement et le nouveau LOD peut immédiatement être sélectionné pour le rendu.

Le serveur peut se trouver dans une situation qui l'empêche de répondre rapidement à toutes les requêtes parce que d'autres clients peuvent demander des données. De plus, le réseau peut être lent ou encombré. Malheureusement, plus une requête est en attente, plus il est probable qu'elle soit devenue obsolète quand les données sont enfin reçues, par exemple si l'utilisateur a déplacé le point de vue vers une autre partie de la carte. Pour éviter de surcharger le réseau et le serveur de requêtes non pertinentes, nous fixons

chez le client un nombre maximum de requêtes serveur en attente. Nous choisissons les requêtes à transmettre au serveur en fonction d'une mesure d'importance lors d'une étape de sélection supplémentaire. A tout moment donné, notre méthode charge les données les plus importantes. Ainsi, nous nous adaptons implicitement à la bande passante et la latence du réseau tandis que la qualité de rendu s'améliore constamment.

Mises à jour asynchrones de la base de données

Le client utilise trois processus légers s'exécutant en parallèle (voir la Figure 14). Le processus de chargement des LODs transmet les requêtes de données sélectionnées au serveur puis ajoute les données reçues à la base de données du client. Le processus de mise à jour de l'arbre effectue continuellement des opérations de découpe et de fusion de blocs. Enfin, le processus principal effectue la sélection et le rendu des LODs. Il s'occupe également de déclencher les requêtes de mises à jour de la base de données et de sélectionner parmi ces requêtes celles qui doivent être effectuées en priorité par les autres processus. Cette étape de sélection des requêtes s'applique à la fois pour les requêtes de chargement que celles de mise à jour de l'arbre, bien que dans le second cas il n'y ait pas de latence supplémentaire due au réseau.

En pratique, l'étape de sélection adaptative des LODs assigne une valeur d'importance à chaque requête déclenchée. En plus de la liste des requêtes déclenchées pour chaque type d'opération, nous utilisons une file de requêtes sélectionnées de taille fixe et relativement réduite. A chaque étape de rendu, nous remplissons cette file en sélectionnant les requêtes les plus importantes. Les requêtes qui ne sont pas sélectionnées sont alors supprimées ; à la prochaine étape de rendu, l'étape de sélection adaptative déclenchera de nouvelles requêtes avec des valeurs d'importance mises à jour. Notez que de pouvoir sélectionner plusieurs requêtes à la fois est utile dans les cas où plus d'une opération peut être effectuée par étape de rendu et/ou où plusieurs requêtes peuvent être transmises simultanément au serveur.

En parallèle, les processus de mise à jour de l'arbre et de chargement des LODs effectuent les copies de données correspondant aux requêtes sélectionnées. Puisque ces opérations sont effectuées en parallèle avec la sélection et le rendu, leur impact sur la fluidité du rendu est limité, en particulier sur des architectures multi-processeurs. Toutefois, pour éviter des conflits avec le processus de rendu, les mises à jour de la structure de l'arbre et les suppressions de données sont effectuées par le processus principal au début de chaque étape de rendu (voir l'Algorithme 1).

Sélection adaptative des niveaux de détail pour le rendu

Nous pouvons effectuer le rendu de la base de données partielle du client à tout moment, en suivant les étapes présentées dans la Figure 14 et l'Algorithme 1 et décrites dans cette section.

Notez que seuls les blocs qui sont des feuilles de l'arbre incomplet peuvent être rendus parce qu'eux seuls ont des grilles d'échantillons. Cependant, la carte d'échantillons est rarement entièrement visible et nous pouvons donc éviter de traiter les blocs qui ne

sont pas visibles. Nous décidons rapidement de la visibilité d'un bloc en comparant sa position avec le sous-ensemble de la carte qui est visible. Par exemple, dans le cas d'un rendu 3D, ceci s'effectue par comparaison entre la pyramide de rendu et un volume englobant le bloc. A chaque étape de rendu, nous obtenons la visibilité de tous les blocs par un parcours en profondeur de l'arbre en évitant de parcourir une branche quand nous savons que tous les blocs qui la constituent sont entièrement visibles ou invisibles. Les blocs qui ne sont pas visibles ne sont pas traités lors du rendu et ne peuvent pas déclencher de requêtes de chargement de données ou de découpe.

Algorithm 1: Opérations effectuées chez le client par le processus principal de sélection adaptative à chaque étape de rendu.

```
terminer les opérations de mise à jour de l'arbre
pour chaque bloc faire
    calculer la visibilité
    calculer l'importance
    sélectionner le LOD à afficher
    si LOD sélectionné n'est pas disponible alors
        déclencher une opération de mise à jour
        sélectionner le LOD le plus proche à la place
sélectionner les requêtes à effectuer en parallèle
pour chaque bloc visible faire
    extraire les échantillons du LOD sélectionné
    effectuer le rendu de ce LOD
```

Après avoir sélectionné des LODs disponibles pour effectuer le rendu des blocs visibles, nous envoyons les échantillons de ces LODs au système de rendu en utilisant un ensemble de masques. Un masque est un tableau d'indices qui définit le sous-ensemble de la grille d'échantillons du bloc qui est utilisé par un LOD. Puisque ces sous-ensembles sont les mêmes pour tous les blocs, nous pouvons calculer et stocker une fois pour toutes un seul masque par LOD.

Mesure d'importance

Comme nous l'avons mentionné dans les sections précédentes, nous utilisons une mesure d'importance pour sélectionner les LODs à afficher, pour déclencher les opérations de mise à jour de la base de données et pou définir l'ordre dans lequel nous effectuons ces opérations. Dans cette section, nous décrivons cette mesure dont le but est d'assurer une bonne capacité d'adaptation de la solution à la fois pour le chargement et le rendu. Nous pouvons obtenir une valeur d'importance pour tout bloc à tout moment ; elle représente la qualité désirée pour la partie de la carte que ce bloc couvre. L'importance dépend d'un certain nombre de facteurs. N'importe quels facteurs peuvent être utilisés en fonction du système de rendu et de l'application, mais nous pouvons les regrouper selon plusieurs familles.

Premièrement, l'importance dépend d'un facteur de qualité générique commun à tous les blocs de la carte, qui permet d'assurer que la solution s'adapte à une vitesse ou une qualité de rendu demandée. Deuxièmement, nous prenons en compte le rayon de la partie de la carte couverte par le bloc. Ceci donne notamment une plus grande importance aux blocs qui couvrent des parties plus grandes de la carte. Troisièmement, nous prenons en compte les interactions de l'utilisateur, par exemple via la distance entre le bloc et le point de vue utilisé pour le rendu. Nous pouvons ensuite ajouter d'autres facteurs qui représentent des caractéristiques intrinsèques des blocs.

Nous considérons qu'un LOD, en quadruplant le nombre d'échantillons, double la qualité visuelle comparé au précédent. De plus, comme nous l'expliquons dans la section suivante, nous sélectionnons les LODs selon des seuils d'importance décroissants. Pour refléter ceci, nous prenons d'abord le logarithme en base deux du produit des facteurs, puis l'opposé de cette valeur. La mesure d'importance finale utilisée dans notre solution est présentée dans l'Equation 1 :

$$importance = -\log_2 (facteurQualité \times rayon \times facteurInteractif) \qquad (1)$$

Sélection des niveaux de détail

A chaque étape de rendu, nous utilisons l'importance de chaque bloc pour sélectionner un LOD à utiliser pour le rendu de ce bloc. Chaque LOD a une valeur d'importance associée ; le LOD est sélectionné quand ce seuil est atteint.

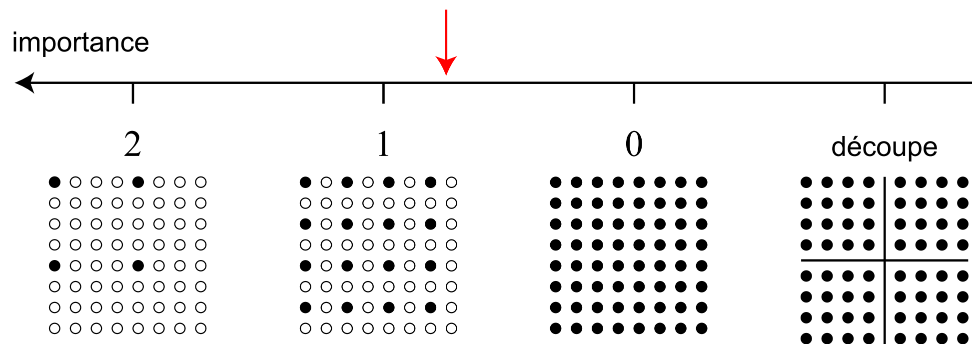


Figure 15 – Seuils de sélection des LODs sur l'échelle d'importance pour un bloc (le même bloc que dans la Figure 11). Par exemple, lorsque l'importance du bloc se situe au niveau de la flèche rouge, le LOD numéro 1 est sélectionné. Puisque nous utilisons des seuils décroissants, une basse valeur d'importance signifie qu'une haute qualité est demandée. Un seuil supplémentaire est utilisé pour déclencher les opérations de découpe, présenté dans la section suivante.

Les LODs étant des grilles régulières d'échantillons sont la résolution est successivement doublée dans chaque sens, nous pouvons utiliser des seuils d'importance prédéfinis. En pratique, nous utilisons simplement le numéro de chaque LOD comme seuil. Comme nous l'avons mentionné auparavant, nous numérotons les LODs en ordre décroissant.

Ceci assure que le LOD numéro 0 utilise la résolution maximum du bloc quel que soit le nombre de LODs du bloc. Ceci permet aux valeurs d'importance et aux seuils des LODs d'être comparables entre des bases de données qui utilisent un nombre de LODs différent. Le fait d'utiliser les numéros des LODs comme seuils d'importance offre une échelle linéaire comme le montre la Figure 15 et permet de sélectionner les LODs simplement en arrondissant la valeur de l'importance à la valeur entière supérieure.

Déclenchement des mises à jour de la base de données

Quand un LOD sélectionné n'est pas disponible, nous utilisons le plus proche parmi ceux qui sont disponibles pour le rendu, puis nous déclenchons une requête pour l'opération de mise à jour de la base de données nécessaire à l'obtention du LOD manquant. Cette requête a une valeur d'importance associée qui est utilisée pour sélectionner les requêtes à effectuer en priorité. Nous obtenons cette valeur en soustrayant la valeur d'importance du bloc à la valeur du seuil du LOD disponible qui est utilisé.

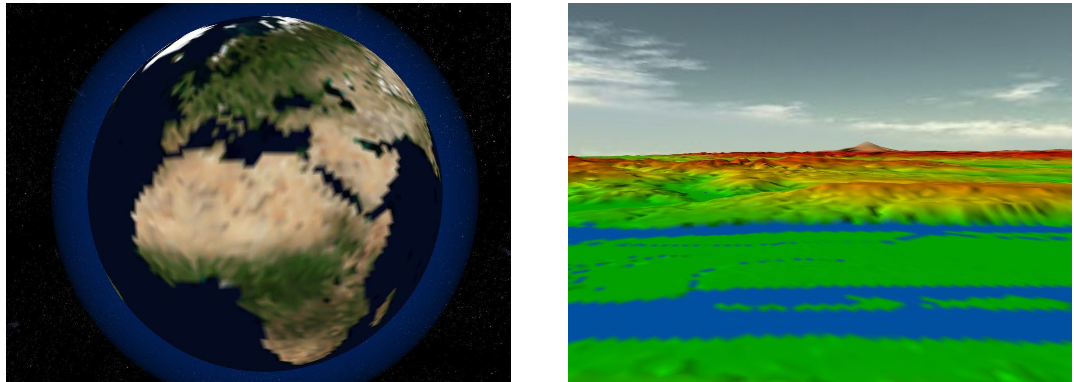
Le premier cas dans lequel un LOD sélectionné n'est pas disponible est celui où ce LOD a une résolution plus grande que celle du meilleur actuellement disponible. C'est par exemple le cas où le LOD numéro 1 de la Figure 15 n'est pas chargé. En ce cas, nous déclenchons une requête de chargement de données qui mènera éventuellement à une opération de raffinement.

