Università degli Studi di Pisa

Ph.D. Thesis

Multiresolution Techniques for Real–Time Visualization of Urban Environments and Terrains

Marco Di Benedetto

Supervisor	Supervisor
Dr. Paolo Cignoni	Dr. Fabio Ganovelli

Referee	Referee
Prof. Valerio Pascucci	Prof. Pere Brunet Crosa

June 13, 2011

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org diglib.eg.org

a mia madre

Abstract

In recent times we are witnessing a steep increase in the availability of data coming from real–life environments. Nowadays, virtually everyone connected to the Internet may have instant access to a tremendous amount of data coming from satellite elevation maps, airborne time-of-flight scanners and digital cameras, street–level photographs and even cadastral maps. As for other, more traditional types of media such as pictures and videos, users of digital exploration softwares expect commodity hardware to exhibit good performance for interactive purposes, regardless of the dataset size.

In this thesis we propose novel solutions to the problem of rendering large terrain and urban models on commodity platforms, both for local and remote exploration. Our solutions build on the concept of multiresolution representation, where alternative representations of the same data with different accuracy are used to selectively distribute the computational power, and consequently the visual accuracy, where it is more needed on the base of the user's point of view.

In particular, we will introduce an efficient multiresolution data compression technique for planar and spherical surfaces applied to terrain datasets which is able to handle huge amount of information at a planetary scale. We will also describe a novel data structure for compact storage and rendering of urban entities such as buildings to allow real–time exploration of cityscapes from a remote online repository. Moreover, we will show how recent technologies can be exploited to transparently integrate virtual exploration and general computer graphics techniques with web applications.

Acknowledgments

This has been one of the most intense experiences of my life. Joy in successes and letdown in failures have deeply influenced my life, as well as the excitation in thinking in a way that was very unusual for me.
The Visual Computing Laboratory, ISTI - CNR, where I work, is one of the best places I know, both from a working and a human point of view. I thank Paolo Cignoni and Fabio Ganovelli, my advisors, for the great, precious conversations we had and their continuous support before and during my Ph.D., Roberto Scopigno, the head of the lab, for believing in me, and all my irreplaceable colleagues. Thanks goes to Prof. Giuseppe Attardi and Dr. Antonio Cisternino for their work in my Ph.D. committee, and to my referees Prof. Valerio Pascussi and Prof. Pere Brunet Crosa for their reviews and advices for enhancing this thesis.
I thank my family for supporting me in all my decisions, my best friends Marco Fichera, Roberto Guanciale, Gianluigi "Sherpya" Tiesi and all the others that I don't mention because I'm sure I would forget some.

My final thanks goes to my beloved Irene, a special person who had to endure me in this work and whose smile makes me wish an endless life.

Contents

1 Introduction **1**
- 1.1 Motivations and Contribution 1
- 1.2 Thesis Structure 5

2 Background and Related Work **7**
- 2.1 Representing Three–Dimensional Objects 7
- 2.2 The Graphics Hardware 9
  - 2.2.1 The Graphics Pipeline 9
  - 2.2.2 Evolution of Graphics Hardware 10
- 2.3 Adaptive Multiresolution Rendering 12
  - 2.3.1 Visual and Temporal Fidelity 12
  - 2.3.2 Geometric Level-Of-Detail 13
- 2.4 Multiresolution in Terrain Rendering 16
- 2.5 Texture-Based Accelerated Rendering 18
  - 2.5.1 Billboards 19
  - 2.5.2 Portal Textures 22
  - 2.5.3 Textured Depth Meshes 23
  - 2.5.4 Layered Depth Images 26
  - 2.5.5 Multi-Layered Impostors 27
- 2.6 Adding Visual Detail to Coarse Geometric Representations 28
  - 2.6.1 Normal Mapping 29
  - 2.6.2 Parallax Mapping 31
  - 2.6.3 Relief Mapping 31
- 2.7 Texture Management and Multiresolution in Urban Models 35
- 2.8 Deploying 3D Graphics to the World Wide Web 37

3 Data Compression and Rendering for Terrain Models **39**
- 3.1 Terrain Models Rendering 39
- 3.2 Batched Dynamic Adaptive Meshes 41
- 3.3 Compressing BDAM 43
- 3.4 Out-of-Core Construction of Massive Variable Resolution Terrain Datasets 47
  - 3.4.1 Compressed Data Representation. 49

3.5 Rendering a C-BDAM . . . 49
3.5.1 Data Access and Cut Selection . . . 49
3.5.2 Rendering: Elevation, Colors, and Normals . . . 52
3.6 Implementation and Results . . . 53
3.6.1 Preprocessing . . . 55
3.6.2 Adaptive Rendering . . . 56

4 A Multiresolution Representation for Urban Models 59
4.1 Exploring Urban Areas . . . 59
4.2 The BlockMap Representation . . . 61
4.2.1 Data Structure . . . 62
4.2.2 Rendering a BlockMap . . . 65
4.3 Construction of a Multiresolution Representation for Urban Environments . . . 67
4.3.1 Hierarchy Construction . . . 67
4.3.2 BlockMap Generation . . . 70
4.4 Rendering Urban Models with BlockMaps . . . 74
4.5 Implementation and Results . . . 77
4.5.1 Preprocessing . . . 78
4.5.2 Streaming and Rendering . . . 78

5 Computer Graphics for the Web Platform 83
5.1 Computer Graphics and the World Wide Web . . . 83
5.2 The SpiderGL Graphics Library . . . 86
5.2.1 Library Architecture. . . . 86
5.3 Using SpiderGL . . . 98
5.3.1 WebGL Limitations and Standard CG Algorithms . . . 98
5.4 Large Datasets on the Web . . . 99
5.4.1 Terrain Models . . . 101
5.4.2 Urban Models with BlockMaps . . . 103
5.4.3 Polynomial Texture Maps . . . 104
5.5 Biological Data Visualization . . . 106
5.5.1 Visualizing Molecular Properties . . . 106
5.6 MeShade: deploying 3D content on the Web . . . 110

6 Conclusions 113
6.1 Advancements in Multiresolution Techniques . . . 113
6.2 Future Work . . . 115
6.3 List of Publications . . . 116

Bibliography 117

List of Figures

2.1 The Graphics Pipeline . . . 9
2.2 Adaptive Multiresolution Rendering Pipeline . . . 14
2.3 Multiresolution Rendering Example . . . 16
2.4 Semi-Regular Adaptive Triangulation . . . 17
2.5 Geometry Clipmaps . . . 18
2.6 Billboard Rotation . . . 19
2.7 Bounding Box Impostors . . . 20
2.8 Impostor Error . . . 21
2.9 Nailborards . . . 21
2.10 Portal Textures Smooth Warping . . . 22
2.11 Portal Textures Continuity Preserving . . . 24
2.12 Textured Depth Meshes . . . 25
2.13 Layered Depth Images . . . 26
2.14 LDI Memory Layout . . . 27
2.15 Multi-Layered Impostors . . . 28
2.16 Normal Map Construction . . . 30
2.17 Normal Map Rendering . . . 30
2.18 Parallax Mapping Offset . . . 32
2.19 Parallax Mapping Rendering . . . 32
2.20 Cube Relief Maps . . . 33
2.21 Relief Map Warping . . . 33
2.22 Relief Mapping Rendering . . . 34
2.23 Relief Mapping Ray-Surface Intersection . . . 34
2.24 Relief Mapping of Planar Surfaces . . . 35
2.25 Relief Mapping of Arbitrary Surfaces . . . 35
2.26 Omni-Directional Impostors . . . 36
2.27 Multi-Level Relief Impostors . . . 37

3.1 CBDAM Earth View . . . 40
3.2 BDAM Hierarchy . . . 42
3.3 BDAM Subdivision . . . 43
3.4 BDAM on a regular grid . . . 44
3.5 Neville filters . . . 45
3.6 Variable resolution input . . . 47

3.7 Rendering Pipeline 50
3.8 Procedural refinement 52
3.9 View-space error control 53
3.10 Inspection sequences 54

4.1 The BlockMap Concept 60
4.2 The BlockMap Texture 62
4.3 BlockMap Ambient Occlusion 63
4.4 BlockMap Components 64
4.5 BlockMap Ray Intersection Cases 66
4.6 BlockMap Wall Hit 67
4.7 BlockMaps Rendering Steps 68
4.8 Rendered BlockMap 69
4.9 Walls Attributes Sampling 72
4.10 BlockMap Remote Rendering Architecture 75
4.11 BlockMap Child Composition 77
4.12 Paris Dataset 79
4.13 Bandwidth Comparison 79
4.14 BlockMap Network Cache Misses Statistics 79
4.15 Detail Perception with Ambient Occlusion 80
4.16 Pompei Dataset 81
4.17 Simple Shapes with Ambient Occlusion 81

5.1 SpiderGL Library Architecture 88
5.2 SpiderGL Mesh Rendering Setup 94
5.3 SpiderGL Shadow Mapping 99
5.4 SpiderGL Large Meshes Rendering 100
5.5 SpiderGL Terrain Tile Skirts 102
5.6 SpiderGL Multiresolution Terrain Rendering 103
5.7 SpiderGL BlockMaps Rendering 104
5.8 SpiderGL Polynomial Texture Maps 105
5.9 SpiderGL Molecular Lipophilic Potential 107
5.10 SpiderGL Molecular Electrostatic Potential 108
5.11 SpiderGL Molecule Atom Structure 109
5.12 SpiderGL MeShade Authoring Tool 111

List of Tables

3.1 Preprocessing and Compression Evaluation 55

List of Listings

5.1 SpiderGL Naming Convention 87
5.2 SpiderGL Object Info Structure 90
5.3 SpiderGL Mesh Rendering 95
5.4 SpiderGL Molecule Electrostatic Potential Shader 108

Chapter 1

Introduction

Real–time 3D exploration of large environments is a challenging task for the Computer Graphics research community. The huge amount of data, typical of these datasets, often exceeds the capabilities of commodity PCs and thus requires visualization techniques that are able to provide a representation of the original dataset at multiple resolutions.
In this chapter we describe the motivations that lead us to develop novel techniques for real–time exploration of large environments, and outline our contribution to the state–of–the–art.

1.1 Motivations and Contribution

Real–time virtual exploration of large environments is a fundamental requirement of many practical applications. Systems dedicated to critical purposes, such as seismic monitoring, crisis prevention and handling, construction planning of urban areas and traffic management, often need a detailed geometrical view of the region of interest to make complex decisions. On the other side, applications that must provide a general visual feedback, such as tourism–oriented and advertising campaigns, or even targeted to pure entertainment purposes like video games, require detailed information about the actual appearance of the environment, like high–definition images taken from artificial satellites and even at street–level. The continuous growth of interest in 3D exploration applications mainly depends on the increasing availability of terrain elevation data and urban models. In fact, several efforts are focused on the production of digital urban environments, based on a number of different technologies, like semi–automatic reconstruction solutions (e.g. specialized 3D scanning), assisted reconstruction from digital photos and video streams, and parametric reconstruction methodologies.
Due to the very large geometric and image complexity of datasets spanning large geographic regions, and to the restrictions that system and graphics hardware imposes on memory footprint and processing power, it is not always possible to provide the

user with a highly interactive virtual immersion in the environment represented by the raw input data. One technique for providing real–time performance is through the use of adaptive multiresolution rendering algorithms. These techniques assume that datasets are represented at different levels of detail, usually arranged in a hierarchical structure, and select at rendering time the representation that can be rendered within user–imposed time or quality constraints.
Throughout the years, the computer graphics research community has developed several families of multiresolution techniques, whose common denominator is the adoption of strategies aimed at distributing the workload in a view–dependent manner. Most of the developed techniques have a domain–specific nature and are heavily conditioned by the capabilities of the underlying graphics system. In fact, in the last decade, we have witnessed a serious evolution of commodity graphics hardware, both in terms of performances and capabilities, that allowed to develop a large amount of new real–time rendering algorithms previously confined to professional workstations or even off–line processing. Thanks to the possibility to execute user–defined programs, modern graphics hardware is referred to as *Graphics Processing Unit* (GPU). However, despite the increase in computational resources and the widespread adoption of 3D accelerators even in low–end personal computers, user expectations for real–time graphics software continue to grow at a rate that does not match the processing capabilities.
At the same time, the increase in bandwidth and diffusion of network connections allows large volumes of data to be transferred worldwide in a relatively short amount of time. This leads to the development and commercialization of several software applications capable of visualizing global–scale content in real–time. In this context, terrain models with associated aerial images are the most common type of dataset that can be accessed through remote servers. However, despite the speed of data channels, the richness in detail typical of these datasets often translates in transmission bottlenecks that risk to degrade the exploration experience. To deal with these issues, current research has focused on the development of data compression techniques.

Even though, historically, virtual exploration applications have been devoted to textured digital elevation models, the interest is now rapidly shifting towards urban environments. Focusing on such environments is extremely important, since a large part of Earth's population lives and works in dense urban areas. Moreover, recent advances in city acquisition and modeling have lead to a rapid increase in the availability of urban models that are very complex due to the large amount of data needed to represent a city as a whole. Exploring this kind of datasets, seamlessly going from high altitude flight views to street level views, is extremely challenging. What makes urban models so peculiar is that their geometry is made of many small connected components, i.e. the buildings, which are often similar in shape, adjoining, rich in detail, and unevenly distributed. Usually, each building is represented with a relatively small number of polygons but a large amount of color information, typically one photograph for each façade. While multiresolution

texturing and geometric levels of detail may provide acceptable solution for buildings near to the viewer and moderate degrees of simplification, major problems arise when rendering clusters of distant buildings, In fact, several triangles and a large amount of input texels typically project to a single pixel, making joint filtering of geometry and texturing an important issue. Moreover, relying on surface meshes to drive an aggressive simplification process for distant urban areas is very complicated, as the triangles mapping to a given pixel may stem from disconnected surfaces, or totally different objects. Therefore, state-of-the-art solutions propose switching to radically different, often image-based, representations for distant objects having a small, slowly changing on-screen projection. Typically, impostors are used to fulfill these needs. However, away from their supported set of views, impostors introduce rendering artifacts, such as parallax errors, disocclusions, or rubber sheet effects. Thus, the development of new techniques able to offer a higher quality in multiresolution urban dataset rendering is still needed.

On the technological side, remote exploration of three–dimensional digital models of our environments has been achieved, to date, by using *native* software solutions, that is, compiled applications that have full access to the system resources. However, accessing and visualizing 3D content via a web browser has been a topic of great interest since the graphics hardware of commodity personal computers became powerful enough to handle non–trivial 3D scenes in real–time. Many attempts have been made to allow the user of standard web documents to directly access and interact with three–dimensional objects or, more generally, complex environments from within the web browser. Historically, these solutions were based on software components in the form of proprietary and often non–portable browser plug–ins. The lack of a standardized API did not allow web and computer graphics developers to rely on a solid and widespread platform, thus losing the actual benefits that these technologies could provide. This issue became even more evident when the evolution of the technology behind web browsers allowed interpreted languages to perform quite efficiently in general purpose computations, thanks to novel just-in-time compilers. In this scenario, the need for a standardized computer graphics API for the web platform became a high–priority issue. In very recent times, the problem seems to be solved by the introduction of a new API standard for interpreted languages in web documents that allows the access to graphics accelerators directly within web pages. Although scripting languages can not be considered as performant as a compiled one, the tendency of delegating the most time–consuming parts of a computer graphics algorithm to the graphics hardware helps mitigating the performance gap. Indeed, this tendency is the common thread that drove our research path.

In this work we present a series of novel multiresolution techniques and software solutions targeted to real–time, 3D virtual exploration of massive environments. More in detail we focus on the following areas:

Terrain Data Compression and Rendering. Our contribution to the field con-

sists of an extension to an existing technique, based on a compressed multiresolution representation for the management and interactive rendering of very large planar and spherical terrain surfaces. Our approach modifies the underlying, patch–based hierarchical structure by using the same regular triangulation connectivity for all patches and incrementally encoding their vertex attributes when descending in the multiresolution hierarchy. The encoding follows a two-stage, wavelet–based near-lossless scheme. The proposed approach supports both mean-square error and maximum error metrics allowing to introduce a strict bound on the maximum error introduced in the visualization process. The structure provides a number of benefits: overall geometric continuity for planar and spherical domains, support for variable resolution input data, management of multiple vertex attributes, efficient compression and fast construction times, ability to support maximum-error metrics, real-time decompression and shaded rendering with configurable variable level-of-detail extraction, and runtime detail synthesis. While there exists other techniques that share some of these properties, they typically do not match the capabilities of the proposed method in all of the areas.

Urban Models Streaming and Rendering. We introduce a network- and GPU–friendly simplified representation that efficiently exploits the highly structured nature of urban environments, and illustrate how it can be used to construct an efficient output-sensitive system for urban models rendering. Central to our approach is a novel discrete representation for textured sets of buildings. The proposed texture–based representation encodes both the geometry and the sampled appearance (e.g. color and normals) of a small group of buildings that exhibit a predominant 2.5D structure. The proposed representation is stored into small, fixed size texture chunks, and can be efficiently rendered through GPU ray casting. Since the resulting encoding is a simplified representation of the original dataset, it provides full support to visibility queries, and, when built into a tree hierarchy, offers multiresolution adaptability. By working in a network setting, we also test the effectiveness of the novel structure as a scalable, output-sensitive level–of–detail technique that is able to provide a bounded size representation for large urban scenes.

Three–Dimensional Content on the Web Platform. We describe a novel library aimed at bringing to the web platform the common tools used by computer graphics developers. The proposed software can be used directly into standard html pages to create multiresolution visualization web application for terrain, urban, and image datasets by taking full advantage of the graphics accelerator. Moreover, we will also show how customizable remote 3D models repositories can be created by introducing an online 3D content authoring tool that allows users to apply complex shading techniques to mimic the original materials of the objects.

1.2 Thesis Structure

In this chapter we have exposed the fundamental motivations that inspired our work and outlined our contribution to the field. The rest of this document is organized as follows.
Chapter 2 describes the general architecture of the rendering pipeline and shows how the capabilities of graphics hardware evolved through the years. We then introduce the fundamental concept behind multiresolution techniques and present a brief state–of–the–art on geometry- and texture-based techniques for visualizing large scale datasets.
In Chapter 3 we describe our contribution to terrain datasets compression and rendering. The presented method, called *Compressed Dynamic Adaptive Meshes* (C-BDAM), extends the state–of–the–art in terrain rendering with an effective compression scheme, allowing heavy reduction of memory footprint and transmission latency of huge datasets, coupled with fast, real–time decompression and rendering.
We approach the field of urban models in Chapter 4. We introduce the *BlockMap*, a GPU–friendly representation that exploits the nature of urban environments to ensure multiresolution rendering quality and real–time city exploration tasks. We describe the details behind the novel data representation and show how it can be used to create an efficient, remote multiresolution rendering system.
In Chapter 5 we present *SpiderGL*, a novel JavaScript 3D Graphics library, whose design helps experienced computer graphics developers to implement, in the web platform, well–settled rendering techniques and complex multiresolution algorithms.
Finally, we summarize our work and propose some possible extensions in Chapter 6, where we also provide the list of publications produced by our contribution.

Chapter 2

Background and Related Work

Visualization of large scale environments is an active research area. The evolution of sophisticated techniques that aim at rendering huge amount of geometry and color information can be paired with the constant increase in performances and capabilities of the graphics hardware, allowing complex calculations to be moved from the Central Processing Unit (CPU) to the Graphics Processing Unit (GPU).
In this Chapter we introduce the problem and present a brief state–of–the–art on geometry- and texture-based techniques for visualizing large scale datasets.

2.1 Representing Three–Dimensional Objects

The way in which a three–dimensional object is represented plays a fundamental role in the design and implementation of computer graphics algorithms. As in many computer science disciplines, the data representation (i.e. data structures) determines the nature and efficiency of the operations (i.e. algorithms) that can be performed on it. In the field of computer graphics this means how objects are constructed and manipulated to synthesize an image of a three-dimensional scene.

For a high–level classification, 3D representations can be divided into two main categories: *volumetric* representations, which describe an object by means of the volume it occupies, while *surface* representations concentrate in defining the surface of the object, that is, the *interface* which separates it from the surrounding space.

Volumetric Representations. The most common volumetric representation is a partition of the space with a three–dimensional regular grid. Similarly to an image, the extension in space of the box and the number of subdivisions along each axis (i.e. width, height and depth) determine the resolution at which the information is stored. In this discrete representation, each cell of the 3D array is called a *voxel* in analogy to a 2D pixel. The information held in each voxel is application dependent. For example, 1–bit voxels are commonly used in algorithms that only need to know

whether a certain region of space is occupied by an object or it is empty. In other applications voxels can hold more complex information, like material density (as in the case of CT or MRI scans) or color.
Contrary to the discrete nature of the voxel representation, there exist other volumetric structures which can describe objects in a continuous way. Examples of these are *Constructive Solid Geometry* (CSG), in which the object is described as the result of a series of set operation among solids, and *Binary Space Partitioning* (BSP), where the object interior is identified as the intersection of a set of half–planes.

Surface Representations. Unlike volumetric representations, surface representations only aim at describing the surface of the object and are, generally, irregular. A simple example is a *polygon soup*, that is, a set of (typically planar) polygons which approximates the object surface. Each polygon is represented by an ordered list of *vertices*, where each vertex is a set of values resembling the attributes (e.g. position, color, surface normal, etc.) of a certain point on the surface of the object. Even though a polygon soup is extremely easy to handle, it lacks topological information. In fact, it is not directly possible to establish adjacency relationships among the polygons.
A special case of polygon soup is represented by a polygon *mesh*. A mesh consists of a set of vertices connected to each other to form polygons. A mesh is thus a *cell complex*, a topological space with a collection of connected *cells* (points, edges and faces). In software systems, a polygonal mesh is implemented with an array of vertices, where each item holds the surface attributes relevant to the application, and an array of *faces*, each one implemented as an array of pointers to vertices or, more straightforwardly, indices into the vertex array. Some implementations allow faces to have a variable number of vertices while others fix the polygon ariety, typically to three (triangle meshes) or four (quad meshes).
Other representations, like *subdivision*, *parametric* or *implicit* surfaces, allow the description of the object surface in a smoother way; the first two are particularly versatile and are very often used during the process of modeling 3D objects; on another side, implicit surfaces describe the boundary of the object with mathematical equations, e.g. $F(x, y, z) = 0$.
In some scenarios the problem domain can be restricted, as for of elevation maps (or *height maps*) used to represent land surfaces. In general, height maps describe the height of a surface at a certain point; the term 2.5D is often used to emphasize the analogy with a two–dimensional function.

Throughout our work we will use the triangle mesh and the height map as the main 3D object representation.

2.2 The Graphics Hardware

Since the dawn of computer graphics, almost all of the hardware that has been designed to accelerate the process of image synthesis uses a process, known as *rasterization*, to produce a raster image starting from a vectorial representation of objects. Up to now, the most computationally efficient vectorial primitives handled by the rasterization hardware are points, line segments and triangles. Thus, a triangular mesh is the most appropriate data structure for representing 3D objects.

2.2.1 The Graphics Pipeline

As the input representation undergoes a series of processing stages, the whole process can be modeled as a *pipeline*. The standard composition of the graphics pipeline is depicted in Figure 2.1. Data flows from the *application* (or, more generally, the client of the graphics system) to the *framebuffer*, which is a memory region that holds the final raster image. The whole process is started by the application, which is responsible to setup and issue the drawing commands, specifying data such as vertex attributes, connectivity information, material parameters, and viewing parameters.

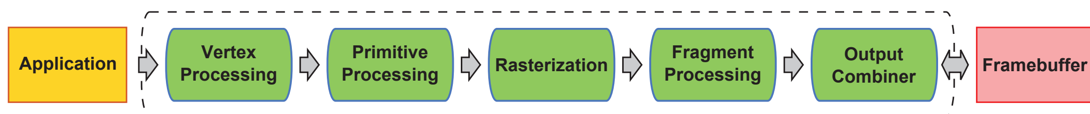


Figure 2.1: **The Graphics Pipeline.** The rendering process is started by the *application*, which sets up and issues the drawing command, sending to the graphics system data such as vertex attributes, connectivity information, material properties and viewing parameters. Data flows through the whole graphics pipeline to the *framebuffer*, a memory region that holds the final raster image.

Vertex Processing. The Vertex Processing stage processes one input vertex at a time and is responsible to apply geometric transformations to vertex positions. It can also calculate other *per–vertex* attributes such as the lighting contribution at the point of the surface that the vertex represents.

Primitive Processing. Once vertices have been processed, the Primitive Processing stage assembles them into geometric primitives such as points, line segments and triangles, and perform *per–primitive* operations like clipping and frustum culling.

Rasterization. The Rasterization stage receives in input a geometric primitive expressed in a vectorial form and is in charge to identify the pixels in the final raster image that are covered by the primitive itself. For each pixel belonging to the primitive, the rasterizer outputs a *fragment*, that is, a data structure containing all the necessary information that will be used in the following stages to compute the final pixel color. It is important to note that the attributes associated to the vertices of the primitive will be interpolated at each fragment.

Fragment Processing. The Fragment Processing stage is responsible to use the interpolated information coming from the Rasterization stage to calculate the final color of the fragment. This is the phase where, typically, texture mapping is used to apply images to the surface of the object.

Output Combiner. The Output Combiner stage performs a series of tests to determine if the fragment must be written to the framebuffer, thus becoming an actual pixel. In this last stage the incoming color and depth values are combined to the existing ones already present in the framebuffer. Typical operations of this stage are depth testing for hidden surface removal and color blending for transparency effects.

2.2.2 Evolution of Graphics Hardware

Graphics hardware has undergone a continuous evolution in terms of performances and capabilities. The first graphics accelerator chips performed very simple operations like fast copy of memory blocks (*blitting*) with a simple set of logical bitwise *raster operations* (ROPs). They could only act on the framebuffer, mandating the CPU to execute all the stages of pipeline. After the introduction of OpenGL [64] in 1992, some stages started to be implemented in hardware. The rasterization process was one of the first processing units with a dedicated chip, although the fragments generated had to flow back to the CPU to process texturing operations. With the release of the Voodoo Graphics 3D Accelerator by 3dfx Interactive, all the stages from rasterization to the framebuffer were completely executed in hardware. Moreover, thanks to the drop in cost of fast memory, texture images were directly stored in video RAM, allowing unprecedented performances during texture mapping operations.
A big step in graphics processing occurred in the late '90s, when the GeForce 256 accelerator was proposed by the NVIDIA Corporation. For the first time, all the operations of the rendering pipeline were performed in hardware, including geometric transformations and lighting calculations. This innovation led the computer community to use the term *Graphics Processing Unit* (GPU) to identify fully capable graphics hardware systems. Concurrently, clock and memory speed and capacity exhibited a steady increase, causing 3D content detail to be pushed to higher level of

complexity and allowing applications to use more computational resources for non–graphics tasks. Up to this point the actual capabilities of graphics systems were fixed, that is, all the operations were tied to a predefined processing step. Even if most of the stages could be configured, it was not possible to implement complex graphics techniques that involved custom data processing.
The beginning of a paradigm shift was set up with the introduction of programmable vertex processing units. The Vertex Processing stage became able to execute *Vertex Shaders*, that is, user–defined code routines expressed through ad–hoc programming languages. This new capability was soon followed by the introduction of *Fragment Shaders* for the Fragment Processing stage. At the same time, the usage of graphics memory was generalized, allowing the storage of geometry data and to fed it directly into the pipeline. This is the case of *Vertex Buffer Objects* (VBOs) for vertex attributes and *Index Buffer Objects* (IBOs) for connectivity information. It is evident how being able to program some of the most important stages of the graphics pipeline, coupled with high memory bandwidth for geometric data, allows the implementation of novel and/or complex techniques whose execution can be mostly or even fully accelerated by the hardware.
Further evolutions introduced *Geometry Shaders* in the Primitive Processing stage, which allowed *on–chip* generation of new geometric primitives and, more recently, the extension of the graphics pipeline with stages for programmable geometry tessellation.
Although the growth in processing speed and capabilities has been considerably rapid, the data transfer performance between system memory and graphics memory is still an issue. Even if the introduction of faster communication interfaces (e.g. PCI Express) reduced the communication time, transferring large amounts of data can cause noticeable stalls during the frame rendering. For this reason, in the context of multiresolution systems where, usually, the dataset is subdivided into several regions, it is often preferable to distribute transfers of large sections (e.g multiple terrain tiles, textures or mesh patches) among several frames. This strategy has the effect of delaying the visual feedback during resolution switches, but prevents perceivable and distracting framerate drops.
Despite communication latencies, modern graphics accelerator exhibit a processing power on the order Teraflops: exploiting the intrinsic parallel nature of the rendering process, recent hardware is built as a collection of several *stream multiprocessors*, each equipped with many processing cores. From a quantitative point of view, clock speed can reach 2 GHz while fast dedicated memory can extend up to 2 GBytes.

The practical effect of these improvements is that, as evolution proceeds, the execution of more and more kinds of heavy calculations required by sophisticated techniques can be delegated to the graphics hardware, thus diminishing the workload of the application (and, more generally, of the CPU), which can dedicate more computational resources to other tasks.

2.3 Adaptive Multiresolution Rendering

Many application domains require the ability to visualize complex datasets with real–time interaction. Due to the very large geometric and image complexity of such datasets, and to the restrictions that system and graphics hardware imposes in terms of memory and processing power, it is not always possible to provide the user with a highly interactive virtual immersion in the environment represented by the raw input data. One technique for providing real–time performance is through the use of adaptive multiresolution rendering algorithms. These techniques assume that datasets are represented at different levels of detail. The representation can be rendered within user imposed time or quality constraints selected at rendering time. We introduce the subject of visual and temporal fidelity and review techniques that rely on detail selection for rendering scenes composed of many independent objects, as well as view–dependent techniques for handling large multiresolution objects spanning a wide range of different distances from the observer.

Very large and geometrically complex scenes, exceeding millions of polygons and possibly thousands of objects, arise naturally in many areas of interactive computer graphics. Handling this kind of scenes presents important challenges to application developers. This is particularly true for highly interactive 3D programs, such as visual simulations and virtual environments, with their inherent focus on interactive, low–latency, and real–time processing. Since there is no upper bound on the complexity of a scene visible from a specific viewpoint, occlusion and view frustum culling techniques are not sufficient alone for meeting the performance requirements dictated by the human perceptual system. Achieving this goal requires the ability to trade rendering quality for speed. Ideally, this time/quality conflict should be handled with adaptive techniques, to cope with the wide range of viewing conditions while avoiding worst–case assumptions.

2.3.1 Visual and Temporal Fidelity

Vision is generally considered the most dominant sense, and there is evidence that human cognition is oriented around vision [70]. High–quality visual representation is thus critical for visual simulation applications.

The major aspects of the visual sense that have an impact on display requirements are the following:

- **depth perception**: stereoscopic viewing is a primary human visual mechanism for perceiving depth. However, because human eyes are located only on average 6.3 centimeters apart, the geometric benefits of stereopsis are lost for objects more distant than 30 meters, and it is most effective at much closer distances. Other primary cues (eye convergence and accommodation) and secondary cues (e.g. perspective, motion parallax, size, texture, shading, and

shadows) are essential for far objects and of varying importance for near ones;

- **accuracy and field-of-view**: the total horizontal field of vision of both human eyes is about 180 degrees without eye/head movement and 270 degrees with fixed head and moving eyes. The vertical field of vision is typically over 120 degrees. While the total field is not necessary for a user to feel immersed in a visual environment, 90 to 110 degrees are generally considered necessary for the horizontal field of vision [121]; when considering accuracy, the central fovea of a human eye has a resolution of about 0.5 minutes of arc [58];

- **critical fusion frequency**: visual simulations achieve the illusion of animation by rapid successive presentation of a sequence of static images. The critical fusion frequency is the rate above which humans are unable to distinguish between successive visual stimuli. This frequency is proportional to the luminance and the size of the area covered on the retina [29, 73]. Typical values for average scenes are between 5 and 60 Hz [121]. A rule of thumb in the computer graphics industry suggests that below about 10–15 Hz, objects will not appear to be in continuous motion, resulting in distraction [85]. High-speed applications, such as professional flight simulators, require visual feedback frequencies of more than 60 Hz [15].

Additional performance constraints derive from the fact that, often, multimodal outputs have to be integrated into a single system, introducing therefore synchronization constraints. Even mono–modal interactive 3D applications (e.g. applications with only visual feedback) have to face similar problems, and low–latency constraints have to be met. In this case, synchronization is between synthesized and real–world sensory input.

Human beings are very sensitive to synchronization delays. For instance, it has been reported that, depending on the task and surrounding environment, lag of as little as 100 ms degrades human performance [80, 50] and, if lag exceeds 300 ms, humans start to dissociate their movements from the displayed effects, thus destroying any immersive effects [50].

This means that spatio–temporal realism, i.e. the ability to meet frequency, lag, and accuracy constraints within low tolerances, is a required feature for all visual simulation systems. The goal of adaptive multiresolution rendering techniques is to strike a compromise between visual and temporal fidelity by dynamically adapting the complexity of the rendered models (see Figure 2.2).

2.3.2 Geometric Level-Of-Detail

The idea of using a cheaper representation for distant geometry that has a small, slowly changing on–screen projection is central to many approaches for massive model rendering. The visualization of a scene at different resolutions is the core aspect of *Level-Of-Detail* (LOD) techniques.

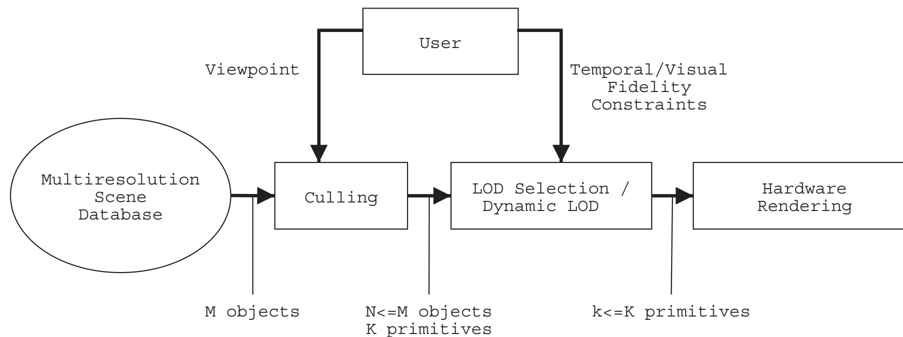


Figure 2.2: **Adaptive Multiresolution Rendering Pipeline.** The adaptive multiresolution rendering subsystem aims at meeting visual and/or temporal fidelity constraints by dynamically adaptive the complexity of the rendered models.

Discrete vs. Continuous LODs

A multiresolution model provides a way to manage approximate representations of the same geometric object at different resolutions. *Discrete LOD* representations store a 3D model in a fixed number of independent resolutions. An improvement to this method consists in representing objects as multiresolution meshes, also known as *Continuous LOD* (CLOD). In this case, the LODs are stored in a dedicated data structure from which representations of the mesh can be extracted with any number of polygons.

The choice of the type of multiresolution representation has important consequences. In particular, having a small number of LODs might introduce visual artifacts due to the sudden changes of resolution between differing representations [49] and, more importantly, limits the degrees of freedom of LOD selection algorithms.

LOD selection vs. view–dependent tessellation

Many types of graphic scenes contain a large number of distinct and possibly animated small–scale objects (e.g. rigid body simulations, virtual engineering prototypes [115]).

In an adaptive renderer, these scenes are modeled as a collection of distinct objects, each possessing multiple graphic representations. The graphic representations range from a few discrete LODs per object, both purely geometric or image–based (i.e. impostors, [106, 31]). In this case, the core of the adaptive renderer is the *LOD selection* algorithm that decides at each frame which representation to use for each of the visible objects.

The key to real–time rendering of large–scale surfaces is, instead, to locally adapt surface geometric complexity to changing view parameters. As dynamic re–tessellation adds a run–time overhead, this approach is suitable when the time gained because of the better simplification is larger than the additional time spent in the selection algorithm. For this reason, this kind of technique has been applied when

the entire scene, or most of it, can be seen as a single large multiresolution object from which to extract view–dependent variable resolution representations.

Continuous LODs

It has been demonstrated that the accuracy and running time problems associated with discrete LOD selection are overcome when using appropriate multiresolution data structures that allow the expression of predictive LOD selection in the framework of continuous convex constrained optimization [44, 45].

In general, the assumption of a *continuous LOD* technique is that the underlying multiresolution geometric information is represented as a triangular mesh M. A *cost* function is defined to calculate the time and space resources associated to the rendering of a subset of M. Similarly, a *degradation* function calculates the drop in quality that the visualization of a subset of M introduces with respect to the rendering of the original data. This allows to state the continuous LOD selection problem as an optimization problem which minimizes both the *cost* and *degradation* functions, whose nature largely influences the approach to its solution.

The work presented in [44] assumes that the *cost* depends linearly on the number of triangles and that *degradation* is a convex and smooth function of the object resolution. The authors use a *sequential unconstrained minimization* algorithm to solve optimization problem with a guaranteed specified accuracy ϵ. The algorithm solves a sequence of unconstrained minimization problems for decreasing values of ϵ, starting at each step from the previously computed sub–optimal value. Each minimization problem is solved using an interior point algorithm for convex optimization, which transforms the original problem into an effectively unconstrained problem by incorporating in the objective a barrier function which is smooth, convex, and tends to infinity as the parameter approaches the boundary of the feasible set. The framework presented in [44] is tested against a walkthrough in a scene with more than one million polygons and hundreds of objects rendered at ten frames per second on a low end PC. The solution error is below 5%, which is an order of magnitude better than what can be guaranteed by current combinatorial optimization approaches.

The work presented in [45] restricts the *degradation* function to have a resolution–dependent part. The authors demonstrate that the particular optimization problem associated to this kind of function can be solved more efficiently using an active–set strategy, which searches for a solution of the original inequality–constrained optimization problem along the edges and faces of the feasible set by solving a sequence of equality–constrained problems. By exploiting the problem structure, Lagrange multiplier estimates and equality constrained problem solutions are computed in linear time. The authors demonstrate the performance of the approach with walkthrough test cases, for models of up to one thousand independent parts and over one million polygons rendered at ten frames per second on a low end PC. Figure 2.3 shows an example of adaptive CAD visualization.