Le second cas est celui où le bloc a été découpé auparavant, mais maintenant un LOD de ce bloc est demandé. C'est par exemple le cas où le bloc de la Figure 15 a été découpé. En ce cas nous déclenchons une requête de fusion, en utilisant l'opposé de la valeur d'importance pour la requête parce que les fusions les plus importantes sont celles où la plus basse qualité est demandée.

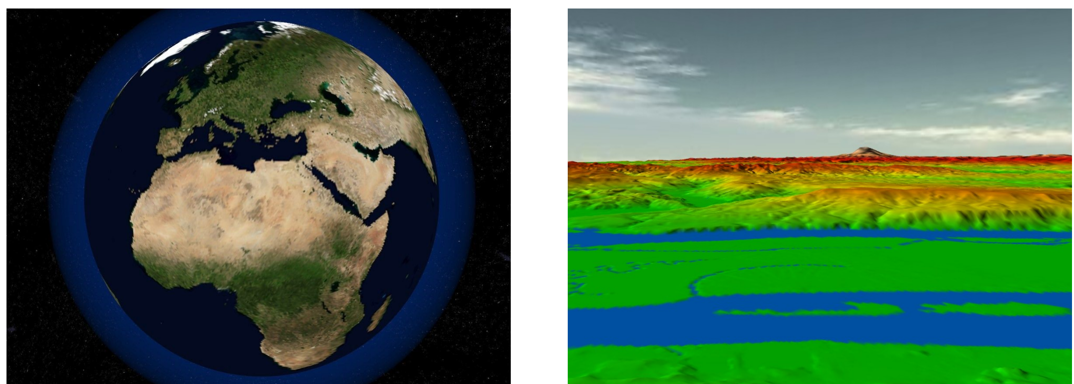
Enfin, nous déclenchons une requête de découpe d'un bloc quand l'un de ses fils devrait charger son second LOD. Nous évitons de découper un bloc dès qu'il est entièrement chargé car cela pourrait mener à une instabilité de la structure de l'arbre si l'importance varie constamment. Toutefois, les fils du bloc ne sont pas encore présents dans l'arbre dont nous devons supposer la valeur d'importance la plus haute parmi les fils en fonction de celle du père. Nous utilisons un seuil supplémentaire sur l'échelle de l'importance, introduit dans la Figure 15. En pratique, celui-ci correspond avec le seuil d'un LOD virtuel numéro -1.

Résultats

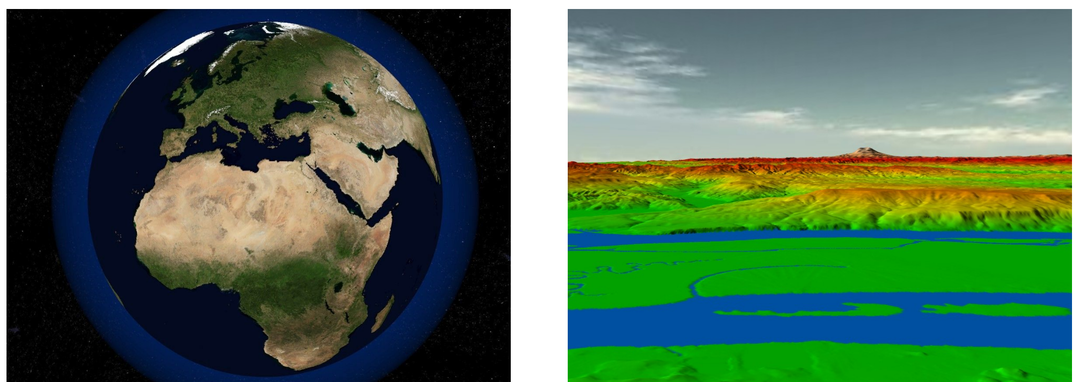
La structure de données et les méthodes que nous venons de présenter sont génériques : nous manipulons des cartes d'échantillons qui peuvent être rendues de plusieurs façons. Nous avons testé notre solution en utilisant deux applications de rendu 3D de terrains, en utilisant des cartes dont les échantillons contiennent à la fois des valeurs d'élévation et de couleur. La première application manipule des terrains dont l'élévation est relative à un plan ; l'autre application supporte des terrains planétaires. Les sommets 3D calculés à partir des valeurs l'élévation sont reliés avec des triangles pour reconstruire la surface 3D du terrain. Les problèmes spécifiques à ces applications sont discutés dans la partie



(a) Après 2 secondes de chargement



(b) Après 10 secondes de chargement



(c) Après 40 secondes de chargement

Figure 16 – Rendus 3D des bases de données BMNG (**à gauche**) et Puget Sound (**à droite**) après avoir chargé des données à 40Ko/s pendant la période indiquée.

suivante de cette thèse.

Nous avons testé les deux applications avec des bases de données et des interactions utilisateur typiques. Les fichiers de base de données présents sur le disque dur du serveur ont été pré-calculés, avec notamment une étape de compression des LODs afin d'améliorer la vitesse de chargement (voir les résultats des étapes de pré-calcul). Les données ont été chargées via une connexion Internet par ADSL avec une bande passante d'environ 40 kilo-octets par seconde.

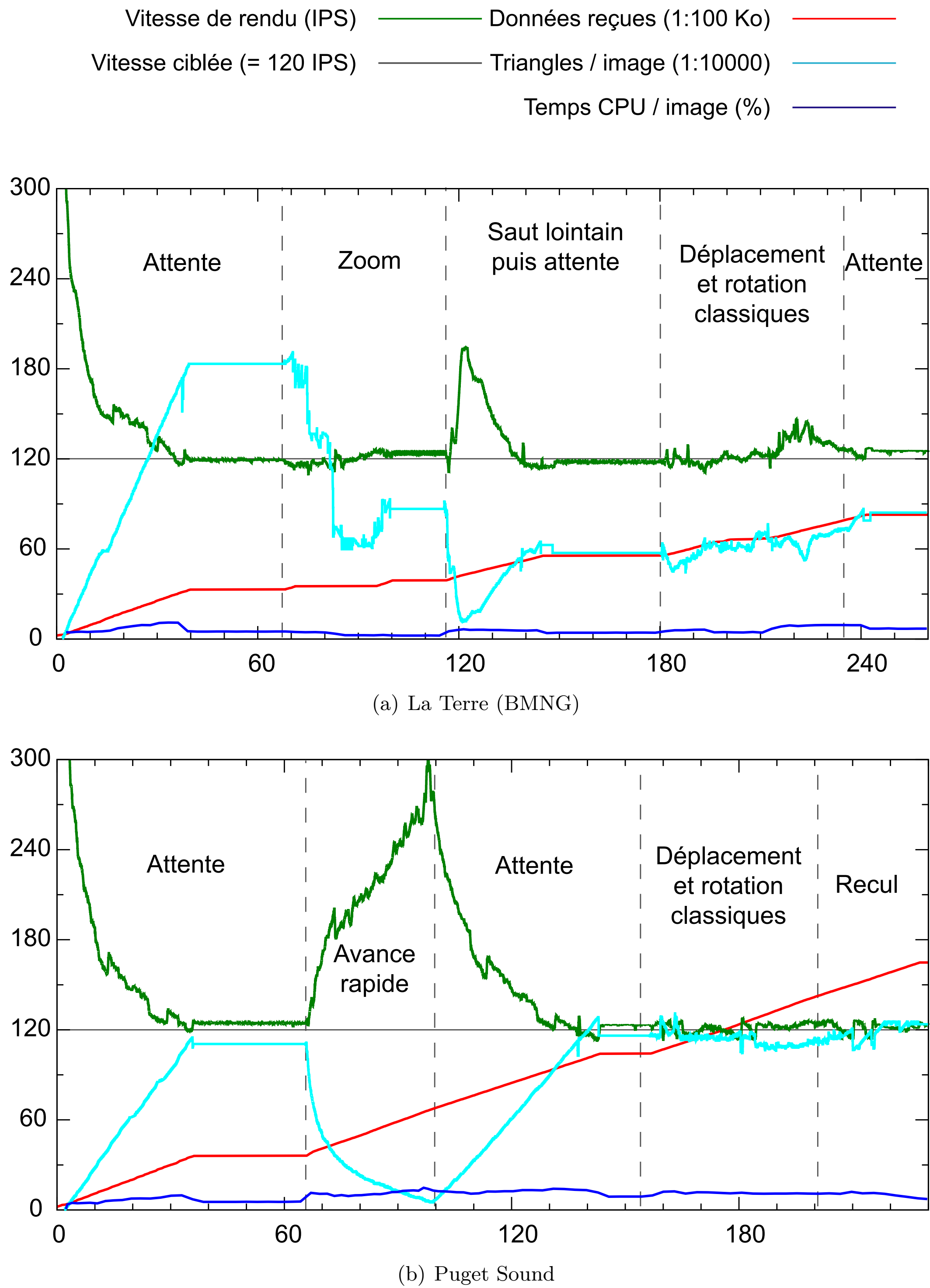
Nous avons exécuté l'application client sur un ordinateur équipé d'un processeur Core 2 Duo 3.2GHz et d'une carte graphique GeForce 9800GTX+, en effectuant le rendu avec une résolution d'écran de 1680×1050 pixels et en activant l'anti-crénelage matériel. Nous avons demandé 120 images par secondes (IPS) pour guider le facteur de qualité de la mesure d'importance. Les statistiques issues de nos tests sont présentées sur la Figure 17, et des rendus 3D obtenus après avoir chargé les données pendant des périodes définies sont présentés sur la Figure 16. Nous avons aussi effectué les mêmes tests sur un ordinateur portable peu performant, en lui demandant 15 images par secondes. La solution s'est correctement adaptée à ces nouvelles conditions, affichant un comportement similaire à ce qui est présenté sur la Figure 17.

Sur la Figure 17(a), nous présentons les résultats obtenus avec la base de données terrestres BMNG qui fait 2.76Go. Nous pouvons voir que le chargement s'arrête dès que la vitesse de rendu passe en-dessous du seuil déterminé : à ce moment là, le facteur de qualité converge vers une valeur constante. Aucun chargement supplémentaire n'est nécessaire puisque les données ont été chargées dans l'ordre de leur importance. Le nombre de triangles affichés par image devient également constant jusqu'à ce qu'une nouvelle interaction de l'utilisateur modifie un facteur de la mesure d'importance. Dans des conditions standard à partir de la seconde 180, le réseau n'est pas toujours assez rapide pour offrir la meilleure qualité possible à une vitesse de rendu stable, cependant plus de 95% des données reçues sont immédiatement utilisées pour le rendu. Comme nous pouvons le voir, nos méthodes de sélection et de mise à jour des données effectuées sur le processeur central (CPU) prennent environ 5 à 10% du temps pour chaque image.

Les résultats du test sur la base de données Puget Sound (terrain non planétaire de 700Mo) sont présentés sur la Figure 17(b). Quand le point de vue se déplace rapidement vers l'avant, la plupart des données précédemment disponibles ne sont plus affichées puisqu'elles passent derrière le point de vue. La latence du réseau nous empêche de les remplacer immédiatement par des données pour les parties visibles du terrain, d'où le pic en termes d'images par seconde. Cependant, dans des conditions standard à partir de la seconde 155, la vitesse de rendu est stable alors que des données sont constamment chargées car le facteur de qualité adaptatif compense la latence du réseau.

Conclusion

Dans cette partie, nous avons proposé une solution générique pour le chargement progressif et la sélection adaptative de données pour le rendu d'immenses cartes d'échantillons. Nos méthodes s'appliquent pour tous types de données : elles peuvent être appliquées à différentes situations seulement en adaptant le système de rendu et la mesure



(a) La Terre (BMNG)

(b) Puget Sound

Figure 17 – Statistiques issues de sessions de rendu 3D interactif, en chargeant les données à 40Kos. L'axe horizontal est le temps en secondes.

d'importance. Nous pouvons, par exemple, charger progressivement une photographie aérienne pour en effectuer le rendu avec une fonction de zoom, ou bien visualiser la Terre en 3D dans une application de géo-positionnement. Nous nous adaptons à la vitesse de rendu et de chargement afin que les performances ne dépendent pas de la taille de la base de données. Nous pouvons donc utiliser une unique base de données sur un unique serveur avec n'importe quel type de client, comme un téléphone avec connexion 3G ou un ordinateur personnel avec une carte graphique 3D et une connexion câblée.

Nous avons utilisé une structure de données disposant de bonnes propriétés et avons ajouté de nouvelles méthodes pour la manipuler plus efficacement. Nous avons utilisé cette structure pour transporter nos données depuis le disque dur du serveur jusqu'au système de rendu du client. En particulier, nous avons évité de charger des données non pertinentes, par exemple en assurant que les données ne sont pas redondantes entre les chargements successifs et en transmettant toujours les données les plus importantes en premier. Nous avons aussi évité autant que possible les opérations de structure coûteuses, en privilégiant des mises à jour sur place via des masques d'échantillons.

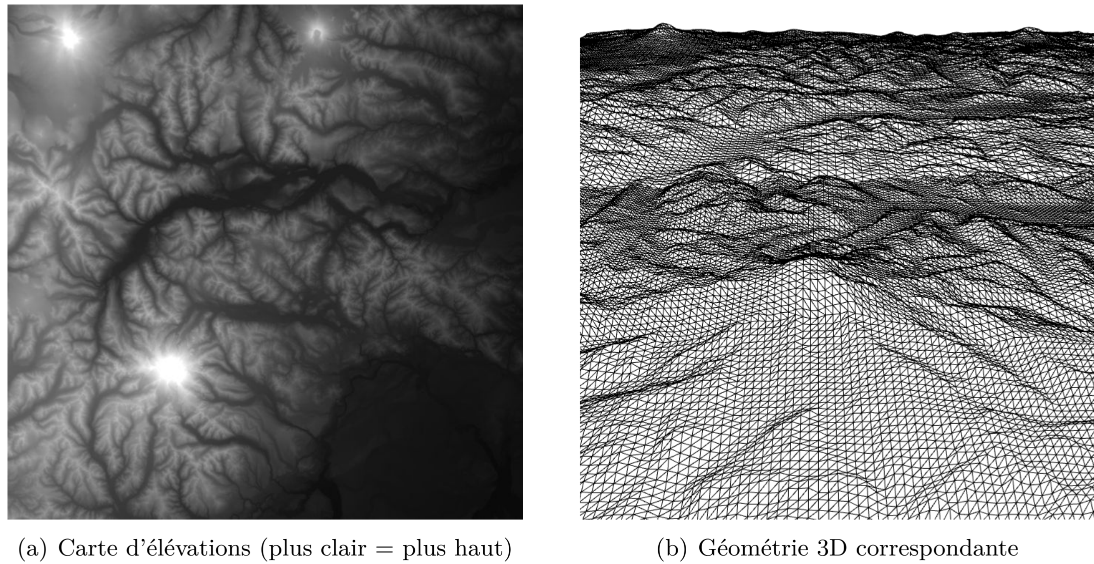
Application au rendu 3D de vastes terrains et planètes

Rendu de géométrie 3D à partir de cartes d'élévations

L'application la plus intéressante de la solution générique présentée dans la partie précédente est le chargement et le rendu adaptatifs de vastes terrains. En particulier, cette application consiste à manipuler des cartes d'élévations représentant la surface 3D du terrain et à effectuer le rendu 3D de ce terrain en temps réel sur du matériel dédié. Pour cela, nous devons dans un premier temps transformer les valeurs d'élévation des échantillons en coordonnées 3D en virgule flottante dans un repère général. Les échantillons sont alors appelés sommets et sont connectés par des triangles pour obtenir une représentation 3D de la géométrie du terrain comme nous pouvons le voir sur la Figure 18, qui peut être traitée par le matériel actuel de rendu 3D.

Nous utilisons une solution de chargement et de sélection de données qui subdivise la carte d'élévations avec des blocs d'échantillons qui sont alors rendus indépendamment les uns des autres. Pour obtenir un ensemble continu de triangles connectant les sommets de deux blocs adjacents, nous stockons de manière redondante les échantillons de l'arête commune dans les grilles des deux blocs. Chaque bloc dispose donc d'une ligne et d'une colonne supplémentaires dans sa grille d'échantillons, respectivement sur les arêtes du bas et de droite. Cependant, des discontinuités peuvent toujours apparaître lorsque des blocs adjacents ne sont pas situés sur le même niveau de l'arbre, et/ou n'ont pas sélectionné le même LOD. Nous résolvons ces fissures dans une autre section.

Dans le cas où les valeurs d'élévation sont relatives à un plan de référence comme le niveau de la mer, la conversion des échantillons en sommets 3D se résume à élever chaque sommet placé sur ce plan de la valeur d'élévation correspondante. Pour des raisons de vitesse de rendu, nous convertissons les échantillons une fois pour toutes lors des opérations de raffinement avant de les stocker dans la grille d'échantillons.



(a) Carte d'élévations (plus clair = plus haut) (b) Géométrie 3D correspondante

Figure 18 – Rendu 3D d'une carte d'élévations avec des sommets et des triangles.

Optimisations

Lorsque nous utilisons du matériel dédié au rendu 3D, les masques servant à extraire les échantillons d'élévation de chaque LOD depuis la grille de son bloc peuvent être implémentés avec des « triangle strips ». Cette structure standard définit un ensemble de triangles contigus grâce à une succession d'indices pointant vers un tableau de sommets 3D, comme indiqué sur la Figure 19. De cette façon les masques d'échantillons sont appliqués directement dans le matériel de rendu, ce qui économise une copie de données en plus de réduire le nombre de transferts vers la mémoire graphique.

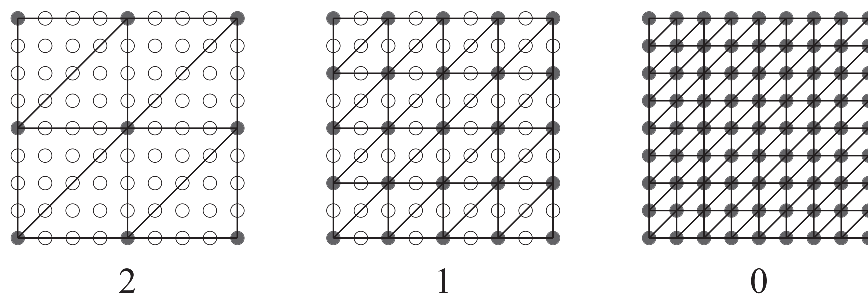


Figure 19 – Masques de triangle strips pour le rendu 3D d'un bloc d'une carte d'élévations (le même bloc que dans la Figure 11). Pour chaque LOD, les échantillons correspondants dans la grille sont connectés par des triangles. Leurs valeurs d'élévations sont utilisées pour obtenir les coordonnées 3D des sommets.

De plus, nous évitons de transférer les données d'un même LOD vers la mémoire graphique à chaque étape de rendu en les stockant dans cette mémoire grâce aux « buffer

objects ». Ces structures permettent de stocker aussi bien les sommets que les triangle strips constituant les LODs. Nous remplaçons ces données par des nouvelles uniquement quand un nouveau LOD est sélectionné pour le bloc. Sur du matériel récent permettant d'afficher des millions de triangles par image à une vitesse acceptable, ceci évite de nombreux transferts de données. En pratique, cette technique a augmenté la vitesse de rendu jusqu'à 40%.

Mesure d'importance

Les facteurs de la mesure d'importance générique peuvent être spécialisés au contexte du rendu 3D de cartes d'élévations. En particulier, nous guidons alors le facteur de qualité général par le nombre d'images calculées par secondes par le système de rendu. En pratique, l'utilisateur choisit une vitesse de rendu désirée : quand la vitesse de rendu s'éloigne de la valeur demandée, le facteur de qualité s'adapte proportionnellement de façon à ce que nous obtenions cette vitesse une fois toutes les données nécessaires chargées. Nous pouvons également ajouter d'autres informations pour obtenir une mesure d'importance plus spécifique, tels que l'angle incident du point de vue ou un facteur de rugosité ou de dénivellation du terrain.

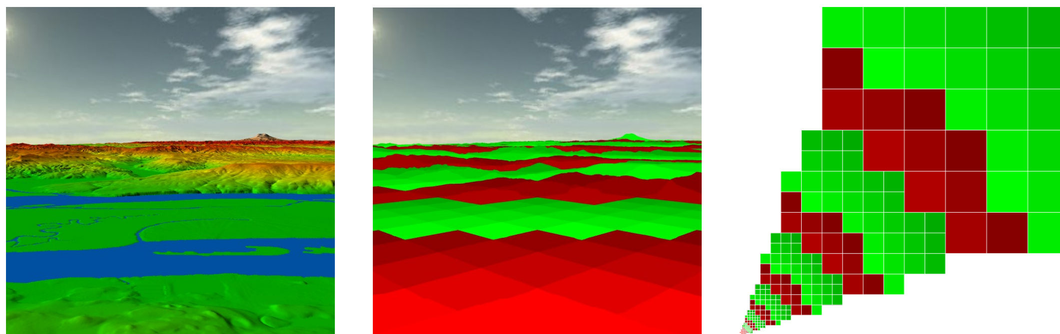
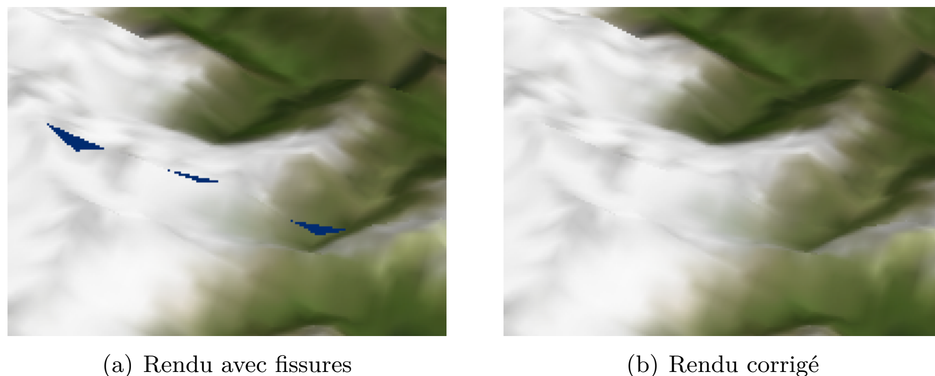


Figure 20 – Impact de la mesure d'importance sur le rendu 3D. **A gauche** – Rendu 3D du terrain. Les couleurs sont incluses dans les échantillons. **Au milieu** – Rendu 3D du terrain utilisant l'importance des blocs comme couleur : le rouge est plus important que le vert, et plus une couleur est claire plus le bloc est important. En utilisant deux LODs par bloc, nous pouvons alors les sélectionner comme sur l'image : vert pour le premier LOD et rouge pour le second. **A droite** – Vue de dessus du terrain avec les boîtes englobantes des blocs utilisées pour éliminer les blocs qui ne sont pas visibles. Le point de vue se trouve en bas à gauche de la carte et regarde en haut à droite.