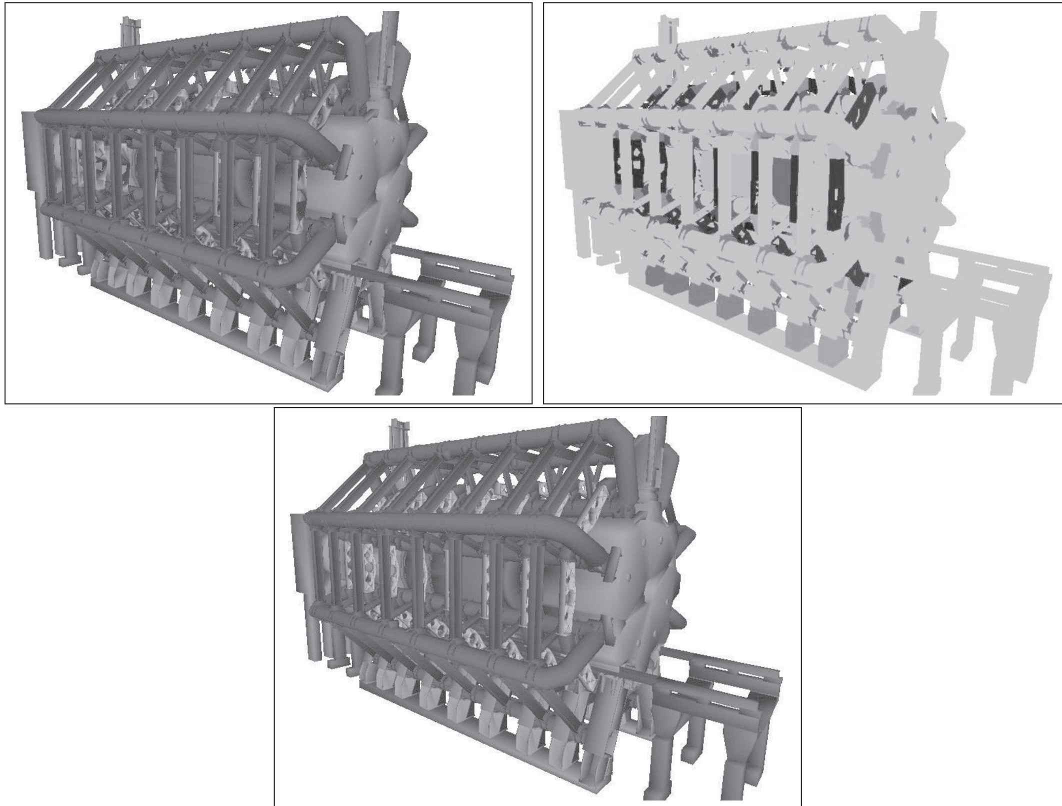


Figure 2.3: **Multiresolution Rendering Example.** These three snapshots illustrate an example rendering of the ATLAS Experiment Pit, part of the LHC facility at CERN using the system presented in [44]. In the bottom image, all 985 objects are rendered at maximum resolution (246K triangles). The top left image is what the viewer actually sees during interactive navigation, with the resolution of each object modified to meet timing constraints (10 frames per second on a low end PC). The total number of displayed polygons is reduced to 42K triangles). The top right image depicts the resolution chosen for each object, lighter shades representing more detail.

2.4 Multiresolution in Terrain Rendering

Adaptive rendering of huge terrain datasets has a long history, as attested in well established surveys [79, 92].
The vast majority of the adaptive terrain rendering approaches have historically dealt with large triangle meshes computed on the regularly distributed height samples of the original data, using either irregular (e.g., [30, 52]) or semi-regular adaptive triangulations (e.g. [77, 39, 91, 78], see Figure 2.4).

The main objective of this kind of algorithms was to compute the minimum number of triangles to render each frame, so that the graphic board was able to sustain the rendering. More recently, the impressive improvement of the graphics

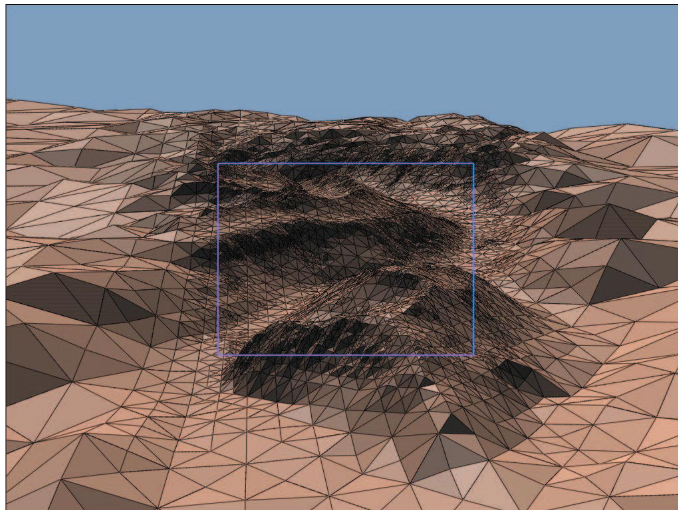


Figure 2.4: **Semi-Regular Adaptive Triangulation.** The approach in [78] shows the quick drop of the mesh resolution as the terrain geometry goes outside the viewing frustum, virtually represented by the violet rectangle.

hardware both in term of computation (transformations per second, fill rate) and communication speed (since the introduction of AGP1x to the PCI-E) shifted the bottleneck of the process from the GPU to the CPU. In other words, the approaches which select the proper set of triangles to be rendered in the CPU did not have a sufficient throughput to feed the GPU at the top of its capability. For this reason many techniques proposed to reduce per-triangle workload by composing at run-time pre-assembled optimized surface patches, making it possible to employ the *retained-mode* rendering model instead of the less efficient direct rendering approach. Tiled blocks techniques (e.g., [51, 118]), originally designed for external data management purposes, partition the terrain into square patches tessellated at different resolutions. The main challenge is to seamlessly stitch block boundaries. The first methods capable to producing adaptive conforming surfaces by composing precomputed patches with a low CPU cost, were explicitly designed for terrain rendering. RUSTIC [100] and CABTT [76] are extensions of the ROAM [39] algorithm, in which subtrees of the ROAM bintree are cached and reused during rendering. A similar technique is also presented in [32] for generic meshes. BDAM [19, 21] constructs a forest of hierarchies of right triangles, where each node is a general triangulation of a small surface region, and explicitates the rules required to obtain globally conforming triangulations by composing precomputed patches. A similar approach, but described in terms of a 4-8 hierarchy, is described in [54], which store textures and geometry using the same technique.

Recently various authors have concentrated on combining data compression methods with multiresolution schemes to reduce data transfer bandwidths and memory footprints. Tiled block techniques typically use standard 2D compressors to in-

dependently compress each tile. Geometry Clipmaps [81] organize the terrain height data in a pyramidal multiresolution scheme and the residual between levels are compressed using an advanced image coder that supports fast access to image regions (see Figure 2.5).

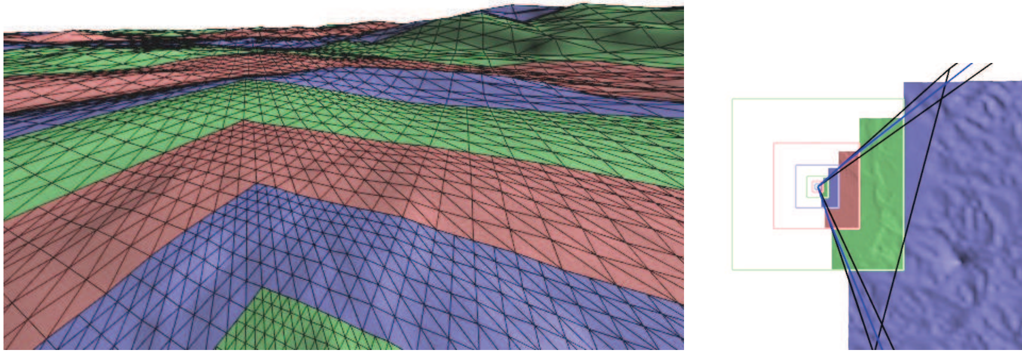


Figure 2.5: **Geometry Clipmaps.** The terrain height data is organized in a pyramidal multiresolution scheme. At rendering time, the pyramid apex coincides with the observer's position.

A compact representation even for large datasets is obtained by storing in a compressed form just the heights and reconstructing at runtime both normal and color data (e.g. using a simple height color mapping). The pyramidal scheme limits, however, adaptivity. In particular, as with texture clipmap-based methods that relies on a single hierarchy, the technique works best for wide field of views and nearly planar geometry, and does not apply, as it is, to planetary reconstructions. In fact, non–planar dataset are typically build on a set of hierarchies and require, for a given perspective, more than one nesting neighborhood, each belonging to a different hierarchy. In [54], the authors point out that, when using a 4-8 hierarchy, the rectangular tiles associated to each diamond could be also compressed using standard 2D image compression methods. More information about terrain data compression schemes can be found in [5].

In Chapter 3 we will describe our contribution to the field of terrain data compression and visualization.

2.5 Texture-Based Accelerated Rendering

In the last 15 years a set of hybrid techniques have been proposed to accelerate the rendering of a scene replacing complex geometry with textures. The obvious achievement is that texture rendering is a constant time operation, while the time for rendering geometry is proportional to its complexity (i.e. number of polygons).

2.5.1 Billboards

A **billboard** is a planar polygon which orientation changes to always face the viewer [86]. It is used to replace geometric representations of objects that have a rough cylindric symmetry, like a tree. There are different types of billboard, which differentiate on their use and on the way their rotation matrix is computed. The most common is the *axial billboard*, where the billboard rotates only on a specified axis (this is the case of a tree). Let w_{up} be that axis, v_{dir} the viewer direction and n the normal to the billboard. The angle of rotation is computed as the angle between n and the vector T_{dir} that lies on the plane individuated by w_{up} and v_{dir} and perpendicular to w_{up} (see Figure 2.6). The vector t_{dir} is computed as:

$$t_{dir} = w_{up} \times (v_{dir} \times w_{up})$$

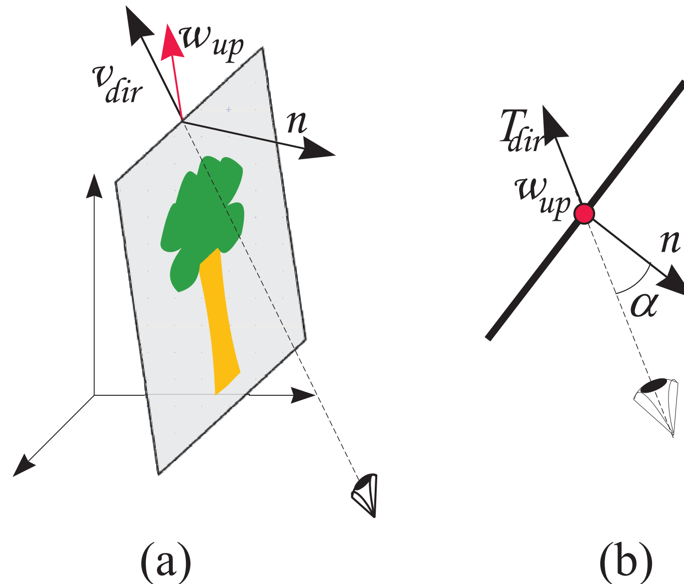


Figure 2.6: **Billboard Rotation.** (b) Shows the same billboard as in (a) from the w_{up} direction. The vector T_{dir} lays on the plane formed by w_{up} and v_{dir}. The angle α between n and T_{dir} is the rotation to be applied.

The drawback of the axial billboards is that if the viewer looks down the w_{up} axis the illusion is ruined. A solution to this problem is proposed in [82], considering a more sophisticated type of impostor that uses a bounding box and projecting on each face the appropriate texture. To decide when the texture or the geometry have to be used, a set of sample directions is taken and for each one the result of using geometry or texture are compared, producing a value representing the *benefit* of using the texture (see Figure 2.7) and which is used at run time.

In [106] a similar idea is used but the impostors are generated on-the-fly and the same holds for the evaluation of the benefit of using impostor. The impostor is

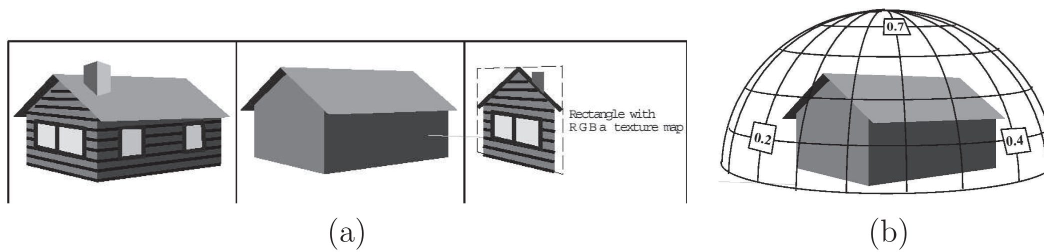


Figure 2.7: **Bounding Box Impostors.** (a) Model of a house with texture (left); only geometry (right) and projection of the front part on a bounding box side. (b) The hemisphere used to choose which is the proper side of the textured box to use for rendering.

created at run time, pointing the view direction towards the center of the bounding box of the object, rendering the object and copying in the texture buffer the region corresponding to the bounding rectangle of the image. The alpha channel of the frame buffer is set to 1 where the object is present and 0 elsewhere. Once created, it is used as a billboard, rotating the plane to always face the viewer. In this technique the screen resolution and the object distance are also considered to guarantee not to produce a texture with more than a texel per pixel. Since that generally objects do not have spherical symmetry, an impostor is valid only in a range of viewer positions. If the viewer position translates the validity of the current impostor is evaluated as the angle shown in Figure 2.8. The case (a) shows how to compute an error angle which increases as the viewer moves parallel to the plane of the impostor. The error angle is the one formed by the extreme points of the bounding box (which were collinear with the previous viewer position) with the new point of view. A similar reasoning holds for the case (b), when the viewer gets closer. There is no error associated to change of orientation, since the viewing projection is perspective.

A further improvement is the *nailboard* [109, 107]. The idea is to use a depth buffer for each texture, so obtaining a RGBΔ texture. This depth buffer stores, for each texel, the distance between the plane and the point on the object that projects on that texel. This can avoid artifacts when the impostor is close to geometry, offsetting the Δ value of each pixels by the distance of the nailboard from the viewer and obtaining a correct depth buffer. The precision of the method depends on how many bits are used to store the Δ value (see Figure 2.9). Instead of using a single texture with depth values, in [108] multiple layer of textures are used, where each layer stores the projection of the part geometry that is contained in a specified depth interval. In other words, each texture is an impostor on its own, but related to a single layer of the object.

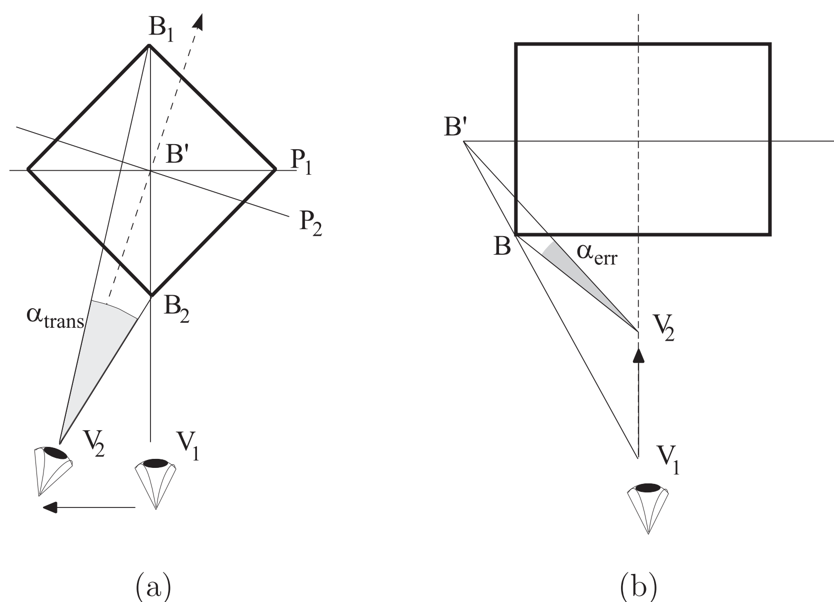


Figure 2.8: **Impostor Error.** (a) Error due to translation of the point of view; (b) Error due to moving-in the point of view

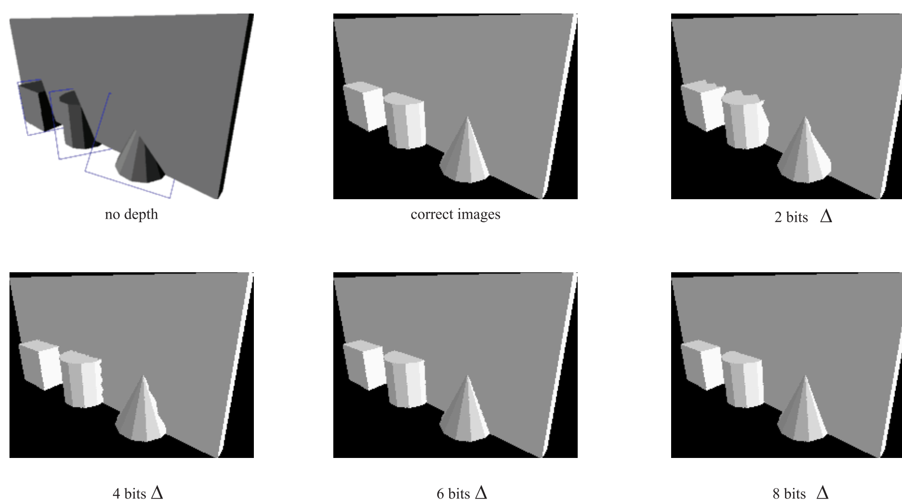


Figure 2.9: **Nailboards.** The top left image shows the error produced using no depth information when the objects represented with textures (box, cylinder and cone) overlap a polygon; the next image shows the result with a correct visibility. The subsequent images show the improvement of visibility as the precision (number of bits) of the depth information increase.

2.5.2 Portal Textures

There are environments which are naturally subdivided in cells with reduced mutual visibility. A typical example is the inside of a building, where adjacent rooms can be connected by doors of windows and if the observer is in a room he/she can see the inside of the adjacent cells only through those openings.
This feature can be exploited in visibility culling, disregarding all the geometry which is outside the perspective formed by the observer position and the opening. If the observer is not too close to the opening and/or the opening is not too wide, it makes sense to put a texture on the opening instead of displaying the geometry. In [4] *portal textures* are introduced to this aim. In this approach, the portal textures are computed in a pre-processing phase and fetched appropriately as the observer moves from a cell to another. Since the view through an opening is not the same from any point in the cell, a set of views equally spaced along a semicircle centered in the opening are sampled. At run time, the image to display is the texture corresponding to the sample point that is the closest to the current position of the observer. When the observer is closer than a fixed threshold to the opening, the portal texture is replaced by geometry and the opposite happens when the observer is further than the threshold.

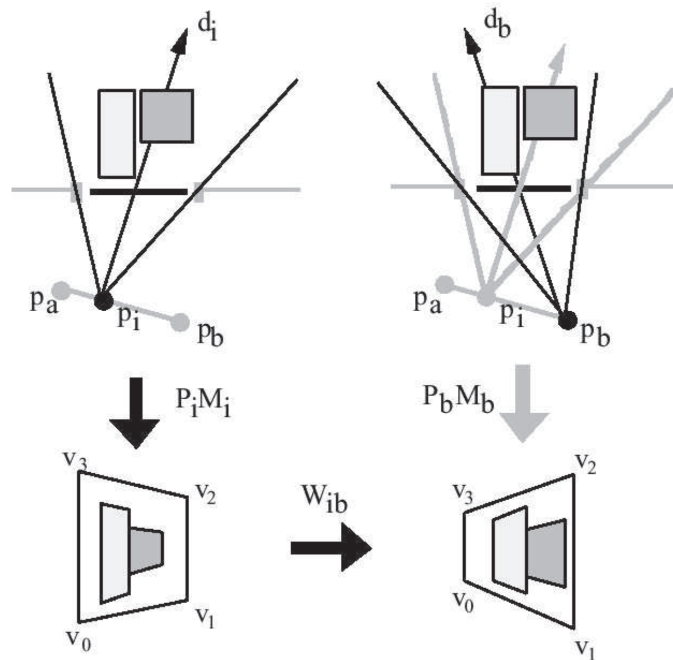


Figure 2.10: **Portal Textures: Smooth Warping.** The *popping* effect arising whenever the *mode* is changed or the viewpoint does not correspond to a sample point is mitigated by introducing a smooth transition with warping.

To limit unpleasant *popping* effects that would happen every time the *mode* has to be changed and the point of view does not correspond to a sample point the

authors make use of *warping* to have a smooth transition. This warping procedure is done under the assumption that the height of the point of view is constant and the opening is perpendicular to the floor.
Let p_a be the closest sampled point from the current position p_b and assume the texture has to be replaced by geometry from the current position. The method consists of defining a warping procedure that warps the geometry to match the texture and using that procedure for a set of few intermediate points between p_a and p_b (see Figure 2.10).
Let M_i and P_i be respectively the model-space and projection transformation: the pair of matrices for the view frustum $[v_0 - v_3, p_i]$ are easily computed. The matrices for $[v_0 - v_3, p_b]$ are found as: $M_w = P_i M_i$, that is, the geometry as seen by p_i, and $P_w = W_{ib}$, where W_{ib} is a perspective warp from p_i to p_b, obtained as the mapping of the four corners $[v_0 - v_3]$ from their projections $P_i M_i$ to their projection $P_b M_b$:

$$\left(\begin{array}{ccc|c} a & b & 0 & c \\ d & e & 0 & f \\ 0 & 0 & 1 & 0 \\ \hline g & h & 0 & i \end{array}\right)$$

This warping leave the z untouched, so not to modified the depth ordering relation of the polygons.
In [2] the idea of warping geometry is extended in the more general case where a texture is used to replace geometry in a portion of view which is not through an opening. In this case there are no walls along the texture border to occlude the rest of geometry, here called *surrounding geometry*, which therefore is warped to match the texture, so providing C_0 continuity (see Figure 2.11).

Although warping geometry is a way to obtain pleasant results, it should be noted that a texture provide the correct view of the scene only from the point where it has been sampled and not elsewhere. Consequently, it could sound improper to warp the surrounding geometry because it means "display a geometry wrong to match the information of the texture, which is wrong". At a first glance one would think to warp the texture to match the geometry instead but this is not possible because the RGB texture itself does not contain any depth information.
This introduce another problem, that is the possibility that (part of the) objects that was occluded from the sampled point of view could be visible from a different point of view where the same texture is still used. Even if the depth of each pixel is known and the texture is warped preserving visibility, it would contain black holes in the regions that were occluded from the sampled point of view.

2.5.3 Textured Depth Meshes

The Textured Depth Meshes technique, which avoids black holes in the texture, appears in [28, 111]. Textures are triangulated and a depth value is associated to

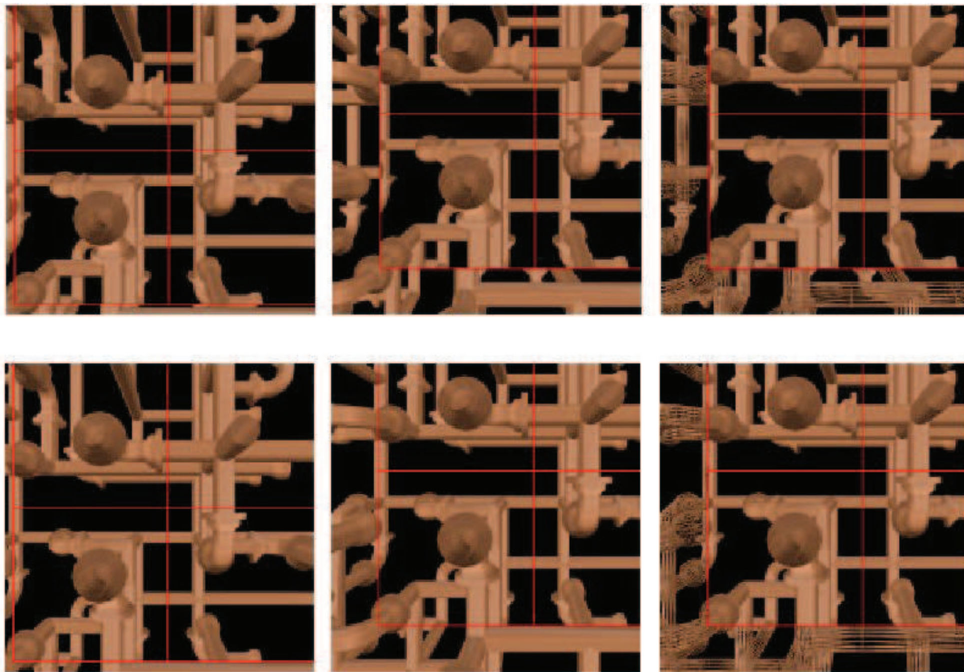


Figure 2.11: **Portal Textures: Warping for Preserving Continuity.** The top three images are produced with no geometry warping. Note the crack between the texture (the part inside the red grid) and the geometry (shown in wireframe on the right most image). The crack is not present when the geometry surrounding the texture is warped to preserve continuity (bottom three images).

each vertex so that the image is not just a discrete grid of points but a continuous surface. The consequence is that when the texture is warped for a new view the de-occlusions cannot create holes anymore (see Figure 2.12).

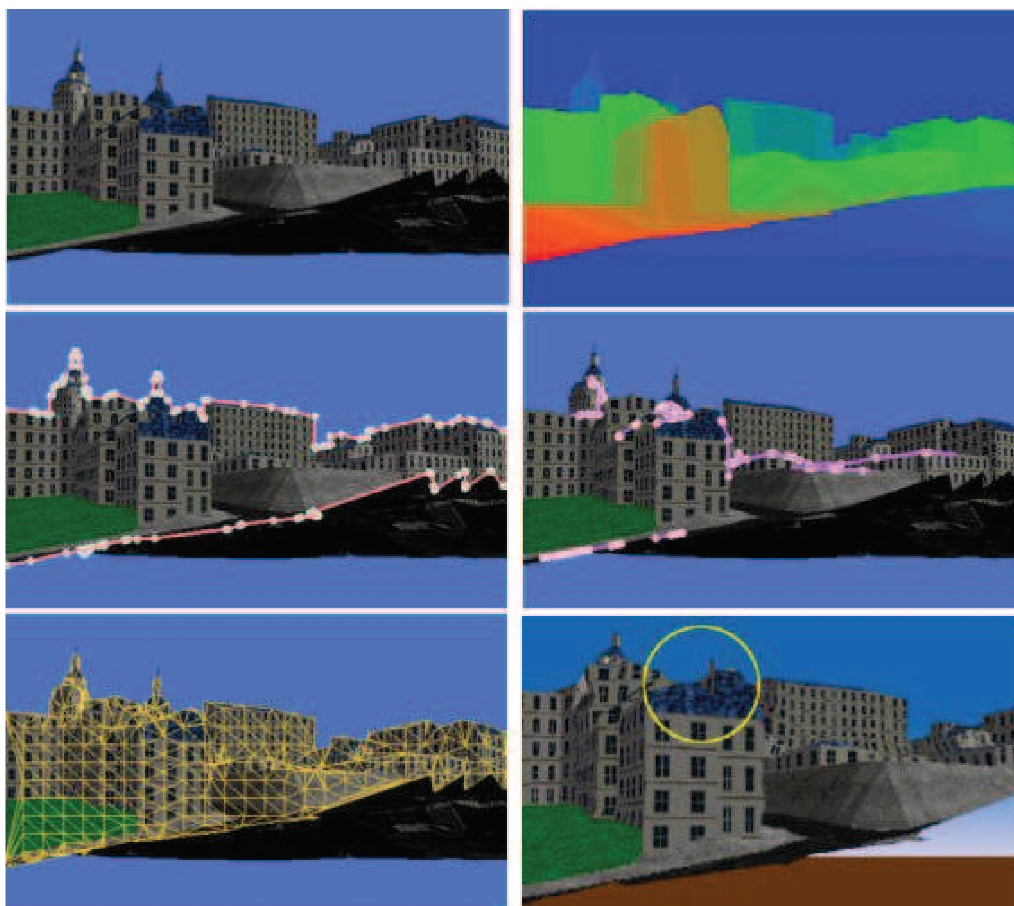


Figure 2.12: **Textured Depth Meshes.** From left to right and from top to bottom: original view with texture and geometry; corresponding depth image; separation of the object from the background; depth disparity lines; meshing; rendering from a different view point.

The creation of the impostor proceeds taking an image of the desired view, individuating the contours of "interesting" features and the line separating regions with big gaps between their depths and then using that piecewise segments to perform a constrained Delaunay triangulation.

It is important than the texture triangulation is done so that edges do not cross borders of regions with very different depth which would result in a erroneous warping of the borders.

2.5.4 Layered Depth Images

A *Layered Depth Image* [110] (LDI) is an image that store more than one sample per pixel. The additional samples at a pixel belongs to surfaces, which are not visible from the original center of projection of the image, along the same ray from the viewpoint. When the viewpoint is moved to a new location, pixels that were occluded and that can now be seen are no more overwritten in the frame buffer.

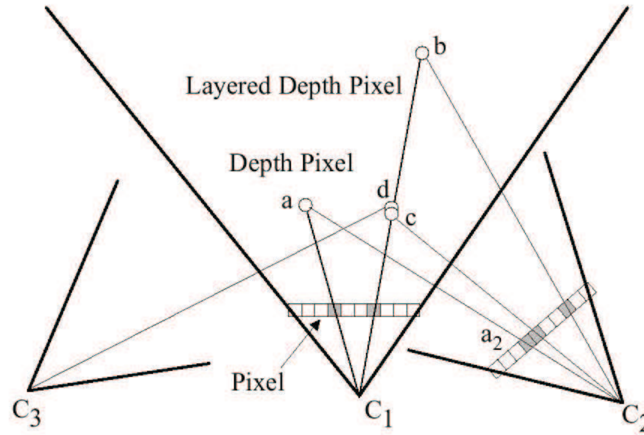


Figure 2.13: **Layered Depth Images.** Several samples are stored per pixel and belong to surfaces not visible from the original center of projection of the image.

Referring to Figure 2.13, let C_1 be the the viewpoint and frustum associated with the LDI and C_2 a new viewpoint and M_1 and M_2 the related 4×4 transformation matrices (from the world coordinates to the viewport coordinates). A pixel in first image is identified by

$$\begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{pmatrix}$$

where (x_1, y_1) are the image coordinates and z_1 is the depth. The projection of this pixel for C_2 is given by $T_{1,2} = M_2 \cdot {M_1}^{-1}$, that is, the pixels is re-projected back to the global coordinates and the projected for C_2. The linearity of matrix operations allows to decompose the product as:

$$T_{1,2} \cdot \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{pmatrix} = T_{1,2} \cdot \begin{pmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{pmatrix} + z_1 \cdot T_{1,2} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = start + z_1 \cdot depth$$

Once *start* has been computed for a pixel, only the value *depth* needs to be computed for the other layered pixels with the same image coordinates. Furthermore, since the value *start* for a adjacent pixel (say on the X axis) is:

$$T_{1,2} \cdot \begin{pmatrix} x_1 + 1 \\ y_1 \\ 0 \\ 1 \end{pmatrix} = T_{1,2} \cdot \begin{pmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{pmatrix} + T_{1,2} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = start + z_1 \cdot x_{incr}$$

In this manner, to find any layered depth pixel, one simply offsets to the beginning of the scanline and then further to the first depth pixel at that location. Much of the efficiency of LDI's depends on the implementation, since that they are not supported by the current graphics hardware. To exploit the second level cache line, spatial locality is taken into account: the depth pixels are stored in a linear array, bottom to top and left to right in screen space and back to front along a ray For each scanline an array of integers cumulates the number of depth pixels for each ray (i.e. the value in the ith position is the number of depth pixels along the first $i-1$ rays in the scanline). Similarly, another array cumulates the number of depth pixels on each scanline (see Figure 2.14).

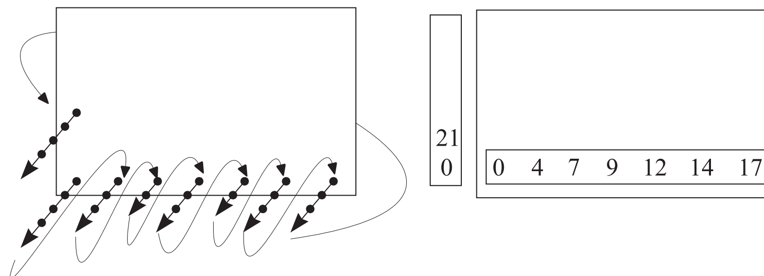


Figure 2.14: **LDI Memory Layout** Pixels order on memory (left) and double array of offsets in LDI implementation (right).

In [101] a parallel implementation of LDI's is proposed and used in the context of portal textures, as in [3].

2.5.5 Multi-Layered Impostors

The *Multi-Layered Impostors* [31] use an heuristic to make textures out of objects that do not overlap if seen from the region where the texture is used. In a preprocessing phase, a graph with a node for each object is defined. The graph is fully connected (i.e. there is an edge for each pair of nodes) and the weight of the edge w_{ij} quantifies the extent of the region from which the objects i and j are seen as overlapping. The bigger is w_{ij} the more occlusion artifact would be seen if they were put in the same texture layer. The graph is then partitioned in subgraphs so

that the weight of any edge connecting two nodes of the same subgraph is under a predefined threshold. Clearly this work is done for each one of cells partitioning the environment (see example in Figure 2.15).
The technique is especially suitable for urban environments (which is the context of its introduction) where the buildings are clearly identifiable as the objects composing the scene. Furthermore, it is assumed that buildings have walls all orthogonal to the viewing plane and the viewing cell is a segment. These assumption make easier to estimate the objects overlapping.

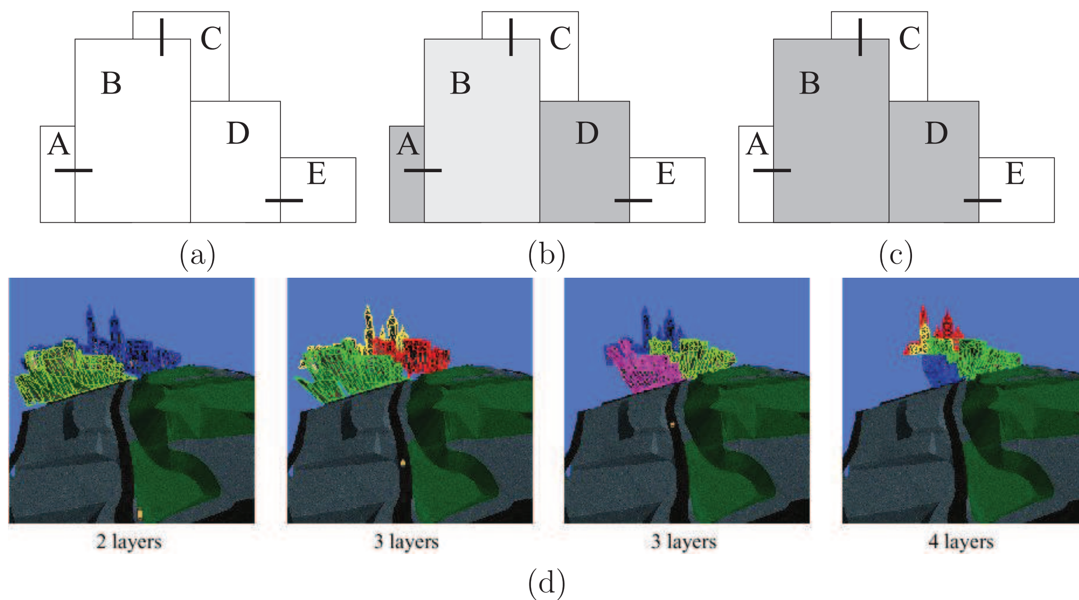


Figure 2.15: **Multi-Layered Impostors.** (a) Example of graph. The weight of the arcs shown is over the predefined threshold, therefore the corresponding nodes will not stay in the same layer. (b) and (c) are two possible partitions respecting such constraint. (d) Application example using 2, 3 or 4 layers (from left to right).

2.6 Adding Visual Detail to Coarse Geometric Representations

The rapid evolution of the graphics hardware (see Section 2.2.2), both in flexibility and processing power, allowed the computer graphics research community to explore a wide range of new techniques for real–time rendering of large datasets. In particular, the switch from a fixed–function, hardwired rendering pipeline to a highly programmable one made it possible to develop ad–hoc algorithms and data structures geometric and optical detail encoding and visualization. In this section we deal with techniques which exploit the ability of performing relatively general

purpose computation to calculate the final color of a pixel in the framebuffer by using custom code fragments, known as *vertex* and *fragment shaders*, to process vertices and pixels, respectively. As a turning point in real–time graphics, these new approaches use textures not only as sources of color information but rather as containers of generic data to add visual detail to 3D objects with simple geometric representation.

2.6.1 Normal Mapping

An observation which holds in a wide range of typical 3D objects is that a large part of the geometric detail consists in small scale features with respect to the overall shape. Due to their small scale, they contribute in a small amount to the silhouette of the object, but the *wrinkles* and *bumps* they represent are clearly noticed when the surface is illuminated. This kind of features are most likely to be removed as a result of a simplification process. In a typical scenario, a high resolution model consisting of a large amount of polygons is simplified to produce a low resolution one which can be quickly rendered in real–time. In 1978, Blinn introduced the concept of *bump maps* [12], where the interpolated surface normal at a pixel is perturbed accordingly to an underlying height In 1998, this technique has been revised with the introduction of *normal maps* [24, 23]. With normal mapping it is possible to add visual details to the low resolution model by encoding the higher resolution surface normals in standard RGB 2D textures which are then used in the lighting equation during rendering. The approach presented by [24] uses a simplification procedure which keeps track of the detail lost during simplification operations. Differently, the technique proposed in [23] is agnostic of the simplification procedure and works directly on the input high and low resolution models. Figure 2.16 depicts the main normal map construction step for the latter approach. The main idea for computing the normal map value at a surface point relies on the fact that the normal to be stored in the normal map should be the one on the high resolution model which is most likely to be visible when looking at the object. This preferred direction of sight is selected by averaging all the rays shot from the simplified mesh toward the hemisphere identified by the surface normal at that point that are not occluded by the high resolution model.
When using normal maps, rather than computing the illumination contribution at each vertex and then interpolate the result inside the polygon primitive, the lighting process is executed per pixel: the polygon texture coordinates are used to fetch the normal map to retrieve the higher resolution surface normal, which is then feed into the lighting equation. Figure 2.17 shows the result of rendering a low resolution model with normal maps: although the underlying geometric representation consists of only a few triangles, the result after computing illumination with normal maps is comparable to the rendering of the original unsimplified model.