Sans facteurs additionnels, l'impact de la mesure d'importance sur notre application de rendu 3D de terrains est montré sur la Figure 20. Nous pouvons voir que de guider l'importance selon la distance entre les blocs et le point de vue génère des bandes de couleur autour du point de vue. Celles-ci correspondent aux niveaux de l'arbre : plus une partie du terrain est importante, plus elle est découpée et donc les blocs qui la constituent obtiennent une importance moindre puisqu'ils sont plus petits.

Correction des fissures aux arêtes de blocs

Dans le contexte du rendu de géométrie 3D avec des triangles, la subdivision de cette géométrie en blocs avec niveaux de détail donne naissance à un artefact typique illustré sur la Figure 21 : des fissures verticales apparaissent quand deux blocs adjacents n'utilisent pas la même définition sur leur arête commune.



(a) Rendu avec fissures (b) Rendu corrigé

Figure 21 – Rendus obtenus avec et sans fissures (zoom).

Nous évitons l'apparition des cracks en adaptant les échantillons utilisés sur l'arête commune de deux blocs adjacents ou « voisins », en utilisant des masques supplémentaires sur ces arêtes comme nous pouvons le voir sur la Figure 22. Dans tous les cas, c'est le bloc qui utilise la plus haute définition qui s'adapte, en excluant les échantillons qui ne sont pas présents dans l'autre bloc. Autrement, nous pourrions avoir besoin de données non encore chargées. Quand un bloc doit s'adapter sur l'une de ses arêtes, nous effectuons le rendu de ce bloc en utilisant cinq triangle strips au lieu d'un seul : un triangle strip central qui correspond au LOD sélectionné, et un triangle strip par arête qui relie le triangle strip central avec le voisin correspondant.

Les LODs sont échantillonnés régulièrement donc nous savons d'avance quels échantillons sont utilisés sur les arêtes d'un bloc pour peu que nous connaissions le LOD sélectionné. Nous pouvons donc calculer le triangle strip utilisé sur une arête pour s'adapter à un bloc en connaissant seulement la différence de définition entre les deux blocs, ou « différence de LOD », obtenue à partir des LODs sélectionnés par les deux blocs et leur niveau dans l'arbre. En pratique, nous pré-calculons tous les triangle strips d'arête possibles et les sélectionnons en fonction de la différence de LOD avec le voisin.

Ajout à la structure de données : blocs voisins

La décision de savoir si un bloc doit s'adapter sur ses arêtes demande de connaître ses voisins. Nous maintenons donc un tableau de quatre voisins par bloc, un par arête. Les voisins peuvent être sur d'autres niveaux de l'arbre car celui-ci n'est pas équilibré. Pendant les opérations de découpe et de fusion, nous obtenons les nouveaux voisins des blocs concernés via leur parent commun. Cependant, le tableau de voisins d'un bloc

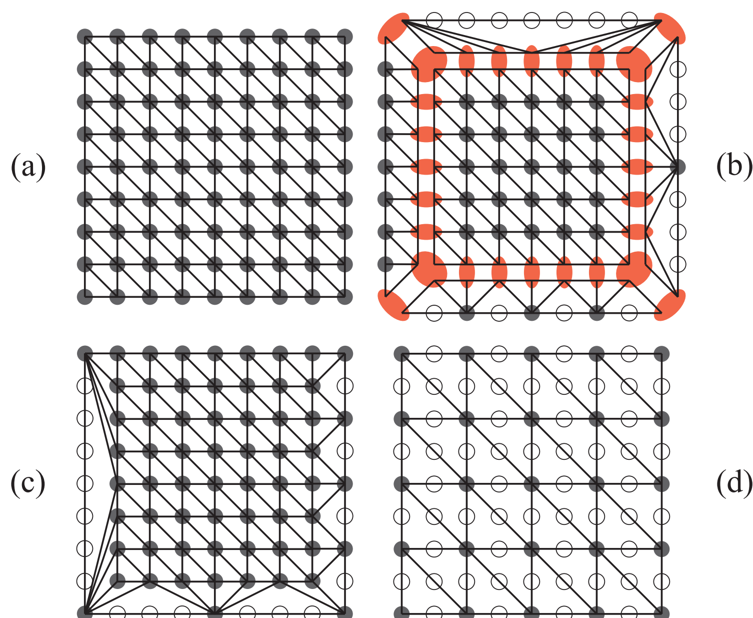


Figure 22 – Masques de triangle strips supplémentaires pour les arêtes des blocs. **a)** Bloc utilisant son dernier LOD. **b)** Bloc utilisant son avant-dernier LOD, avec des triangles strips supplémentaires sur ses arêtes pour d'adapter aux voisins selon la différence de définition. A gauche : aucune différence, en bas : un niveau, à droite : deux niveaux, en haut : trois niveaux. Les échantillons rouges sont élargis ici afin de mieux différencier les triangle strips. **c)** Miroir de (b), avec les triangle strips reliés comme lors du rendu. **d)** Bloc utilisant son avant-dernier LOD.

n'est pas mis à jour quand l'un de ses voisins situé sur le même niveau de l'arbre se découpe car ceci demanderait plus d'un voisin par arête. Dans ce cas, nous savons déjà que le bloc a une définition inférieure ou égale à celle des fils de son voisin, donc que le bloc ne s'adaptera pas sur cette arête.

Au moment d'effectuer le rendu d'un bloc, nous vérifions d'abord pour chaque arête que le voisin est visible et n'est pas découpé. Si c'est pas le cas, nous calculons alors la différence de LOD. Le bloc s'adapte sur l'arête en question si la différence de LOD est positive, autrement dit s'il utilise une définition supérieure à celle de son voisin.

Cas impossibles à corriger

La plupart des solutions existantes forcent les blocs voisins soit à n'avoir qu'un seul niveau de différence dans l'arbre. Certaines empêchent également les blocs d'utiliser des LODs internes afin de réduire le nombre de cas où des fissures apparaissent. Ceci assure que les voisins n'ont à s'adapter qu'à une seule différence de définition possible. Nous avons choisi d'autoriser plus de différences de définition entre voisins pour obtenir plus de libertés dans la sélection des LODs et pour éviter de devoir répercuter une opération

de découpe sur de nombreux blocs lorsqu'un seul en a besoin. Toutefois, dans certains cas nous ne pouvons pas recoller l'arête commune de deux blocs voisins s'il y a une grande différence entre leurs niveaux dans l'arbre.

Pour assurer qu'aucune fissure insolvable n'apparaît, nous fixons une différence maximum de niveau de l'arbre entre voisins, que nous calculons en fonction des paramètres de la base de données. Avant d'effectuer les opérations de découpe et de fusion, nous vérifions des contraintes basées sur cette différence maximum de niveau afin de savoir si l'opération est autorisée. De telles contraintes pourraient cependant empêcher des blocs à haute importance de se découper, par exemple si l'un de ses voisins n'est pas visible et ne se découpe donc pas lui-même. Dans ce cas, nous forçons les voisins concernés à déclencher des opérations de découpe et/ou de raffinement jusqu'à ce que les contraintes nécessaires à la découpe du bloc important soient vérifiées.

Cartes de texture

Les cartes de texture pour un terrain sont généralement plus grandes que les cartes d'élévation. Elles représentent les propriétés photométriques de la partie de terrain correspondante. Les cartes de texture peuvent contenir des valeurs de couleur, ou des vecteurs dits « normales » qui améliorent la qualité d'éclairage du terrain. Nous souhaitons pouvoir choisir les niveaux de détail de ces cartes de texture aussi indépendamment que possible de ceux des cartes d'élévations. Nous avons donc utilisé notre solution générique pour supporter le chargement et la sélection adaptatifs de cartes de textures, en utilisant une mesure d'importance spécifique.

Ajout à la structure de données : arbres liés

Un bloc de texture ne s'affiche pas seul, il est utilisé par un ou plusieurs blocs d'élévations lors de leur rendu. Comme les cartes de texture et d'élévations utilisent des arbres distincts, nous maintenons un lien entre ces deux arbres incomplets de façon à ce que les blocs d'élévations puissent savoir quels blocs de texture utiliser.

Les méthodes classiques de rendu matériel de texture nécessites qu'un bloc d'élévations rendu avec un triangle strip n'utilise qu'un seul bloc de texture. En pratique, il utilise le bloc d'élévations associé si celui-ci est présent, sinon il demande récursivement à son père jusqu'à obtenir un bloc de texture. Réciproquement, il n'est pas possible d'effectuer le rendu d'un bloc de texture s'il n'a pas de bloc d'élévations associé. Par conséquent, nous devons utiliser des blocs de texture de résolution supérieure à celle des blocs d'élévations si nous souhaitons pouvoir obtenir une meilleure définition pour la texture que pour la géométrie. Nous vérifions également plusieurs contraintes avant d'effectuer les opérations de découpe et de fusion pour assurer que nous n'obtenons jamais de blocs de texture qui ne peuvent pas être rendus.

Rendu

Les blocs d'élévations ont besoin de coordonnées de textures pour faire la correspondance entre ses sommets et les échantillons de texture. Nous stockons un seul masque

pour les coordonnées de textures de tous les blocs d'élévation. Quand le bloc de texture associé à un bloc d'élévations couvre une partie du terrain plus grande que la sienne, nous utilisons la matrice de texture du matériel de rendu pour zoomer et décaler automatiquement les coordonnées de textures afin d'obtenir le bon sous-ensemble.

Enfin, les LODs des textures sont implémentés en tant que « mipmaps » pour leur rendu 3D. Les mipmaps sont stockées en mémoire graphique et le matériel de rendu peut alors sélectionner indépendamment un LOD à utiliser pour le rendu. Cependant quand seul le premier LOD d'un bloc est disponible, aucune sélection n'est possible. Pour éviter cela, lors des opérations de découpe nous créons des mipmaps supplémentaires qui correspondent à des LODs virtuels de basse résolution.

Filtrage

Bien qu'un sous-échantillonnage régulier soit généralement acceptable pour les cartes d'élévations, ce n'est pas le cas pour les cartes de texture. Nous choisissons donc d'utiliser un filtrage des échantillons lors de la construction du fichier serveur correspondant. Nous pouvons utiliser n'importe quelle méthode de filtrage tant que celle-ci produit des LODs à échantillonnage régulier. Nous avons choisi d'utiliser la méthode « Progressive Texture Map » [MB03] car elle est très rapide à décoder et peut donc être utilisée sur des clients basse performance. Une comparaison visuelle peut être vue sur la Figure 23.