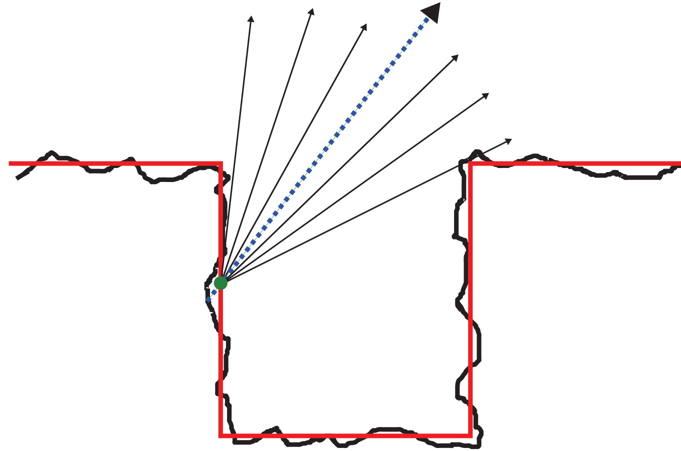


Figure 2.16: **Normal Map Construction.** The normal vector stored in a texel of the normal map (green point) corresponds to the surface normal on the high–resolution model (black outline) which is intersected by the *average visibility direction* (dashed blue arrow) on the low–resolution model (red outline).

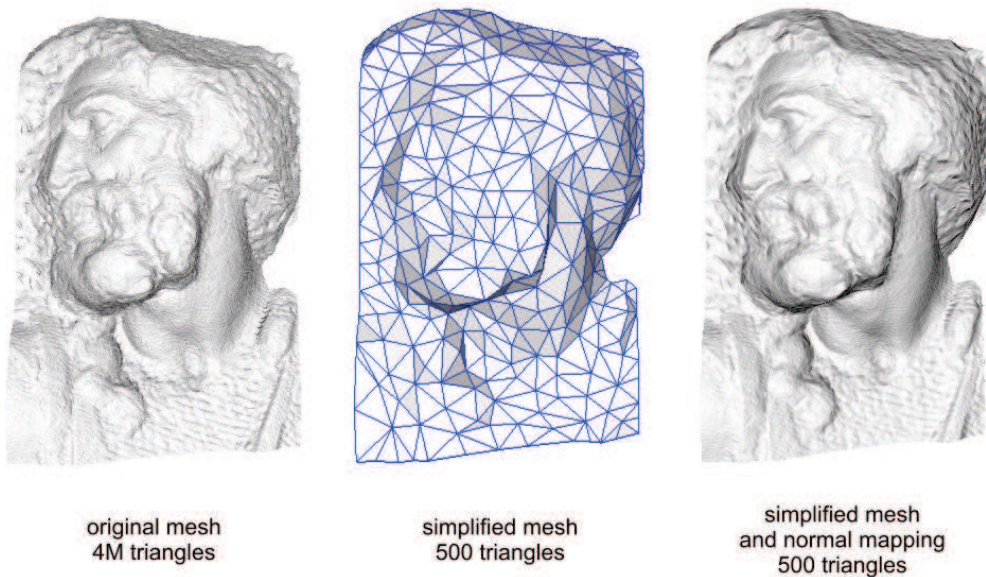


Figure 2.17: **Rendering with Normal Mapping.** *Left*: rendering of a 3D scanned high-resolution mesh consisting of million of triangles. *Center*: the same mesh simplified down to few hundreds of triangles, rendered with wire–frame overlay. *Right*: the simplified mesh rendered with normal mapping. Note how the surface responds to illumination in a way very close to the original high–resolution model, making up for the lack of detail lost during the simplification process.

2.6.2 Parallax Mapping

The most important limitation of normal mapping is that it only affects the shading process. The lighting calculations with normal mapping enrich the surface response to light but it does not generate any *parallax* effect, that is, the apparent displacement in position of the surface when observed from different viewpoints. This perceived disparity in surface position gives the observer the illusion of depth.
Parallax mapping [61] aims at mitigating this problem by using a height map to offset the surface texture coordinates in function of the viewpoint. Rather than distorting the original texture image, this technique uses the programmable graphics hardware to compute, at rendering time, an offset to be added to the polygon texture coordinates, thus simulating a relative surface *shift*, which results in a motion parallax effect.
Figure 2.18 shows how this offset is calculated. For a point A'' in a polygon with texture coordinates u and v, a displacement value $depth(u, v)$ is first accessed from a depth map (typically encoded in the alpha channel of the color texture image). Then, given the viewing vector e from the observer to point A'', the new texture coordinates (u', v') are calculated as

$$(u', v') = (u, v) + \tan \theta_{(u,v)} \times depth(u, v) \tag{2.1}$$

A result of the application of parallax mapping is shown in Figure 2.19. One of the major advantages of this techniques is that the pixel shader code required to compute the offset for the original texture coordinates is particularly easy to implement and has a small impact on the overall performances. However, implementing the parallax effect with texture coordinates offsets results in inaccurate surface displacement, especially noticeable when the underlying structure does not exhibit a considerable level of noise.

2.6.3 Relief Mapping

To add visual 3D surface detail to texture mapped polygons, Oliveira et al. [90] introduced the concept of *relief texture maps*. Basically, as in parallax mapping, a relief texture map is a texture extended with surface displacement information. In this case, the authors make use of image pre-warping techniques to prepare source color and depth images to be eventually used as input to the texture mapping process. Since the rendering of a height field requires to find the closest intersection of the viewing ray with the surface, Oliveira et al. divided the process in two steps: the height filed is first converted to a standard 2D texture image using a forward transform, then the resulting texture is used as in conventional texture mapping. Starting from six images of an object aligned to its bounding box (Figure 2.20), relief maps are warped (Figure 2.21) to obtain a multi–view impostor which can be rendered using only the bounding box geometry and which exhibits a high level of

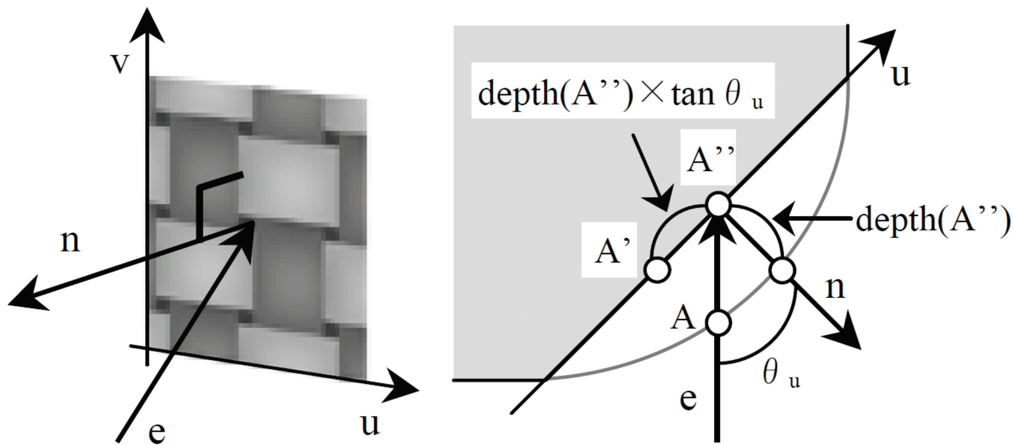


Figure 2.18: **Parallax Mapping: Offset Computation.** The texture coordinates space associated with a polygon is warped before texture fetching as a function of the surface height map, the polygon normal and the direction of the viewing ray.

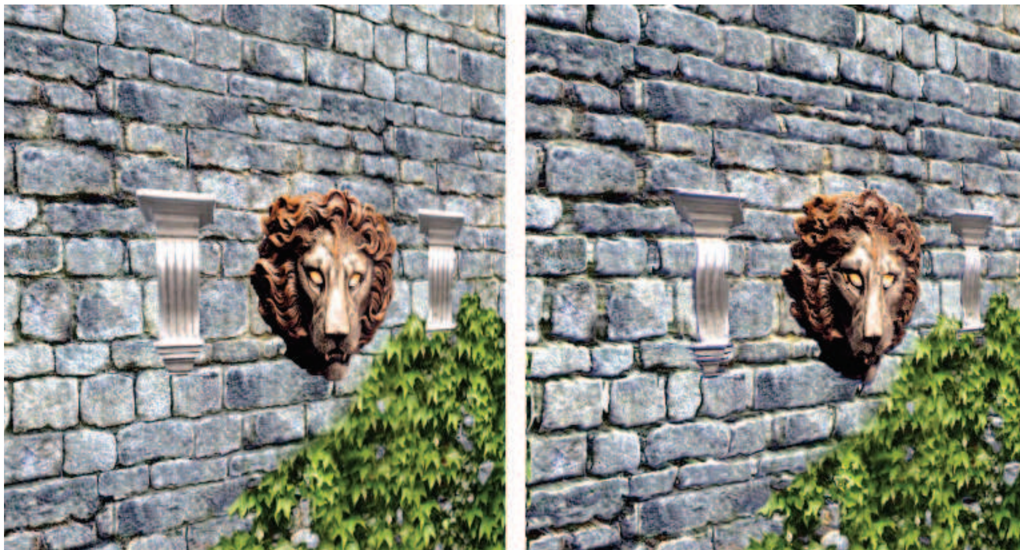


Figure 2.19: **Rendering with Parallax Mapping.** *Left*: A flat wall rendered with standard texture mapping. *Right*: the application of parallax mapping modify texture coordinates in a view–dependent manner, resulting in an illusion of depth which produces a parallax effect whenever the relative position and orientation of the scene and the observer change.

parallax and self occlusion, as shown in Figure 2.22.

Starting from this concept, Policarpo et al. [98] applied relief mapping to arbitrary polygonal surfaces by moving the calculation in the polygon tangent space. The main contribution of Policarpo's work was to exploit the programmability of the

Figure 2.20: **Bounding Box Relief Mapping.** The six relief maps of an object are associated with the faces of its bounding box.

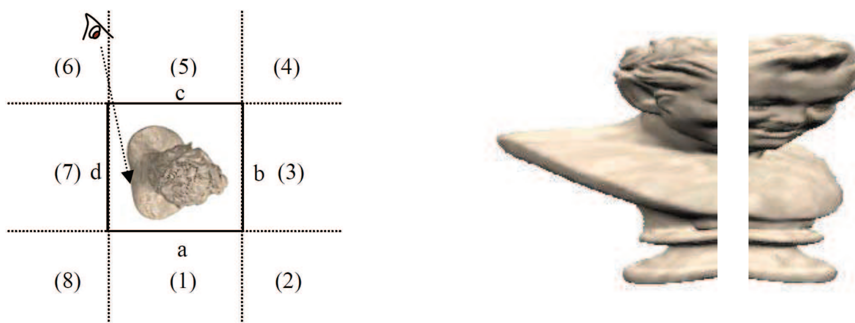


Figure 2.21: **Relief Mapping Warp.** *Left*: in the warping phase, depending on the view direction, samples of one axis–aligned plane can be fetched from different faces of the relief cube map. *Right*: two relief maps after warping.

graphics hardware to efficiently compute the intersection of the viewing ray with the virtual displaced surface. In their implementation they use a two–step ray marching approach: first, a fixed binary search is employed to find a relatively small range in which the viewing ray first hit the surface (Figure 2.23 left), then a linear search is performed to refine the point of intersection (Figure 2.23 right). In both phases, the number of iterations of the search algorithm is user–defined, thus allowing a tunable balance between performances and quality. In more recent graphics hardware, the possibility of dynamically terminate the linear search before reaching the maximum number of iteration contributes in a performances gain.

As shown in Figure 2.24 and Figure 2.25, this technique is able to produces images which exhibit a deep and consistent parallax effect for both planar and arbitrary polygonal meshes.

Although the relative simplicity of implementation, the process involved in the search

Figure 2.22: **Rendering with Relief Mapping.** A view-independent impostor of a statue is rendered by only drawing the object bounding box with pre–warped relief maps.

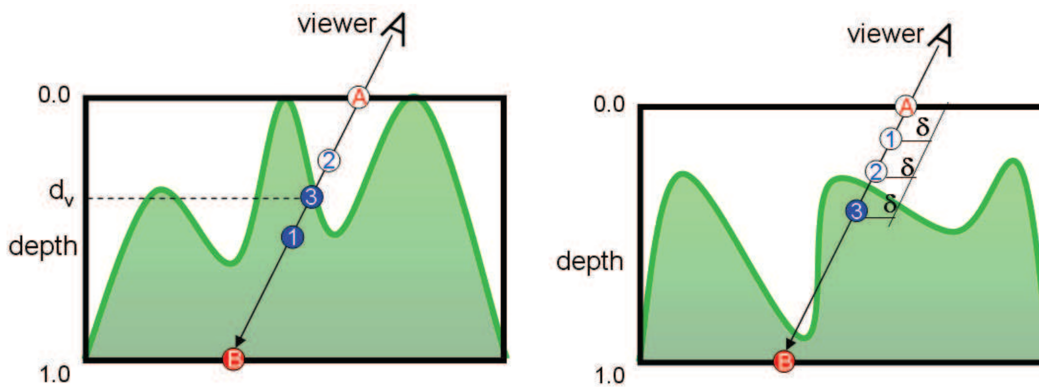


Figure 2.23: **Relief Mapping Ray-Surface Intersection.** The search for intersection of the view-ray with the height-map surface is a two–phases process. *left*, a binary search is performed to spatially reduce the interval in which the view ray first intersect the height map. *Right*: a linear search with fixed ray step is then performed.

for the first hit of the viewing ray with the height map could produce wrong results. During the binary search phase it is assumed that there is only one possible surface hit between two steps of the search, which is not the case when high–frequency *valleys* or *bumps* are present; also, these small features could be skipped in the linear search due to the fixed ray step. These artifacts can be mitigated by using a larger number of iterations but this will result in significant performances loss.

For this reason, several algorithm have been introduced which exploit precomputed information on the height map. The common underlying idea is to adapt the ray marching at each iteration of the search in order to conservatively perform the longest possible step which avoid undetected hit with the surface. These techniques include *distance maps* [38], *interval mapping* [104], *cone step mapping* [40] and *relaxed cone step mapping* [97].

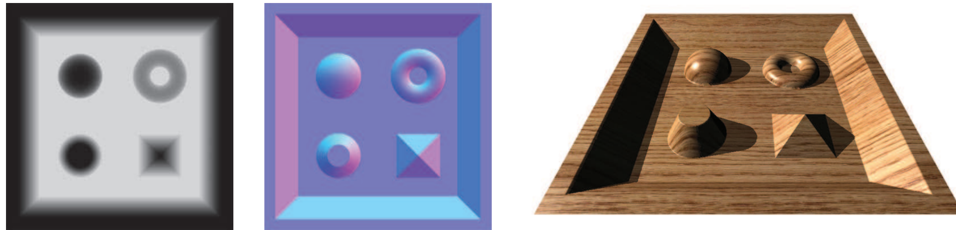


Figure 2.24: **Relief Mapping: Planar Surfaces.** A height map (left) and a normal map (center) are used to produce the illusion of depth in a planar surface (right).

Figure 2.25: **Relief Mapping: Arbitrary Surfaces.** By working in the polygon tangent space, relief mapping can be applied to arbitrary polygonal meshes.

2.7 Texture Management and Multiresolution in Urban Models

Massively textured urban models, where the facade of each building is represented by a different image, require the management of very large amounts of textures.

The data management issue has been historically addressed in the context of terrain datasets with very simple parameterization (e.g., the Clipmap approach [113], that required specialized hardware). These solutions, however, consider a single smooth, uniform texture parameterization, and are not directly applicable to detailed urban datasets with many vertical walls. Just a few solution were proposed for the more general situation of complex meshes with large texture information. The GoLD approach [13] is one of the most recent representatives in this area. It consists of a CLOD hierarchical data structure that partitions the geometry and the texture over it into patches. This approach, like most of the CLOD multiresolution schemes, is more oriented to the management of large unstructured meshes with simple topology than to scenes composed of tens of thousands of single textured components. In [16] the authors present a hierarchical structure called texture–atlas tree that is tailored, to the management of large textured urban models. Their work, however, considers multi-resolution textures but single resolution geometry, and does not take into account network streaming issues.
Recently, Andujar et al. [10] proposed a technique which exploits the base concept of relief mapping to create *omni–directional relief impostors*: by acquiring color, normal and depth information of an object from multiple views (Figure 2.26), the authors create a set of relief maps which are then selected at runtime to reproduce the appearance of the original object. Although this method is not artifact–free and require a non–trivial selection of the original views, given its output–sensitive nature it can be used to produce a multiresolution representation of the scene. In fact, the same basic methodologies have been used in [9] and [8] to produce real–time renderings of large urban environments (Figure 2.27).

In Chapter 4 we introduce a multiresolution framework based on a novel data structure, the BlockMap, which is used to efficiently encode and render urban data.

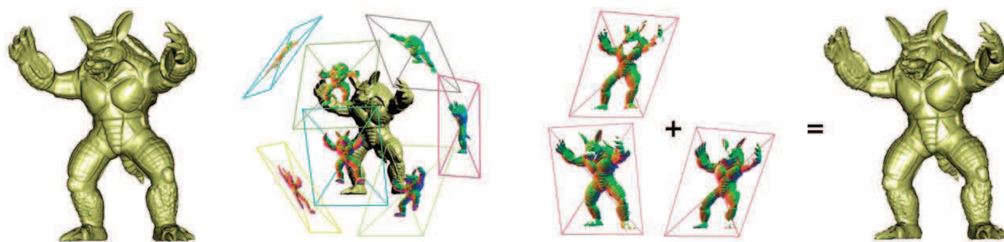


Figure 2.26: **Omni-Directional Relief Impostors.** At preprocessing time, a series of relief maps of an object are generated from different view direction. At rendering time, a subset of the generated reliefs is selected to draw the object from any viewpoint.

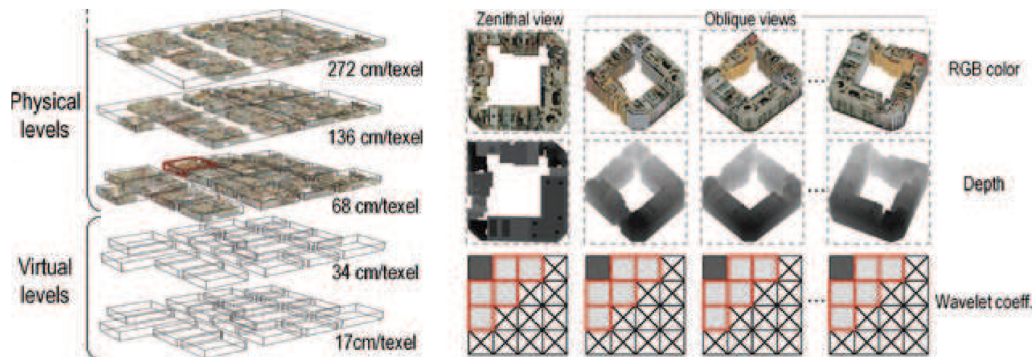


Figure 2.27: **Multi-Level Relief Impostors.** Several relief impostors are generated from the input dataset at different resolutions. The geometry and appearance of the buildings are encoded in a set of textures generated from different viewpoints.

2.8 Deploying 3D Graphics to the World Wide Web

The delivery of 3D content through the web comes with a considerable delay with respect to other digital media such as text, still images, videos and sound. Just like it already happened for commodity platforms, 3D Computer Graphics is the latest of the abilities acquired by the web browsers. The main reason for this delay is likely the higher requirements for 3D graphics in terms of computational power. However, nowadays sophisticated rendering techniques can be implemented thanks to the ability of modern GPUs to perform complex tasks and the possibility to interact with the graphics system directly within web pages, even using relatively slow interpreted languages. Before the introduction of these capabilities inside web browsers, several technologies have been developed over the years.

The Virtual Reality Modeling Language (VRML) [103] (then superseded by X3D [37]) was proposed as a text based format for specifying 3D scenes in terms of geometry and material properties, while for the rendering in the web browser it is required the installation of a platform specific plug-in.

Java Applets are probably the most practiced method to add custom software components, not necessarily 3D, in a web browser. The philosophy of Java applets is that the URL to the applet and its data are put in the HTML page and then executed by a third part component, the Java Virtual Machine. The implementation of the JVM on all the operating systems made Java applets ubiquitous and the introduction of binding to OpenGL such as JOGL [59] added control on the 3D graphics hardware. A similar idea lies behind the ActiveX [1] technology, developed by Microsoft since 1996. Unlike Java Applets, ActiveX controls are not bytecode but dynamic linked Windows libraries which share the same memory space as the calling process (i.e. the browser), and so they are much faster to execute.
These technologies allow to incorporate 3D graphics in a web page but they all do

it by handling a special element of the page itself with a third party component.

More recently, Google started the development of a 3D graphics engine named O3D [48]. O3D is also deployed as a plug-in for browsers, but instead of a black-box, non programmable control, it integrates into the browser itself, extending its JavaScript with 3D graphics capabilities relying both on OpenGL and DirectX. O3D is *scene graph*-based and supplies utilities for loading 3D scenes in several commonly used formats.

The turning point for 3D graphics and the Web is represented by the introduction of WebGL [68], an API specification produced by the Khronos Group [66], that, as the name suggests, defines the JavaScript analogous of the OpenGL API for C++. WebGL closely matches OpenGL|ES 2.0 and, most important, uses GLSL as the language for shader programs, which means that the shader core of existent applications can be reused for their JavaScript/WebGL version. Since WebGL is a specification, it is up to the web browsers developer to implement it. At the time of this writing WebGL is supported in the nightly build versions of the most used web browsers (Firefox, Chrome, Safari), and a number of JavaScript libraries are being developed to provide higher level functionalities to create 3D graphics applications. For example WebGLU [33], which is the WebGL correspondent of GLU [63], provides wrappings for placing the camera in the scene or for creating simple geometric primitives, other libraries such as GLGE [14] or SceneJS [62] uses WebGL for implementing a scene graph based rendering and animation engines.

In Chapter 5 we will discuss a novel JavaScript 3D graphics library, SpiderGL, based on WebGL and show how this new technology can be effectively used to bring multiresolution visualization to web users.

Chapter 3

Data Compression and Rendering for Terrain Models

Visualization of terrain data is one of the most important components of a 3D exploration application. Many terrain datasets consist of several Gigabytes of memory and, thus, require the visualization system to be able to perform its operations in an out-of-core fashion. In this scenario, being able to efficiently transfer data between the different level of the cache hierarchy (such as remote network server or even local hard disks) become a challenge.
In this chapter we describe C-BDAM, a multiresolution representation and rendering technique that extends the state–of–the–art in terrain rendering with an effective compression scheme, allowing heavy reduction of datasets memory footprint and transmission latency.

3.1 Terrain Models Rendering

A fundamental piece of information which is at the core of a large virtual environment is represented by terrain data. By consequence, the ability to explore digital elevation models in real-time is a key component of several practical applications. Nowadays, high accuracy datasets contain billions of samples, exceeding memory size and graphics processing capability of even the highest-end graphics platforms. To cope with this problem, there has been extensive research on output sensitive algorithms for terrain rendering (see Section 2.4) which use multiresolution techniques. At present time, the most successful methodologies for efficiently processing and rendering very large datasets are based on two approaches: adaptive coarse grained refinement from out-of-core multiresolution data structures (e.g., BDAM [19], P-BDAM [21], 4-8 tiling [54]) and in-core rendering from aggressively compressed pyramidal structures (e.g., Geometry Clipmaps [81]). The first set of methods is very efficient in approximating a planar or spherical terrain with the required accuracy and in incrementally communicating updates to the GPU as the viewer moves,

but multiresolution structure footprints typically require out-of-core data management. The second approach, limited to planar domains, uses nested regular grids centered about the viewer. This technique ignores local adaptivity, but is able to exploit structure regularity to compress data so that it typically succeeds in fitting most if not all data structures entirely in core memory, thereby avoiding the complexity of out-of-core memory management for a large class of practical models.

In this chapter we describe a compressed multiresolution representation for the management and interactive rendering of very large planar and spherical terrain surfaces. The technique, called Compressed Batched Dynamic Adaptive Meshes (*C-BDAM*) [46], is an extension of the BDAM [19] and P-BDAM [21] chunked level-of-detail hierarchy, and strives to combine the generality and adaptivity of chunked bintree multiresolution structures with the compression rates of nested regular grid techniques. Similarly to BDAM, coarse grain refinement operations are associated

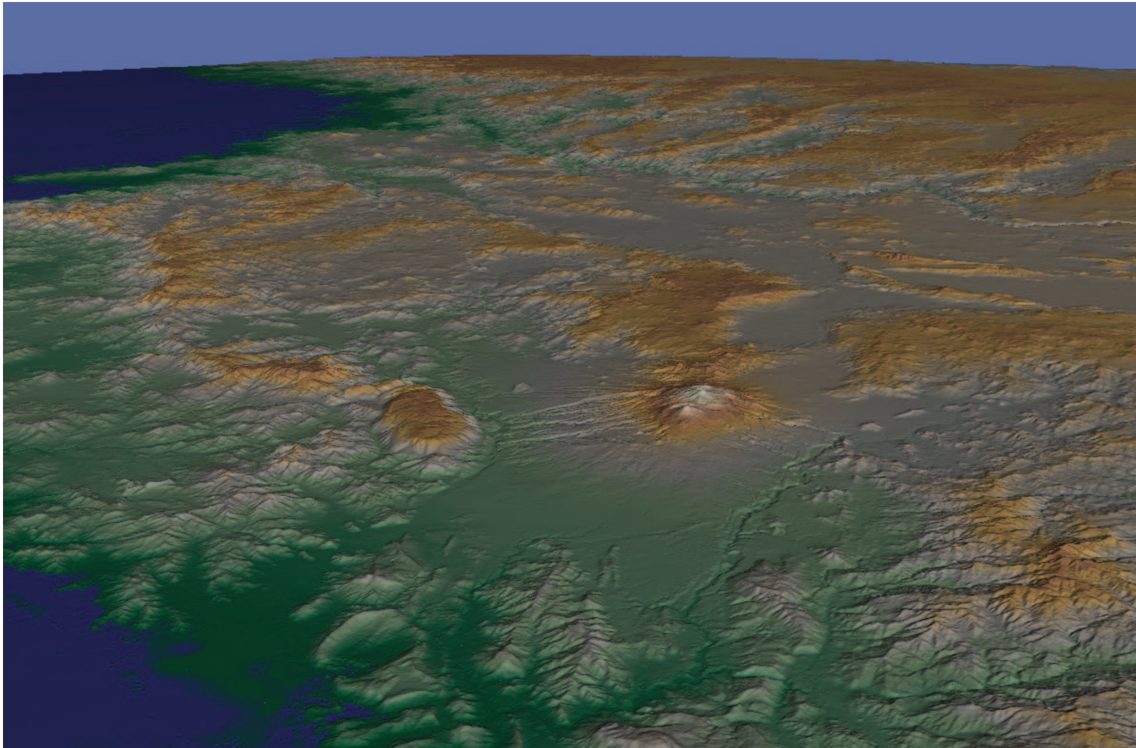


Figure 3.1: **View of the Earth near Guadalajara.** This $29G$ samples sparse global dataset is compressed to $0.25bps$. At run-time, decompression and normal computation are performed incrementally, allowing interactive full-screen flights at video rates with pixel sized triangles.

to regions in a bintree hierarchy. Each region, called diamond, is formed by two triangular patches that share their longest edge. In BDAM, each patch is a general precomputed triangulated surface region. In the C-BDAM approach, however, all

patches share the same regular triangulation connectivity and incrementally encode their vertex attributes when descending in the multiresolution hierarchy. The encoding follows a two-stage wavelet based near-lossless scheme. The proposed approach supports both mean-square error and maximum error metrics allowing to introduce a strict bound on the maximum error introduced in the visualization process. The scheme requires storage of two small square matrices of residuals per diamond, which are maintained in a repository. At run-time, a compact in-core multiresolution structure is traversed, and incrementally refined or coarsened on a diamond-by-diamond basis until screen space error criteria are met. The data required for refining is either retrieved from the repository or procedurally generated to support runtime detail synthesis. At each frame, updates are communicated to the GPU with a batched communication model.

The structure provides a number of benefits: overall geometric continuity for planar and spherical domains, support for variable resolution input data, management of multiple vertex attributes, efficient compression and fast construction times, ability to support maximum-error metrics, real-time decompression and shaded rendering with configurable variable level-of-detail extraction, and runtime detail synthesis (see Figure 3.1). As highlighted in Section 2.4, while other techniques share some of these properties, they typically do not match the capabilities of our method in all of the areas.

Our approach adopts the philosophy of the BDAM breed of techniques, that we summarize in Section 3.2. The new approach for efficiently encoding planet sized datasets is described in Section 3.3. Section 3.4 illustrates the pre-processing of digital elevation models, while Section 3.5 is devoted to the presentation of the run-time rendering algorithm. The efficiency of the method has been successfully evaluated on a number of test cases, including the interactive visualization of the Earth created from 3 arcsec SRTM data (Section 3.6).

3.2 Batched Dynamic Adaptive Meshes

A multiresolution surface model supporting view-dependent rendering must encode the steps performed by a mesh refinement or coarsening process in a compact data structure from which a virtually continuous set of variable-resolution meshes can be efficiently extracted.

The BDAM approach is a specialization of the more general Batched Multi-Triangulation framework [22], and is based on the idea of moving the grain of the multiresolution surface model up from points or triangles to small contiguous portions of mesh. BDAM exploits the partitioning induced by a recursive subdivision of the input domain in a *hierarchy of right triangles*. The partitioning consists of a binary forest of triangular regions, whose roots cover the entire domain and whose

other nodes are generated by triangle bisection. This operation consists in replacing a triangular region σ with the two triangular regions obtained by splitting σ at the midpoint of its longest edge (see Figure 3.2). To guarantee that a conforming mesh (i.e. without T-junctions) is always generated after a bisection, the (at most) two triangular regions sharing σ's longest edge are split at the same time. These pairs of triangular regions are called diamonds and cover a square. In this multiresolution schema a diamond represents the basic item of the hierarchy. Figure 3.3 illustrates the parental relationship of a diamond and their children at the next finer level of the hierarchy: the resulting dependency graph encoding the multiresolution structure is a DAG with at most two parents and at most four children per node.

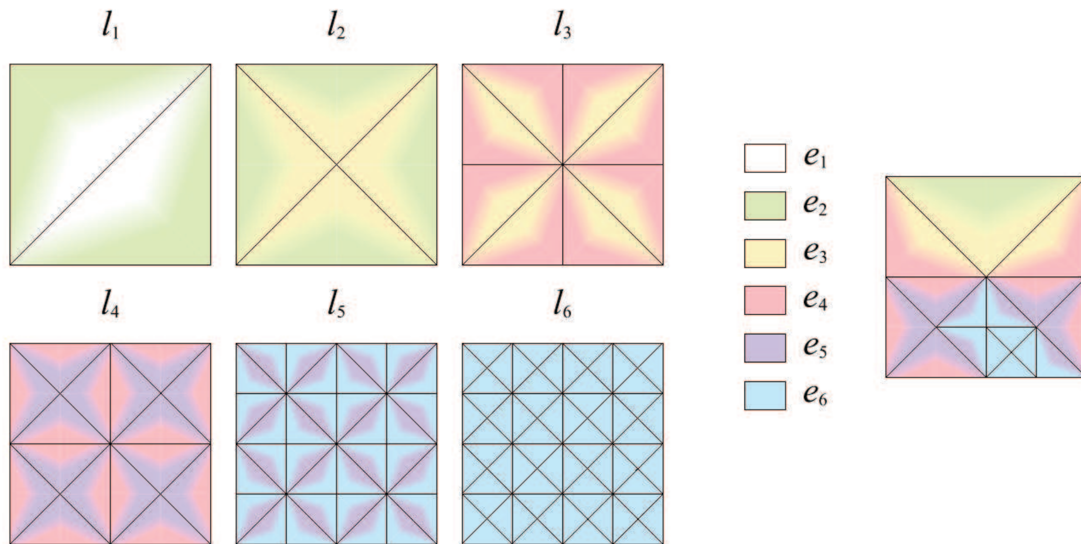


Figure 3.2: **BDAM right triangles hierarchy.** The initial diamond (composed by two triangular regions) at level 1 (l_1) of the hierarchy is subdivided five times ($l_{2..6}$). Here, each triangular region represents a terrain patch composed by many triangles (a *triangles batch*). The color–coded error values e_i show how the error smoothly varies inside each patch.

This structure has the important property that, by selectively refining or coarsening it on a diamond by diamond basis, it is possible to extract conforming variable resolution mesh representations. BDAM exploits this property to construct a coarse grained level-of-detail structure for the surface of the input model. This is done by associating to each triangle region a small tessellated patch, up to a given triangle count, of the portion of the mesh contained in it. Each patch is constructed so that vertices along the longest edge of the region are kept fixed during a diamond coarsening operation (or, equivalently, so that vertices along the shortest edge are kept fixed when refining). In this way, it is ensured that each collection of small

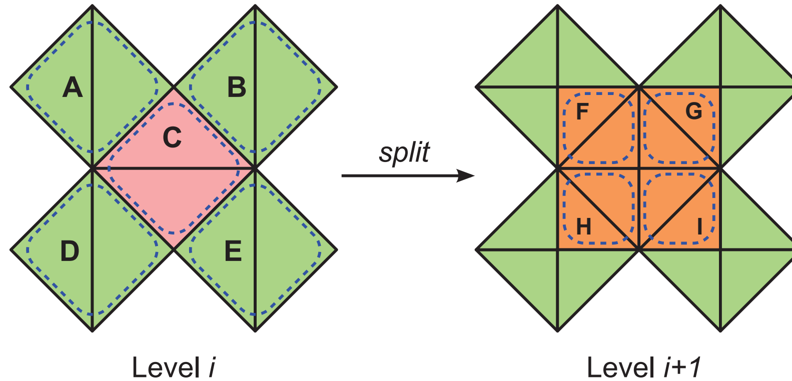
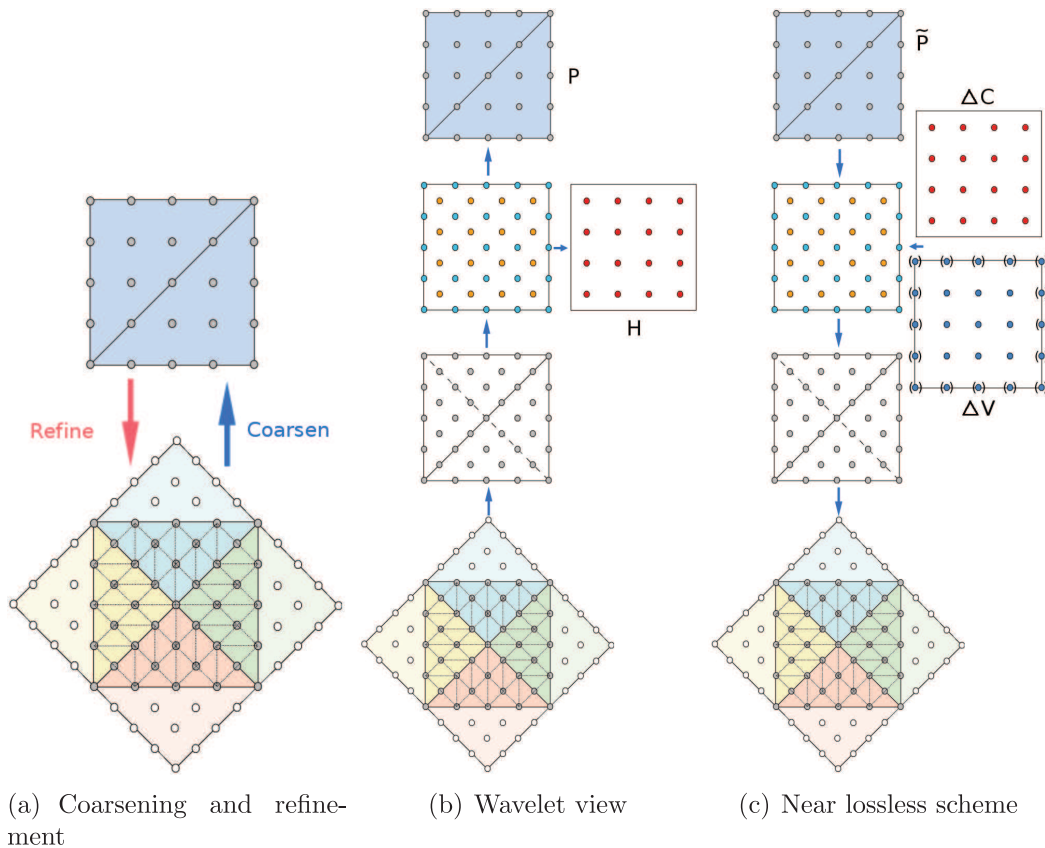


Figure 3.3: **BDAM diamond subdivision.** New diamonds are created after a split along the longest edge of the two triangular regions. Each diamond is split into four right-triangular regions, each forming a diamond with adjacent child regions. This implies that a diamond has two parents (except for the regions at the border of the dataset). More precisely, the resulting hierarchy is a DAG in which each node has at most two parents and at most four children. For example, diamond C has diamonds F, G, H and I as children and diamond F has diamond A and C as parents.

patches arranged as a correct hierarchy of triangular regions generates a globally correct and conforming surface triangulation. These properties are exploited in [21] to define an efficient parallel simplification method in which patches composing diamonds at levels of resolution matching the input data are sampled, while coarser level patches contain TINs constructed by constrained edge-collapse simplification of child patches.

3.3 Compressing BDAM

In order to construct a BDAM hierarchy, three operations are required: original data sampling, diamond coarsening (simplification), and its reciprocal, diamond refinement. At diamond coarsening time, data is gathered from child patches and simplified while keeping the diamond boundary fixed. Diamond refinement has to undo the simplification operation, pushing new patches into child diamonds. As noted elsewhere [81, 55, 22], given current hardware rendering rates, exceeding the hundreds of millions of triangles per second, it is now possible to interactively render scenes with approximately one triangle/pixel thresholds. At this point, controlling triangle shapes during simplification to reduce triangle counts is no longer of primary importance, and it is possible to work on regular grids rather than on irregular triangular networks, replacing mesh simplification/refinement with digital signal processing operations.



(a) Coarsening and refinement (b) Wavelet view (c) Near lossless scheme

Figure 3.4: **BDAM on a regular grid.** We recast diamond processing in the framework of wavelet update lifting. In this framework, diamond coarsening and refinement are associated to wavelet analysis and synthesis.

Wavelet Transformation. The structure of a BDAM mesh, when operating on regular grids, is depicted in Figure 3.4(a). As we can see, the two patches that form a diamond have vertices placed on a uniform square grid, while the vertices of the refined patches are placed at the centers of the square cells of the grid. To support compression, we recast diamond processing in the framework of wavelet update lifting [60], with the purpose of transforming data into a domain with a sparser representation. In this framework, diamond coarsening is associated to wavelet analysis, while diamond refinement corresponds to wavelet synthesis (see Figure 3.4(b)).

The analysis process for constructing a level l diamond starts by gathering vertex data from child diamonds at level $l+1$ (or from the input dataset) into a square matrix $P^{(l+1)}$, which is then decomposed into two square matrices: a matrix $V^{(l+1)}$ of vertex centered values, and a matrix $C^{(l+1)}$ of cell centered values. A new set of vertex values $P^{(l)}$ and a matrix of detail coefficients $H^{(l)}$ can then be computed by

the following low- and high-pass filtering operations:

$$P_{ij}^{(l)} = \alpha_{ij} V_{ij}^{(l+1)} + (1 - \alpha_{ij}) \mathcal{P}_{ij}^{(V)}(C^{(l+1)}) \tag{3.1}$$

$$H_{ij}^{(l)} = C_{ij}^{(l+1)} - \mathcal{P}_{ij}^{(C)}(P^{(l)}) \tag{3.2}$$

Here, $\mathcal{P}_{ij}^{(A)}(B)$ are prediction operators that predict the value at $A(i,j)$ from the values in matrix B, and $0 < \alpha_{ij} \leq 1$ weights between smoothing and pure subsampling. To comply with BDAM diamond boundary constraints, it is sufficient to set $\alpha_{ij} = 1$ for all diamond boundary vertices. Even though weighting could be data dependent (see, e.g., [96]), we assume for speed and simplicity reason a constant weighting of $\alpha_{ij} = \frac{1}{2}$ for all inner points. For prediction, we use a order 4 Neville interpolating filter if all support points fall inside the diamond, and a order 2 filter otherwise [71]. These filters predict points by a weighted sum of 12 (order 4) or 4 (order 2) coefficients, and are therefore very fast to compute (see Figure 3.5).

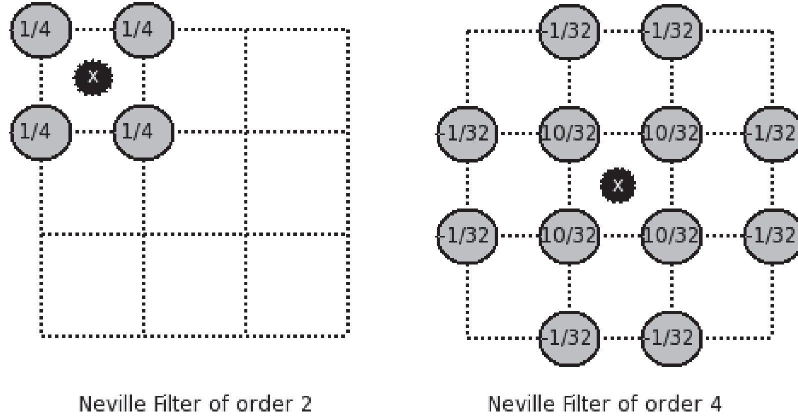


Figure 3.5: **Neville filters.** The filters predict points by a weighted sum of 12 (order 4) or 4 (order 2) coefficients.

By iterating the analysis process from leaf diamonds up to the roots, all the input data gets filtered up to the coarsest scale. The resulting wavelet representation is multiscale, as it represents the original terrain dataset with a set of coarsest scale coefficients associated to the root diamonds, plus a set of detail coefficients with increasingly finer resolution. During synthesis it is possible to produce variable resolution representations by refining a diamond at a time, using a process that simply reverses the analysis steps. At diamond refinement time, first vertex and face centered values are computed from the diamond vertex values:

$$C_{ij}^{(l+1)} = H_{ij}^{(l)} + \mathcal{P}_{ij}^{(C)}(P^{(l)}) \tag{3.3}$$

$$V_{ij}^{(l+1)} = \frac{P_{ij}^{(l)} - (1 - \alpha_{ij}) \mathcal{P}_{ij}^{(V)}(C^{(l+1)})}{\alpha_{ij}} \tag{3.4}$$

Then, this data is reassembled and scattered to child diamonds.

Lossy Data Compression. For typical terrain datasets, the wavelet representation produces detail coefficients that decay rapidly from coarse to fine scale, as most of the shape is captured by the prediction operators. By entropy coding the detail coefficients it is thus possible to efficiently compress the input dataset. The achievable lossless compression ratio is however limited. More aggressive lossy compression can be achieved by quantizing detail coefficients or discarding small ones. Quantizing the wavelet coefficients to meet a maximum error criterion is however a complex problem, and while there are many excellent wavelet-based lossy compression schemes under the L^2 norm, none of them can offer a tight bound on the maximum reconstruction error of each value, since it is difficult to derive meaningful relations between distortions in the wavelet domains and in the signal domain in the L^∞ sense [120]. Using a L^2 norm for general terrain management applications is hardly acceptable, since reconstruction errors are averaged over the entire domain and results suffer from high variance in the quality of data approximation. Therefore, using a L^2 norm can severely bias reconstructions in favor of smooth regions, and does not offer error guarantees for individual approximate answers to variable resolution queries. In C-BDAM, we solve the problem using a simple two-stage near-lossless scheme that ensures that each diamond is within a given maximum value from the original filtered data. In contrast to other two stage schemes from the signal processing literature [120], we thus keep under strict control each level's error, and not only the finest level reconstruction, in order to avoid view-dependent rendering artifacts.

In our two-stage scheme, we first compute a full wavelet representation fine to coarse. We then produce a quantized compressed representation in a diamond-by-diamond coarse to fine pass that, at each step corrects the data reconstructed from quantized values by adding a quantized residual. For a diamond at level l, the residuals required for refinement are thus computed as follows:

$$\Delta\tilde{C}_{ij}^{(l+1)} = Quant_\epsilon(\mathcal{P}_{ij}^{(C)}(P^{(l)}) + H_{ij}^{(l)} - \mathcal{P}_{ij}^{(C)}(\tilde{P}^{(l)})) \quad (3.5)$$

$$\tilde{C}_{ij}^{(l+1)} = \mathcal{P}_{ij}^{(C)}(\tilde{P}^{(l)}) + \Delta\tilde{C}_{ij}^{(l+1)} \quad (3.6)$$

$$\Delta\tilde{V}_{ij}^{(l+1)} = Quant_\epsilon\left(\frac{P_{ij}^{(l)} - \tilde{P_{ij}}^{(l)} - (1-\alpha_{ij})(\mathcal{P}_{ij}^{(V)}(C^{(l+1)}) - \mathcal{P}_{ij}^{(V)}(\tilde{C}^{(l+1)}))}{\alpha_{ij}}\right) \quad (3.7)$$

$$\tilde{V}_{ij}^{(l+1)} = \frac{\tilde{P_{ij}}^{(l)} - (1-\alpha_{ij})(\mathcal{P}_{ij}^{(V)}(\tilde{C}^{(l+1)}))}{\alpha_{ij}} + \Delta\tilde{V}_{ij}^{(l+1)} \quad (3.8)$$

where $Quant_\epsilon(x)$ is a uniform quantizer with step size 2ϵ. This scheme requires storage of two small square matrices of residuals per diamond, which are maintained in a data repository indexed by diamond id. For a diamond of N^2 vertices, matrix $\Delta\tilde{C}$ is of size $(N-1)^2$, while matrix $\Delta\tilde{V}$ is of size $(N-2)^2$, since, due to BDAM diamond graph constraints, all boundary values are kept constant (we use a point sampling filter), and corrections are therefore implicitly null.

3.4 Out-of-Core Construction of Massive Variable Resolution Terrain Datasets

The off-line component of our method constructs a multiresolution structure starting from a collection of input terrain datasets, a required diamond size, and a required maximum error tolerance.

Diamond Graph Construction. The first construction phase – graph construction – generates a diamond DAG, whose roots partition the input domain and whose leaves sample the dataset at a finer or equal density to that of the input dataset. The graph is constructed coarse to fine, by refining the diamond graph a diamond at a time until sample spacing is sufficiently fine. This process makes it possible to efficiently process variable resolution terrain datasets (see Figure 3.6). In our implementation, the input dataset is described by a hierarchy of layers, ordered by resolution, where each layer contains a number of tiles. For graph construction, each diamond provides the input dataset with the range of coordinates it covers, and receives as a result the sample spacing of the finest resolution tile contained in it. If this number is smaller than the diamond's sample spacing, the diamond is refined.

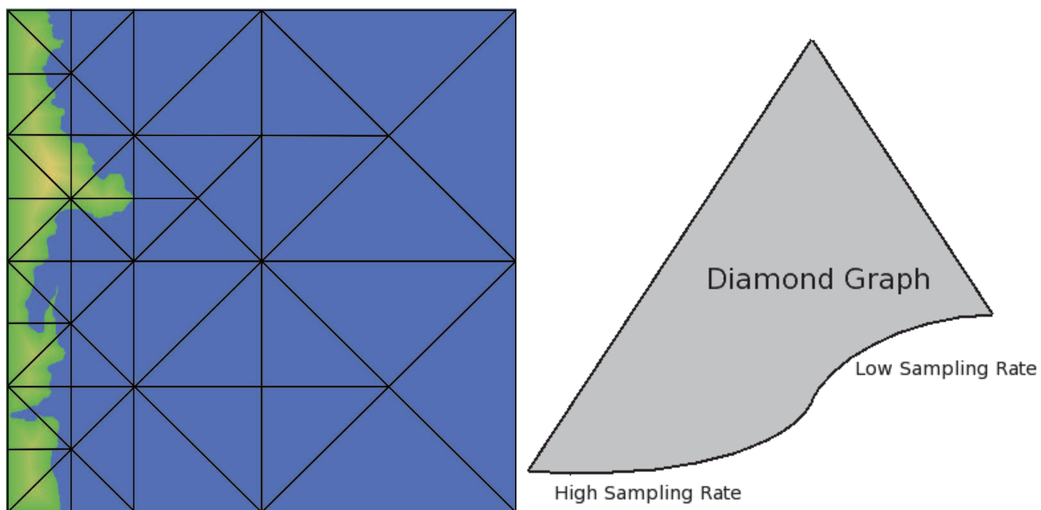


Figure 3.6: **Variable resolution input.** Diamond graph roots partition the input domain, while leaves leaves sample the dataset at a finer or equal density to that of the input dataset. In this simple example of a sparse input dataset covering emerged land, leafs are denser where data exists (terrain area) and rapidly decrease density where there is no data (blue area).

Fine to Coarse Filtering. Once the graph is constructed, it is traversed fine to coarse to construct a filtered representation by wavelet analysis (equations 3.1

and 3.2). For each filtered diamond, we store in a out-of-core repository both the filtered data $P^{(l)}$ and the detail coefficients $H^{(l)}$. Storing both matrices per diamond allows us to locally have all the information required to compute the reference values in the compression step. When sampling variable resolution input datasets at leaf diamonds, a value is always returned from the higher resolution tile that contains the query coordinate. A null value (typically 0) is returned when data is missing. An alternative approach would be to define transition areas and return weighted averages in case of overlapping tiles.

Coarse to Fine Compression. Once the filtered data repository is created, the root diamond values are copied to the output repository, and the diamond graph is then traversed coarse to fine to generate the final compressed representation of residuals. At each visited diamond, the reconstructed values of its parent diamonds are retrieved from a auxiliary repository updated while compressing, while the filtered values are retrieved from the repository generated in the first pass. Equations 3.5 to 3.8 are used to determine from this data the reconstructed vertex values for the diamond, as well as the residual matrices required to generate children. The relevant temporary and output repository are then updated, and the process continues until all diamonds are visited.

Boundary Management. For non-global datasets, some of the diamonds fall on a boundary and are therefore incomplete. This would lead, in general, to deal, during both filtering and compression, not only with square coefficients matrices, but also with triangular ones. To simplify processing, we choose instead the approach of synthetically generating missing data by a symmetric extension (mirroring along the boundary). This data is only virtual, and will never be used for rendering, since no patches will be generated for fully out-of-bounds children.

Out-of-Core Parallel Construction. The whole construction process is inherently parallel, because the grain of the individual diamond processing tasks is very fine and synchronization is required only at the completion of each level. In our parallel implementation, a coordinator process traverses all the diamonds in the required order (coarse to fine for filtering, and fine to coarse for compression) and distributes the diamond processing job to a number of workers, which execute it and send the result back to the coordinator for updating the repository and generating output file. Load balancing is ensured if the coordinator initially seeds each worker with a single diamond and subsequently always sends a new job request to the worker that sent back a result. The generation of the output file in the correct order can be ensured with a small buffer for handling out-of-sync output requests. In this solution, the main memory required for each worker is that required for processing a single diamond, while the coordinator simply needs to reserve enough memory for holding out-of-sync requests in addition to the memory required to store

the structure of the diamond graph, which is orders of magnitude smaller than the input data, since each diamond represents thousands of samples.

3.4.1 Compressed Data Representation.

The compression process generates a set of square matrices of integer coefficients that have to be compactly stored in the repository. These matrices are expected to be sparse and mostly composed of very small numbers. A number of efficient entropy coding methods for such matrices have been presented in the image compression literature. For our work, we have adopted a very simple storage scheme that, even though it is far from being optimal, it has the advantage of being straightforward to implement and extremely fast to decompress. In this scheme, the matrix is interpreted as a quadtree and recursively traversed in depth first order until a sub-matrix contains only zeros or is smaller than 2×2 in size. A single bit per split is used to encode the quadtree structure. At non-empty leaves, the coefficients are mapped to positive integers and encoded using a simple Elias gamma code [41], in which a positive integer x is represented by: $1 + \lfloor \log_2 x \rfloor$ in unary (that is, $\lfloor \log_2 x \rfloor$ 0-bits followed by a 1-bit), followed by the binary representation of x without its most significant bit.

Managing Multiple Vertex Attributes. Quite often, other vertex attributes need to be mapped over geometry. In our framework, we interpret these attributes as separate layers, and assume that different repositories are created independently for each of the attributes. In the implementation we support elevation, represented as a scalar value, and color, represented in input as a RGB triple. For color compression, we map colors to the YCoCg-R color space [83] to reduce correlation among components, and then compress each component separately. Normals are neither stored in the repository nor passed to the GPU, but directly computed on demand by finite differences from elevation data.

3.5 Rendering a C-BDAM

Our adaptive rendering algorithm works on a standard PC, and preprocessed data is assumed to be either locally stored (in memory or on a secondary storage unit directly visible to the rendering engine) or remotely stored on a network server (see Figure 3.7).

3.5.1 Data Access and Cut Selection

The result of pre-processing is stored in repositories that contain root diamond values and residual matrices for each diamond. We assume that a main repository contains

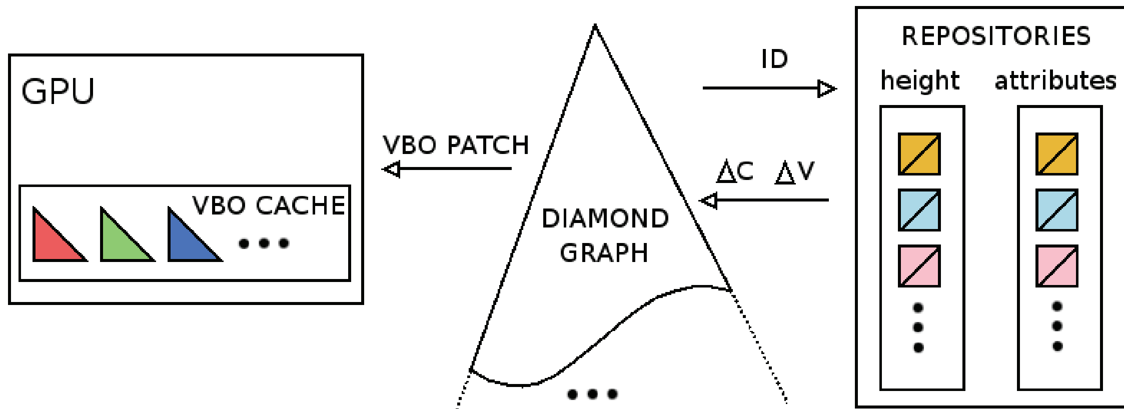


Figure 3.7: **The rendering pipeline.** The renderer accesses compressed data through a data access layer that hides whether data is local or remote.

elevation values, while an optional secondary repository contain color values. The DAG hierarchy itself resides in core memory.

Data Layout and Fetch. The repositories are required to cover the same domain and to use the same diamond patch granularity but can be, and generally will be, of different depth, as the original data is typically provided at different resolutions. Access to a data repository is made through a data access layer, that hides from the renderer whether data is local or remote. This layer internally uses memory mapping primitives for local data, and a TCP/IP protocol for remote data.
We chose TCP instead of UDP for two main reasons. First, the limited size of a UDP packet (typically 1.5KBytes) implies that a single patch must be split into multiple chunks that need to be eventually reassembled in the correct order at the client side, mimicking an important phase natively implemented in the TCP protocol. Secondly, in a multiresolution visualization system, contrarily to audio/video streaming applications, the packet loss events that occur during a UDP transfer session have great impact on the underlying algorithms and, most of all, on the data layer implementation: after sending a patch request to the remote data server, the client would not know whether missing data has been lost or it is still in transit, implying that the data layer should continue issuing requests at regular intervals. Nonetheless, managing packet loss in an efficient way is already one of the main purposes of the TCP protocol, even at the cost of additional control packets to be exchanged and whose overhead (a few bytes) is very small compared to the patch size.
By using a reliable network transfer protocol and secondary storage devices such as hard disks, the developed data access layer makes it possible to asynchronously move in-core a diamond's associated data by fetching it from the repository, to test whether a diamond's data is immediately available, and to move available data to

RAM.

Refinement Algorithm. For the sake of interactivity the multiresolution extraction process should be able to support a constant high frame rate, given the available time and memory resources. This means that the algorithm must be able to fulfill its task within a predetermined budget of time and memory resources, always ending with a consistent result, or in other words, it must be interruptible. Our extraction method uses a time-critical variation of the dual-queue refinement approach [39] to maintain a current cut in the diamond DAG by means of refinement and coarsening operations. For this purpose we store the set of operations that are feasible given the available budget and compatible with the current cut, in two heaps: the *CoarseningHeap* and the *RefinementHeap*, with a priority dictated by the screen space error. The heaps are reinitialized at each frame, given the current cut and the current view parameters. When initializing the refinement heap, only operations for which data is immediately available are considered. In case of missing data, an asynchronous request is posted to the data access layer, which will be responsible of fetching the needed patches without blocking the extraction thread. The renderer maintains in-core all nodes of the diamond graph above the current cut, in a streamlined format suitable for communicating with the GPU and for quickly evaluating visibility and screen space error.

For each node, we maintain an object space error, the diamond corner information, links to parents and children, an oriented bounding rectangle, and one set of fixed size vertex arrays for each of its two patches. Each time a refinement or coarsening operation is performed, the current cut is updated. For coarsening, this simply amounts to freeing the unreferenced memory, while for refinement children data must be constructed, starting from current vertex data and detail coefficients retrieved from the repositories, or, optionally, procedurally generated (see Figure 3.8). In our implementation, all the decompression steps are performed on the CPU. The fact that we are able to rapidly generate details is exploited to render scenes where color and elevation are not available at the same resolution in the repository. In that case, data is fetched from repository where available, and generated otherwise. Moreover, the fact that coarsening operations are basically no cost is exploited to support non-monotonic error metrics in a time-critical setting. To that end, the extraction algorithm performs refinement operations until the graph is locally optimal or a time-out occurs, while coarsening has the role of a garbage collector and is performed only once in a while to remove unneeded detail.

View-Space and Object-Space Errors. As for other graph based methods, the presented technique is not limited to using distance based LODs, but can employ configurable data-dependent metrics. In the examples presented in this chapter, we use oriented bounding boxes as bounding volume primitives, and we simply use the average triangle areas as a measure of object space error. Thanks to the

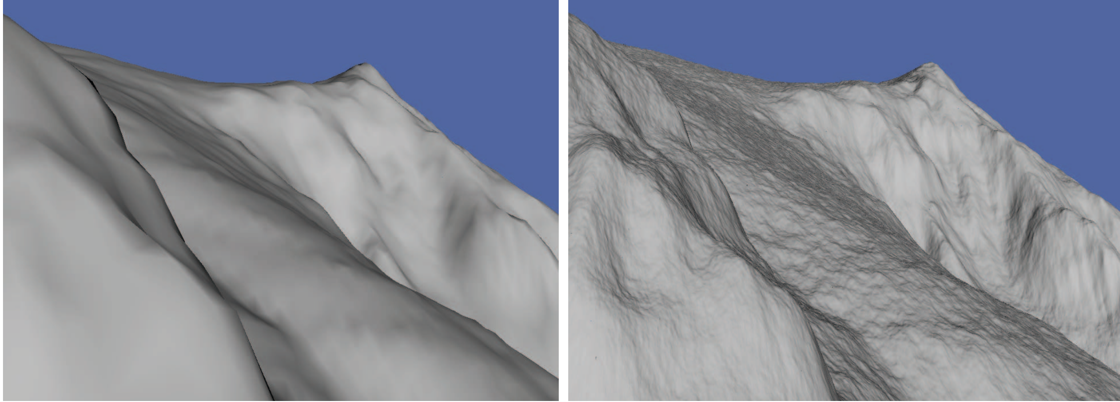


Figure 3.8: **Procedural Detail Synthesis.** On the left, refinement stops when no more data is available in the repository, i.e., when reaching the input datasets original sampling density; on the right, artificial detail is synthesized by generating procedural detail coefficients (in this case, simply a zero-centered uniform random number less than 30% of triangle edge length for ΔV, and 0 for ΔC.

computation of the $L^{\inf}$ error during compression, and, given the relatively modest size of the diamond graph, the use of elaborate metrics at run-time does not slow down the rendering process. Bounding boxes are computed on-the-fly each time a diamond is updated with a new patch during a refine. View space errors are estimated by dividing the projected size of the box on the screen by the number of triangles contained in a diamond. Using such data depended measures provides higher quality results than the simple distance based techniques that have to be employed in pyramidal schemes (see Figure 3.9).

3.5.2 Rendering: Elevation, Colors, and Normals

When constructing a given node, data is required for elevation, and optionally for color and normals. Since our target is to render very small (pixel sized) triangles, texture maps offer limited benefits, since no attribute interpolation would be employed anyway. In the most simple implementation, we have thus the option of managing everything using per-vertex attributes. In addition, since vertices are tightly packed, the normals required for shading can be effectively computed at refinement time from vertex elevation rather than managed in an external repository. Consistently with BDAM diamond constraints, boundary normals are kept fixed, while the others are computed by central differences from elevation data. Communication with the GPU is made exclusively through a retained mode interface, which reduces bus traffic by managing a least-recently-used cache of patches maintained on-board as OpenGL *Vertex Buffer Object*. GPU memory and bandwidth costs are further reduced by compactly coding vertex data, and using vertex shaders to perform decompression. All data is specified in local (u, v) patch coordinates. Since we are using regular

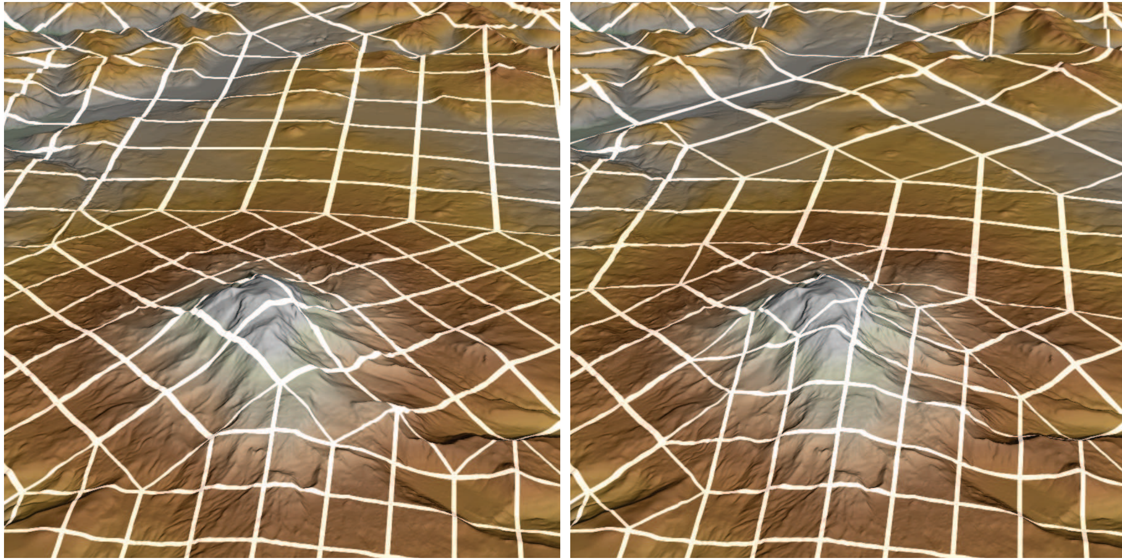


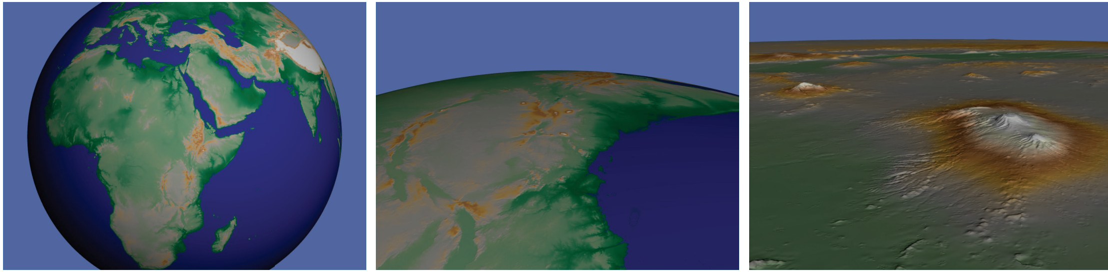
Figure 3.9: **View-space error control.** A distance based error metric (left) is less efficient in distributing detail than a data dependent measure based on projected diamond size (right).

grids, (u, v) locations as well as triangle strip indices are shared among all patches and stored only once in GPU memory. A single per patch 3-component vector stores elevation $h(u, v)$ and its derivatives dh/du and dh/du, from which positions and normals are computed on the GPU. In a texture-less approach, a second optional vertex array stores color in RGB8 format. A texture-based approach can provide a better anisotropic filtering at the cost of higher implementation complexity.

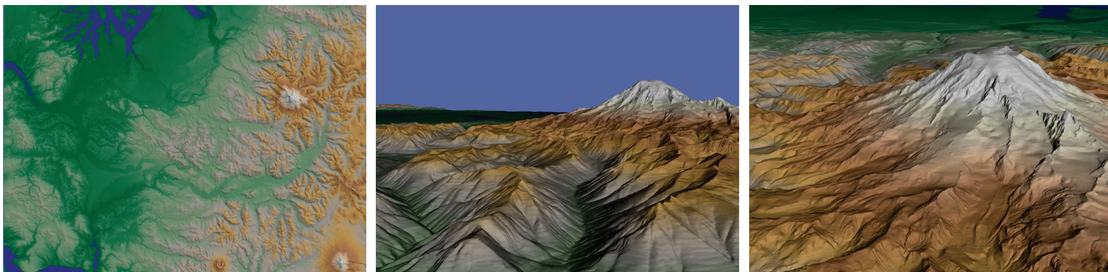
3.6 Implementation and Results

An experimental software library and a rendering application supporting the C-BDAM technique have been implemented on Linux using C++ with OpenGL and NVIDIA Cg. We have extensively tested our system with a number of large terrain datasets. The discussed results have been obtained on three terrain datasets with different characteristics (see Figure 3.10).

The main dataset is a global reconstruction of the planet Earth from SRTM data at 3 arcsec resolution (one point every $90m$ at the Equator. Source: CGIAR Consortium for Spatial Information. *srtm.csi.cgiar.org*). The dataset is very large (29 billion samples) and provides an example of variable resolution planetary data, as it is described by a sparse set of tiles providing data only for emerged land in the 66S-66N latitude range. The two other models are local terrain datasets. The first dataset is the standard 265 million samples Puget Sound 10m DTM dataset presented here for the sake of comparison with previous work (source: USGS and



(a) SRTM 3arcsec Earth (Elevation: 29G samples (sparse); Color: 2D lookup-table indexed by latitude and elevation)



(b) Puget Sound 10m resolution (Elevation: 265M samples); Color: 1D lookup-table indexed by elevation)

(c) Paris (Elevation 67M samples; Color 872M samples)

Figure 3.10: **Inspection sequences: selected frames.** All images were recorded live on a AMD 1.8 GHz PC with 2 GB RAM and PCI Express NVIDIA GeForce 7800GT graphics using pixel sized triangles.

Peter Lindstrom. *www.cc.gatech.edu/projects/large_models/ps.html*). The second one is a 67 million samples dataset covering the city of Paris at 1 $sample/m$, textured with an high resolution 872 million samples ortho-photo (source: InfoTerra. *www.infoterraglobal.com*). The Paris elevation dataset is a worst case situation, as it is a model derived by sampling at 1m spacing a vector representation of building outlines, and, as such contains lots of discontinuities. The texture is a typical city-scale high resolution photograph, and is here to demonstrate the ability to handle color information within our framework.
In Chapter 4 we will discuss how to deal with the structural discontinuities introduced by the presence of buildings.

3.6.1 Preprocessing

All the preprocessing tests were executed on a single PC running Linux 2.6.15 with two Dual Core AMD Opteron 1.8 GHz processors, 2 GB of RAM, SATA 10000 rpm 150 GB hard drive.

	Sample #	XY Step	Tolerance	Time	Output Size	bps	Rate
Puget Sound	265 M	10 m	1 m RMS	11 m	9.1 MB	0.28	57:1
Puget Sound	265 M	10 m	1 m AMAX	10 m	19.2 MB	0.61	26:1
Earth SRTM	29 G	90 m	16 m AMAX	30 h 38 m	869.9 MB	0.25	64:1
Paris	67 M	1 m	0.1 m AMAX	6 m	14.6 MB	1.80	9:1
Paris color	872 M	0.25 m	10/255 AMAX	1 h 17 m	254.2 MB	2.45	10:1

Table 3.1: **Numerical results for terrain preprocessing.** Preprocessing results for tests datasets.

Table 3.1 lists numerical results for our out-of-core preprocessing method for the Puget Sound and the Earth datasets. We constructed all multiresolution structures with a prescribed diamond dimension of 64×64 vertex side which gives a good granularity for the multiresolution structure. Each triangular patch is composed by 4K triangles. Preprocessing time is quite equally subdivided into the filtering and the encoding phases (see Section 3.4). In all the results here presented we make use of four threads (one per CPU core), with a speed-up of 3X with respect to the sequential version. The sub-linear speed-up is due to the fact that time is dominated by I/O operations on the input, temporary and output files. We expect a performance increase when distributing them on multiple disks. Processing speed ranges from 260K to 450K input samples per second, more than 7 time faster than P-BDAM [21]. This is because regular grid filtering and compression is faster than general mesh simplification. Still, processing speed is about 4 times slower than 4-8 hierarchies[54], which, however, does not perform compression, and about 2 times slower than Geometry Clipmaps [81], which, however, generates half the number of resolution levels.

Compression Rates. The compression rates are presented in table 3.1. The tolerances used for compressing were chosen to provide near-lossless results. In the Puget Sound and in the Paris dataset the chosen tolerances correspond to 1% percent of the sample spacing, while in the Earth dataset the tolerance is $16m$, which corresponds to the vertical accuracy of the input dataset. For the color dataset, we imposed a maximum error tolerance of 10/255 per component, which is comparable with the error introduced by the S3TC compression algorithm, commonly used in terrain rendering applications (see, e.g, [19]).

The three datasets have been preprocessed using maximum absolute error tolerance to drive the compression. For the sake of comparison with previous work based on Geometry Clipmaps [81], we also compressed the Puget Sound dataset using

RMS error control. In this case, our result of $0.28bps$ is comparable with a Geometry Clipmap [81] representation of the same dataset ($0.26bps$), especially since we provide the double of resolution levels. It is interesting to note that the maximum error in this case goes up to $18m$, showing the inherent lack of local control when using a L^2 norm to drive compression. When using a maximum error tolerance, the bit rate increases to $0.61bps$, but all points are guaranteed to be within $1m$ from the original.

The bit rate for planet Earth is even better ($0.25bps$), even though compression rates are given considering the whole amount of memory occupied by the stored dataset versus the number of samples present in the leaves. The Earth dataset is sampled in the poles area and over the sea, where there is no input data, at the minimum resolution imposed by the continuity constraints of the multiresolution structure. Sampling rate in this variable resolution input test case varies from $53m$ to $23Km$, since our graph adapts to the sparseness of the input grid. This way, compression rates appear worse than what they would be when using a full resolution input of constant elevation for flat areas.

To test the performance in an extreme case, we applied our compressor to the Paris elevation dataset. The bit rate increases to $1.80bps$, which illustrates that the method works much better for terrains which are locally smooth. Compression results for the texture are similar, since texture in this case is also dominated by very sharp boundaries, due to buildings and their shadows. Nonetheless, the bit rate for the Paris dataset is still considerably better than what reported for uncompressed terrain representations, which are typically at least one order of magnitude larger [81]. Handling discontinuities, e.g., through adaptive lifting [96] or a geometric bandelet approach [95], is a promising avenue of future work.

3.6.2 Adaptive Rendering

The rendering tests were executed on a medium end single PC running Linux 2.6.15 with single AMD 1.8 GHz CPU, 2 GB of RAM, SATA 10000 RPM 150 GB hard drive and PCI Express NVIDIA GeForce 7800GT graphics. These tests were performed with a window size of 1280x1024 and a target of pixel-sized triangles.

We evaluated the rendering performance of the technique on a number of flythrough sequences on all the terrain datasets. The sessions were designed to be representative of typical flythrough and to heavily stress the system, and include abrupt rotations and rapid changes from overall views to extreme close-ups. In addition to shading the terrain, we apply a color, coming from an explicit color channel (Paris dataset), a 1D look-up table indexed by elevation (Puget Sound), or a 2D look-up table indexed by elevation and latitude (Earth).

The average frame rates is around 90Hz, while the minimum frame rate never goes below 60Hz using a 1280x1024 window. For all planar datasets, the average triangle throughput is $130M\Delta/s$, and its peak performance is $156M\Delta/s$. In the flythrough of planet Earth, we achieve a average performance of $100M\Delta/s$ and a

peak of $125M\Delta/s$. The difference in performance between planar and spherical datasets is due to the different complexity in vertex shaders, which transforming the height information in a vertex position, and compute also the normal values starting from tangent information. These operations are simpler in the planar shader, than in the spherical shader. Normal computation is the most costly operation. With a simpler shader which computes only position and no shading information the maximum reachable performance is $190M\Delta/s$ in both cases. This increase in performance demonstrates that the technique is GPU bound. Even in the current implementation, the triangle rate is high enough to render over $4M\Delta/frame$ at interactive rates. It is thus possible to use very small pixel thresholds, effectively using pixel sized triangles, virtually eliminating popping artifacts without the need to resort to geomorphing techniques.

Network Streaming. Some network tests have been performed on all test models, on a ADSL 1.2Mbps connection using the TCP/IP protocol to access the data. The rendering rate remains the same as the local file version, only the asynchronous updates arrive with increased latency due to the network connections. Since only few diamonds per frame needs update, and diamond data is extremely compressed, the flythrough quality remains excellent.

Limitations of the Approach. The proposed method has also some limitations. As for geometry clipmaps [81], the compression method is lossy and assumes that the terrain has bounded spectral density, which is the case for typical remote sensing datasets. Moreover, as for all regular grid approaches, the rendered mesh has a higher triangle count than in the BDAM and P-BDAM schemes, which can exploit irregular connectivity to follow terrain features.

Chapter 4

A Multiresolution Representation for Urban Models

Virtual exploration of urban areas represents an interesting challenge in the computer graphics research community. The challenge originates from the difficulty to represent and render in real–time the very large amount of geometry and image data typical in urban environments. In particular, standard geometric multiresolution techniques exhibit problems when building low resolution representation of the original datasets due to the already simple structure of the entities (e.g. buildings) being simplified.
In this chapter we introduce the BlockMap, a GPU–friendly technique that exploits the nature of urban environments to ensure multiresolution rendering quality and real–time city exploration tasks.

4.1 Exploring Urban Areas

In the past few years we have witnessed a continuous growth of interest on the possibility to remotely explore in real–time a three–dimensional representation of the environment in which we live. This is due in part to the widespread adoption of relatively cheap but powerful 3D graphics accelerators that are able to display high detailed content, both in terms of geometry and image data. Even though, historically, virtual exploration applications have focused on textured digital elevation models, the interest is now rapidly shifting towards urban environments. Focusing on such environments is extremely important, since a large part of Earth's population lives and works in dense urban areas. Moreover, recent advances in city acquisition and modeling (e.g., specialized 3D scanning [114, 43, 25], assisted reconstruction from photos and video streams [99], or parametric reconstruction [93, 88]) lead to a rapidly increasing availability of highly detailed urban models.

Exploring large detailed urban models, seamlessly going from high altitude flight views to street level views, is extremely challenging. What makes urban models so

peculiar is that their geometry is made of many small connected components, i.e. the buildings, which are often similar in shape, adjoining, rich in detail, and unevenly distributed. Usually, each building is represented with a relatively small number of polygons but a large amount of color information, typically one photograph for each façade. Moreover, geometry is highly discontinuous and different views of the model have widely different depth complexity, ranging from full visibility of flyovers to nearly full occlusion at ground level.
While multiresolution texturing and geometric levels of detail may provide acceptable solution for buildings near to the viewer and moderate degrees of simplification, major problems arise when rendering clusters of distant buildings, In fact, several triangles and a large amount of input texels typically project to a single pixel, making joint filtering of geometry and texturing an important issue. Moreover, relying on surface meshes to drive an aggressive simplification process for distant urban areas is very complicated, as the triangles mapping to a given pixel may stem from disconnected surfaces, or totally different objects.

Therefore, state-of-the-art solutions propose switching to radically different, often image-based, representations for distant objects having a small, slowly changing on-screen projection. Typically, impostors are used to fulfill these needs. However, away from their supported set of views, impostors introduce rendering artifacts, such as parallax errors, disocclusions, or rubber sheet effects, which can only be reduced by introducing constraints on viewpoint motion or by radically increasing storage and transmission costs. The limitations of current techniques for simplification of distant geometry thus impose important scalability limits their usability. In general, this dual representation approach, increases implementation complexity, while introducing hard to manage bandwidth bottlenecks at representation changes.