(a) Sous-échantillonnage régulier

(b) Progressive Texture Map

Figure 23 – Rendus en basse définition de la base de données de texture terrestre TrueMarble avec et sans filtrage d'échantillons. Sans filtrage l'Espagne et le Maroc semblent reliés et les bords de mer ou chaînes de montagnes ne sont pas continus. Avec filtrage tout est plus flou mais les détails topologiques sont mieux conservés.

Quand nous utilisons une méthode de filtrage des échantillons, les différents LODs d'un bloc utilisent différentes valeurs pour un même échantillon. Ceci nous empêche d'utiliser une unique grille d'échantillon par bloc. Nous stockons donc chaque LOD dans une grille séparés. Bien que ceci implique de la redondance de données, nous

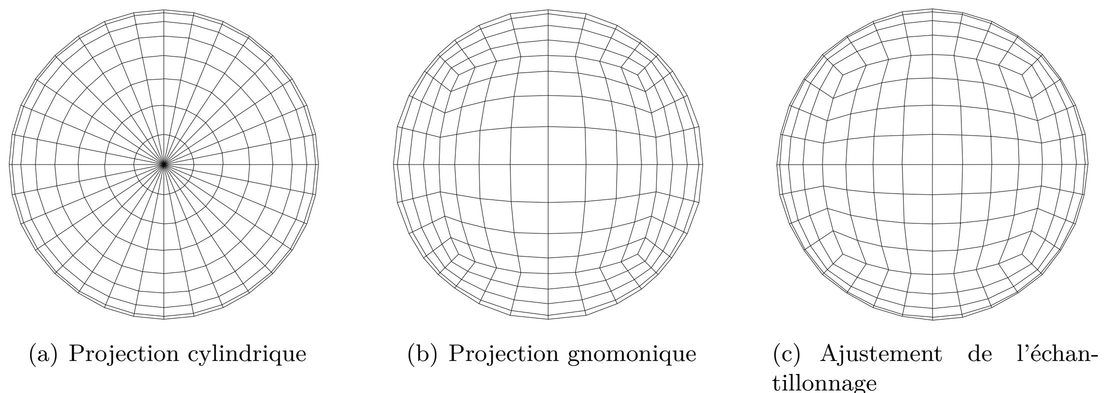
remarquons que l'utilisation de mipmaps pour le rendu de texture crée déjà le même supplément en mémoire graphique : l'utilisation de filtrage d'échantillons tire mieux parti de ce système. Enfin, pendant les opérations de découpe, nous découpons tous les LODs au lieu de ne le faire que pour le dernier.

Terrains planétaires

Notre solution générique pour le chargement et le rendu adaptatifs de terrains supporte des bases de données de n'importe quelle taille. Dans cette section, nous tirons parti de cette particularité pour supporter des terrains planétaires et nous présentons les techniques spécifiques aux planètes que nous ajoutons pour en effectuer le rendu.

Projection gnomonique sur un cube

Afin de pouvoir représenter un terrain planétaire avec des cartes d'échantillons, il est nécessaire d'effectuer une projection 2D d'une surface 3D. Les planètes sont généralement modelées à partir d'une sphère de référence puis projetées sur un rectangle avec une projection cylindrique. Cependant, ce type de projection implique d'importantes distorsions visuelles autour des pôles, comme nous pouvons le voir sur la Figure 24(a).



(a) Projection cylindrique (b) Projection gnomonique (c) Ajustement de l'échantillonnage

Figure 24 – Comparaison entre plusieurs projections : vues isométriques centrées sur le pôle nord de planètes reconstruites en 3D, avec les bords des blocs carrés subdivisant la carte 2D. Nous pouvons voir que les blocs autour du pôle deviennent très petits et étirés avec la projection cylindrique, bien qu'ils utilisent la même résolution que les autres. La projection gnomonique souffre moins de ces distorsions et disparités. L'échantillonnage ajusté du (c) est expliqué dans la section suivante.

Au lieu de cela, nous choisissons de projeter la planète sur les six faces d'un cube tangent, obtenant ainsi six cartes carrées. Nous utilisons la projection gnomonique [Weia] pour projeter les points depuis la surface de la sphère de référence sur la plus proche face du cube comme indiqué sur la Figure 25. Nous avons choisi cette projection car elle offre un échantillonnage plus uniforme qu'une projection cylindrique comme nous pouvons le voir sur la Figure 24. Ceci nous permet d'obtenir des bases de données 25%

plus petites et moins de distorsions lors du rendu. Cette projection a aussi l'avantage de permettre une reconstruction 3D rapide via une simple normalisation de vecteur 3D.

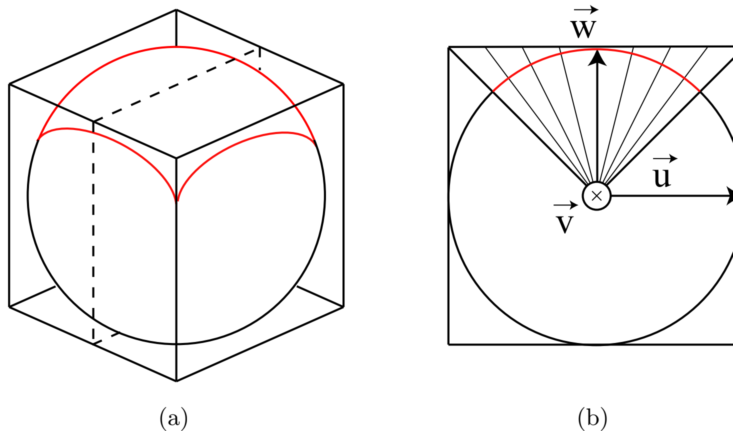


Figure 25 – Projection d'une planète sphérique sur un cube avec la projection gnomonique. **a)** Vue de côté du cube. La partie supérieure de la sphère (en rouge) est projetée sur la face supérieure du cube. **b)** Coupe 2D du cube le long du plan pointillé dans (a). La projection gnomonique projette les points depuis la surface de la sphère sur le plan de projection selon leur direction depuis le centre de la sphère.

Les valeurs d'élévation sont relatives à une surface de référence de la planète définie par un datum. Le datum définit un repère de coordonnées qui permet de manipuler des coordonnées sphériques tout en reconstruisant un modèle ellipsoïde plus précis de la planète. Pour obtenir le sommet 3D utilisé pour effectuer le rendu d'un échantillon, nous devons inverser la projection en gardant les proportions et la courbure de la planète. Pour cela, nous normalisons d'abord la direction du point de la surface correspondant depuis le centre de la planète. Nous appliquons ensuite le datum et ajoutons la valeur d'élévation de l'échantillon pour obtenir ses coordonnées 3D dans un repère dont les axes sont alignés avec les arêtes du cube.

Ajustement pour un échantillonnage plus uniforme

Comme nous pouvons le voir sur la Figure 24(b), la projection gnomonique sur un cube conserve des incohérences d'échantillonnage comme pour toute projection de planète. En particulier, l'aire couverte par un échantillon est environ 75% plus petite aux coins du cube qu'au centre de ses faces. Nous limitons ce problème en appliquant un simple ajustement à la projection.

Au lieu d'échantillonner régulièrement le plan de projection, nous choisissons d'échantillonner la carte directement au niveau de la surface de la sphère, comme présenté sur la Figure 26. Nous effectuons des rotations 2D indépendantes autour de deux axes principaux $\vec{u}, \vec{v}$ alignés avec les arêtes du cube, en utilisant un pas angulaire afin d'obtenir un échantillonnage 2D de la surface de la planète qui peut être projeté sur la face du

cube. Les échantillons sont distribués plus uniformément sur la surface comme nous pouvons le voir sur la Figure 24. L'aire couverte par un échantillon est seulement 33% plus petite aux coins du cube qu'au centre de projection.

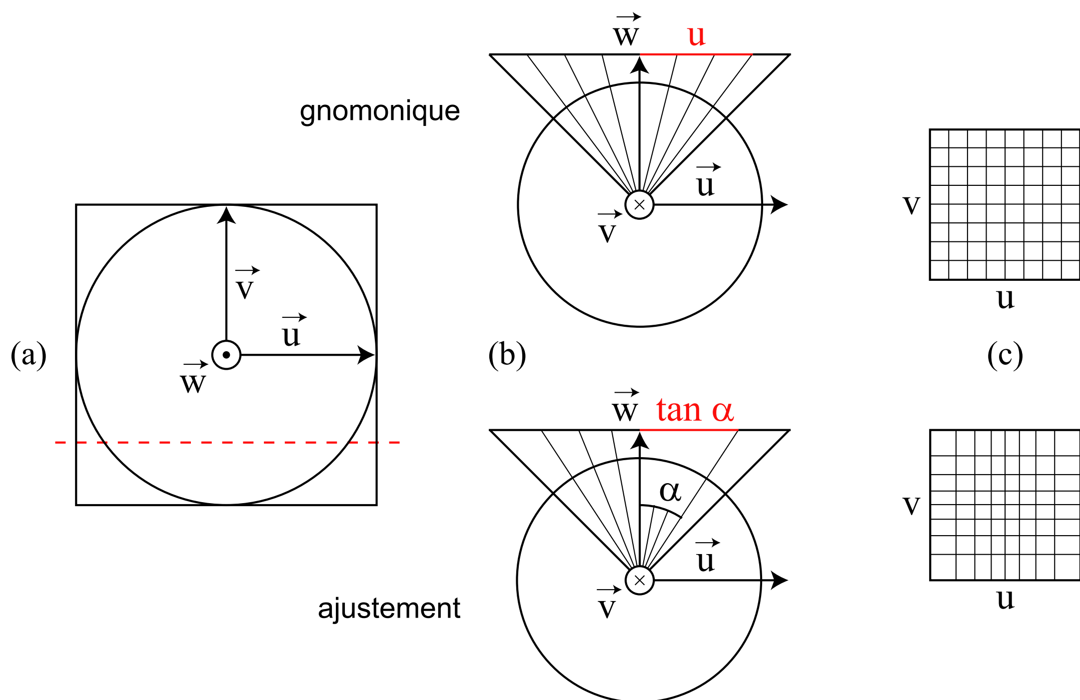


Figure 26 – Ajustement de l'échantillonnage pour la projection gnomonique. **a)** Vue de haut du cube. Nous utilisons les vecteurs $\vec{u}$ et $\vec{v}$ pour paramétrer la face supérieure. **b)** Coupes 2D du cube le long de la ligne rouge pointillée dans (a). Projection gnomonique : la ligne de projection est échantillonnée régulièrement par translation selon $\vec{u}$. Echantillonnage ajusté : la surface du cercle est régulièrement échantillonné par rotation autour de $\vec{v}$. **c)** Vues de haut de la face, avec les échantillons projetés. Projection gnomonique : la face est une carte d'échantillons régulière. Echantillonnage ajusté : les coordonnées d'un échantillon de la face sont les tangentes de ses coordonnées angulaires.

Lorsque nous projetons nos échantillons en utilisant la projection gnomonique, le plan de projection n'est pas échantillonné régulièrement. Cependant, en fonction d'un axe principal donné, la coordonnée 1D de l'échantillon projeté est la tangente de l'angle de rotation. Nous pouvons donc stocker nos échantillons dans une carte régulière, puis utiliser les tangentes de leurs coordonnées angulaires pour les convertir sur le plan de projection gnomonique lors de la reconstruction 3D.