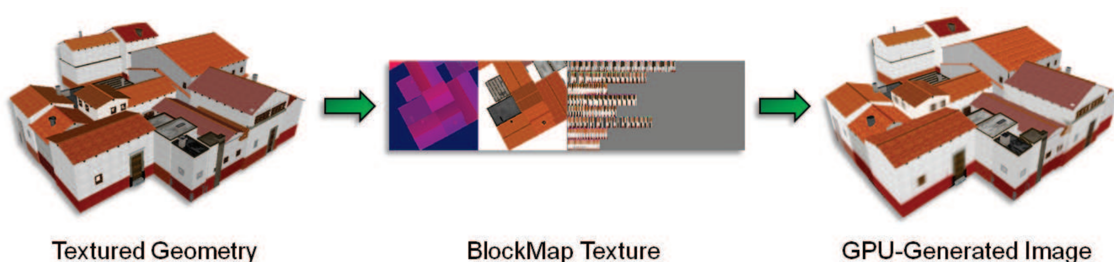


Figure 4.1: **The BlockMap Concept.** A *BlockMap* replaces groups of buildings (left) with a concise, texture-based representation (center), which is efficiently rendered by a GPU ray caster (right).

In this chapter, we introduce a network- and GPU–friendly simplified representation that efficiently exploits the highly structured nature of urban environments, and illustrate how it can be used to construct an efficient output-sensitive system. Central to our approach is a novel discrete representation for textured sets of build-

ings, that we call *BlockMap* [20, 34]. The BlockMap encodes both the geometry and the sampled appearance (e.g. color and normals) of a small group of buildings that exhibit a predominant 2.5D structure (see Figure 4.1). BlockMaps are stored into small, fixed size texture chunks, and can be efficiently rendered through GPU ray casting. Since BlockMaps are simplified representations of the original dataset, they provide full support to visibility queries, and, when built into a tree hierarchy, offer multiresolution adaptability. By working in a network setting, we also test the effectiveness of the BlockMap as a scalable output-sensitive LOD technique that is able to provide a bounded size representation for large urban scenes.

The main purpose of a rendering system based on BlockMaps is to visualize the most salient aspects of large urban environments as a whole, enhancing the information provided by modern exploration applications that allow to explore our environment in a very wide scale range. As discussed in Chapter 2, due to efficient and robust simplification algorithms that operate on dense models, state–of–the–art visualization methods allow smooth exploration of planet–sized dataset, as well as small–scale complex models (e.g. 3D–scanned objects). In this context, the proposed technique places itself between very high altitude navigations, above which building shapes are not perceivable, down to very low altitude flyovers, below which more detailed representations (e.g. high–resolution textured triangle meshes) need to be used. Moreover, the compact data representation allows BlockMaps to be used in situations where handling an large amount of detail is not feasible due to hardware restrictions, such as modern smartphones and tablets.

4.2 The BlockMap Representation

The goal of the BlockMap representation is to compactly and faithfully represent a portion of a urban model. For scalability reasons, this means that the representation should be able to rapidly render many buildings projecting to few pixels, with a memory and time complexity ideally depending only on the region screen footprint. An important characteristic of urban models is that a set of textured 2.5D shapes typically provides a good approximation of the geometry and appearance of a region, while preserving the large scale features, i.e. the silhouette and color of the buildings. This assumption holds not only because the vast majority of buildings have a prism-like shape, but also because coarse automated methods for ground-based acquisition of large-scale 3D city models impose geometric constraints on reconstruction to make the acquisition tasks more tractable.

Starting from this assumption, with BlockMaps we introduce a GPU–friendly representation that is able to handle a discrete height field with the hard discontinuities introduced by vertical surfaces (e.g. building façades). We exploit the regularity of this discrete description to obtain a very compact encoding and a fast rendering algorithm, and use the discretization step size to control reconstruction quality. The

simple structure and the *image–space–driven* data acquisition approach of this novel representation allows us to easily arrange several BlockMaps in a quadtree hierarchy to obtain a multiresolution version of the original urban environment.

4.2.1 Data Structure

The BlockMap representation stores in a rectangular portion of texture memory all the data required to represent a region of the input dataset enclosed in an axis–aligned bounding box.

The fundamental idea behind the BlockMap concept is that the shape must be effi-

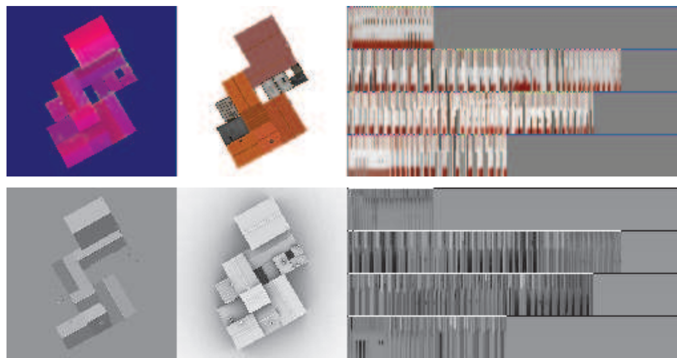


Figure 4.2: **The BlockMap Texture.** A standard 32–bit RGBA image is used to encode both shape and appearance information of a city block (RGB channels at top, while Alpha channel at bottom

.

ciently accessed and drawn by a GPU–based, heightmap ray casting algorithm that, depending on the hit location and the incident angle, will quickly retrieve surface attributes belonging to ground, building roofs and vertical façades and use them for lighting computations.

For this purpose, a BlockMap texture (see Figure 4.2) stores surface attributes such as color, normal for roofs/ground and visible building walls, as well as a heightmap to encode terrain elevation and the 2.5D shape of buildings.

In general, as we will show in Section 4.3.2, a BlockMap can virtually encode any surface attribute that the original dataset may expose, either local (e.g. specular roughness) or global. In particular, since we aim to enhance shape perception during visualization, we include in our structure information that allows us to adopt a shading model based on the Ambient Occlusion term [74], which approximates the amount of irradiance on a point on the surface by computing how much it is geometrically occluded or, conversely, how much it is exposed to light. From a perceptual point of view this choice has a grounded foundation, as there is evidence that it improves shape perception over direct lighting [75]. Shape perception enhancements introduced by the use of ambient occlusion can be appreciated in the side–to–side

comparison in Figure 4.3.

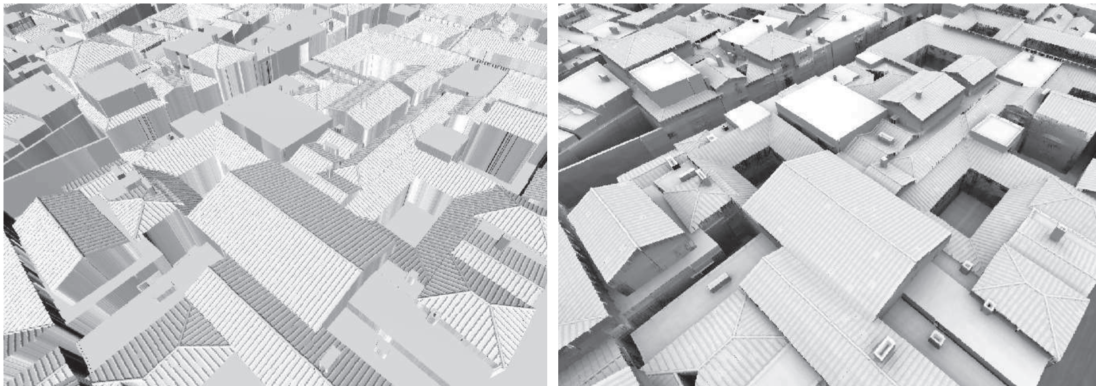


Figure 4.3: **Direct Lighting vs. Ambient Occlusion.** Shape perception can be greatly enhanced by using an ambient occlusion term. Here a close view of a block in the Pompeii dataset is shown without (left) or with (right) ambient occlusion.

It is important to note that, unlike most height field representations, which assume a smoothly–varying surface, a BlockMap is able to handle hard discontinuities such as perfectly vertical surfaces (i.e. building façades).

A BlockMap texture is obtained by packing together several components, as depicted in Figure 4.4. The resulting structured texture is stored in an RGBA image with 8 bits per channel, which can be *physically* subdivided into three rectangular sections. From a *logical* point of view, a BlockMap can be also divided into three sections, discussed in the following, that separately store information about 1) shape, 2) building roofs and ground surface attributes, and 3) buildings vertical walls surface attributes. For our purposes, we encode normal, color and ambient occlusion (AO) term as surface attributes.

Shape. Each BlockMap texture is associated to a region of the original dataset enclosed in an axis–aligned bounding box with square base. The shape information of buildings and ground inside this region is represented with a discretized heightmap, which is stored in the R channel of the first physical section. In our implementation, the Z axis is aligned to the world vertical direction and each 8–bit height value is used to quantize the space in $[Z_{min}, Z_{max}]$ into 256 possible elevations. At rendering time, the shape represented by the heightmap is drawn with a GPU ray casting algorithm implemented in a fragment shader (see Section 4.2.2).

Roof/Ground Data. Regions of the BlockMap texture are dedicated to store surface attributes for buildings roofs and ground. . We pack the RGB color and the AO term in the RGBA channels of the second physical section, and the x and y components of the surface normal in the BA components of the first physical section.

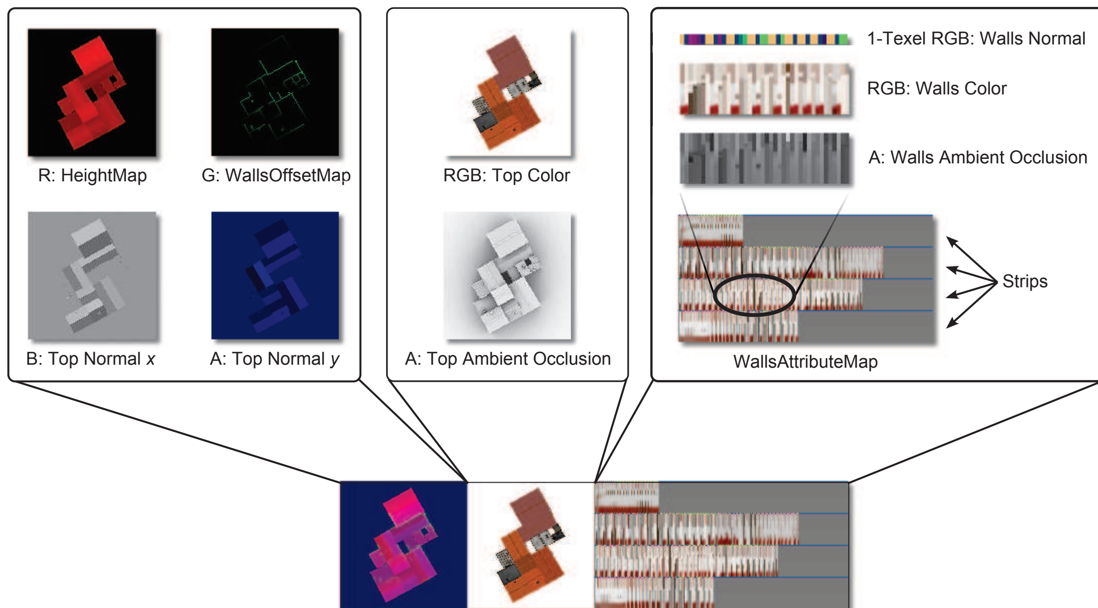


Figure 4.4: **Components of a BlockMap Texture.** The BlockMap texture can be divided in three physical sections. The first section holds the heightmap for buildings and ground, the WallsOffsetMap and the x and y components of the roof/ground surface normal. The second section stores color and ambient occlusion term for roofs and ground. The third section encodes the surface attributes (normal, color and ambient occlusion term) for walls.

The z component of the surface normal can be easily reconstructed from the first two by observing that the normal is unit–length and it is pointing towards the upper hemisphere (i.e. the surface is not *back–facing*).

Walls Data. The ability to handle hard discontinuities in a heightmap (i.e. the vertical surfaces of building façades) is a fundamental capability of the BlockMap representation. We say that a texel is *exposed* if and only if it is more than a predefined amount of units "taller" than any of its 4–connected neighbors, that is, a valuable portion of the wall is visible.
The key idea is to have a *parametrization* that maps exposed texels to one–texel–wide columns of surface attributes. This is accomplished by using a map that stores, for each exposed texel, a unique offset used to address the corresponding surface attributes column. We call this map the *WallsOffsetMap* and store it in the 8–bits G channel of the first physical section of the BlockMap texture. The special value 0 means that the texel does not belong to a vertical surface (i.e. it is a roof or ground surface). The WallsOffsetMap is thus able to address up to 255 discontinuities.
The *WallsAttributeMap* is in the third physical section of the BlockMap stores sur-

face attributes of buildings walls. To cope with a large amount of discontinuities that, for example, may arise in those regions of the dataset where the walls perimeter is particularly long, the WallsOffsetMap and the WallsAttributeMap can be vertically divided into several *slices* with the same height (see Figure 4.6). In this case, each slice has its own parametrization, resulting in an extended addressing space for the WallsOffsetMap, but in a reduced amount of memory dedicated to each wall column.
In each one–texel–wide column we pack the color and AO term in the RGBA channels. We compute the average normal of the column and store it in its last texel. We found that storing only the average normal is an acceptable approximation, given that the normal for a wall column does not change along its height, and allows us to save memory.

The data structure needed for a BlockMap is thus composed of a texture image, a bounding box and an integer representing the number of slices. By adopting the described packing solution, if we build the square heightmap with a side length of K texels, then the resulting dimension of the BlockMap texture would be $(2K + 256) \times K$, as we want roof/ground surface attribute maps to be in one–to–one correspondence with the heightmap, and the 8 bits used to parametrize visible vertical walls in the WallsOffsetMap allows us to address a maximum of 256 attribute columns. In our experiments, the best result are obtained by using the values of 64 or 128 for K. In fact, smaller values often lead to poor sampling of vertical surfaces, while higher ones produce large amounts of exposed vertical texels that could not be addresses with only 8 bits.

The choice of an RGBA image with 8 bits per channel has the advantage of being compact and encodable using standard, lossless image formats such as PNG or BMP. However, for each image channel, the domain limited to 256 representable values impose a quantization of the original data source. Considering that in almost all datasets the surface color is already expressed with 24 bit RGB images, and that high quality commercial software that uses normal maps adopts the same encoding, the information significantly affected by quantization is actually the height field. Similarly, due to the BlockMap structure, embedding the surface on a discrete heightmap causes aliasing on the building footprint. Nonetheless, the precision of the height field, both in terms of footprint and height values quantization, scales with the BlockMap size. Moreover, applying supersampling to the rendered image allows mitigating the produced *staircase* effect. It should be noted that, however, this structural error is acceptable in the visualization context in which BlockMaps are located, that is, exploration from high to low altitude flyovers.

4.2.2 Rendering a BlockMap

The rendering of a BlockMap node consists of a single–pass technique that uses a vertex shader and a fragment shaders to produce a depth–consistent image of the city

block. The algorithm can be subdivided into several steps, depicted in Figure 4.7. The process is started by the application, which issues the rendering of the node bounding box as a proxy geometry to activate the ray casting procedure for all covered fragments (Figure 4.7(a)). For each vertex of the proxy geometry, a vertex shader computes the view ray origin (Figure 4.7(b)) and direction (Figure 4.7(c)) in texture space and passes them to the rasterizer, which will interpolate the values at each generated fragment. The first step of the fragment shader is to compute the maximum distance the ray can travel inside the bounding box (Figure 4.7(d)) to avoid multiple range checks against the box planes during ray marching. Then, a 2D optimized version of the ray marching algorithm presented by Amanatides and Woo [6] is run on the BlockMap heightmap. The different outcomes of the ray marching determine the following fragment operations (see Figure 4.5). Fragments

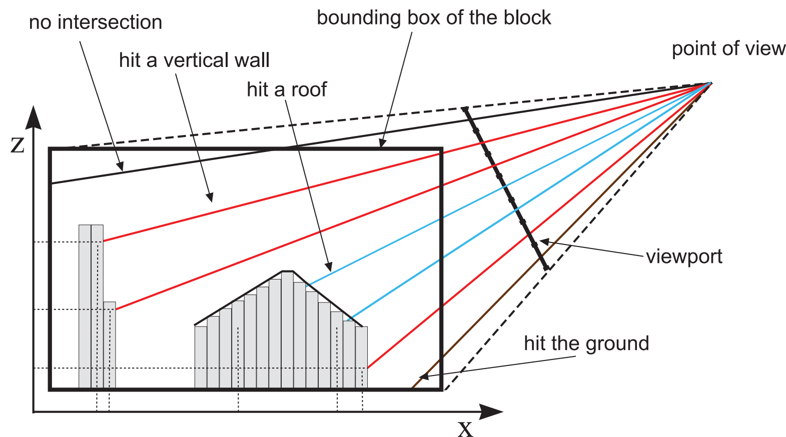


Figure 4.5: **Ray Intersection Cases.** The ray marching algorithm determines the possible intersection cases of the viewing ray against the BlockMap heightmap. Rays are classified to be hitting roof/ground surfaces or vertical walls. In case of no intersection, the fragment is discarded.

corresponding to rays that do not hit the surface are discarded (Figure 4.7(e)), that is, the do not output any value to the framebuffer. Rays that hit the heightmap are further classified to be hitting roof/ground surfaces or vertical walls (Figure 4.7(f)). If a roof/ground surface is hit, the corresponding surface attributes are accessed through a straightforward mapping of the uv coordinates of the hit point: the x and y components of the surface normal are retrieved from the first physical section using the same coordinates, while color and ambient occlusion term are accessed by simply displacing the u coordinate to fall on the second physical section.
If the ray hits a vertical surface, the hit point Z coordinate (Figure 4.7(g)), strip index (Figure 4.7(h)), and column offset (Figure 4.7(i)) are used to fetch the surface attributes, as illustrated in Figure 4.6.
Once all attributes have been fetched, that is, surface normal (Figure 4.7(j)), color (Figure 4.7(k)), and ambient occlusion term (Figure 4.7(l)), they are used in the

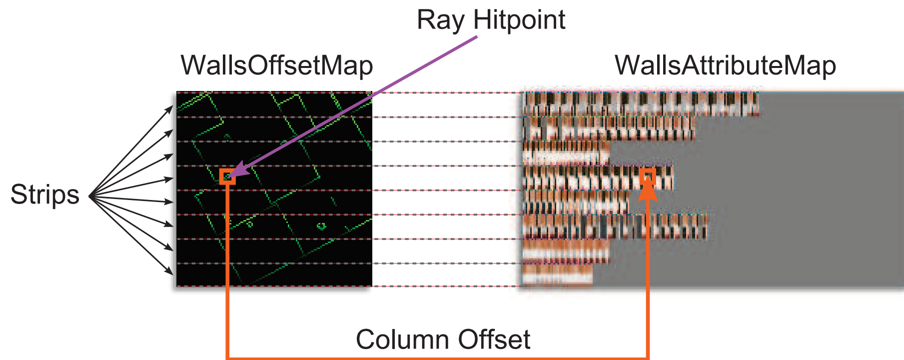


Figure 4.6: **Wall Attributes Fetch.** Whenever a ray hits a vertical surface, the offset of the corresponding texel in the WallsOffsetMap is used to address the wall attributes column in the WallsAttributeMap. The Z coordinate of the hit point is used to access the right texel in the attribute column.

lighting equation to produce the final result, as shown in Figure 4.8.
Finally, to generate a depth–consistent rendering, the fragment shader calculates the correct depth buffer value based on the ray hit point and viewing parameters. This is done with a simple geometric transformation and allows to consistently integrate the self–contained BlockMap rendering procedure into more general frameworks that use different techniques to draw other parts of the environment, like crowds, vehicles and vegetation.

4.3 Construction of a Multiresolution Representation for Urban Environments

The construction process used to build a BlockMap multiresolution representation of a urban environment is in general agnostic of the representation used in the input dataset. More precisely, we only require that the it must be possible to render a bounded region of the dataset to take snapshots of surface attributes and create an associated depth buffer. This means that a BlockMap dataset can be constructed starting from several kinds of input representations, like textured polygons, height fields, volumetric data and even implicit surfaces.
The produced multiresolution dataset is composed of a quadtree hierarchy in which each node represents the dataset region enclosed in the node bounding box, encoded as a BlockMap texture.

4.3.1 Hierarchy Construction

The out–of–core construction process is composed of several steps and starts by embedding the input dataset in a quadtree. The height of the tree is user–defined

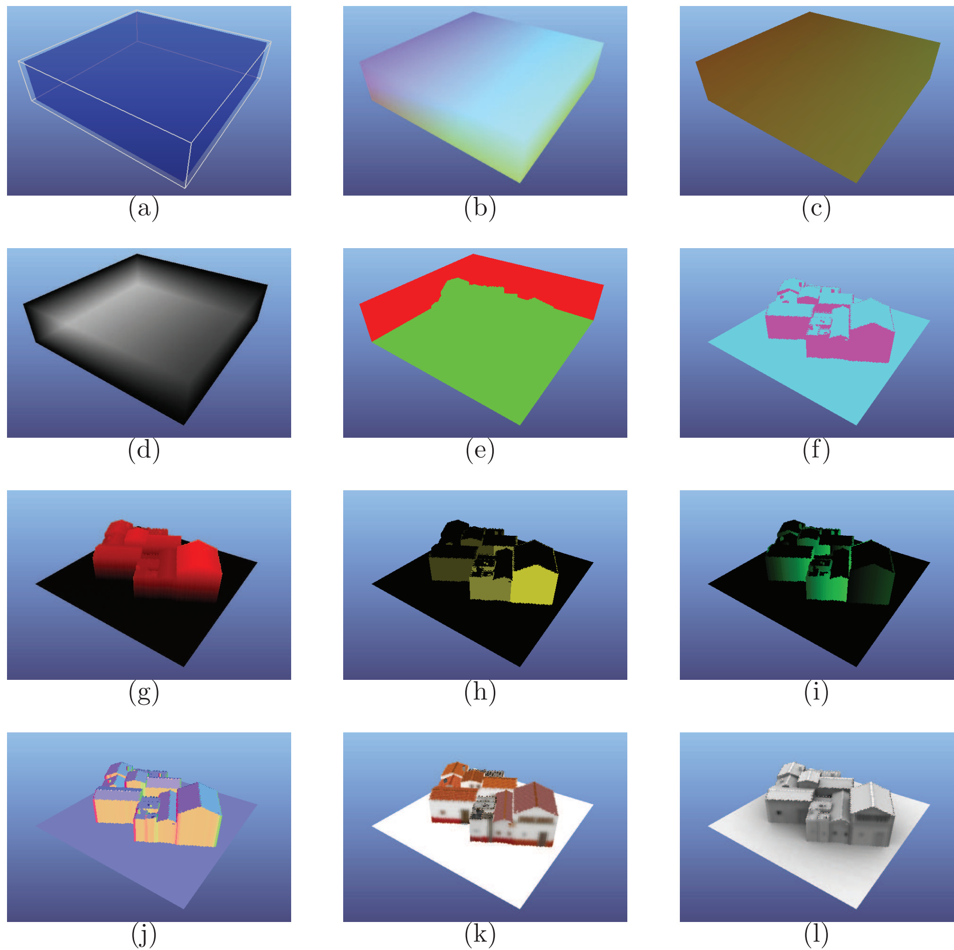


Figure 4.7: **BlockMap Rendering Steps.** The figures show the steps used to draw a city block using the BlockMap texture in Figure 4.2 to produce the final rendering in Figure 4.8.
(a) The rendering of the BlockMap bounding box is issued to activate the ray casting shader for each covered fragment. The vertex shader computes (b) ray origin and (c) ray direction, which will be interpolated across the box primitives. (d) The fragment shader determines the maximum length the ray can travel inside the box. (e) Fragments that do not hit the heightmap are discarded (marked in red). (f) Roof/ground (cyan) and walls (magenta) surfaces are identified. The Z coordinate of (g) the ray hit point, (h) strip index and (i) column index are used to address the wall surface attributes in the WallsAttributeMap. (j) Surface normal, (k) color and (l) ambient occlusion term are accessed and used in the lighting equation to produce the final rendering (see Figure 4.8).

and, together with the BlockMap texture size, determines the final dataset resolution. The construction is carried one level at a time, starting from leaf nodes and

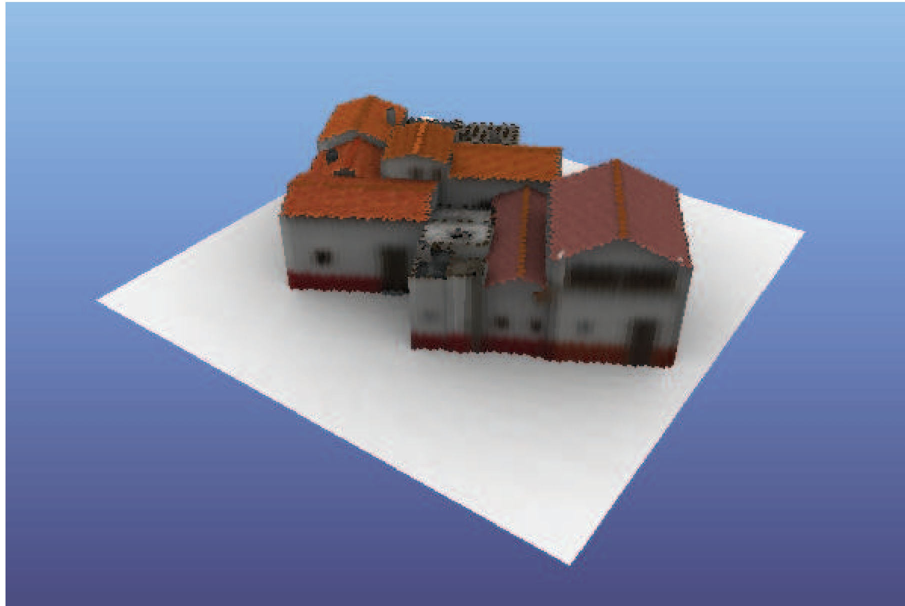


Figure 4.8: **Rendered BlockMap.** The result of the rendering process of the BlockMap in Figure 4.2.

proceeding up to the root.

Input Partitioning. Once the quadtree has been defined, for each leaf node, the region of the dataset enclosed in its bounding box is extracted and stored in a local database. In case of textured geometry input, the extraction consists of cutting the geometry at the leaf boundary. Given the local nature of this operation, the extraction step is run in parallel on each leaf.
A drawback of this locality is the introduction, at rendering time, of *seams* where two spatially adjacent nodes at different height are rendered (see Section 4.4). In theory, a partial solution to this problem would be to generate an *object/entity-oriented* hierarchy (e.g. on a *per-building* basis) rather than a hierarchy based on spatial subdivision. This would intrinsically avoid seams but would introduce other problems. For example, this partitioning strategy would assume that every building can be, at its highest detail, entirely encoded in a BlockMap texture. Otherwise, sub–partitioning it in a set of several BlockMaps would reintroduce the original problem whenever the whole set is not available at rendering time, a typical situation that arises when working on a memory–limited system and/or with a high–latency data transmission channel (e.g. during a remote streaming exploration). Even if buildings could be represented within a single BlockMap, the hierarchy resulting from such an object–oriented partitioning strategy would be impractically large whenever the urban model contains hundreds of thousands of buildings, which is typical for large cities. In addition, this kind of partitioning would not prevent the

city ground to generate seams.

Leaf Nodes BlockMaps Generation. We start by constructing the BlockMaps for leaf nodes (see Section 4.3.2). The data associated to each leaf is loaded from the local database and uploaded to the graphics hardware in the most appropriate form (e.g. vertex buffer objects and textures in case of textured geometry). To generate occlusion data for the accessibility term, we also load leaf nodes that are within a user–defined distance from the node being constructed. Nodes are processed following a Z–order filling curve, allowing us to use a simple cache with *least recently used* policy to minimize secondary storage access time. The resulting BlockMap texture is similarly stored to a local database.

Internal Nodes BlockMaps Generation. We can see the BlockMaps multiresolution hierarchy as a special form of image mipmap pyramid. When creating the mipmap pyramid from an image, the usual process is to start with the original data as the base level, and then generate each subsequent level at half the resolution from the previous one. Following this intuition, BlockMaps for internal nodes are generated from the BlockMaps associated to their four children. This is justified by the fact that the data associated to internal nodes can not be processed at once (e.g. the root node would require the entire dataset to be in graphics memory). Moreover, in our tests we experimented that the low resolution BlockMaps constructed from high resolution ones closely match the results produced by using the original data.

4.3.2 BlockMap Generation

The generation of a BlockMap texture is composed of several steps and it is almost totally handled by the graphics hardware. In particular, we use vertex, geometry and fragment shaders to acquire shape and surface attributes from the input dataset, and to build and assemble the various components of the BlockMap. The output of the rendering is redirected to texture memory (*render–to-texture*) by using OpenGL *Framebuffer Objects* (FBOs). Whenever a construction step outputs several values, we exploit the *Multiple Render Targets* (MRT) ability of GPUs to simultaneously write onto several textures.
The following steps are referred to the generation of a $(2K + 256) \times K$ BlockMap, where K is a user–defined power of two.

HeightMap and Roof/Ground Surface Attributes. The first step of the generation process consists of creating the heightmap of the dataset region enclosed in the bounding box associated to the BlockMap. This is easily done by rendering the input data (or the four child BlockMaps in case of internal nodes) from above the region with an orthogonal viewing volume coincident with the bounding box and a $K \times K$ viewport. In our coordinate frame the Z axis maps to the world

vertical direction; thus, for each fragment generated by the rendering process, we map world–space Z coordinate between the bounding box Z_{bottom} and Z_{top} values to the range $[0, 1]$ and write the result on the texture, representing the heightmap. Moreover, by exploiting MRT, we also output the color–coded surface normal and surface color. The target FBO for this step thus consists of three $K \times K$ textures: one 1–channel, 8–bit texture for the heightmap, and two 3–channels, 24–bit textures for the surface normal and color.

Walls Parametrization. Once the heightmap has been generated, it is read back to system memory to identify the discontinuities introduced by vertical surfaces and construct the 8–bit WallsOffsetMap. The parametrization process consists in assigning a unique offset to each exposed texel (see Section 4.2.1). We do this by simply assigning to each exposed texel the value of a running counter starting from 1 (the special value 0 is used to identify roof or ground surfaces). We start with one slice comprising the whole heightmap. Whenever the counter exceeds the value of 255, we double the number of slices and repeat the parametrization process independently on each slice until all offsets are in a representable range or the height in texels of the slice reaches a user–defined minimum (note that the minimum height for a slice is 2 texels, as one texel is dedicated to the column average normal). Finally, the resulting WallsOffsetMap is uploaded to texture memory to be used in the following steps.
This parametrization process could be also implemented to run on the GPU, for example by using parallel prefix sum (i.e. *scan*) and stream compaction techniques [53, 122]. However, the computationally simple process and the small memory footprint of the heightmap and the WallsOffsetMap allow us to use the CPU without introducing significant delays.

Wall Surfaces Attributes. To acquire surface attributes belonging to vertical walls, we developed an elegant and effective sampling strategy. We take views of the whole geometry from uniformly distributed directions in the hemisphere centered at the BlockMap center, and reproject the result of each view onto the BlockMap's prisms. Since each surface point can be seen from several directions, the final value is obtained by gathering all the contributions and combining them to a single value per point.

We developed a GPU–accelerated technique that harnesses the performance and programmability of current hardware to efficiently perform multi–view sampling. The sampling process is performed by iterating the following steps for each view and each slice of the BlockMap:

1. render surface attributes of input data from the viewpoint with an orthogonal projection to a $n \times n$ texture (called *AttributeBuffer* in Figure 4.9);

2. ray cast the BlockMap with the same viewing settings, targeting rendering to a

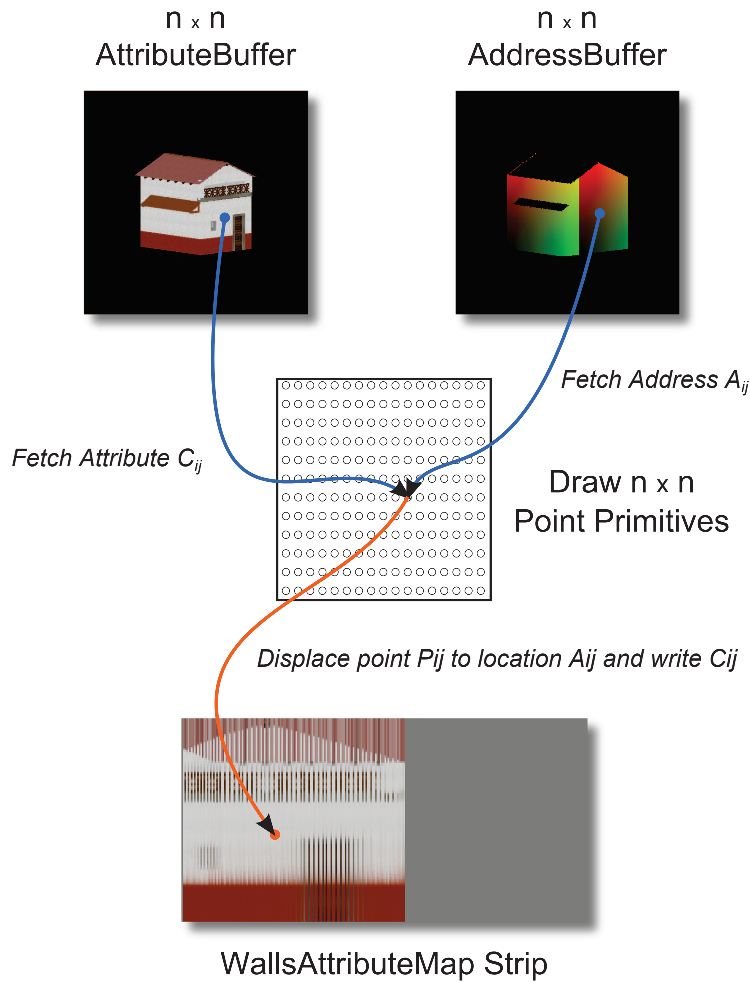


Figure 4.9: **Walls Attributes Sampling.** In the first pass, surface attributes of vertical walls are drawn to a $n \times n$ *AttributeBuffer* texture. At the same time, the BlockMap heightmap is ray casted and the corresponding value of the WallsOffsetMap is written. In the second pass, $n \times n$ point primitives are drawn. In the vertex shader, each point P_{ij} fetches the corresponding attribute C_{ij} and address A_{ij}. The point is displaced to the final address A_{ij} in a strip of the WallsAttributeMap, where the value of C_{ij} is accumulated. The final pass averages the accumulated attribute values.

$n \times n$ texture (called *AddressBuffer* in Figure 4.9); whenever a ray hits vertical surface, the corresponding address in the WallsOffsetMap (i.e. a pointer to a column in the *wall textures*) is written to the data that should be fetched for shading.

3. set a $256 \times K$ floating point texture as rendering target and draw $n \times n$ point primitives, one for each pixel of the *AttributeBuffer*. In the vertex shader, the color corresponding to pixel i, j is fetched from the *AttributeBuffer*, while the

vertex position is read from the *AddressBuffer* at the same coordinates.

The purpose of these steps is to build the BlockMap that, when ray casted with the same viewing parameters, would produce the same result as rendering the geometry. This approach is used to sample all the shading attributes in the BlockMap, i.e. *color* and *normal*. For the color, the rendering of geometry at step 1 is done with texture mapping alone (i.e. without lighting), while the value written at step 3 is blended additively with the value possibly present because of other samples fell in the same texel (either in the same or in an earlier view). The same is done for the normals, encoded as color for the sampling. In this manner, colors and normals stored on each texel of the BlockMap slice are averaged among all the views from which the corresponding portion of surface is visible.
On modern GPUs it is possible to set an array of textures as the target color buffer and to select on a per–primitive basis the destination layer on the array. We exploit this capability to simultaneously render to the entire slice set in which wall surfaces have been partitioned. More in detail, in step 2 we also output the slice number associated to the hit column and use it in a geometry shader to redirect the point primitives in step 3 to the corresponding layer of the target texture array.

Ambient Occlusion. For the accessibility, we also need the geometry outside the BlockMap because it may occlude part of the geometry inside the BlockMap. Here, we conservatively assume to use the entire geometry as potential occluder, although we verified that, in practical cases, using only the geometry closer than a fixed distance also leads to good visual results. Step 1 of the wall attributes sampling process is modified by rendering all the potentially occluding geometry and saving the depth buffer. Step 3 is modified by checking, for each point, if the depth value of the corresponding fragment in the AttributeBuffer is greater than the depth value stored at step 1. If not, the contribution of the point is ignored (the sample is occluded), otherwise the value written into the BlockMaps is the value already present plus one. With this approach, the final value per texel corresponds to the number of views from which the corresponding portion of the surface is visible. The accessibility is then obtained by dividing that result by the total number of views.
To calculate the accessibility factors for roof an ground surfaces we adopt a similar procedure, but in this case we do not need to ray cast the heightmap and displace the point primitives in the vertex shader because a one–to–one correspondence from the depth buffer to the render target of the same size is simply established by superimposition.
The computation of the ambient occlusion term is executed only on the leaf nodes of the hierarchy. For internal nodes, the ambient occlusion is considered as a local surface attribute exposed by child BlockMaps and is thus extracted with the same process used to retrieve color and normal information.

Assembly. The final step of the BlockMap generation process consists in assembling all the components into the final structure. Before packing, slice textures holding wall colors and ambient occlusion term are read back to system memory for a final processing step. In particular, missing samples that may be present on columns are filled by linear interpolation between the values at the hole boundary, and then the whole texture is downsampled to match the final slice height. Similarly, the slices with surface normal information are collapsed to a single row by averaging the samples present in each column.

4.4 Rendering Urban Models with BlockMaps

The proposed rendering system is based on an out–of–core algorithm that accesses data from a local secondary storage device (i.e. hard disk) or a remote network server. The output–sensitive rendering algorithm is targeted to run on a commodity PC equipped with programmable graphics accelerator.