Correction des fissures aux arêtes du cube

Les faces du cube que nous utilisons pour projeter la planète sont des cartes d'échantillons différentes, mais elles représentent le même terrain 3D. Nous devons donc correc-

tement recoller les arêtes de ces faces pour éviter l'apparition de fissures lors du rendu en 3D. Nous gérons ce cas particulier de façon à ce que la solution pour corriger les fissures entre les blocs proposées précédemment puisse s'appliquer implicitement.

Nous numérotons et orientons les faces du cube comme indiqué sur la Figure 27 afin de faciliter la gestion des faces voisines. En particulier, les axes principaux le long de l'arête commune de deux cartes adjacentes ont le même sens, ce qui assure que les blocs le long de cette arête ont le même ordre des deux côtés. Par contre, deux cartes adjacentes ne peuvent pas toujours avoir le même axe le long de leur arête commune. Nous utilisons alors des tables de correspondances prédéfinies pour, en fonction d'une face et d'une arête, identifier directement la face voisine et son arête concernée.

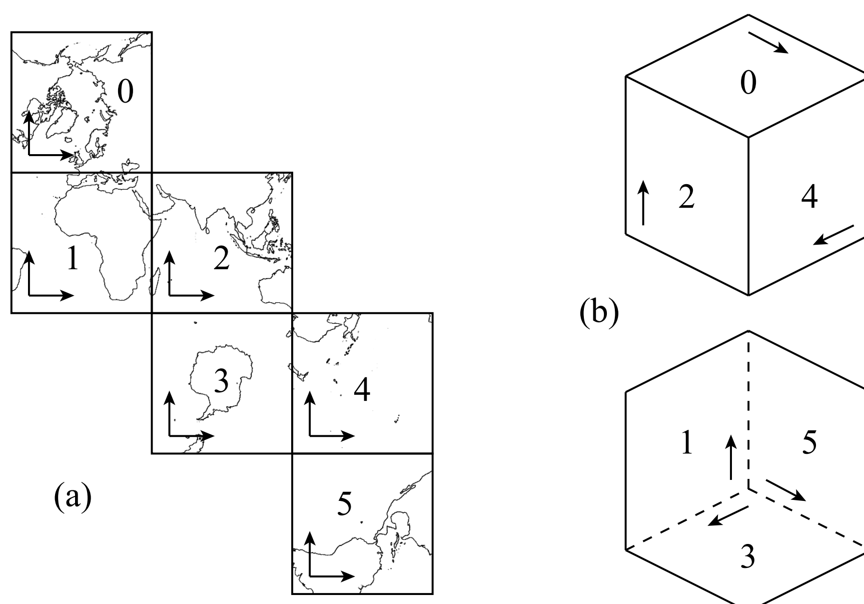


Figure 27 – Orientation et numérotation des faces du cube. **a)** Patron du cube, avec les continents de la Terre. Axes horizontaux : $\vec{u}$, axes verticaux : $\vec{v}$. **b)** Vue incidente du cube (en haut : faces visibles, en bas : faces cachées). Les axes $\vec{v}$ sont représentés le long de l'arête « gauche » de la face (u minimum).

Plans limites de profondeur adaptatifs

Le matériel de rendu 3D utilise des plans minimum (*near*) et maximum (far) pour limiter la profondeur de sa la pyramide de rendu et donc de la géométrie traitée. Nous prenons également en compte ces plans lors du calcul de la visibilité des blocs. Ces plans sont définis par leur distance par rapport au point de vue. Dans le cas d'une planète, nous souhaitons que la distance far soit suffisamment importante pour assurer qu'aucune partie visible de la planète n'est ignorée. Cependant, si nous utilisons une différence trop importante entre *near* et far la précision du rendu diminue, ce qui cause des problèmes de scintillement à cause de la précision limitée du tampon de profondeur

matériel. De plus, en plaçant le plan far derrière la planète, la face cachée de la planète est inutilement traitée car incluse dans la pyramide de rendu. Pour éviter ces problèmes, nous adaptons les deux plans de profondeur en fonction de la position du point de vue. Des exemples de résultats sont présentés sur la Figure 29.

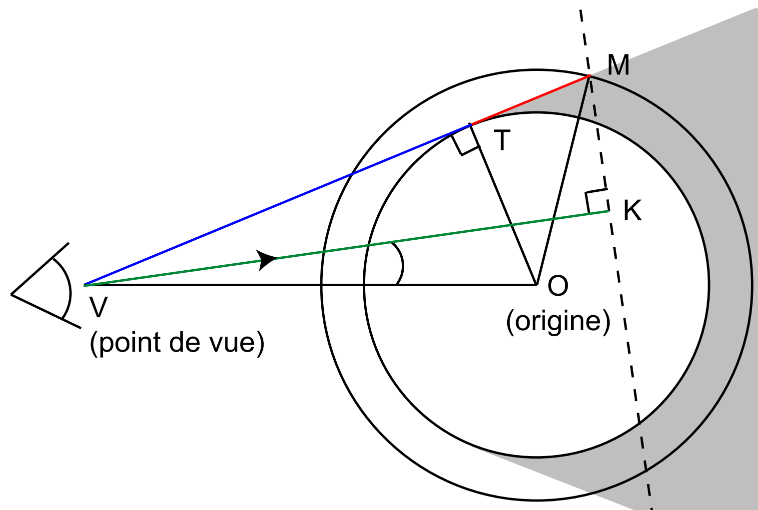


Figure 28 – Calcul de l'horizon : nous cherchons la distance $\|VK\|$ (en vert) du plan far par rapport au point de vue V. La ligne pointillée KM est la projection 2D de ce plan : tout ce qui se trouve plus loin n'est pas affiché. La partie grisée est cachée par la planète elle-même. Nous obtenons $\overrightarrow{VM}$ en fonction de $\|OT\|$ et $\|OM\|$ les rayons minimum et maximum de la planète et V la position du point de vue, puis le projetons sur le vecteur direction du point de vue pour obtenir $\|VK\|$.

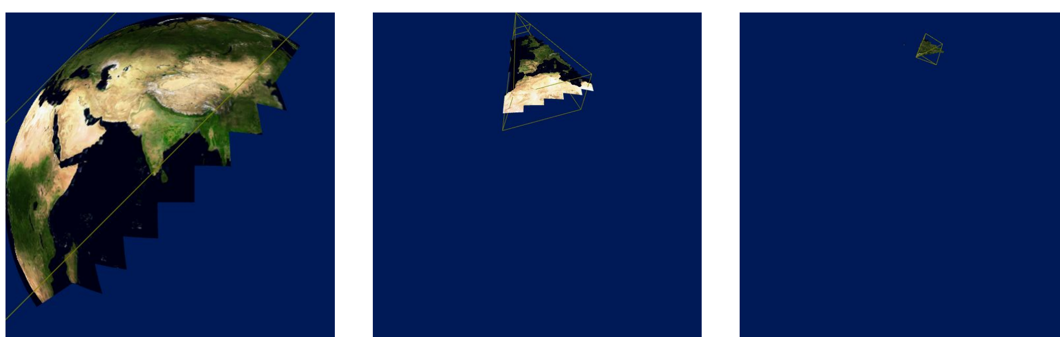
Nous plaçons la distance far à l'horizon ; son calcul est résumé sur la Figure 28. Avec cette méthode, la surface de la planète traitée lors du rendu rétrécit lorsque le point de vue s'en rapproche. Nous pouvons alors utiliser plus de données pour les parties visibles et importantes. Nous calculons la distance *near* bien plus simplement : nous utilisons la distance minimum entre le point de vue et la planète $\|VO\| - \|OM\|$, que nous seuillons à une valeur minimum arbitraire.

Résultats

Nous avons testé les méthodes proposées avec plusieurs bases de données. Après avoir construit les fichiers sur le serveur, nous avons connecté le même ordinateur client que dans la partie précédente et nous avons testé les performances de rendu. Plusieurs captures d'écran sont visibles dans la Figure 32. Afin de pouvoir directement comparer la vitesse de rendu dans différentes conditions, nous avons demandé un nombre donné de 2 millions de triangles à la solution générique au lieu d'un nombre d'images par secondes. Nous avons obtenu les résultats présentés dans la Table 1.

Nous pouvons d'abord voir qu'utiliser des blocs de plus haute résolution offre un rendu plus rapide puisque pour un nombre donné d'échantillons, moins de blocs sont

(a) Rendus 3D de la Terre depuis plusieurs distances différentes



(b) Surfaces affichées correspondantes, à l'échelle de la planète

Figure 29 – Plans limites de profondeur adaptatifs améliorant la précision du rendu. Les parties de la planète situées devant le point de vue mais derrière l'horizon ne sont pas traitées lors du rendu, sans avoir besoin d'effectuer de test supplémentaire.

traités et moins de lots de géométrie sont affichés. L'utilisation de cartes de texture a un impact sur le temps processeur (CPU) puisque cela multiplie le nombre de blocs à traiter. Les cartes de texture ont aussi un impact sur la vitesse de rendu d'environ 15%. A l'inverse, la correction des fissures a un impact très réduit sur la vitesse de rendu.

Nous avons également testé la vitesse des opérations de raffinement. Avec filtrage d'échantillons, la création d'un LOD de résolution $blockWidth = 168$ prend 0.55ms comparé à 0.45ms sans filtrage, soit le coût de copier 25% d'échantillons supplémentaires. Avec notre ajustement de l'échantillonnage de planètes, la reconstruction des sommets 3D à partir des valeurs d'élévation du second LOD d'un bloc de résolution $blockWidth = 52$ prend 0.44ms, comparé à 0.25ms avec la projection gnomonique standard et 0.12ms dans le cas de terrain non planétaire. Bien que cette augmentation soit conséquente, la durée reste négligeable comparé à la latence du réseau.

Enfin, nous avons testé la solution sur un ordinateur peu performant. A 30 images par secondes pour afficher 100 000 triangles, le temps processeur pour chaque image est de 0.3ms soit moins de 1% du temps total. Ceci s'explique car sur ce système, le matériel de rendu 3D est particulièrement lent et c'est donc lui qui limite les performances. Le coût issu de l'utilisation de cartes de textures et de la correction des fissures est comparable en proportion à ce que nous avons obtenu précédemment.

Base de données	*blockWidth*	**IPS**	**CPU par image**		**Avec fissures**	
			Durée	Proportion	IPS	Impact
Puget Sound	64	133	0.9ms	12%	141	+6%
+ texture	64, 256	113	1.05ms	12%	121	+7%
BMNG	84	164	0.45ms	7%	161	-2%
+ texture	84, 168	141	0.8ms	11%	135	-4%
SRTM	104	185	0.2ms	4%	172	-7%
+ TrueMarble	104, 168	162	0.35ms	6%	160	-1%

Table 1 – Résultats de rendu 3D temps réel sur un client performant à environ 2 millions de triangles par image, avec et sans textures et corrections des fissures. La valeur de CPU par image est le temps utilisé par nos méthodes logicielles de sélection de données à chaque image ; le reste étant utilisé par le rendu 3D matériel. La dernière colonne donne la vitesse de rendu obtenue sans correction des fissures, en utilisant exactement les même données qu'avec. Toutes les bases de données utilisent $subDepth = 1$ (deux LODs par bloc) pour obtenir des temps de raffinement des blocs comparables.

Conclusion

Dans cette partie, nous avons tout d'abord étendu la solution générique de chargement et sélection adaptatifs de données afin de l'appliquer au contexte du rendu matériel 3D de terrains. Nous avons ensuite corrigé les fissures apparaissant entre des blocs adjacents utilisant différents niveaux de détails, en adaptant une solution existante au contexte du chargement progressif de données. Nous avons également étendu la solution générique afin de supporter le rendu de terrains texturés et avons ajouté la possibilité d'améliorer la qualité du rendu grâce au filtrage d'échantillons.

Ensuite, nous avons représenté des terrains planétaires en les projetant sur un cube, via une version ajustée de la projection gnomonique qui permet d'obtenir un échantillonnage plus uniforme de la surface de la planète qu'avec des représentations classiques. Ceci évite de transférer et traiter des données redondantes et évite des incohérences de rendu autour des pôles. Enfin, nous avons adapté en temps réel les plans limites de la profondeur du rendu matériel afin d'améliorer la précision de rendu et d'éliminer la plupart des parties de la planète situées derrière l'horizon.

Pré-calcul de fichiers serveur à partir d'immenses cartes

Dans cette partie, nous décrivons les deux étapes de pré-calcul utilisées pour obtenir une base de données serveur utilisable par notre solution générique à partir de n'importe quelle base de données d'entrée. Les étapes sont présentées dans la Figure 30.

Les algorithmes utilisés dans ces deux étapes de pré-calcul sont conçus afin de supporter des cartes de taille quelconque. Ils consistent à subdiviser ces cartes en parties plus petites qui peuvent être chargées entièrement en mémoire. Pour simplement les entrées/sorties sur d'immenses cartes, nous avons développé un outil dédié. Cet outil

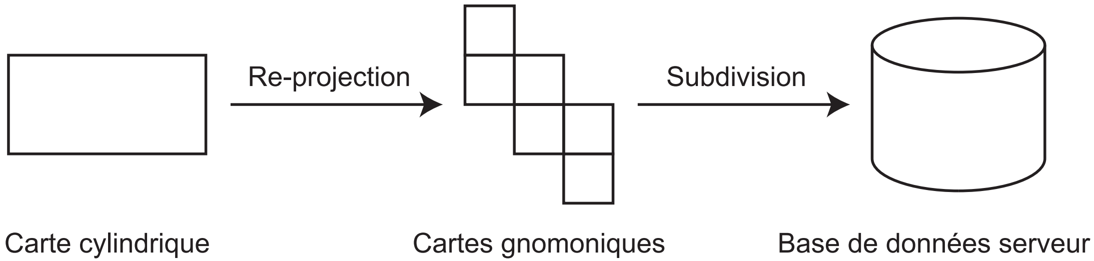


Figure 30 – Etapes de pré-calcul. L'étape de re-projection, utilisée seulement pour les bases de données planétaires, convertit une carte planétaire cylindrique en six cartes que nous pouvons utiliser dans notre solution. L'étape de subdivision génère un fichier de base de données qui peut être utilisé par l'application serveur de notre solution à partir d'une carte non planétaire ou bien de six cartes obtenues via l'étape de re-projection.

permet d'utiliser des cartes constituées d'un nombre quelconque de tuiles rectangulaires, et permet de charger en mémoire seulement un sous-ensemble rectangulaire donné de la carte, potentiellement à cheval sur les bords des tuiles.

Re-projection de carte planétaire

Dans cette section, nous expliquons comment calculer les six cartes carrées utilisées dans notre solution de rendu de planètes en re-projetant les échantillons d'une carte planétaire qui utilise une projection cylindrique. Nous choisissons la résolution des cartes créées afin d'obtenir la même définition que la carte d'entrée à l'équateur. Ceci permet de gagner 33% de données principalement redondantes tout en conservant la même définition générale que la carte d'entrée.

Re-projection d'un échantillon

Pour chaque échantillon de chaque carte générée, nous calculons ses coordonnées re-projetées dans la carte d'entrée pour obtenir la valeur de l'échantillon correspondant. Nous utilisons les Equations 2 (issues de [Weia] et simplifiées) pour convertir un point de la surface d'une planète depuis ses coordonnées gnomoniques $u, v \in [-1, 1]$ vers ses coordonnées cylindriques $\phi' \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ et $\lambda' \in [-\pi, \pi]$ (latitude et longitude) :

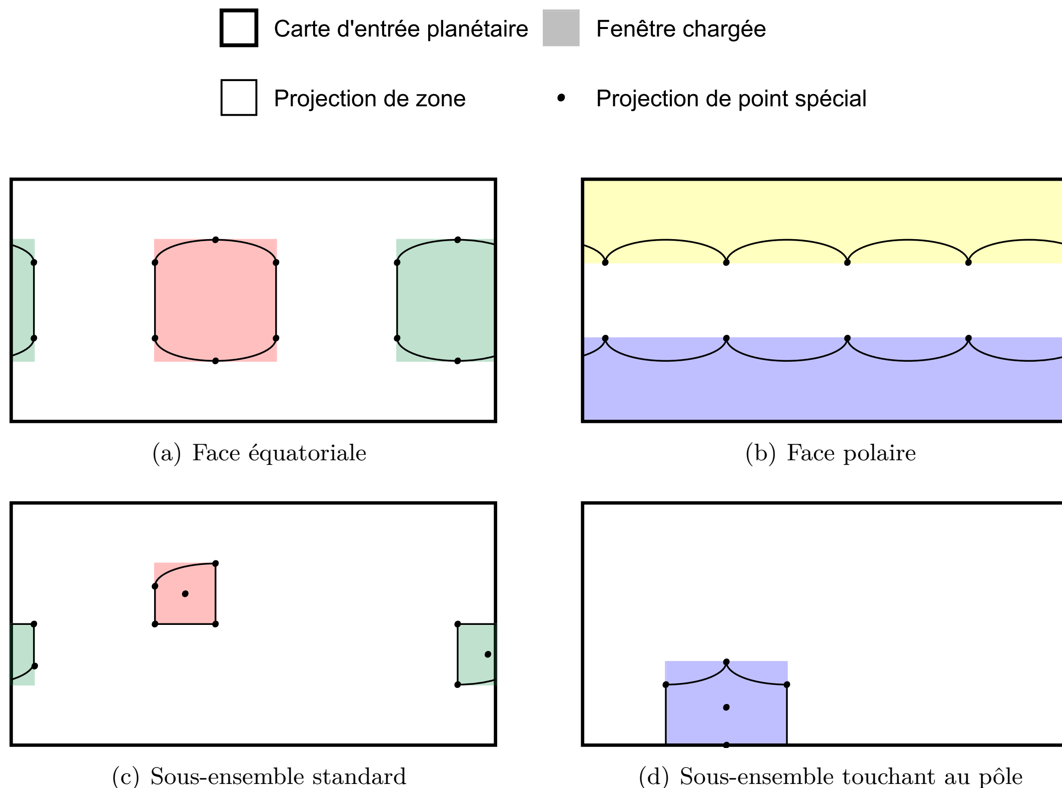
$$\begin{aligned} \lambda' &= \arctan\left(\frac{u}{\cos\phi - v\sin\phi}\right) + \lambda \\ \phi' &= \arcsin\left(\frac{\sin\phi + v\cos\phi}{\sqrt{u^2+v^2+1}}\right) \end{aligned} \quad (2)$$

où ϕ, λ sont les coordonnées dans la carte d'entrée du centre de projection utilisé pour la projection gnomonique, autrement dit le centre de la face correspondante du cube. Pour appliquer notre ajustement de l'échantillonnage, nous utilisons $u = \tan u'$ et $v = \tan v'$ où $u', v' \in [\frac{-\pi}{4}, \frac{\pi}{4}]$ sont les coordonnées angulaires du point.

Re-projection de la carte d'une face du cube

Nous re-projetons les six cartes correspondant aux six faces du cube indépendamment avec le même algorithme. Cet algorithme consiste à subdiviser récursivement et uniformément la carte gnomonique en quatre jusqu'à ce nous puissions charger un de ces sous-ensembles de la carte entièrement en mémoire. Nous re-projetons alors chaque échantillon de cette partie de la carte pour en obtenir la valeur. Cet algorithme assure que nous ne chargeons jamais de sous-ensemble de la carte dont la place mémoire dépasse une valeur maximum prédéfinie. Cependant, cet algorithme nécessite de pouvoir calculer les coordonnées de la partie rectangulaire de la carte d'entrée qui contient tous les échantillons re-projetés de la carte générée.

Pour simplifier les calculs et éviter de devoir charger de très grandes parties de la carte autour des pôles de la planète (autrement dit les bordures supérieure et inférieure de la carte d'entrée), nous forçons les faces du haut et du bas du cube à utiliser les pôles nord et sud comme centres de projection. Dans ces conditions, nous pouvons re-projeter des points spécifiquement choisis afin d'obtenir la partie englobante correspondante dans la carte d'entrée. Les différents cas possibles sont présentés dans la Figure 31.



(a) Face équatoriale

(b) Face polaire

(c) Sous-ensemble standard

(d) Sous-ensemble touchant au pôle

Figure 31 – Points spéciaux de la carte générée devant être re-projetés pour obtenir la partie rectangulaire englobante de la carte d'entrée.

Génération de la base de données du serveur

L'étape finale de pré-calcul consiste à créer le fichier de base de données du serveur à partir d'une ou plusieurs cartes d'échantillons carrées, potentiellement issues de l'étape de re-projection. Cette étape effectue une subdivision uniforme de chaque carte et un dés-entrelacement de ses échantillons, sans en changer le nombre.

Lors de la construction d'un LOD, l'organisation du fichier présentée précédemment nécessite de connaître la position et la taille du ou des LOD suivants, qu'ils appartiennent au même bloc ou à ses fils. Ceci est adapté pour une génération de l'arbre commençant par ses plus bas niveaux. Nous subdivisions d'abord récursivement la carte d'échantillons en blocs en effectuant un parcours en profondeur de l'arbre virtuel définit par les paramètres *nbSubs* et *subDepth*. Quand nous atteignons une feuille de l'arbre, nous obtenons la grille d'échantillons correspondante depuis la carte d'entrée. Nous pouvons ensuite calculer les LODs du bloc et les écrire dans le fichier puis transmettre les échantillons de son premier LOD à son père, récursivement afin de construire l'arbre de bas en haut. D'autre part, nous chargeons les données de la carte d'entrée correspondant à un bloc dès que celui-ci est suffisamment petit pour que puissions loger toutes ces données en mémoire, avant de subdiviser le bloc récursivement. Ceci évite d'effectuer un chargement de données pour chaque feuille de l'arbre.

Résultats

Nous avons testé l'étape de re-projection avec plusieurs cartes planétaires, puis nous avons utilisé la seconde étape de pré-calcul pour générer les fichiers serveur correspondants. Nous avons effectué les tests sur un ordinateur équipé d'un processeur Xeon 5160 3GHz avec 3Go de mémoire vive. Les fichiers d'entrée et de sortie se situent tous sur un miroir RAID de deux disques durs rapides. Les résultats sont présentés sur la Table 2.

Carte d'entrée			**Re-projection**		**Création fichier**	
Nom	Type	Taille	Durée	Taille	Temps	Durée
Puget Sound	élévation	512M	N/A	N/A	50s	280M
	couleur	768M			1m25	458M
SRTM	élévation	174G	8h55	126G	4h50	14.9G
TrueMarble	couleur	41.7G	1h35	31.0G	36m35	6.84G
BMNG	élévation	6.95G	27m20	5.16G	6m45	879M
	couleur	10.4G	33m40	7.75G	9m25	1.89G

Table 2 – Résultats des deux étapes de pré-calcul.