Remote Rendering Architecture. The main application of our system is networked browsing of very large urban environments. We implemented a prototype client-server architecture that enabled multiple clients to explore city models stored on a remote server. Since BlockMaps are able to provide approximate representations of blocks of buildings which are visually valid for a wide range of viewpoints, we can create a full quadtree hierarchy of levels of detail by associating BlockMaps to larger and larger areas. Thanks to the constant footprint of BlockMaps, memory management is particularly simple and effective. In particular, no fragmentation effects occur throughout the memory hierarchy, and data transfers at all levels can be optimized by grouping BlockMaps for tuning message sizes. Upon connection, the node hierarchy structure is transmitted to the client, which stores it in main memory. The hierarchy (amounting to a few Kilobytes) is the only data structure permanently kept in client memory, while all the BlockMaps reside only on the server and are transmitted only on demand. Each client maintains a 2-level cache of recently used BlockMaps in graphics memory and local storage, and requests data to the server only at cache misses.

Figure 4.10 illustrates the architecture of our multi-threaded rendering client. The main client thread, called *RenderThread*, is responsible for visiting the hierarchy to determine the level-of-detail required for all parts of the scenes, and for performing the rendering. Other two threads, the *TextureThread* and the *NetworkThread*, transparently handle the BlockMap requests made by the RenderThread to the remote data server. The texture cache is organized as a set of texture atlases of fixed size, where the BlockMaps are copied from local storage. The local storage of the BlockMaps received from the remote server is implemented using a Berkeley DB, which provides a level of abstraction towards the main memory and secondary storage (e.g. hard disk).

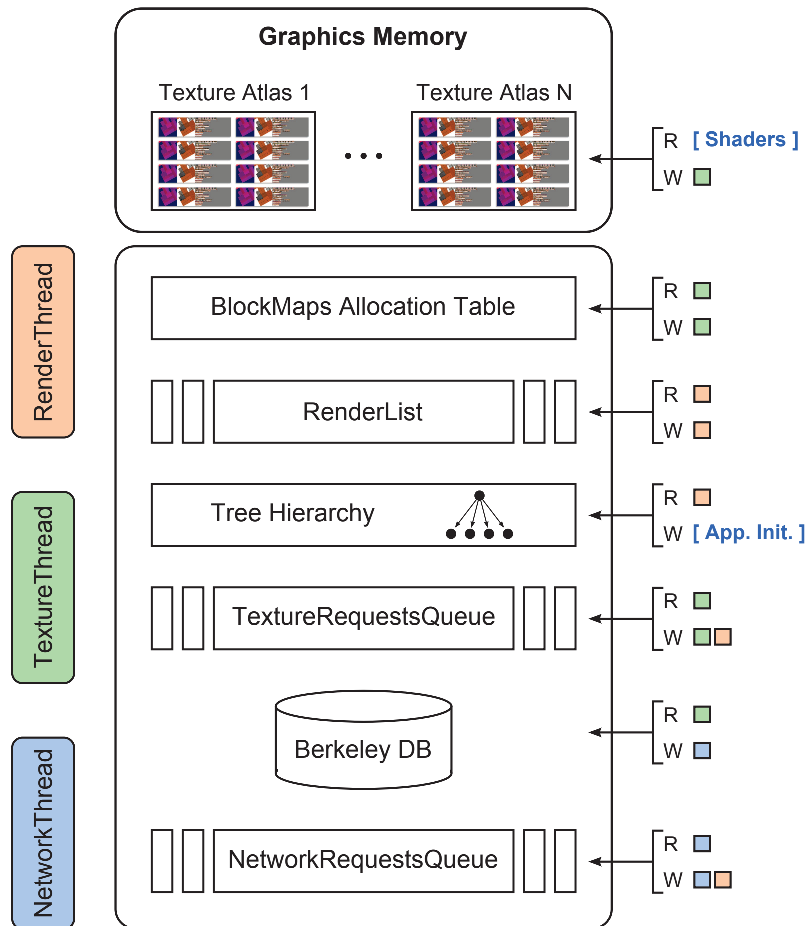


Figure 4.10: **Remote Rendering Architecture.** The client side of the remote rendering system is a multithread architecture aimed at handling network/disk data requests and transfers to provide real–time rendering of large urban environments. BlockMaps are organized in a cache hierarchy that is constantly kept up–to–date in response to the viewing parameters.

The RenderThread works in three stages: *RequestingVisit*, *LockingVisit*, and *Rendering*.

In the *RequestingVisit* stage, the RenderThread performs a top down traversal of the hierarchy according to the current viewing parameters and issues a request for each visited node. If the node is present in the local database, then the request is queued to the *TextureRequestsQueue*, meaning that the BlockMap is waiting to be loaded in graphics memory, otherwise the request is queued in the *NetworkRequestsQueue*. In order to prioritize requests, we distinguish among different request *classes*: class `A` requests, if the requested node is inside the view frustum, class `B` requests if the node is not inside the view frustum but is a neighbor of a class `A` node, or class `C` requests if the node error is below the accepted error threshold. Note that

requests of class `B` and **C** define the neighborhood of the set of nodes required for rendering the current view; if the amount of cache is enough, by prefetching them we are able to offer a more responsive detail refinement.

In the *LockingVisit* stage, the RenderThread performs a second top-down traversal stopping at nodes of class `A`. The goal of the visit is to lock all the nodes that satisfy the error criterion and add them to the *RenderList*, i.e. the list of nodes that will be rendered in the current frame. A node can be locked if the associated BlockMap is in graphics memory. Locking a node sets a flag that prevents its unloading. The visit is implemented as a recursive procedure which starts by trying to lock the current node. If locking is successful, but the error is above the user–specified threshold, then its children are visited. If the error is within the threshold, the node is added to the *RenderList*. In case all children of a locked node have been locked, their parent is unlocked. Otherwise, the region occupied by the children that cannot be locked will be drawn using the corresponding portion of the parent node, as show in Figure 4.11. This is done by inserting the parent node in the *RenderList* with a code that specifies which of the four portions of the BlockMap must be used for rendering.

The use of parent data in case of cache misses illustrates an intrinsic limitation of the technique: due to the local nature of the BlockMap representation, seams are introduced where two spatially adjacent nodes at different height are rendered (see Section 4.3.1). This also happens whenever data is available but the projected screen error causes the node selection algorithm in the RequestingVisit stage to select adjacent nodes belonging to different levels of the quadtree.

The final stage is *Rendering*, which simply consists of scanning the *RenderList* and sending the BlockMaps to the GPU ray casting process, described in Section 4.2.2. To minimize the number of texture switches, the *RenderList* is ordered in buckets on the base of the texture atlas containing the BlockMap and processed front to back within each bucket, thus reducing pixels overdraw and enabling the use of occlusion strategies (e.g. hardware occlusion queries).

Prioritization of requests is crucial for interactive rendering. Both the requests to load BlockMaps from local storage to graphics memory and from the network to local storage are kept in dedicated priority queues (*TextureResquestsQueue* and *NetworkRequestsQueue*, respectively). The comparison operator to sort the requests is implemented by combining three criteria. The first criterion is the *time* the request is issued. At every frame all the necessary nodes that are not present in graphics memory are enqueued in the *NetworkRequestsQueue*, while the queue is purged of the requests older than a predefined *life time*. This means that after having satisfied all the requests issued in the last frame, recent unsatisfied requests can be processed. In this manner we avoid to continuously delete/reinsert in the queue the nodes close to the boundary of the view frustum. If the time does not discriminate the request and prefetching is enabled, the *class* of the requests is used as criterion, giving higher

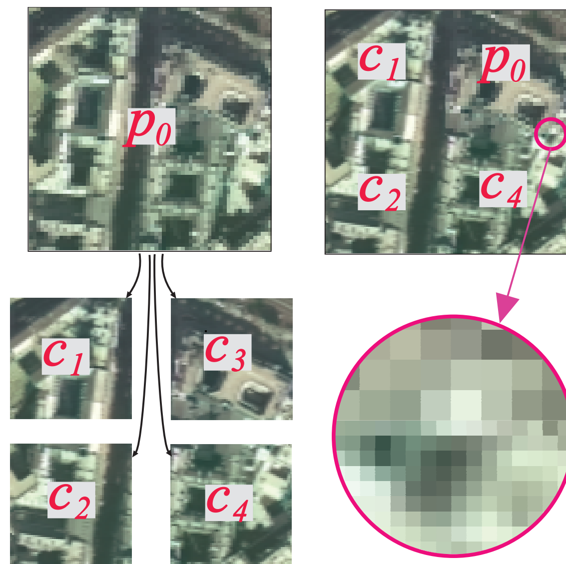


Figure 4.11: **Using Parent BlockMap for Missing Data.** On the left side an example of a node with its four children (top-view). In the right side the representation used when only nodes c_1, c_2 and c_3 can be locked, with a zoomed image of the border region between the two resolutions.

priority to class A requests, followed in order by class B and class C requests. Finally, if both previous criteria fail, which happens for all the nodes in the view frustum, the screen–space error is used.

4.5 Implementation and Results

We implemented the described system using C++, OpenGL, and GLSL shaders. We have extensively tested our system with a number of urban models. The quantitative and qualitative results discussed here are for the Paris urban environment, which shows an example of large scale models created from cadastral maps, and two smaller models of the ancient Pompeii environment and the ancient city of Rome, representative of a smaller scale but detailed model created by procedural means. The Paris model is created from the cadastral maps, containing a vector representation of 80,414 buildings described with 3.7M triangles. The original dataset has no texture information for the building façades, so, for the sake of testing, we have created and stored for each building a different 512^2 procedural texture. Roof textures were taken from aerial photographs composed in 64 $2K \times 2K$ tiles. Overall, the texture information for the façades is composed by 20G texels (almost 60GB of uncompressed data). The Pompeii model represents a reconstructive hypothesis of the ancient Pompeii described by 30M triangles and 30M texels [102]. The Rome model is made of 35M triangles and 20M texels and is part of the Rome Reborn

project [94].

4.5.1 Preprocessing

We implemented a preprocessing framework that takes as input datasets of urban environments in the form of textured triangle meshes.

For the Paris dataset the creation of all the geometry from the cadastral profiles, the geometric partitioning and the quadtree construction took less than a couple of minutes and generated a tree of 20K nodes with 15K leaves, with maximum depth of 10. The recursive partitioning of the tree targeted less than 10000 polygons and buildings for each leaf and a set of textures that could be arranged in a 2048×2048 atlas. The construction of the atlas-tree, used to speed-up sampling, took approximatively four hours starting from the original 80K 512^2 façade textures. Once the data is reorganized this way, in the case of 128×512 BlockMaps used in our experiments, the creation of a tree of 3K BlockMaps took 140 minutes.
For the Pompeii and Rome datasets we created a quadtree with 8 levels, whose leaves have a side length of 8 meters, for a total of about 22K BlockMap nodes for each dataset. The datasets creation took, respectively, 6 and 7 hours, whose vast majority was spent in I/O disk transfers.

4.5.2 Streaming and Rendering

The reported times were obtained on a commodity PC with a quad–core Intel i7 920 Processor running at 2.66 GHz, 6 GB of RAM, two 500 GB SATA hard disks, and a NVIDIA GeForce 260 GTX with 896 MB of graphics RAM for the client application, and a Pentium IV @ 3 GHz, 2 GB of RAM, and a 500 GB SATA hard disk for the server side. The goal of our tests was to show how our system can provide an efficient solution for remote visualization of large urban models.

We run a fly–through over the Paris model (see Figure 4.12) performed with a controlled bandwidth of 100, 20, 8 and 4 Mb/s on a 1200×1000 viewport (see Figure 4.13) to evaluate the system.

Since the system is multi-threaded and the rendering cycle does not have to wait for BlockMaps to be received from the server, the frame rate is is always above 50 fps, with peaks of 200 fps, as in all the other tested datasets. A more interesting data is the number of network cache misses, i.e. the number of nodes that should have been rendered according to the screen–space error threshold, but whose data was not available locally (see Figure 4.14). The ratio between the cache misses tends to be (inversely) proportional to the bandwidth only when the point of view moves at a sufficient speed, so that the set of nodes in the frustum changes rapidly, while for a normal inspection, where the user is not interested to the maximum level of detail while moving from a point to another, but only when stopping to a region of interest, the 4 Mb/s bandwidth provides a result comparable to much larger data channels.

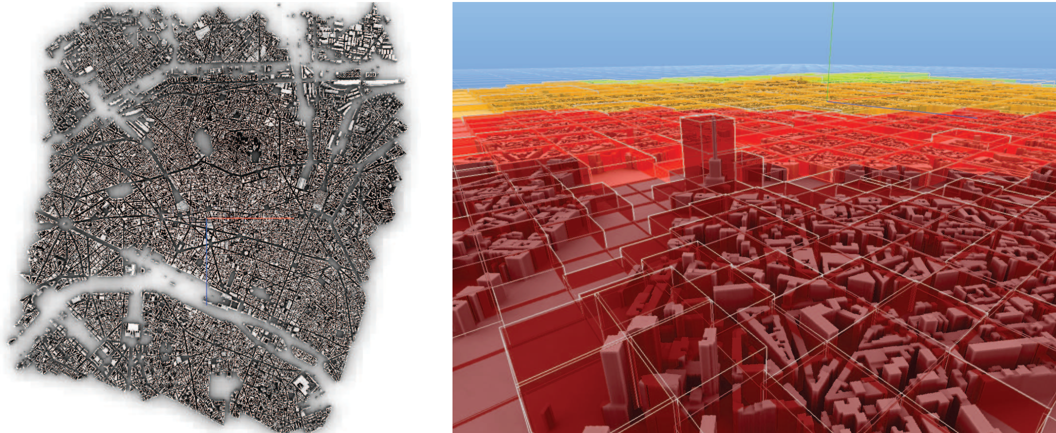


Figure 4.12: **Paris Dataset.** Left: a top view of the Paris dataset with the ambient occlusion term. Right: a lower view with the bounding boxes of the BlockMap nodes overlaid.

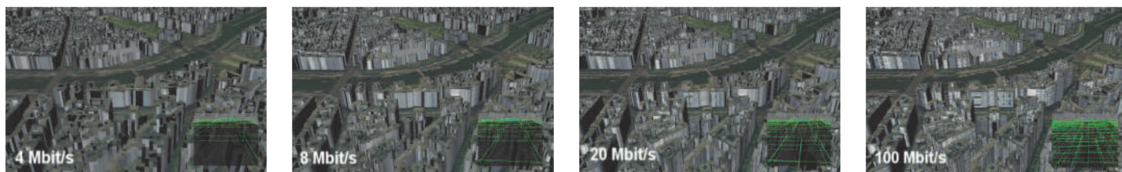


Figure 4.13: **Detail vs. Bandwidth**. Four snapshots of a fly–through over the Paris dataset with different bandwidth settings. As expected, with larger bandwidths a greater amount of detail is presented.

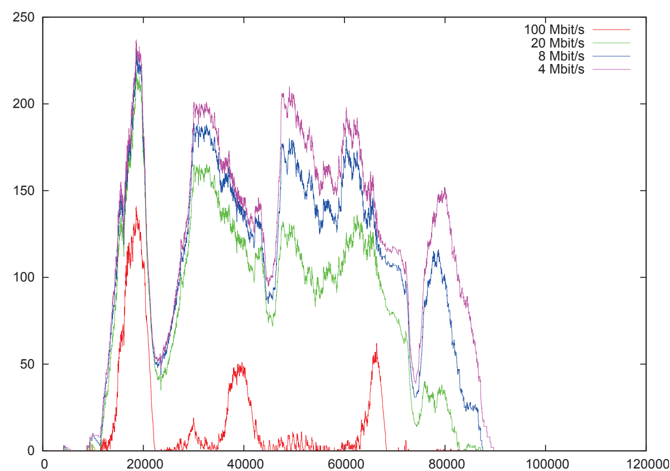


Figure 4.14: **Network Cache Misses.** The graph of recorded cache misses during a fly–through over the Paris dataset shows how, as expected, data availability increases as a result of a larger bandwidth.

Figure 4.15: **Detail Perception with Ambient Occlusion** A view of ancient Rome with only AO term (left) and also with color (right). Note that not only sloped rooftops are represented but also the covering roof tiles are realistically rendered.

Rendering quality is easier to appreciate when looking at the detailed Pompeii (see Figure 4.16) and Rome (see Figure 4.15) models, which contains fine geometric features and many sloped roofs. In the frame shown in Figure 4.3 it can be appreciated how the ambient occlusion term enhances the perception of small features, for example the details of the roofing, and gives the impression of jutting out roofs. Even in extreme cases in which the dataset does not contain image information and whose shapes are represented by extruded prisms like in the Paris dataset (see Figure 4.17), the use of the ambient occlusion term allows a quick and clean perception of the environment.

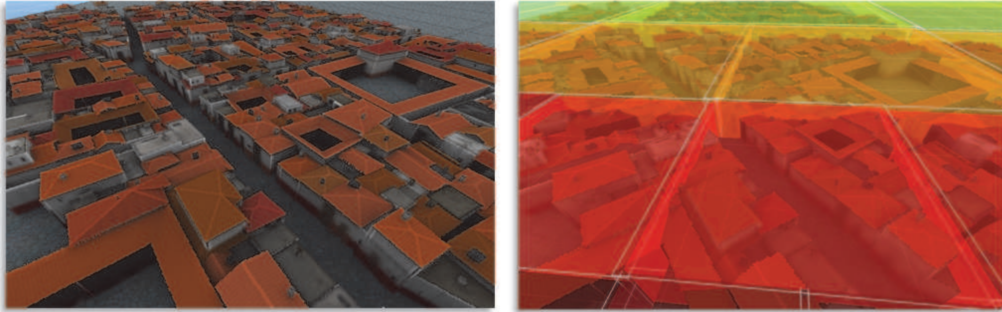


Figure 4.16: **Pompei Dataset.** The result of using the ambient occlusion in combination with a standard Phong illumination model enhances the exploration experience. Left: a region of the Pompeii dataset viewed from above. Right: the same view showing the nodes bounding boxes.

Figure 4.17: **Simple Shapes with Ambient Occlusion** With the use of the Ambient occlusion term, even simple, non–textured shapes as the extruded prisms of the Paris dataset provide a well–defined understanding of the urban environment.

Chapter 5

Computer Graphics for the Web Platform

Worldwide data providing and sharing entered daylife since high–speed network connections became available at a large scale. While accessing high–quality audio and video media is now a common activity, it is still not a widespread conception to have complex 3D content available inside web documents. Thanks to the availability of low–cost graphics hardware and recent advances in standardized web technologies, users of web platforms can now benefit from the power of graphics hardware directly within web browsers.
In this chapter we present SpiderGL, a novel JavaScript 3D Graphics library, whose design helps experienced computer graphics developers to implement, in the web platform, well–settled common and complex multiresolution algorithms.

5.1 Computer Graphics and the World Wide Web

The delivery of 3D content via the web platform started to be a topic of interest since the graphics hardware of commodity personal computers became enough powerful to handle non–trivial 3D scenes in real–time. As described in Section 2.8, many attempts have been made to allow the user of standard web documents to directly access and interact with three–dimensional objects or, more generally, complex environments from within the web browser. Historically, these solutions were based on software components in the form of proprietary and often non–portable browser plug–ins. The lack of a standardized API did not allowed web and computer graphics (CG) developers to rely on a solid and widespread platform, thus losing the actual benefits that these technologies could provide.
In the same period of time in which GPUs showed a tremendous increase in performances and capabilities, the evolution of the technology behind web browsers allowed interpreted languages such as JavaScript to perform quite efficiently in general purpose computations, thanks to novel just-in-time (JIT) compilers such as

TraceMonkey [87], V8 [72] and SquirrelFish [11].
Thus, on one side, the hardware and software components have reached a level of efficiency and performances which could fit the requirements for high–quality and interactive rendering of 3D content to be visualized, on the other the increase of bandwidth for accessing the Internet allowed large volumes of data to be transferred worldwide in a relatively short amount of time.

In this scenario, the need for a standardized computer graphics API became a high–priority problem to be solved. In fact, in late 2009, the Khronos Group [66] officialized a new standard, WebGL [68], which aims at harnessing the power of graphics hardware directly within web pages through a JavaScript interface. WebGL is an API specification designed to closely match the OpenGL|ES 2.0 specifications [67], with some modifications which make the API more close in look–and–feel to a JavaScript developer. On the other side, as web pages which use WebGL are freely accessible from every potential web client, the new specification impose a series of restrictions to comply with a more strict security policy.
Although this scripting language can not be considered as performant as a compiled one like C++, the tendency of delegating the most time–consuming parts of a CG algorithm to the graphics hardware helps mitigating the performance gap.

In this chapter we will discuss SpiderGL [36, 35], a novel JavaScript library aimed at bringing to the web platform the common tools used by computer graphics developers. In Section 5.2 we will describe the library architecture and provide some general examples in Section 5.3; Section 5.4 will present how SpiderGL can be used to easily create multiresolution visualization Web application for terrain, urban, and image datasets, while Section 5.5 the attention will be focused on biological data presentation. Finally, in Section 5.6 we will introduce MeShade, an online application for deploying 3D models on the Web with customizable material.

Leveraging the new Web Technologies. Thanks to the combination of hardware and software capabilities and performances, coupled with a high–speed data channel, it is nowadays possible to effectively and natively handle real–time 3D graphics within web pages. In particular, by exploiting the asynchronous features provided by the runtime environment of the web browser, it is possible to manage large datasets in a natural out–of–core fashion. The creation of fast and reliable visualization algorithms that allow the user to explore huge environments (like Google Earth [56] and Bing Maps [26]) implies that multiresolution algorithms should be developed with network streaming in mind, both in terms of caching mechanisms and the actual representation of a data packet. Alongside, it is easy to see how the new WebGL 3D technology will bring closer web developers, which are more and more interested in learning 3D graphics and CG developers, which will try to deploy their algorithms to less powerful platforms.

The question is now what still separates a compiled C++ from a JavaScript application with respect to CG algorithms. One obvious answer is execution speed, but there are other gaps to be filled:

- Asynchronous content loading: many CG algorithms, especially when dealing with multiresolution datasets, make intensive use of multithreading for asynchronous (down)loading of textures or geometry data from different cache levels. This is vital to avoid the application to freeze while waiting for a texture to be loaded from RAM, disk or even a remote database to GPU. On the other hand JavaScript still does not officially support multithreaded execution.

- Shape data loading from file: there are many file formats for 3D models and as many C++ libraries to load them [18, 116, 105]. JavaScript includes a series of predefined types of objects for which the standard language bindings expose native loading facilities (i.e. the `Image` object), but such bindings for 3D models have yet to come.

- Math: linear algebra algorithms for 3D points and vectors are very common tools for the CG developer, and a large set of dedicated libraries exists for C++ and other languages. Although many JavaScript demos for mathematical algorithms can be found just browsing the web, a structured library with the specific set of operations used in CG is still missing.

- WebGL wrapping: the WebGL specification is very similar to OpenGL|ES 2.0, which means that there are significant changes w.r.t. OpenGL, for example there are no matrix or attribute stacks and there is no immediate mode. Although these choices comply to the bare-bones philosophy of OpenGL|ES 2.0, they also imply incompatibility even with OpenGL 3.0, which, for example, still provides matrix stack operations.

We designed and implemented the SpiderGL JavaScript library to fill these gaps: it extends JavaScript by including geometric data structures and algorithms and wraps their implementation towards WebGL. In particular, SpiderGL was designed keeping in mind three fundamental qualities:

- Efficiency: With JavaScript and WebGL, efficiency is not only a matter of asymptotic bounds on the algorithms, but the ability to find the most efficient mechanism to implement, for example, asynchronous loading or parameters passing to the shader programs, without burdening the CPU with respect to a bare bone implementation;

- Simplicity and Short Learning Time: Users should be able to reuse as much as possible of their former knowledge on the subject and take advantage of the library quickly. For this reason SpiderGL carefully avoids over-abstraction: almost all of the function names in SpiderGL have a one to one correspondence with either OpenGL or GLU commands (e.g. the SpiderGL function

sglLookAt for setting up the camera pose matrix), or with geometric/mathematics entities (e.g. SglSphere3, SglMeshJS).

- Flexibility: SpiderGL does not try to hide native WebGL functions, instead it provides higher level functionalities that fulfill the most common needs of the CG developer, who can use SpiderGL and WebGL calls almost seamlessly.

5.2 The SpiderGL Graphics Library

Most of the current JavaScript graphics libraries implement the *scene graph* paradigm. Although scene graphs can naturally represent the idea of a "scene", they also force the user to resort to complex schemes whenever more control over the execution flow is needed. There are several situations in which fixed functionalities implemented by scene graph nodes cannot be easily combined to accomplish the desired output, thus requiring the developer to alter the standard behavior, typically by deriving native classes and overriding their methods or, in some cases, by implementing new node types. In these cases, a procedural paradigm often represents a more practical choice. Also, scene graphs contain a large codebase to overcome the limitations of strongly typed imperative programming languages, which is no more required in dynamic languages such as JavaScript.

5.2.1 Library Architecture.

SpiderGL is composed of five modules, distinguished by color in Figure 5.1:

- **MATH**: Math and Geometry utilities. Linear algebra objects and functions, as well as geometric entities represents the base tools for a CG programmer.
- **GL**: Access to WebGL functionalities. The GL module contains a low-level layer, managing low-level data structures with no associated logic, and a high-level layer, composed of *wrapper* objects, plus a series of orthogonal facilities.
- **MESH**: 3D model definition and rendering. This module provides the implementation of a polygonal mesh (SglMeshJS), to allow the user to build and edit 3D models, and its image on the GPU side (SglMeshGL). SpiderGL handles the construction of a SglMeshGL object from a SglMeshJS.
- **ASYNC** : Asynchronous Content Loading. Request objects, priority queues and transfer notifiers help the programmer to implement the asynchronous loading of data.
- **UI** : User Interface. A GLUT–like framework and a series of typical 3D manipulators allows a quick and easy setup of the web page with 3D viewports and provide effective management of user input.

In the following we will describe the main features of each module. In the description we will provide some code fragments whenever they help the understanding of the module. As the JavaScript language does not have the concept of public and private interfaces (every class member is publicly accessible), developers try to make visible the intended access policy by adopting a naming convention which, by the time, has become a *de-facto* standard. SpiderGL wants to be as close as possible to existing behaviors, so it embraces the same philosophy: Listing 5.1 gives an overview of the naming conventions adopted throughout the library.

```
// Public Interface
SGL_PUBLIC_CONSTANT     // Constant
SGL_PublicVariable      // Variable
sglPublicFunction       // Function
SglPublicClass          // Class
obj.PUBLIC_CONSTANT     // Class Constant
obj.publicAttribute     // Class Attribute
obj.publicMethod        // Class Method
obj.publicProperty      // Class Property

// Private Interface
_SGL_PRIVATE_CONSTANT   // Constant
_SGL_PrivateVariable    // Variable
_sglPrivateFunction     // Function
_SglPrivateClass        // Class
obj._PRIVATE_CONSTANT   // Class Constant
obj._privateAttribute   // Class Attribute
obj._privateMethod      // Class Method
obj._privateProperty    // Class Property
```

Listing 5.1: **SpiderGL Naming Convention for Public and Private Interfaces.** The proposed syntax reflects what in JavaScript has become a *de-facto* standard for identifiers naming.

SPACE: Linear Algebra and Geometry

The SPACE module provides low-level mathematical functions and objects as well as space-related object representations and algorithms, as described in the following.

General Math and Linear Algebra. This submodule implements essential mathematical objects such as vectors and matrices, along with basic operations on them. Particular attention has been paid in their implementation in order to reduce the programmer effort and the impact on performances.
The low-level layer is composed by functions that operate on native JavaScript arrays as input parameters and return types. As a usage example, calculating a three-dimensional triangle normal would be as follow:

```
var v0 = [ x0, y0, z0 ]; // 1st vertex coordinates
var v1 = [ x1, y1, z1 ]; // 2nd vertex coordinates
```

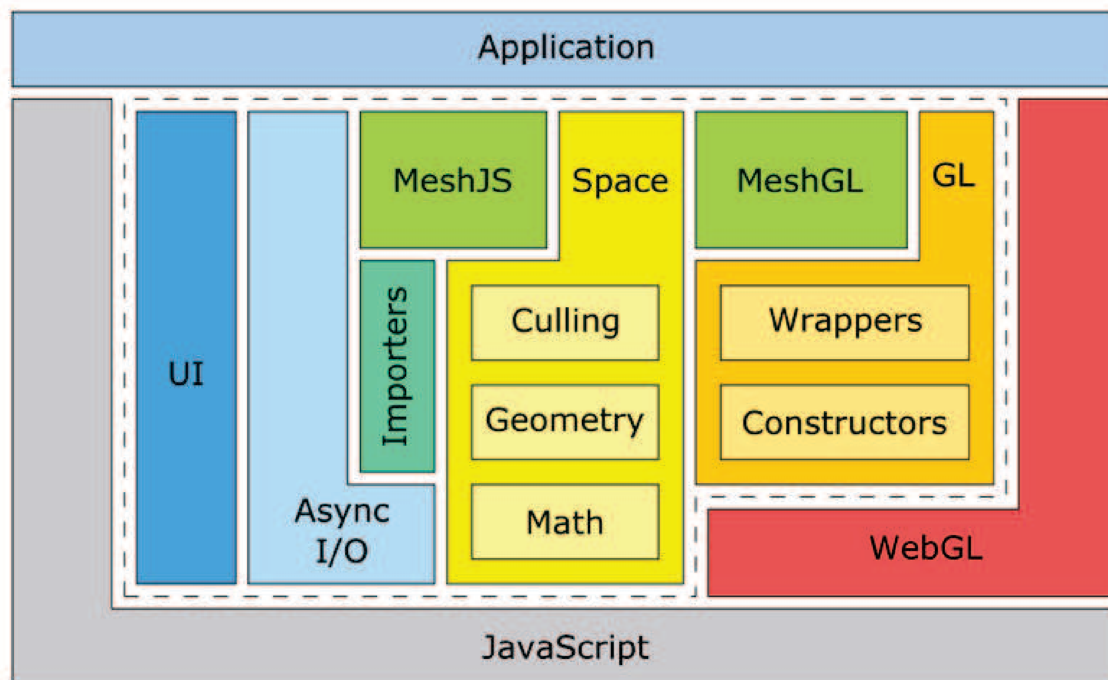


Figure 5.1: **SpiderGL Architecture.** The library is logically composed of five modules: MATH (linear algebra and geometry, GL (WebGL wrapping), MESH (polygonal mesh definition and rendering), ASYNC (asynchronous content loading) and UI (event handling and interactors).

```
3 var v2 = [ x2, y2, z2 ]; // 3rd vertex coordinates
4 var normal = sglNormalizeV3(
5     sglCrossV3(sglSubV3(v1, v0), sglSubV3(v2, v0)));
```

As it can be noted, a complex expression is expressed by nesting function calls. Moreover, the lack of overloading (which does not exist in JavaScript) imposes the use of unique names for distinguishing on the input types (in the above example, the `V3` suffix is used to identify the subset of functions working with 3-dimensional vectors).
The high-level layer is composed of classes that wrap the low-level layer. The above example would become:

```
1 var v0 = new SglVec3(x0, y0, z0);
2 var v1 = new SglVec3(x1, y1, z1);
3 var v2 = new SglVec3(x2, y2, z2);
4 var normal = v1.sub(v0).cross(v2.sub(v0)).normalized();
```

The choice on which level of abstraction to use is up to the developer.

Space-Related Structures and Algorithms. Another important module at the foundation of a 3D graphics library comprises standard geometric objects, as well as

space-related algorithms. SpiderGL offers a series of classes representing such kind of objects, like rays for intersection testing, infinite planes, spheres and axis aligned boxes, coupled with distance calculation and intersection tests routines.

Hierarchical Frustum Culling. When operating over a network, it is reasonable to assume that the content retrieval has an consistent impact on the overall performance. Since multimedia context began to be widely used in web documents, it was clear that a sort of multiresolution approach should have to be implemented to compensate for the transmission lags, giving the user a quick feedback, even if at a lower resolution (i.e. progressive JPEG and PNG). Following this principle, geometric *Level–Of–Detail* (LOD) techniques (see Section 2.3.2) are used to implement a hierarchical description of a three-dimensional scene, where coarse resolution data is stored in the highest nodes of a tree-like structure while full resolution representation is available at the leaf level. To ease the use of hierarchical multiresolution datasets, SpiderGL provides a special class, `SglFrustum`, which contains a series of methods for speeding up the visibility culling process and projected error calculation for hierarchical bounding volumes hierarchies.

Matrix Stack. Users of pre-programmable (*fixed pipeline*) graphics libraries relied on the so called *transformation matrix stacks* for a logical separation among the projection, viewing and modeling transformations, and for a natural implementation of hierarchical relationships in composite objects through matrix composition. Even if this has proven a widely used pattern, it no longer exists since version 2.0 of OpenGL|ES (it was claimed that its introduction in the specifications would have violated the principle of a bare-bones API). We thought that this important component was indeed essential in 3D graphics, so we introduced the SglMatrixStack class, which keeps track of a stack of 4x4 transformation matrices with the same functionalities of the OpenGL matrix stack. Moreover, the SglTransformStack comprises three matrix stacks (projection, viewing and modeling) and represents the whole transformation chain, offering utility methods to compute viewer position, viewing direction, viewport projection of model coordinates to screen coordinates and the symmetric unprojection.
Note that, for practicality of use, we decided to have the modeling and viewing transformation stacks separated, contrarily to the single OpenGL *ModelView* stack.

GL: WebGL Interface

The GL module is the core rendering part of the SpiderGL library. It is composed of two sub–modules, *constructors* and *wrappers* which, in different ways, simplify the management of native WebGL objects by providing a simple but completely configurable abstraction layer.

GL: Constructors. WebGL specifications expose an extremely low level API, according to the base philosophy of being a lightweight, highly configurable and high performances graphics infrastructure. However, a series of typical usage patterns can be extrapolated from most of the 3D graphics applications. As an example, the sequence of command for creating a shader program or framebuffer object are almost always the same block of code. The `GL` module exposes a number of easy-to-use creation functions that hide the most common operations and parameters to the developer.
However, since the user should be able to control all the low level details which are exposed by the WebGL standard, SpiderGL allows to override default parameters with *options* function parameters, in the philosophy of the JavaScript programming style. The general creation function would be:

```
1 var objInfo = sglCreateSomeGLObject(gl, arg1, ..., argN, options);
```

where `gl` is the WebGL context object, `arg1..N` are mandatory object-specific parameters (e.g. pixel format, width and height for a texture image object), and `options` is a JavaScript object of the form

```
1 var options = {
2     objSpecificParam1 : nonDefaultValue1,
3     ...,
4     objSpecificParamK : nonDefaultValueK
5 };
```

That is, whenever the user needs to override default values, a specific field in the `options` parameter can be specified with its respective value. This simple mechanism is indeed a powerful way to lessen the burden of library users in frequently performed tasks, while giving them the all the control they need whenever default parameters do not suffice.
For every type of WebGL object, the developer can use the corresponding SpiderGL creation function which returns a JavaScript object of type `Sgl`*`ObjectType`*`Info` with no associated logic (e.g. methods) where every field corresponds to an attribute of the native object. As an example, Listing 5.2 shows the `SglTextureInfo` structure associated to WebGL textures, created by a call to `sglCreateTexture2DFromData()`.

```
1  // SglTextureInfo structure
2  var texInfo = {
3      handle    : aWebGLTexture,     // native WebGL texture object
4      width     : 512,               // image width in pixels
5      height    : 512,               // image height in pixels
6      format    : gl.RGBA,           // 4-channel pixel format
7      minFilter : gl.LINEAR,         // minification filter
8      magFilter : gl.LINEAR,         // magnification filter
9      wrapS     : gl.CLAMP_TO_EDGE,  // addressing in U direction
10     wrapT     : gl.CLAMP_TO_EDGE   // addressing in V direction
11 };
```

Listing 5.2: **Object Info Structure.** A SpiderGL `Sgl`*`ObjectType`*`Info` structure contains the native WebGL object handle, as well as all the parameters set up at

construction phase. This example shows the structure of a texture object created by a call to `sglCreateTexture2D()`.

In the case of container objects like shader programs, WebGL functions are used to retrieve object specific values which are not part of the construction parameters set. This is the case of shader uniform locations and vertex attributes binding points. Whenever an object is created with these utility functions, the developer can freely use its `handle` field to directly work with WebGL calls.

GL: Wrappers. The use of a low-level API often requires a sequence of calls even to accomplish a simple task and even after object/resource creation and initial setup. For this reason, every WebGL object has a corresponding higher-level wrapper which takes care of the usage details.

Wrapper objects constructors parameters are the same used in their corresponding lower-level creation functions, but they also have overloaded versions which take the single `Sgl`*`ObjectType`*`Info` structure. This is particularly useful when more developers are involved in a large system: in this case, everyone can choose the best level of abstraction which fits his or her habits. If WebGL resources are globally accessible, the developer of a first module can decide to use the native `handle` reference, while in another module other people could choose a higher level approach by creating a wrapper object around the low level object definition. For example, a shader program info structure can be directly passed to the corresponding wrapper object:

```
1 // wrap a shader program
2 var prgWrap = new SglProgram(gl, prgInfo);
```

To ensure consistency, every wrapper contains a `synchronize()` method which retrieve the salient attribute values which could have been changed with native WebGL calls. For performance reasons, the synchronization is not automatically performed but should be explicitly invoked.

MESH: Mesh Manipulation and Rendering

One of the fundamental parts of a graphics library consists of data structures for the definition of 3D objects (meshes) and their rendering.
As in many libraries for polygonal meshes, SpiderGL encodes a mesh as a set of vertices and connectivity information. Following the philosophy WebGL, a vertex can be seen as a bundle of data, storing several kind of quantities such as geometric (position, surface normal), optical (material albedo, specularity) or even *custom* attributes. The *connectivity* describes how these vertices should be connected to form geometric primitives, such as line segments or triangles.
As the representation of meshes is tightly related to their intended use, SpiderGL supplies two different data structures: the first one, `SglMeshJS`, resides in *client scope*, i.e. in system memory, where it can be freely accessed and modified within the

user script; the other is `SglMeshGL`, which is the image of a mesh in the GPU memory under the form of WebGL vertex and index buffer objects. Crucial for memory and execution efficiency is how the vertex set and the connectivity information is laid out; in the following paragraphs we will describe our solution, mainly dictated by the JavaScript language and the WebGL execution model.

Vertices Memory Layout. There are two main data layouts which can be used to store vertex data: *array-of-structs* or *struct-of-arrays*.
In the first case, a vertex is represented as an object containing all the needed attributes: the vertex storage thus consists of an array of such vertex objects.
In the second case, an array is created for each vertex attribute: in this case the vertex storage is a single object whose fields are arrays of attributes, where a vertex object is extracted by selecting corresponding entry in each array.
SpiderGL adopts the struct-of-arrays layout for two reasons: first, JavaScript runtime performs more efficiently when working with homogeneous arrays of numbers rather than arrays of generic object references; second, adding and removing attributes is easily accomplished.
In a similar way, the GPU-side mesh (`SglMeshGL`) stores its vertices with a dedicated *vertex buffer object* (VBO) for each attribute. Such a choice for the GPU layout can be objected by claiming that the use of *interleaved vertex arrays* is more efficient, because interleaving attributes results in a more efficient memory access pattern, which is true when we solely consider the memory bandwidth used by the isolated system composed of the *GPU memory* and the *vertex puller* stage of the graphics pipeline, that is, the first stage of the pipeline, which is in charge of fetching vertex attributes from memory, assemble them in a memory packet and send it to the subsequent stage (in this case, the Vertex Processor stage, executing a custom vertex shader). In this subsystem, the prefetching policy of the *pre-vertex-shader cache* mitigates the transfer latency.
However, when looking at the whole rendering pipeline in real-life scenarios, we see how the most relevant part of the execution time is spent in the vertex shader (whose overall performance is increased by the *post-vertex-shader cache*), in the fragment shader, in texture accesses and framebuffer writes or compositing (blending) operations. In our experiments, these bottlenecks made the benefits of the interleaved layout not even measurable.
These considerations supported our choice in adopting the *struct-of-arrays* layout for both Application-side and GPU-side meshes.

Connectivity Memory Layout. The connectivity can be implicitly derived from the order in which vertices are stored or, more frequently, explicitly described with a set of vertex indices. In SpiderGL it is possible to represent both of them with, respectively, *array primitive* or *indexed primitive streams*.
A SpiderGL mesh may contain more than one primitive stream; for example it may

contain a primitive stream for the triangles and one for the edges in order to render the object in a filled or wireframe mode; the main reason behind this choice is that OpenGL|ES 2.0 (and thus WebGL) specifications does not contain any routine to select the mode in which source geometric primitives should be rasterized. For example, such a routine (known in desktop OpenGL as `glPolygonMode`) could be able to setup the rasterizer in order to draw just the edges of a triangle primitive.

Overcoming WebGL limitations. When using indexed primitives in WebGL, the native type for the elements in the index array can be `UNSIGNED_BYTE` (8-bit unsigned integer) or `UNSIGNED_SHORT` (16-bit unsigned integer); thus, the maximum addressable vertex has index $2^{16}-1$. SpiderGL automatically overcomes this limitation by splitting the original mesh into smaller sub–meshes. In order not to burden the user with special cases when converting an `SglMeshJS` to its renderable representation, we introduced the *packed-indexed primitive stream* for `SglMeshGL`, which transparently keeps track of sub-meshes bounds without introducing additional vertex or index buffers.

Rendering. In WebGL the rendering process involves the use of shader programs, vertex buffers and, often, index buffers and textures. Central to the graphics pipeline is the concept of binding points, that is, named input sites to which resources are attached and from which pipeline stages fetch data (see Figure 5.2).

More in detail, rendering a mesh with WebGL consists of the following steps:

1. Attach mesh vertex buffers to named vertex attributes binding sites
2. In case of indexed primitives, attach the index buffer to the primitive index binding site
3. Bind a shader program to the vertex and fragment processing stages
4. Establish a correspondence between vertex attribute binding points and vertex shader input attributes
5. Invoke the rendering command

In SpiderGL, mesh rendering is efficiently accomplished with the following code:

```
1 sglRenderMeshGLPrimitives(mesh, primitivesName, program,
     vertexStreamMapping, uniforms, samplers);
```

where the arguments are:

`mesh`: the `SglMeshGL` to render

`primitivesName`: a string referring to the named connectivity set in the 3D object

`program`: the shader program to be used for rendering

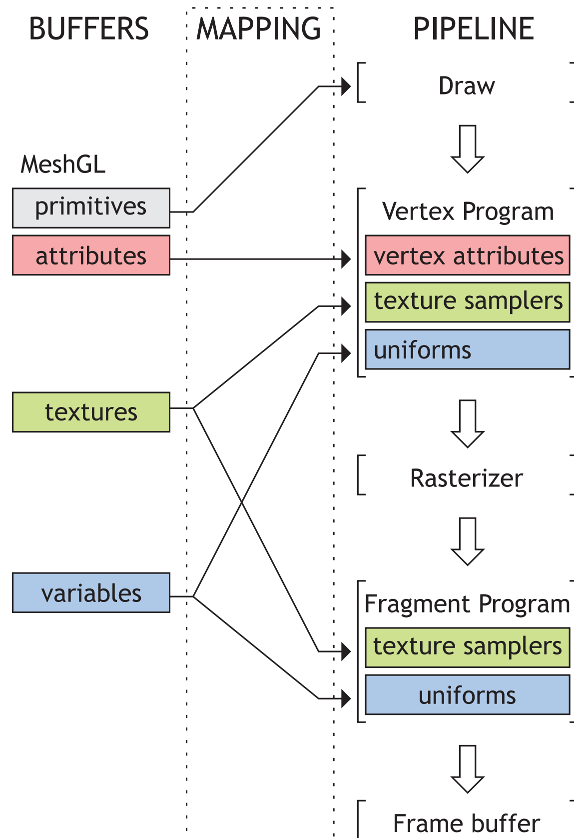


Figure 5.2: **Mesh Rendering Setup.** Vertex buffers, index buffer and textures are attached to named binding sites. Then a series of correspondences is established between: a) mesh vertex streams and vertex shader attributes, b) texture units and shader texture samplers and c) application values and shader uniforms.

`vertexStreamMapping`: a correspondence between named mesh vertex attributes and vertex shader input attributes

`uniforms`: uniform values for the shader program

`samplers`: a correspondence between texture sampler shader uniforms and input texture images.

In particular, the last three arguments are JavaScript objects of the form:

```
{
  parameterName1 : parameterValue1,
  ...,
  parameterNameK : parameterValueK
}
```

Even if a call to `sglRenderMeshGLPrimitives()` is easy to use and suffices for several use cases, it is not the most efficient way to render a mesh when multiple

instances of the same object must be rendered, and each instance differs from the others by a small subset of uniform parameters. As SpiderGL is intended to simplify the usage of WebGL, it must also provide the most efficient mechanisms to meet the developer's special use cases. For this reason, SpiderGL offer the `SglMeshGLRenderer` helper class. This class takes care of all the steps listed above, minimizing the number of binding operations in case of multiple instances rendering and providing smart mechanisms which allow the user to explicitly setup vertex attributes correspondences, shader program uniforms and texture samplers. Listing 5.3 shows how this helper class can be used to efficiently perform the rendering of multiple instances of a same `SglMeshGL` object. In the actual implementation, the function `sglRenderMeshGLPrimitives()` internally uses a `SglMeshGLRenderer` object.

```
1  // -- initialization --
2  // create a renderer object
3  // bound to a shader program
4  var r = new SglMeshGLRenderer(program);
5
6  // -- rendering --
7  // setup a correspondence between mesh vertex attributes
8  // and vertex shader input attributes:
9  // <name : value> : maps mesh vertex attribute <name>
10 // to vertex shader input <value>
11 var streamMapping = {
12   position : "a_position",
13   normal   : "a_normal",
14   texcoord : "a_texcoord"
15 };
16
17 // define uniform values, common to every object instance
18 var commonUniforms = {
19   lightDirection = [ 0.0, 0.0, -1.0 ]
20 };
21
22 // define per-instance uniform values
23 var instanceUniforms = {
24   modelViewProjectionMatrix : null,
25   baseColor : null
26 };
27
28 // define per-instance texture samplers
29 var instanceSamplers = {
30   diffuseMap : null
31 };
32
33 // perform rendering
34 r.begin();
35   r.setStreamMapping(streamMapping);
36   r.setUniforms(commonUniforms);
37   for (var i in meshes) {
38     var mesh = meshes[i];
```

```
      r.beginMesh(mesh.meshGL); // bind vertex buffers
        r.beginPrimitives("triangles"); // bind index buffer
        for (var j in mesh.instances) {
          var inst = mesh.instances[j];
          instanceUniforms.modelViewProjectionMatrix = inst.mvp;
          instanceUniforms.baseColor = inst.baseColor;
          r.setUniforms(instanceUniforms);
          instanceSamplers.diffuseMap = inst.diffuseMap;
          r.setSamplers(instanceSamplers);
          r.render(); // issue draw command
        }
        r.endPrimitives();
        r.beginMesh(meshes[m].meshGL);
      r.endMesh();
    }
  r.end();
```

Listing 5.3: **Mesh Rendering.** The `SglMeshGLRenderer` object is used to efficiently render multiple instances of several meshes, minimizing the cost of binding WebGL buffers and setting up shader uniforms.

Importers: Input Formats. Content formatting (geometry, images, shader effects code etc.) has been carefully kept away from OpenGL-class graphics libraries, due to their application-specific nature; therefore, the importers are not really part of the kernel of SpiderGL. At the present SpiderGL supports COLLADA [65], the Alias|Wavefront `OBJ` format for basic three-dimensional objects and, due to the extremely simple definition of mesh, the `JSON` [27] object serialization framework.

ASYNC: Asynchronous Content Loading

Several CG applications, especially the ones which perform multiresolution rendering, require the ability of asynchronous loading in memory from the disk or from a remote location. Usually this is implemented using a multithread architecture with prioritized request queues which prevent the rendering from freezing during data transfers. JavaScript still does not officially support multithreading, so the asynchronous loading of remote content is typically done by using `XMLHttpRequest` and `Image` objects, appropriately set up with callback functions which will be invoked whenever the transfer of the requested data has completed. SpiderGL uses this mechanism to implement prioritized request queues. The following code snippet shows how to use a priority queue to create textures from remote images:

```
// create a request queue
// with a maximum number of ongoing requests
var requestQ = new SglRequestQueue(maxOngoingReqCount);

// define a callback function that will
// create a texture when the image data is ready
```

```
 7 var textures  = { };
 8 var callback = function(request) {
 9     textures[request.url] = new SglTexture2D(gl, request.image);
10 };
11
12 // instantiate requests and push it into the queue
13 var req1 = new SglImageRequest("sourceURL1", callback);
14 requestQ.push(req1); // continue issuing requests
15
16
17 // usage
18 if (req1.success) {
19   // access texture object with textures[req1.url]
20 }
```

The status of the request can be queried at any time, allowing the application to take different code paths depending on the resource availability or even abort the request. This whole mechanism is particularly useful when multiresolution algorithms and data structures are employed for tasks such as 3D navigation in large environments.

UI: Event Handling and Interactors

To access the WebGL layer within a web page, a specific context object must be requested from an HTML `canvas` object. The HTML rendering engine will then issue a page composition operation whenever it detects changes to the associated WebGL framebuffer. In every interactive application, the displayed content is often a consequence of some kind of user interaction. SpiderGL provides an event handling subsystem which reflects the philosophy of the GLUT library [69]. GLUT allows application developers to selectively install custom callbacks for the most common user input event types, such as keyboard key presses or mouse motion. We translated the callback approach used in GLUT in a more object oriented way. The developer can create a generic object and register it to an HTML `canvas` element; events raised from the canvas will be then intercepted by the user interface module which in turn will dispatch them to the registered object. The correspondences between events and event handlers are resolved by simply inspecting the registered object and looking for methods with predefined names. In addition to event dispatching, the registered object will be augmented by a `ui` field which can be used to access the logging system, as well as important information such as tracked input state and canvas properties.
SpiderGL also provides most of the common tools used in interactive application aimed at conveying user input into 3D scene exploration and manipulation. These include, for example, a camera interactor which implements the typical paradigm used in first-person shooter games (`SglFirstPersonCamera`) and a trackball manipulator (`SglTrackball`) for object inspection with pan, zoom, rotation and scaling operations.

5.3 Using SpiderGL

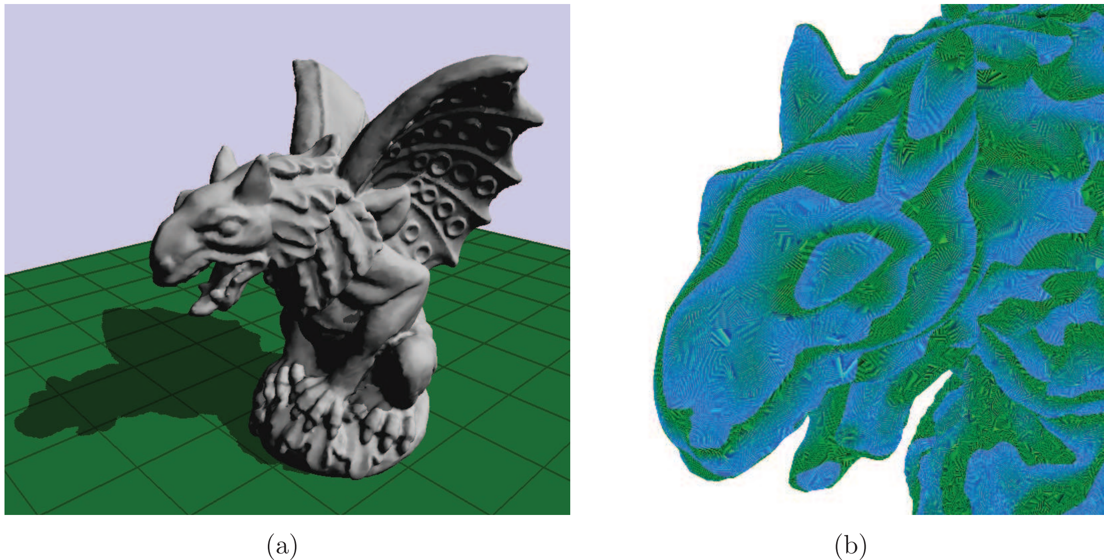
At the time of writing, the WebGL specification is still in draft version and it is only implemented in the experimental version of major web browsers. We successfully tested our library with the latest builds of the most common web browsers on several desktop systems. The results presented here have been run on the Chromium web browser on a Windows Vista system with Intel i7 920 processor, 3 GB RAM, 500 GB Hard Drive and an NVIDIA GT260 graphics board with screen vertical synchronization disabled. The collected results should be analyzed by considering that a minimal HTML/JS page that only clears the color buffer reaches the limit of exactly 250 frames per seconds; we suspect that some kind of temporal quantization occurs in the browser event loop.

5.3.1 WebGL Limitations and Standard CG Algorithms

As noted in Section 5.1, WebGL can be considered as a one–to–one mapping of OpenGL|ES 2.0 functions to a JavaScript API: this means that not every functionality available in even not–so–recent versions of standard OpenGL is available to the developer.

Shadow Mapping. The first example consists of rendering a 100K triangles mesh using Phong lighting model and a 1024^2 shadow map (see Figure 5.3(a)), which can be done at full framerate (250 FPS). As WebGL–capable devices are most likely mobile smartphones and PDAs which uses tile–based rendering hardware, the ability to use integer–format textures as depth buffers is missing. This capability stands actually at the foundations of a shadow mapping algorithm. Thus, in order to implement shadow mapping with WebGL, a standard color texture must be used to encode depth data. In the example, this is accomplished by encoding the single floating point depth value in a conventional 32-bit RGBA texture image (see Figure 5.3(b)).

Large Meshes. To highlight the capabilities of the *packed-indexed primitive stream* (see Section 5.2.1), Figure 5.4(a) shows a 3D scan of Michelangelo's David statue composed of 1M triangles. The model outreaches the maximum value for vertex indices ($2^{16} - 1$) and is thus automatically subdivided into smaller chunks (nine in this case), highlighted by different colors in Figure 5.4(b). The performances here range very inconstantly with peaks of 250. This is probably due to the way the timer event is scheduled by the browser.



(a) (b)

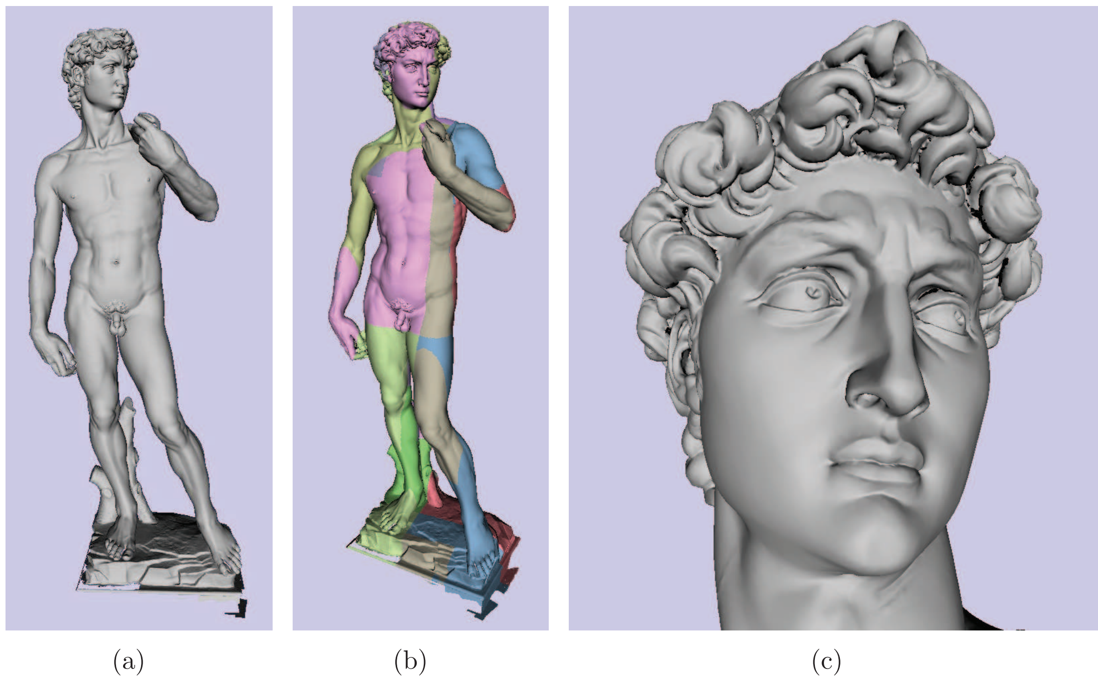
Figure 5.3: **Shadow Mapping.** (a) A scene composed of 100K triangles is rendered at the maximum reachable speed of 250 FPS. (b) The 1024^2 shadow map (here a detail closeup) has been packed on a 32-bit RGBA texture because the WebGL specification does not support depth textures. Note how the background depth has been encoded to pure white.

5.4 Large Datasets on the Web

The large availability of geometric and color data, as well as network connections able to transfer up to several megabytes of data per second, impose a rendering library to be designed to maximize the performances in those areas that are typical of multiresolution rendering algorithms. As described in Chapter 2, the need for a multiresolution approach comes into play whenever we want to show datasets that are too large to be handled w.r.t. the hardware capabilities: in particular, the resources that mainly influence the output of a multiresolution renderer are System and Video RAM speed and amount, as well as the raw CPU and GPU performances. In general, the building blocks of a multiresolution renderer are:

1. a hierarchical layout of the data
2. an algorithm to visit such hierarchy and determine the nodes to use for producing the current frame
3. the ability to load the nodes asynchronously, i.e. to proceed with rendering while missing data are being fetched.

With this in mind, SpiderGL provides the tools needed to perform the full chain of coordinate space transformations (useful for screen error calculation), hierarchical



(a) (b) (c)

Figure 5.4: **Large Meshes.** Vertex index limit is automatically overcome with *packed-indexed primitive stream*. Here a model of the Michelangelo's David statue with 1M triangles is rendered at about 100 FPS on a web browser. The Figure shows: a) the whole mesh, b) the colored chunks after splitting and c) a close-up of the statue head.

visibility culling and asynchronous data fetches. With these facilities built inside the library, we developed a compact and easy–to–use framework to aid the developer during the development of a hierarchical multiresolution visualization system. The proposed framework is as agnostic as possible to the underlying data representation, as long as the following properties hold:

- **tree structure**: the underlying hierarchical structure can be explored as a k–ary tree
- **conservative occupancy**: the bounding volume of an internal node also bounds all its children
- **conservative error**: the error associated to nodes does not increase as the hierarchy is descended.

Once the appropriate callbacks for data fetching, graphics resources creation and destruction, and rendering are set, at each frame the framework will explore the hierarchy to determine nodes visibility and their associated on–screen projected error, will issue data fetches and perform the actual rendering.

Caching. The general data flow and cache hierarchy used in compiled remote applications (remote online repository, disk, system RAM and video RAM) can not be explicitly implemented in web application due to limitations imposed by the restrictive permissions policy adopted for security reasons by web browsers; for example, it is not possible to create and write files in the local file system. This means that we can not explicitly implement the disk cache stage. Actually, even though we will not have explicit control over disk usage and cache eviction policy, by using standard `Image` and `XmlHttpRequest` objects we will automatically take advantage of the caching mechanisms implemented by the browser itself. In fact, every standard web browser caches recently transferred data in the local file system (and even system RAM), thus transparently providing a disk cache level.

Multithreading Considerations. In an out–of–core, multiresolution visualization system we can identify three main tasks that can be theoretically executed in parallel by separate threads: data fetching, hierarchy visit and selection, and rendering. Data fetching is the most time consuming task, because of the relatively slow data transfer channels (network and secondary storage disks), while the hierarchy visit and selection is often, and as in our cases, the lightest one and does not explicitly needs to be run in parallel with the rendering algorithm.
A big implementation effort often arises in the development of the data management, i.e. fetching and caching. As soon as data arrives, it is stored in the appropriate cache level and eventually flows down the cache hierarchy to finally reach the application. The management of cache levels and transfers among them is usually implemented in a multithread framework, where, conceptually, each level is handled by a separate thread. However, as discussed in the previous paragraph, it is not possible to implement an explicit cache hierarchy. Moreover, at the time of writing, multithreading support in web browsers is still in its early stage. For these reasons our use cases are limited to a single thread.

5.4.1 Terrain Models

To show how the algorithms and data structures in SpiderGL can be easily used to integrate virtual 3D exploration inside web pages, we implemented a simple but effective multiresolution terrain viewer. Our approach consists of an offline preprocessing step which creates a multiresolution representation from a discrete elevation and color image, and an online out–of–core rendering algorithm.
As in other existing map–based web applications like Google Maps [47], the multiresolution dataset is organized in tiles. More in detail, in the construction phase, the elevation map is first embedded on a quadtree with user–defined depth. The depth of the tree determines the dimension of the tiles in which the input map is first partitioned. In fact, such tiles correspond to the 2D projection of the bounding

box of the leaf nodes. We build the multiresolution dataset in a bottom–up fashion by first assigning the tile images to their leaf nodes. Then, tiles for internal nodes are generated by assembling the four tiles assigned to the node children in a single square image of twice the dimensions and then downsampled by a factor of two. This means that all the tiles have the same dimension and, in particular, tiles assigned to nodes at level i have half the linear resolution of the ones at level $i+1$. To ensure that the border of adjacent tiles match exactly to avoid cracks and discontinuities, the square regions of the elevation tiles are expanded by one pixel on each side. This also done for the input color map to allow correct color interpolation at borders when bilinear filtering is used. The output of the preprocessing step are two texture images for each node: a RGBA image stores the surface color (RGB channels) and the height map (Alpha channel), and a RGB image for normal maps. Once the nodes to be rendered are identified by the multiresolution framework described above, the rendering algorithm draws each tile by using a vertex shader which performs displacement mapping on a regular triangle grid. As the basic algorithm does not handle junctions whenever two adjacent tiles at a different resolution are drawn, the grid mesh is augmented with *skirts* (see Figure 5.5) to avoid visible holes in the surface.

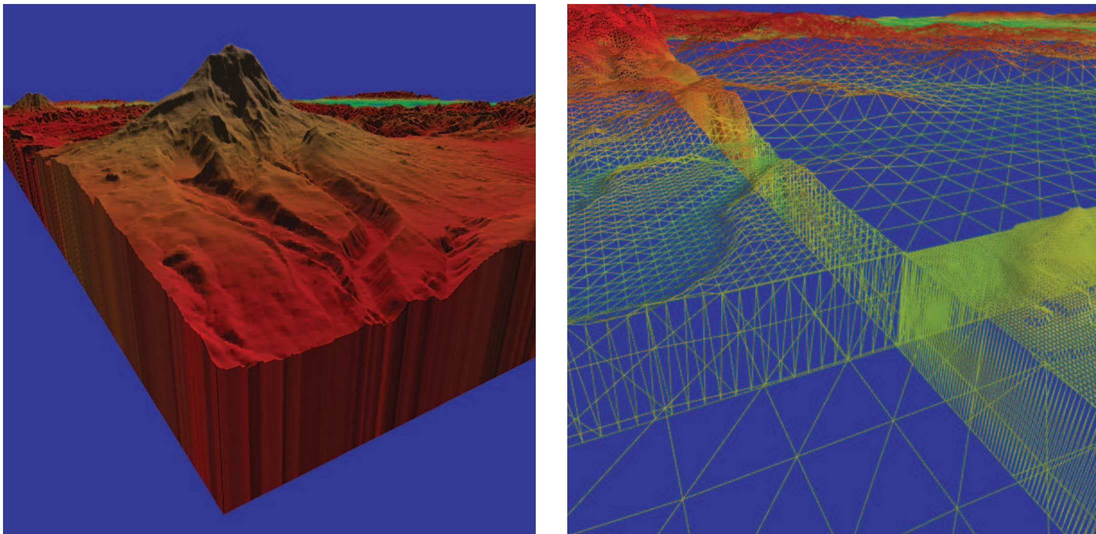


Figure 5.5: **Terrain Tiles with *Skirts*.** To avoid holes that would appear when rendering two adjacent tiles at different resolutions, tile geometry is augmented with *skirts*, that is, vertical surfaces that extrude tile boundaries down into the terrain.

Figure 5.6 shows a snapshot of a terrain rendering demo. The terrain is encoded by a quadtree of a $4K \times 4K$ texture storing elevation data. A regular grid of triangles is used to render each quad, by fetching the elevation of the vertices in a vertex shader.

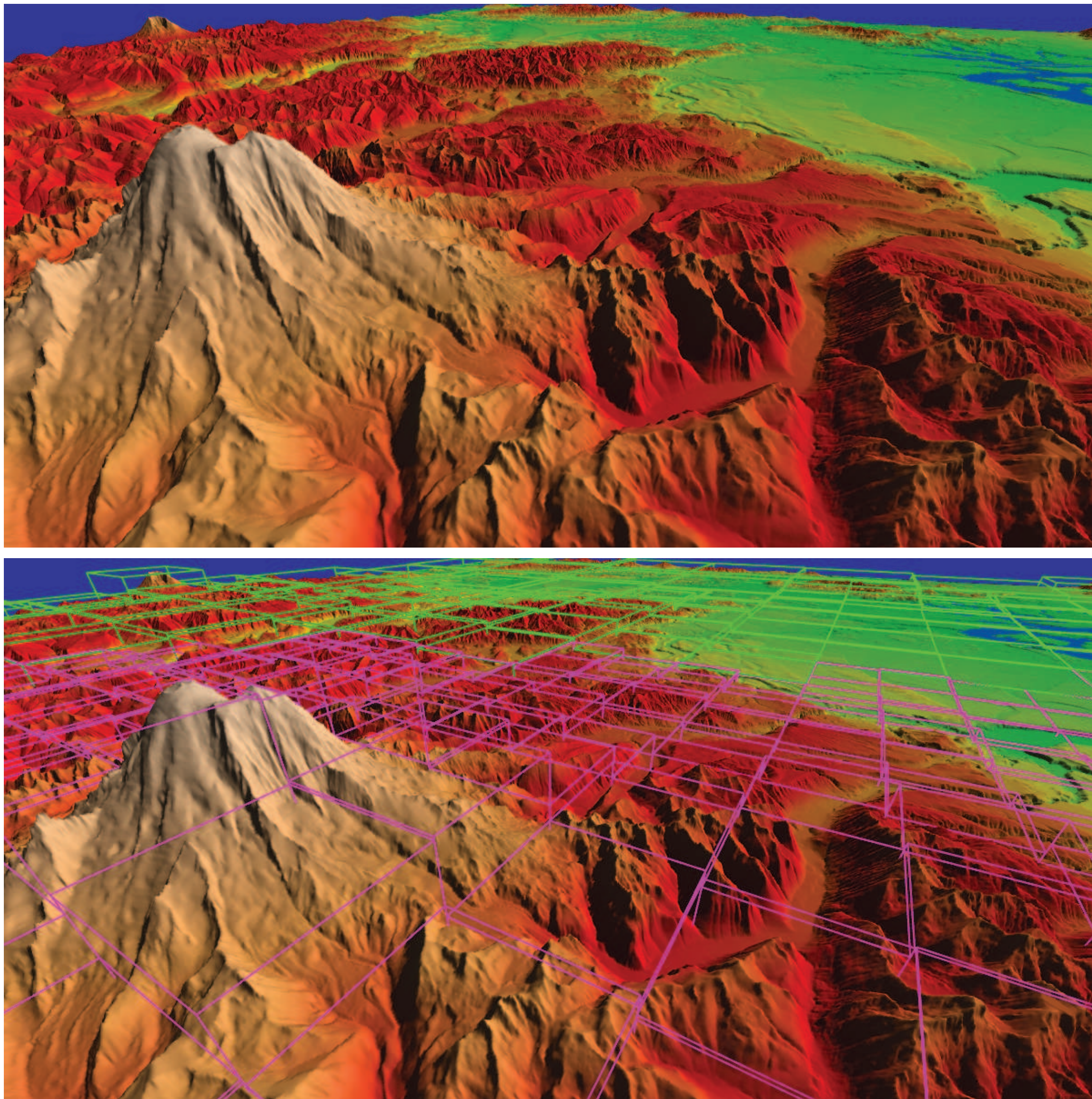
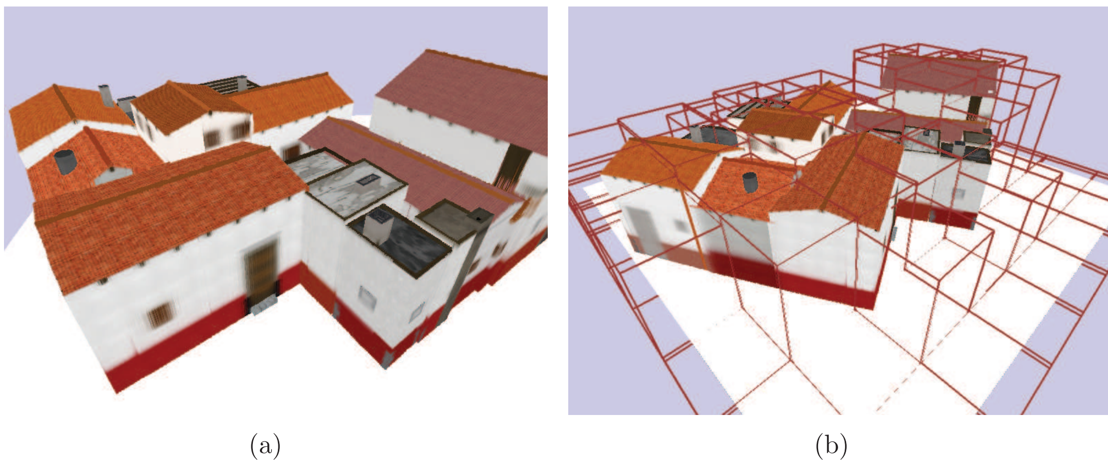


Figure 5.6: **Multiresolution Terrain Visualization.** *Top*: a snapshot of an adaptive multiresolution rendering of a $4k \times 4K$ terrain (Pudget Sound model). *Bottom*: the same terrain view with nodes bounding boxes shown: green and magenta boxes represent, respectively, internal and leaf nodes.

5.4.2 Urban Models with BlockMaps

To show the real potential of WebGL, we implemented a multiresolution renderer of urban environments using the BlockMap data structure described in Chapter 4. One of the main advantages of this novel representation is that it can be directly encoded into plain 32-bit RGBA images, implying that we can use the standard tools natively provided by JavaScript (namely, the `Image`, `Canvas` and `WebGLContext` objects) to

fetch data from remote repositories and upload it to the graphics hardware. Another advantage is that the simplicity of the rendering algorithm and, more importantly, the use of simple instructions in the BlockMap shaders, allow us to write efficient JavaScript code and exploit the actual power of WebGL without any modification. Figure 5.7(a) shows a region of a procedural reconstruction of the ancient Pompei city, rendered using BlockMaps. In this implementation, each BlockMap entry in the original compressed dataset used by the native C++ renderer has been converted to a standard PNG image.



(a) (b)

Figure 5.7: **Ray-Casted BlockMaps.** (a) A fragment shader executes ray-casting over the BlockMaps of a city block at interactive rates (120 FPS). (b) The bounding boxes of the hierarchical structures. The BlockMaps framework exploits the frustum culling and asynchronous data transfer facilities offered by the SpiderGL library.

5.4.3 Polynomial Texture Maps

Although WebGL is primarily designed for three–dimensional graphics, the possibility to use the power of the graphics hardware at pixel processing level makes it an attractive candidate even for complex 2D shading operations. In this case, source images are handled under the form of texture maps, and per–pixel operations are executed by fragment shaders. As an effective use of these capabilities, we used the multiresolution framework to implement a *Polynomial Texture Map* (PTM) viewer. A Polynomial Texture Map [84] is, in brief, a discrete image where each pixel encodes the color as a function of light direction. As in the streaming terrain viewer, a quadtree is built from the the original PTM and, at rendering time, a fragment shader computes the current color as a function of the light position, passed as a global uniform variable.
Figure 5.8 shows 2930×2224 image. The size of the PTM is about 56MB since each pixels must store 9 values (3 for the RGB color and 6 for the PTM coefficients).

Figure 5.8: **Polynomial Texture Maps.** Four frames captured when visualizing a High Quality PTM. Light position is bound to the position of the mouse on the window.

Performances. The use cases discussed in this section were run on a Windows 7 64-bit system, with 6 GBytes of RAM, 1 TByte of disk storage and a NVIDIA GeForce 275 GTX graphics board. As in the examples discussed in Section 5.3, the measured rendering performances for the terrain, BlockMaps and PTM test cases are limited to 250 Hz, probably due to the timer resolution being limited to about four milliseconds. To verify our guess, we performed a very simple test that consisted in just measuring how many times the timer event callback has been invoked in a given amount of time. The result has shown that 250 Hz is the maximum achievable rate. This demonstrates that the presented test cases are able to complete the rendering of a single frame within the shortest possible refresh rate.
In all of our tests we used a texture cache size of 8 MBytes to simulate a low–resource platform (e.g. a low–end smartphone) and we used a controlled bandwidth network as the streaming data channel. Starting from an empty cache, we measured the amount of time needed to fill it entirely. The results have shown that, as expected, the data transfer time over dominates the time needed to prepare the network requests and to upload the textures to video memory. As an example, on a 2 MBit

network channel under standard load, the time elapsed from the first request to the last texture data upload was 36 seconds versus the 32 seconds needed under theoretically optimal performances. In this case the transfer of additional data required by `http` request headers should be considered as a major performance penalty factor.

5.5 Biological Data Visualization

Interactive visualization of molecular structures and physico-chemical data is an important and interesting research field which span from the Computer Graphics world to the Biological and Molecular studies. The amount of complex structures that is available through public repositories and the level of detail of biochemical datasets which can be manipulated by physico-chemical tools has greatly increased in the last years, making it essential to employ dedicated visualization techniques to make an effective use of these data. While it may be easy to draw even large molecular datasets as a series of atoms (Van der Waals spheres) using simple rendering methods based on impostors and other tricks, the precise rendering of a high resolution molecular surface involves the management of more complex geometry. Furthermore, when it is necessary to represent interaction between different molecules or to introduce the rendering of further 3D elements and data layers, the required computational and rendering capabilities do increase significantly.

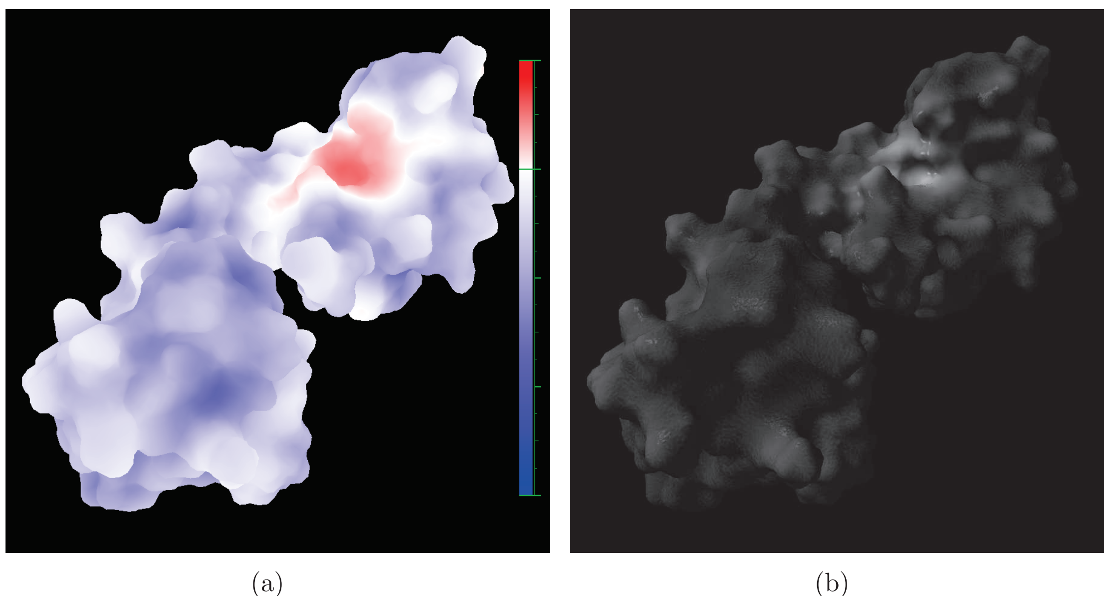
As the SpiderGL library is highly flexible in terms of raw configuration while providing high–level utilities, we implemented an efficient molecular data viewer [17] whose architecture allows high programmability of the final presentation phase.

5.5.1 Visualizing Molecular Properties

Our target is to display two specific biochemical properties on the surface of molecules. The two surface properties were the *Molecular Lipophilic Potential* (MLP) and the *Electrostatic Potential* (EP). The ability of a molecular surface to establish bonds with water is called *Hydrophilicity*; its opposite, which is the ability to establish bonds with fat, is called *Lipophilicity*. The Electrostatic Potential is easier to understand: each atom in a molecule may have a charge, the various charges in the molecule produce an electric field in the surrounding of the molecule.