Comme nous pouvons le voir, les deux étapes de pré-calcul peuvent traiter d'immenses cartes. Les bases de données re-projetées sont environ 25% plus petites que les cartes d'entrée grâce à notre projection qui élimine la plupart des données redondantes. De plus, les fichiers serveur sont encore plus petits car les LODs sont compressés.

Conclusion

Chargement et sélection adaptatifs de données génériques

Dans cette thèse, nous avons d'abord proposé une solution générique pour le chargement progressif et le rendu adaptatif de cartes d'échantillons 2D de taille quelconque. Nos méthodes s'adaptent aux vitesses de chargement et de rendu et ne dépendent pas de la taille de la base de données ni de son contenu. Nous pouvons utiliser une unique base de données sur un unique serveur avec n'importe quel type de client.

Nous avons basé cette solution sur une structure de données générique, qui mélange deux solutions avec de bonnes propriétés et ajoute de nouvelles méthodes pour améliorer l'efficacité du chargement et du rendu. Nous avons utilisé cette structure de données depuis le disque dur d'un serveur jusqu'au système de rendu d'un client, en nous concentrant sur la vitesse d'exécution. En particulier, nous avons évité de charger des données non pertinentes en assurant qu'elles ne sont pas redondantes entre les chargements successifs et en envoyant toujours les requêtes les plus importantes en premier. Nous avons aussi évité autant que possible les modifications de la structure de données chez le client, en privilégiant des mises à jour effectuées sur place de manière asynchrone et l'extraction des échantillons via un système de masques.

Nous avons utilisé une mesure d'importance générique pour guider le chargement et la sélections adaptatifs des niveaux de détail ainsi que pour déclencher les opérations de mise à jour de la base de données du client. Cette mesure dépend d'un facteur de qualité qui permet à la solution de s'adapter à une vitesse ou une qualité de rendu choisie par l'utilisateur. D'autres facteurs peuvent être modifiés ou ajoutés lors de l'application de la solution à un type particulier de données, afin de refléter sa nature et d'améliorer les capacités d'adaptation de la solution.

Du côté du serveur, nous avons utilisé une organisation particulière du fichier de base de données qui assure qu'un unique accès à ce fichier est suffisant pour répondre à une requête de chargement de données. Nous avons décrit comment générer ce fichier sur le serveur au travers d'une étape de pré-calcul qui supporte des cartes d'échantillons de taille quelconque.

Rendu 3D de vastes terrains et planètes

Nous avons dans un second temps appliqué la solution générique au rendu 3D de vastes terrains. Nous avons spécialisé la mesure d'importance et avons tiré parti du matériel de rendu en étendant le concept de masques d'échantillons via des « triangle strips » et en stockant les niveaux de détail géométriques sur la mémoire graphique.

Nous avons corrigé les fissures apparaissant entre les parties de terrain adjacentes qui utilisent différents niveaux de détail, en améliorant une solution existante pour l'adapter au contexte du chargement progressif de données. Contrairement aux solutions existantes, nous avons également permis à ces parties adjacentes d'utiliser n'importe quelle définition tant que ceci ne crée pas de fissures impossibles à corriger.

Nous avons également supporté le rendu de terrains texturés en appliquant la solution générique aux cartes de textures. Nous avons amélioré la qualité du rendu des cartes

de textures en étendant la solution générique afin de tirer parti du mipmapping matériel et de pouvoir proposer un filtrage d'échantillons au lieu du sous-échantillonnage régulier sous-entendu par notre structure de données non redondante.

Dans une autre partie, nous avons projeté des planètes en 3D sur un cube, en utilisant une version ajustée de la projection gnomonique afin d'obtenir un échantillonnage de la surface bien plus uniforme qu'avec des représentations classiques. Ceci évite de stocker, transférer et afficher des données redondantes et évite des incohérences du rendu autour des pôles. Comparé à la projection gnomonique simple, notre ajustement permet d'obtenir une meilleure distribution des échantillons sur le plan de projection, impliquant donc moins d'inégalités au niveau des aires couvertes par les échantillons sur toute la carte. La plupart des bases de données disponibles étant distribuées sous la forme de cartes utilisant une projection cylindrique, nous avons développé une étape de re-projection pour le pré-calcul de nos bases de données, que nous avons conçue pour supporter des cartes de taille quelconque.

Enfin, nous avons mis à jour les plans de la pyramide de rendu matérielle de façon adaptative afin d'améliorer la précision de rendu des planètes et d'éliminer automatiquement la plupart de la surface de la planète située derrière l'horizon.

Perspectives

Notre solution générique pourrait être étendue pour améliorer sa réactivité et ses capacités d'adaptation. Par exemple, il est possible d'utiliser plus de niveaux de détail par bloc sans nécessiter beaucoup de changements. D'autre part, il serait intéressant d'implémenter un système de sauvegarde explicite des données de blocs fusionnés pour éviter d'avoir à les charger à nouveau plus tard.

D'autres améliorations pourraient être apportées pour augmenter la vitesse et/ou la précision du rendu 3D de terrains, par exemple en effectuant la reconstruction 3D des échantillons d'élévation avec le processeur graphique, ou en évitant les calculs en virgule flottante excepté pour le rendu final.

Les travaux futurs pourraient aussi ajouter de nouvelles fonctionnalités à notre solution de rendu 3D de terrains, telles que la génération procédurale de détails, la réduction des artefacts visuels issus de la mise à jour subite de grandes parties de géométrie, le support de meilleures méthodes de compression et/ou de filtrage d'échantillons, etc.

De futurs travaux pourraient enfin ouvrir de nouveaux sujets de recherches en se basant sur notre solution de chargement et rendu de terrains, en particulier en tirant parti de la structuration hiérarchique de la base de données et de l'échantillonnage régulier des niveaux de détail. Par exemple, dans une application de rendu 3D, ceci pourrait simplifier la détection de collisions, le calcul des ombres portées par le terrain sur lui-même à partir de la lumière du soleil, et le positionnement automatique de routes ou frontières 2D sur la surface 3D du terrain.

Nous pensons aussi qu'il serait intéressant d'utiliser de nouvelles méthodes de rendu 3D, telles que la grille de projection, tout en utilisant notre solution générique pour gérer le chargement et la sélection des données.

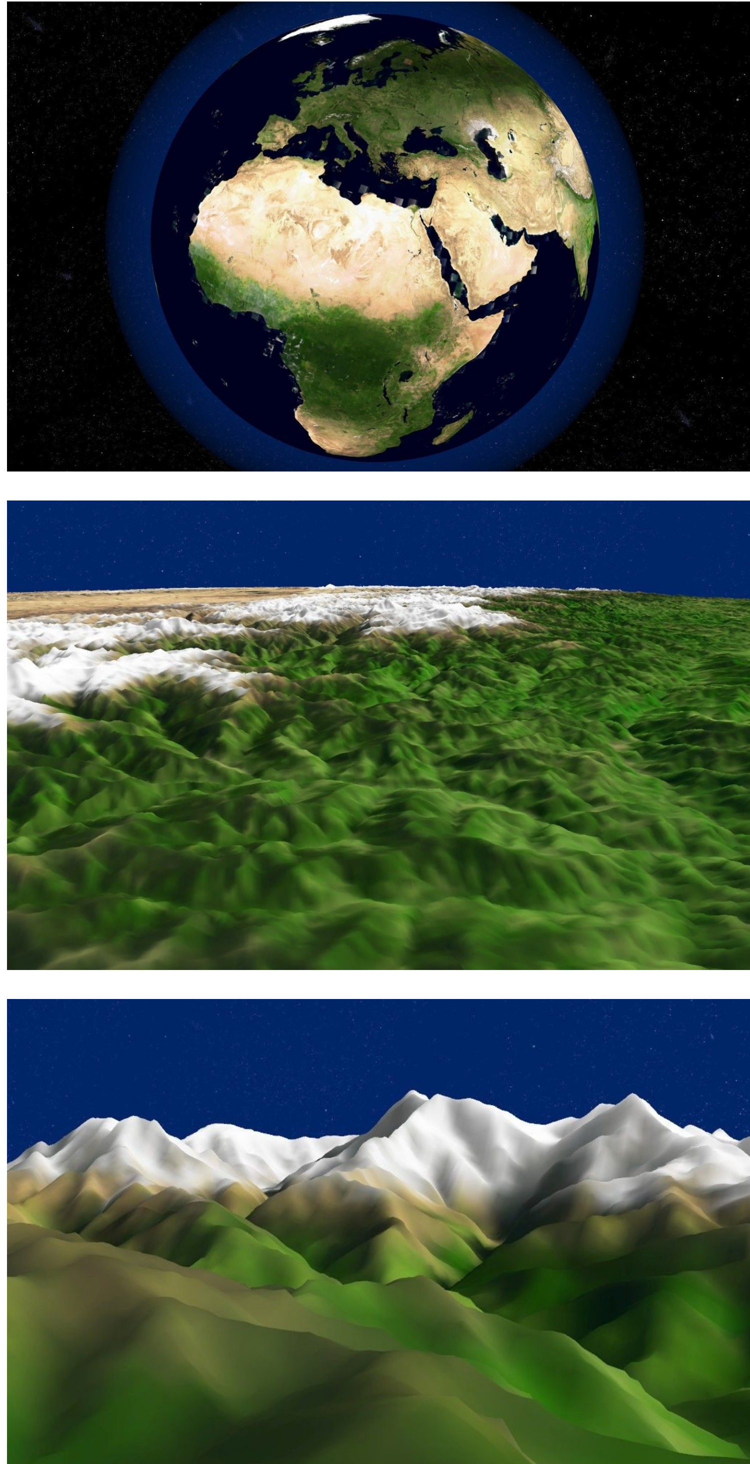


Figure 32 – Captures d'écran prises lors d'une session de rendu 3D temps réel avec notre application, utilisant les bases de données terrestres SRTM et TrueMarble.

Abstract

In this thesis, we propose solutions to perform adaptive streaming and rendering of arbitrary large terrains. One interesting application is to interactively visualize the Earth in 3D on a computer while loading the required data from a huge database over a network.

In the first part of this thesis, we introduce a generic solution to handle sample maps of any size from a server hard disk all the way to a client rendering system. Our methods adapt to the speed of the network and of the rendering and avoid handling redundant data.

In a second part, we build upon the generic solution to propose real-time 3D rendering of large textured terrains on graphics hardware. In addition, we support planetary terrains and use techniques that prevent handling redundant data and limit typical rendering inconsistencies due to map projection and rendering precision.

Finally, we propose preprocessing algorithms that allow to build server databases from huge sample maps.

Résumé

Dans cette thèse, nous proposons des solutions pour le chargement progressif et le rendu adaptatif de vastes terrains. Cela peut servir notamment à visualiser la Terre en 3D sur un ordinateur en chargeant les données depuis une immense base de données via un réseau.

Dans la première partie de cette thèse, nous introduisons une solution générique pour manipuler des cartes d'échantillons de taille quelconque depuis un serveur jusqu'à un système de rendu client. Nos méthodes s'adaptent aux performances du réseau et du rendu et évitent de traiter des données redondantes.

Dans une deuxième partie, nous utilisons cette solution pour permettre le rendu 3D temps réel de vastes terrains texturés. De plus, nous supportons des terrains planétaires et réduisons les incohérences visuelles dues à la projection cartographique et à la précision du rendu.

Enfin, nous proposons des algorithmes permettant de créer des bases de données serveur à partir d'immenses cartes d'échantillons.