The main idea of this visual mapping has been to exploit perceptual associations between the values to be mapped and visual characterization of real-world objects. Ideally, by using already established perceptual association, the viewer would be able to understand the provided information more naturally, without the use of explicit legends.

Lipophilic Potential. The visual mapping of lipophilic potential relies on a combination of color, surface roughness and specularity: these three effects are mapped on the molecular surface according to the local lipophilic potential value. For The result is pleasant and, as visible in the left side of Figure 5.9, the characterization of the surface is quite effective.

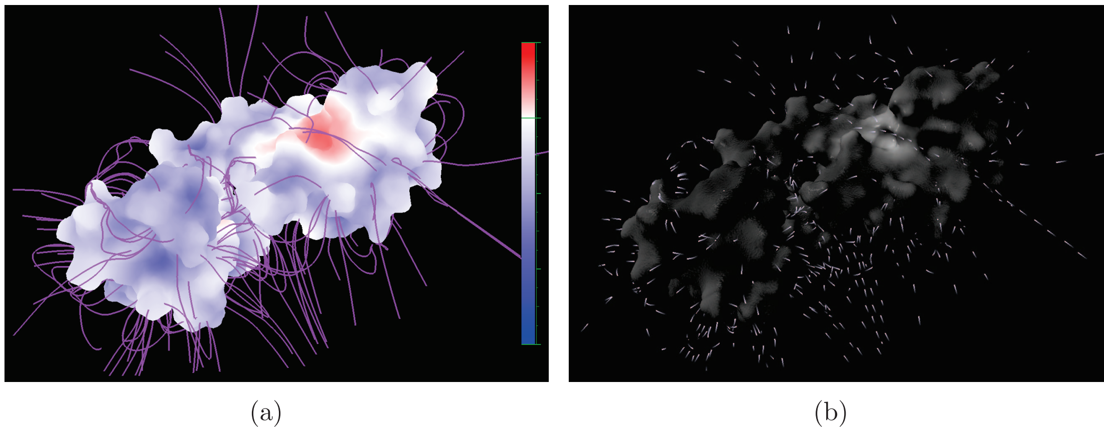


(a) (b)

Figure 5.9: **Molecule Lipophilic Potential** The Lipophilic Potential is mapped on the surface of a Calcium-bound Calmodulin. (a) Visualization using standard color ramp. (b) Visualization using advanced shaders. The light color and the specularity clearly indicates a liphophilic patch on the right part of the molecule, while the dark, dull and rough surface indicates a more hydrophilic area.

Electrostatic Potential. While the MLP value is obviously only observable on the surface itself, electrical phenomena are associated to the idea of an effect projected in the volume surrounding a charged object, and able to affect other objects (like the high school textbook-favorite amber rod attracting paper bits). Field lines are a common way to describe the effect of the electrical field. EP value is therefore represented by showing small particles, moving along the path defined by field lines, visualizing a higher concentration of particles in areas where the electrical fields is stronger.
Particle trajectories can be rendered as a series of solid *line strips* along field isolines, but to emphasize the underlying motion a particle system is often a better choice. As our input test datasets contains explicit particle paths in form of polylines, it is not really necessary to create a particle system, but it is possible to visualize their motion using a GLSL fragment shader which renders only small fragments of the

solid lines according to a periodic function, animated using an offset (see Figure 5.10 and Listing 5.4).



(a) (b)

Figure 5.10: **Molecule Electrostatic Potential.** (a) Particles flow through fixed trajectories described as polylines. (b) The fragment shader in Listing 5.4 sets the output alpha value such that the blending stage will make visible only parts of the line geometry, simulating a particles swarm.

```
1  uniform float u_timeOffset;
2  varying float v_texcoord;
3
4  void main(void)
5  {
6    const float part_density = 4.0;
7    const vec3  part_color   = vec3(0.8 ,0.8, 1.0);
8
9    float val = fract((v_texcoord + u_timeOffset) / part_density);
10   if (val < 0.7)
11     discard;
12
13   val = smoothstep(0.9, 0.7, val);
14   gl_FragColor = vec4(part_color * val, val);
15 }
```

Listing 5.4: **Simulating Particles Motion.** The fragments generated by the polyline rasterizer are blended according to a smooth–varying function of time (see Figure 5.10).

Atomic Structure. In addition to the surface, it is often needed to be able to look at the actual atomic structure of the molecule. For this purpose, we simply use a sphere mesh which represents a single atom and instantiate it once for each atom in the molecule, varying its radius and color accordingly. Figure 5.11 shows

the underlying atomic structure, combined with the lipophilic and electrostatic potential. The rendering is obtained by first drawing the atomic structure and then the surface structure with a modified fragment shader which modulates the output transparency in function of the viewing angle, obtaining a *fresnel-like* effect.

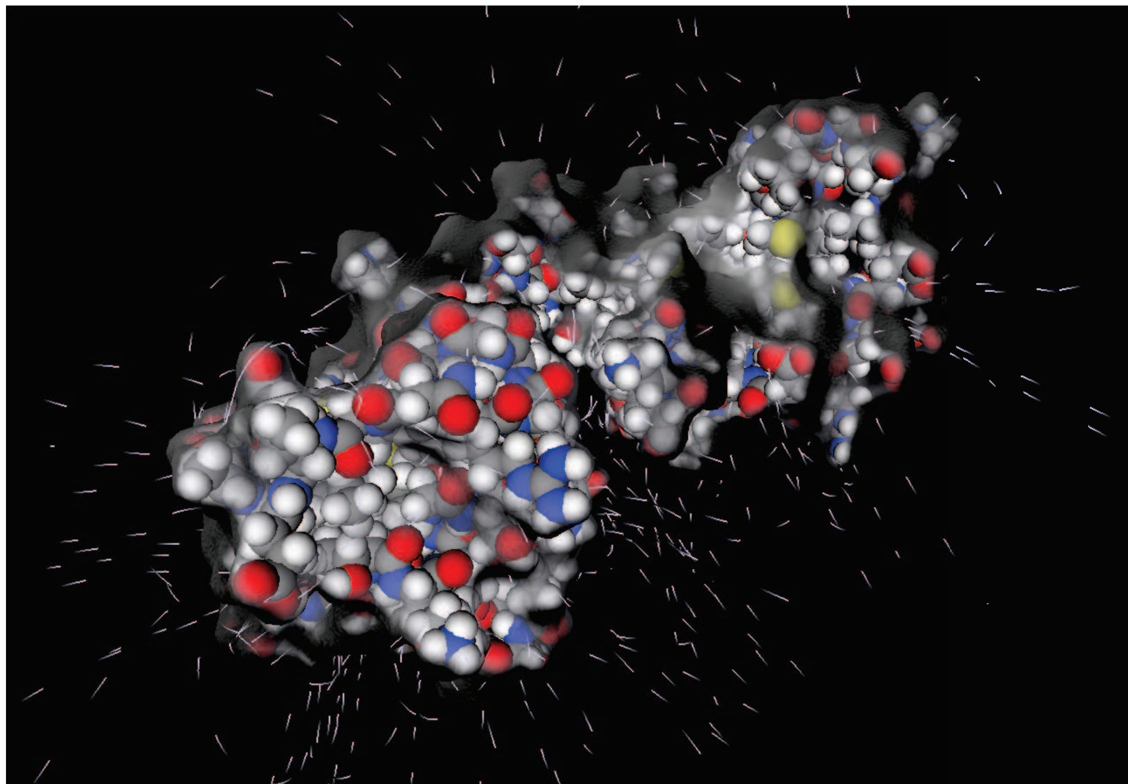


Figure 5.11: **Surface and Atomic Structure**. The superimposition of the Molecular Surface and the underlying Atomic Structure is achieved using a transparency based on view angle.

Performances. Rendering independently a large amount of objects, as the atoms in a complex molecule, evidences the performance penalty that arises from the use of an interpreted language such as JavaScript versus a compiled one like C++. The first implementation of our molecule renderer was composed of a single resource for the atom sphere mesh. The rendering procedure consisted in a loop over the molecule atoms in which, at each iteration, the appropriate transformation matrices were calculated based on the position of the atom relatively to the molecule, the atom color was set in the shader program, and finally the rendering command was issued on a per–atom basis. In this case, using a tessellated sphere with 80 triangles for the atom mesh, a molecule consisting of 2200 atoms was rendered at roughly 25 frames per second, where the 80% of the frame time was taken by matrix computations, evidencing JavaScript performance penalty. In the second implementation we

created the whole molecule mesh by replicating, for each atom, the sphere mesh with per–vertex color and translating it to its final position. This caused the memory needed to store the molecule mesh to rise from 1.4 KBytes to 3.36 MBytes. Paying the larger memory usage had, however, the benefit of increasing the performances to 120 frames per second.

5.6 MeShade: deploying 3D content on the Web

While there are very large repositories for pictures, video or audio files, a web site like Flickr [119] or YouTube [57] for 3D models has yet to come. Up to now there are a few repositories of 3D models made by human modelers that one can browse and also few examples of repository of 3D scanned models [112, 42]. However it is likely that this will change quickly in the near future, both for the increasingly ease of producing 3D models by automatic reconstruction means (for example by cheaper and cheaper laser scanners [89] or by digital photography [117]) and for the ability to use 3D graphics hardware acceleration in the web browser.

MeShade is a web application that allows the user to load a 3D model and images, create a custom shader program (like one can do using, for example, AMD RenderMonkey [7], although at the present with a more limited number of functionalities), and export JSON and HTML code snippets to create a web page which will provide interactive visualization of the mesh using the custom shader.

The user interface of MeShade consists of several collapsible and movable panels (see Figure 5.12), representing the most important parts of a shader composer application. Apart from the interactive preview viewport which displays the loaded 3D model with the current material, the user is provided with text areas for editing the source code of the vertex and the fragment shaders. The user can validate the correctness of the shaders by using the *Validate* button which will show the compiler output (warning and error messages) in the log area. The *Apply* button will apply the shader program to the 3D model.

The way MeShade handles program uniforms and vertex shader attributes is based on predefined names with specific semantic and user-defined input values. In particular:

- every vertex attribute of the mesh is made available to the vertex shader by declaring it with a predefined prefix, i.e. vertex shader attribute `a_position` will be mapped to mesh vertex attribute stream named `position`;
- a series of fundamental and commodity values are exposed via predefined uniform names, like transformation matrices, model bounding box and so on;
- whenever a non-predefined uniform is found, an edit form is added to the HTML DOM which allows for direct editing of the scalar or vector values; the user interface for editing depends on the type of the uniform variable;

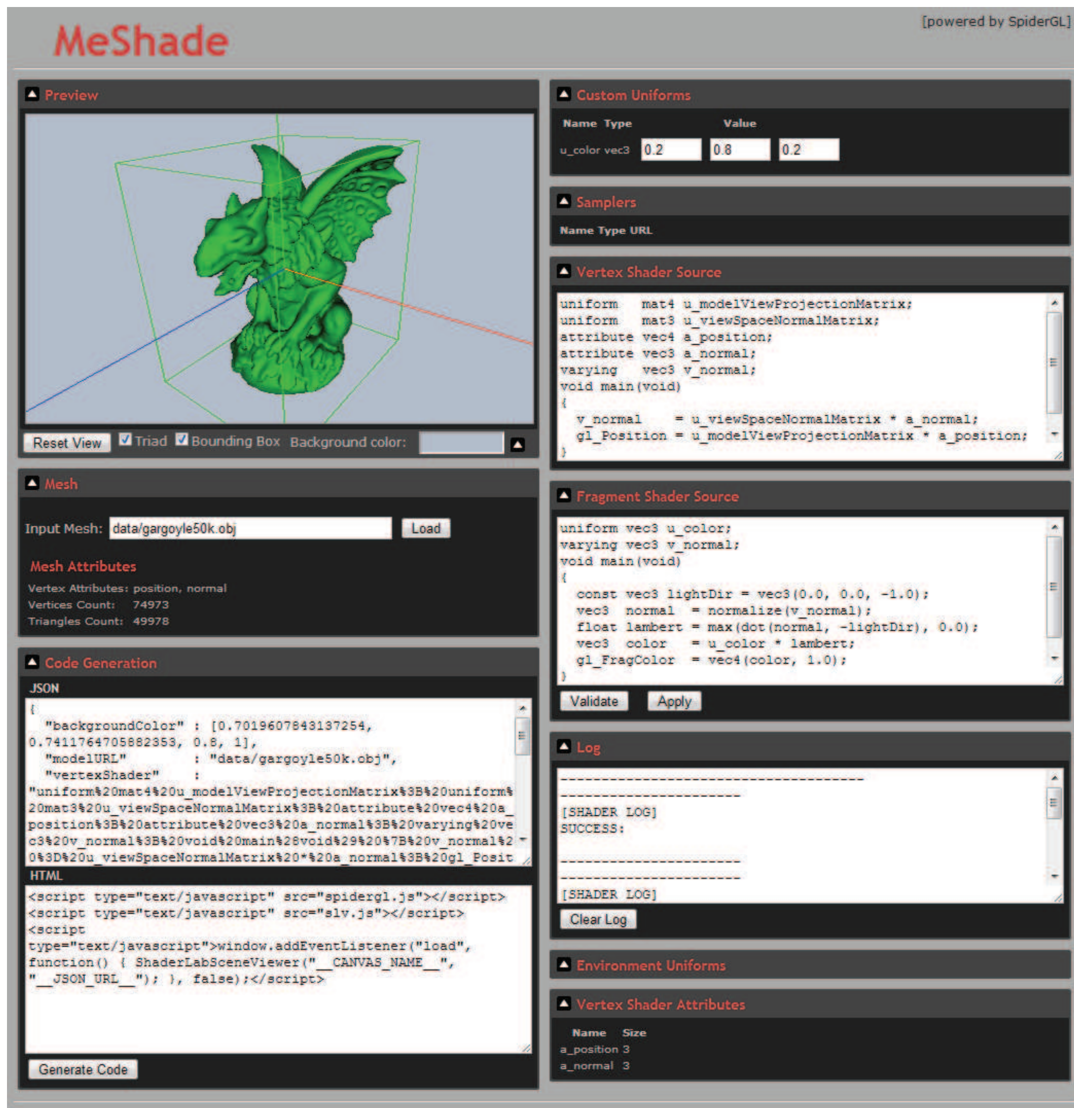


Figure 5.12: **MeShade User Interface.** Once a 3D model URL is specified and data is loaded, the preview viewport displays the mesh using the vertex and fragment shaders written by the user on the page GUI itself. Edit forms for uniforms and texture images are dynamically added to the page as shaders are compiled.

- an image load form is created for every texture sampler uniform; although texture samplers are standard uniforms in the GLSL language, they are grouped in a separate panel to reflect the way SpiderGL handles textures.

The 3D model and the texture images are loaded by specifying their URL and then pressing the corresponding *Load* button. The interface also contains a list of all available predefined uniforms and mesh vertex attributes. The latter ones are updated every time a model is loaded.

Once the user has reached a satisfactory result, he/she can ask MeShade to generate the code to embed the 3D model rendered with the program shader just created within a web page, by pressing the *Generate Code* button. MeShade will generate two code fragments, JSON and HTML, which can be copied to new or existing files.
The JSON section contains the geometry and images locations, as well as the shaders source code and uniform values, and thus serves as a *scene* description file. On the other side, the HTML code contains all the HTML `script` tags to be pasted into existing pages in order to access and visualize the scene.
We decided to generate code snippets rather than a complete HTML page because repository designers are supposed to use their own graphical style throughout their web sites: having only a very few lines of code to embed inside web pages allows for a variety of design choices. Moreover, separating the JSON scene description code allows for sharing the same scene among several web pages without code replication.

Chapter 6

Conclusions

Multiresolution rendering of large urban models and terrain datasets is an active research area. In this thesis we proposed improvements to the state–of–the, introducing a compression scheme for terrain models, a new data representations for urban datasets, and a flexible 3D software library for the web platform.
In this Chapter we summarize our work and propose some possible extensions.

6.1 Advancements in Multiresolution Techniques

The purpose of the research work presented in this thesis was to explore new techniques to efficiently visualize large terrain and urban datasets with commodity hardware.
The major problem that arises when dealing with such datasets is that the available computational resources are often insufficient by orders of magnitude to handle the large amount of detail typical of these models. In fact, not only the memory footprint exceeds by far the the system limits, but also the geometric complexity is too high to be managed even with powerful general purpose or graphics processors. Hence the need for representing the original data at different resolutions. The main idea behind multiresolution techniques is to maintain a description of the dataset at different level of details, usually arranged in a hierarchical structure, and select at rendering time the representation that can be rendered within user–imposed time or quality constraints. These systems must be able to handle the huge amount of memory required by the underlying multiresolution representation, adopting out–of–core strategies to move data among the different levels of the memory hierarchy and even access remote online data sources. In the context of a networked setup, it is clear that high transfer latencies impact on the general exploration experience, thus requiring to adopt compression strategies on the data representation.

With the C-BDAM approach we have extended the BDAM approach to support effective data compression. The presented compressed multiresolution representa-

tion is able to manage and render at real–time rates very large planar and spherical terrain surfaces. Similarly to BDAM, coarse grain refinement operations are associated to regions in a bintree hierarchy. In the C-BDAM approach, all patches share the same regular triangulation connectivity and incrementally encode their vertex attributes using a quantized representation of the difference with respect to values predicted from the coarser level. As illustrated by our experimental result, the structure provides a number of practical benefits: overall geometric continuity for planar and spherical domains, support for variable resolution input data, management of multiple vertex attributes, efficient compression and reasonably fast construction times, ability to support maximum-error metrics, real-time decompression and shaded rendering with configurable variable level-of-detail extraction, as well as runtime detail synthesis.
With our approach we have demonstrated that that it is possible to combine the generality and adaptivity of batched dynamic adaptive meshes with the compression rates of nested regular grid techniques.

Exploring urban environments is another important field for the research community because of the challenges exposed by this kind of datasets. In particular, the geometric structure of buildings and their associated large amount of image data require multiresolution strategies that can not rely only on standard polygonal simplification methods.
We introduced the BlockMap, a GPU-friendly, simplified discrete representation of a portion of urban scenery that provides a replacement for the original color and geometry in the classical level–of–detail sense. BlockMaps are characterized by efficiency in rendering, compactness in space, and bounded geometric error. When built into a tree hierarchy, BlockMaps offer multiresolution adaptability, as demonstrated in our framework for the interactive remote visualization of large urban environments. In some sense, our approach can be interpreted as a robust and practical way to create a new textured simplified representation of a urban environment, and, therefore, as a way of finding a unique parameterization of its surface. Most of the sophisticated high quality real-time rendering techniques need a good uniform parameterization of the surface to correctly work, something that is usually quite difficult to obtain for models such as those used in city rendering application, that often exhibit a wide variety of problems, such as high variance in element size and self intersections. The proposed framework is simple to implement and exposes its robustness when dealing with such ill–formed input models.

Recent multiresolution techniques for exploring large environments, and, more generally, novel rendering algorithms, represent the evidence of the tendency of moving the execution of complex tasks from the CPU to the GPU. This fact, coupled with the large bandwidth provided by nowadays network connections and the performance improvements of interpreted languages typically used in web documents, led to the standardization of the new WebGL API for harnessing the power of 3D graph-

ics accelerators directly within web pages. In this context, we presented SpiderGL, a novel 3D Graphics JavaScript library which uses the WebGL specifications for real–time rendering.
With practical examples, we have shown how the programming facilities exposed by the library help speeding up the creation of 3D web applications without forcing the developer to adopt predefined programming paradigms like scene graphs. By exploiting the JavaScript programming language to override the default behaviors of objects or express complex relations between the different entities involved in the rendering process, the developer is provided with high-level utilities for the creation of complex three-dimensional scenes, while retaining full access to the underlying WebGL layer. Common tools such as linear algebra, space related algorithms, asynchronous content loading and user interface utilities complete the main sectors the SpiderGL library addresses. The procedural style used throughout the library, coupled with mechanisms for object connections make the framework highly configurable and easily integrable into existing systems.
Furthermore we proposed a first potential application of SpiderGL that we called MeShade, to ease the deployment of 3D models on the web, which is likely to become a hot topic in a short while.

6.2 Future Work

Real–time 3D exploration of large environments still offers many opportunities for research, and the advancements in graphics hardware will certainly provide a fertile ground for the development of new techniques.

Although the current C-BDAM implementation already gives satisfactory results, which are state-of-the-art both in terms of compression rates and of rendering speed, there are still open issues that need to be investigated, besides incremental improvements to the various filtering and compression components, which have been chosen here mostly because of simplicity. A first important avenue of research is to determine whether it is possible to obtain in practice a max-error approximation directly in a one-stage wavelet approximation. A second important future work is to improve treatment of discontinuities, verifying whether current state-of-the-art adaptive techniques are fast enough to be employed in a real-time application.

A side effect of the BlockMap representation is that it gives a multiresolution parametrization of the visible surface. We have used it to store a simple ambient occlusion term, but it is quite trivial to extend it for storing more sophisticated information, like precomputed transfer radiance. Thanks to new capabilities of graphics hardware, it should be also possible to move the already few steps of the BlockMap construction process from the CPU to the GPU, having a complete hardware–accelerated preprocessing phase.

Beside the work of upgrading and extending SpiderGL, which is obviously a daily activity, we envisage a promising direction of work in the automatization of

the process of converting large databases of scanned objects to web repositories. The problems are mainly related to the typical large size of scanned objects and to the way to optimize them for a remote visualization. Although there are many available tools to reduce the number of polygons in a mesh, to parameterize it and to recover almost the full detail by bump mapping techniques (just to mention a viable, not unique, optimization pipeline), the whole process still requires a skilled user to be done.

6.3 List of Publications

The following is the list, in chronological order, of publications produced as results of our contribution:

- Enrico Gobbetti, Fabio Marton, Paolo Cignoni, Marco Di Benedetto, Fabio Ganovelli,
  C-BDAM - Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering,
  Computer Graphics Forum, Volume 25, Number 3 - 2006

- Paolo Cignoni, Marco Di Benedetto, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Roberto Scopigno,
  Ray-Casted BlockMaps for Large Urban Models Streaming and Visualization,
  Computer Graphics Forum, Volume 26, Number 3 - 2007

- Marco Di Benedetto, Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Roberto Scopigno,
  Interactive Remote Exploration of Massive Cityscapes,
  VAST 2009, The 10th International Symposium on Virtual Reality, Archaeology and Cultural Heritage - 2009

- Marco Di Benedetto, Federico Ponchio, Fabio Ganovelli, Roberto Scopigno,
  SpiderGL: A JavaScript 3D Graphics Library for Next-Generation WWW,
  Web3D 2010, 15th Conference on 3D Web Technology - 2010

- Marco Callieri, Raluca Mihaela Andrei, Marco Di Benedetto, Monica Zoppè, Roberto Scopigno,
  Visualization Methods for Molecular Studies on the Web Platform,
  Web3D 2010, 15th Conference on 3D Web Technology - 2010

- Marco Di Benedetto, Massimiliano Corsini, Roberto Scopigno,
  SpiderGL: A Graphics Library for 3D Web Applications,
  Proceedings of the 4th ISPRS International Workshop 3D-ARCH - 2011

Bibliography

[1] Microsoft ActiveX Controls. `http://msdn.microsoft.com/en-us/library/aa751968(VS.85).aspx`.

[2] D. G. Aliaga and A. A. Lastra. Smooth transitions in texture-based simplification. *Computers and Graphics*, 22(1):71–81, February 1998.

[3] Daniel G. Aliaga and Anselmo Lastra. Automatic image placement to provide a guaranteed frame rate. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings*, Annual Conference Series, pages 307–316, Los Angeles, 1999. ACM Siggraph, Addison Wesley Longman.

[4] D.G. Aliaga and A.A. Lastra. Architectural walkthroughs using portal textures. In *IEEE Visualization '97 Proc.*, pages 355–362. IEEE Comp. Soc. Press, 1997.

[5] Pierre Alliez and Craig Gotsman. Recent advances in compression of 3d meshes. In M.A. Sabin N.A. Dodgson, M.S. Floater, editor, *Advances in Multiresolution for Geometric Modelling*. Springer, 2005.

[6] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, pages 3–10, 1987.

[7] AMD. Render Monkey. `http://ati.amd.com/developer/rendermonkey/`, 2010.

[8] C. Andújar, P. Brunet, A. Chica, and I. Navazo. Visualization of large-scale urban models through multi-level relief impostors. *Computer Graphics Forum*, 29(8):2456–2468, 2010.

[9] C. Andújar, J. Diaz, and Brunet P. Relief impostor selection for large scale urban rendering. In *Proceedings of the IEEE VR 2008 Workshop on Virtual Cityscapes*, IEEE VR 2008, 2008.

[10] Carlos Andújar, Javier Boo, Pere Brunet, Marta Fairén González, Isabel Navazo, Pere-Pau Vázquez, and Alvar Vinacua. Omni-directional relief impostors. *Computer Graphics Forum*, 26(3):553–560, 2007.

[11] Apple Corps. Squirrelfish : Webkit bytecode interpreter. `http://trac.webkit.org/wiki/SquirrelFish`.

[12] James F. Blinn. Simulation of wrinkled surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, pages 286–292, New York, NY, USA, 1978. ACM.

[13] Louis Borgeat, Guy Godin, François Blais, Philippe Massicotte, and Christian Lahanier. Gold: interactive display of huge colored and textured models. *ACM Trans. Graph.*, 24:869–877, July 2005.

[14] Paul Brunt. GLGE: WebGL for the lazy. `http://www.glge.org/`, 2010.

[15] Stephen T. Bryson and Sandra Johan. Time management, simultaneity and time-critical computation in interactive unsteady visualization environments. In *IEEE Visualization '96*. IEEE, October 1996. ISBN 0-89791-864-9.

[16] Henrik Buchholz and Jurgen Dollner. View-dependent rendering of multiresolution texture-atlases. *Visualization Conference, IEEE*, 0:28, 2005.

[17] Callieri, Marco and Andrei, Raluca Mihaela and Di Benedetto, Marco and Zoppè, Monica and Scopigno, Roberto. Visualization Methods for Molecular Studies on the Web Platform. In *Web3D 2010. 15th Conference on 3D Web Technology*. Web3D Consortium, 2010.

[18] CGAL Project. CGAL, computational geometry algorithms library. `http://www.cgal.org`.

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

[20] Paolo Cignoni, Marco Di Benedetto, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, and Roberto Scopigno. Ray-casted blockmaps for large urban models streaming and visualization. *Computer Graphics Forum*, 26(3), Sept. 2007.

[21] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet-sized batched dynamic adaptive meshes (P-BDAM). In *IEEE Visualization*, pages 147–154, 2003.

[22] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Batched multi triangulation. In *Proceedings IEEE Visualization*, pages 207–214. IEEE Computer Society Press, October 2005.

[23] Paolo Cignoni, Claudio Montani, Claudio Rocchini, and Roberto Scopigno. A general method for preserving attribute values on simplified meshes. In *Visualization '98 Conference Proceedings*, pages 59–66. ACM, 1998.

[24] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 115–122, New York, NY, USA, 1998. ACM.

[25] Nico Cornelis, Kurt Cornelis, and Luc Van Gool. Fast compact city modeling for navigation pre-visualization. In *Proc. CVPR*, pages 1339–1344, 2006.

[26] Microsoft Corporation. Bing maps. `www.bing.com/maps/`.

[27] Douglas Crockford. JSON (JavaScript Object Notation) . `http://www.json.org/`.

[28] Lucia Darsa, Bruno Costa Silva, and Amitabh Varshney. Navigating static environments using image-space simplification and morphing. In Michael Cohen and David Zeltzer, editors, *1997 Symposium on Interactive 3D Graphics*, pages 25–34. ACM SIGGRAPH, April 1997. ISBN 0-89791-884-3.

[29] H. Davson. *Physiology of the Eye*. Pergamon Press, New York, NY, USA, fifth edition, 1994.

[30] L. De Floriani, P. Magillo, and E. Puppo. Building and traversing a surface at variable resolution. In *Proceedings IEEE Visualization 97*, pages 103–110, Phoenix, AZ (USA), October 1997.

[31] Xavier Decoret, François Sillion, Gernot Schaufler, and Julie Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3):61–73, September 1999. ISSN 1067-7055.

[32] Christopher DeCoro and Renato Pajarola. Xfastmesh: fast view-dependent meshing from external memory. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 363–370, Washington, DC, USA, 2002. IEEE Computer Society.

[33] Benjamin DeLillo. WebGLU: A utility library for working with WebGL . `http://webglu.sourceforge.org/`, 2009.

[34] Marco Di Benedetto, Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, and Roberto Scopigno. Interactive remote exploration of massive cityscapes. In K. De Battista, C. Perlingieri, D. Pitzalis, and S. Spina, editors, *The 10th International Symposium on Virtual Reality, Archaeology and Cultural Heritage VAST (2009)*, pages 9–16. Eurographics, 2009.

[35] Di Benedetto, Marco and Corsini, Massimiliano and Scopigno, Roberto. SpiderGL: A Graphics Library for 3D Web Applications. In *Proceedings of 3D-ARCH 2011*, 2011.

[36] Di Benedetto, Marco and Ponchio, Federico and Ganovelli, Fabio and Scopigno, Roberto. SpiderGL: A JavaScript 3D Graphics Library for Next-Generation WWW. In *Web3D 2010. 15th Conference on 3D Web Technology*. Web3D Consortium, 2010.

[37] Leonard Daly Don Brutzmann. *X3D: Extensible 3D Graphics for Web Authors*. Morgan Kaufmann, 2007.

[38] William Donnelly. Per-pixel displacement mapping with distance functions. In *GPU Gems 2*, pages 123–136. Addison-Wesley, 2005.

[39] M.A. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *Proceedings IEEE Visualization '97*, pages 81–88. IEEE, October 1997.

[40] Jonathan Dummer. Cone step mapping: An iterative ray-heightfield intersection algorithm. Available online at `http://www.lonesock.net/files/ConeStepMapping.pdf`, 2006.

[41] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Inform. Theory*, 21(2):194–203, March 1975.

[42] Bianca Falcidieno. AIM@SHAPE project presentation. In *Shape Modeling International*, pages 329–338. IEEE Computer Society, 2004.

[43] Christian Frueh, Siddharth Jain, and Avideh Zakhor. Data processing algorithms for generating textured 3d building facade meshes from laser scans and camera images. *International Journal of Computer Vision*, 61(2):159–184, feb 2005.

[44] E. Gobbetti and E. Bouvier. Time-critical multiresolution scene rendering. In *Proceedings IEEE Visualization*, pages 123–130, Conference held in San Francisco, CA, USA, October 1999. IEEE Computer Society Press.

[45] E. Gobbetti and E. Bouvier. Time-critical multiresolution rendering of large complex models. *Journal of Computer-Aided Design*, 2000.

[46] 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), sep 2006. To appear in Eurographics 2006 conference proceedings.

[47] Google Inc. Google Maps. `http://http://maps.google.com`.

[48] Google Labs. O3D. `http://code.google.com/apis/o3d/`, 2009.

[49] Paul S. Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Proc. Graphics Interface '94*, pages 43–50, Banff, Canada, May 1994. Canadian Inf. Proc. Soc. http://www.cs.cmu.edu/~ph.

[50] R. Held and N. Durlach. Telepresence, time delay, and adaptation. In Stephen R. Ellis, editor, *Pictorial Communication in Real and Virtual Environments*. Taylor and Francis, 1991.

[51] Lewis E. Hitchner and Michael W. McGreevy. Methods for user-based reduction of model complexity for virtual planetary exploration. In *SPIE*, volume 1913, pages 622–636, 1993.

[52] H. Hoppe. Smooth view-dependent level-of-detail control and its aplications to terrain rendering. In *IEEE Visualization '98 Conf.*, pages 35–42, 1998.

[53] Daniel Horn. Stream reduction operations for gpgpu applications. In *GPU Gems 2*, pages 573–589. Addison-Wesley, 2005.

[54] 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*, 11:355–368, July 2005.

[55] Lok Ming Hwa, Mark A. Duchaineau, and Kenneth I. Joy. Adaptive 4-8 texture hierarchies. In *Proceedings of IEEE Visualization 2004*, pages 219–226, Los Alamitos, CA, October 2004. IEEE, Computer Society Press.

[56] Google Inc. Google earth. http://www.google.com/earth/index.html.

[57] Google Inc. YouTube. www.youtube.com.

[58] A.K. Jain. *Fundamentals of Digital Image Processing.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1989.

[59] JOGL Java Binding for the OpenGL API. http://kenai.com/projects/jogl/pages/Home.

[60] Roger L. Claypoole Jr., Geoffrey M. Davis, Wim Sweldens, and Richard G. Baraniuk. Nonlinear wavelet transforms for image coding via lifting. *IEEE Transactions on Image Processing*, 12(12):1449–1459, 2003.

[61] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. In *In Proceedings of the ICAT 2001*, pages 205–208, 2001.

[62] Lindsay Kay. SceneJS. http://www.scenejs.com, 2009.

[63] Khronos Group. GLU OpenGL Utility Library. http://www.opengl.org/documentation/specs/glu/glu1_3.pdf.

[64] Khronos Group. OpenGL - The Industry Standard for High Performance Graphics. http://www.khronos.org/opengl/, 1992.

[65] Khronos Group. COLLADA - 3D Asset Exchange Schema. http://www.khronos.org/collada/, 2008.

[66] Khronos Group. Khronos: Open standards for media authoring and acceleration. http://http://www.khronos.org, 2009.

[67] Khronos Group. OpenGL ES - The Standard for Embedded Accelerated 3D Graphics. http://www.khronos.org/opengles/, 2009.

[68] Khronos Group. WebGL - OpenGL ES 2.0 for the Web. http://www.khronos.org/webgl/, 2009.

[69] Mark J. Kilgard. GLUT - The OpenGL Utility Toolkit . http://www.opengl.org/resources/libraries/glut/.

[70] S.E. Kosslyn. *Image and Brain: The resolution of the imagery debate.* MIT Press, Cambridge, MA, USA, 1994.

[71] Jelena Kovacevic and Wim Sweldens. Wavelet families of increasing order in arbitrary dimensions. *IEEE Transactions on Image Processing*, 9(3):480–496, 2000.

[72] Google Labs. V8 javascript engine. http://code.google.com/p/v8/.

[73] C. Landis. Determinants of the critical flicker-fusion threshold. *Physiological Review*, 34:259–286, 1954.

[74] Hayden Landis. Production ready global illumination. In *SIGGRAPH 2002 Course Notes*, pages 331–338. ACM Press, July 22-26 2002.

[75] Michael S. Langer and Heinrich H. Bulthoff. Perception of shape from shading on a cloudy day. Technical Report 73, Max-Planck-Institut fur biologische Kybernetik, October 1999.

[76] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings of the conference on Visualization '02*, VIS '02, pages 259–266, Washington, DC, USA, 2002. IEEE Computer Society.

[77] P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, N. Faust, and G.A. Turner. Real-time, continuous level of detail rendering of height fields. In *Comp. Graph. Proc., Annual Conf. Series (SIGGRAPH 96), ACM Press*, pages 109–118, New Orleans, LA, USA, Aug. 6-8 1996.

[78] P. Lindstrom and V. Pascucci. Visualization of large terrains made easy. In *Proc. IEEE Visualization 2001*, pages 363–370, 574. IEEE Press, October 2001.

[79] 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*, 8:239–254, July 2002.

[80] A. Liu, G.Tharp, L. French, S. Lai, and L. Stark. Some of what one needs to know about using headmounted displays to improve teleoperator performance. *IEEE Transactions on Robotics and Automation*, 9(5):638–648, 1993.

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

[82] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 95–102. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[83] H. Malavar and G. Sullivan. YCoCg-R: A color space with RGB reversibility and low dynamic range. In *JVT ISO/IEC MPEG ITU-T VCEG*, number JVT-I014r3. JVT, 2003.

[84] T. Malzbender. Enhancement of shape perception by surface reflectance transformation. In *Vision, Modeling, and Visualization*, page 183, 2004.

[85] M. McKenna and D. Zeltzer. Three dimensional visual display systems for virtual environments. *Presence*, 1(4):421–458, 1992.

[86] T. McReynolds and D. Blythe. Advanced graphics programming techniques using openGL. *SIGGRAPH'98 Course Notes*, 1998.

[87] Mozilla. Tracemonkey: Mozilla javascript just-in-time compiler. `https://wiki.mozilla.org/JavaScript:TraceMonkey`, 2010.

[88] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics (TOG)*, 25(3):614–623, 2006.

[89] NextEngine. `http://www.nextengine.com/`.

[90] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 359–368, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[91] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In H. Rushmeier D. Elbert, H. Hagen, editor, *Proceedings of Visualization ‘98*, pages 19–26, 1998.

[92] Renato Pajarola. Overview of quadtree based terrain triangulation and visualization. Technical Report UCI-ICS TR 02-01, Department of Information, Computer Science University of California, Irvine, Jan 2002.

[93] Y.I.H. Parish and P. Müller. Procedural modeling of cities. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, 2001.

[94] Past Perfect Productions. Rome reborn. `http://www.pastperfectproductions.com/`, 2007.

[95] Gabriel Peyré and Stéphane Mallat. Surface compression with geometric bandelets. *ACM Trans. Graph*, 24(3):601–608, 2005.

[96] Gemma Piella and Henk J. A. M. Heijmans. An adaptive update lifting scheme with perfect reconstruction. In *ICIP (3)*, pages 190–193, 2001.

[97] Fábio Policarpo and Manuel Oliveira. Relaxed cone stepping for relief mapping. In *GPU Gems 3*, pages 409–428. Addison Wesley, 2007.

[98] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, I3D ’05, pages 155–162, New York, NY, USA, 2005. ACM.

[99] M. Pollefeys, L. Van Gool, M. Vergauwen, F. Verbiest, K. Cornelis, J. Tops, and R. Koch. Visual Modeling with a Hand-Held Camera. *International Journal of Computer Vision*, 59(3):207–232, 2004.

[100] Alex A. Pomeranz. Roam using surface triangle clusters (rustic). Master’s thesis, University of California at Davis, 2000.

[101] V.S. Popescu, A. Lastra, D.G. Aliaga, and M.M. de Oliveira Neto. Efficient warping for architectural walkthroughs using layered depth images. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization ’98*, pages 211–216. IEEE, 1998.

[102] Procedural, Inc. Procedural modeling of cg architecture. `http://www.procedural.com/cityengine/production-pipeline/export-samples.html`, 2007.

[103] D. Raggett. Extending WWW to support platform independent virtual reality. *Technical Report*, 1995.

[104] Eric Risser, Musawir Shah, and Sumanta Pattanaik. Interval mapping. Technical report, University of Central Florida, 2006.

[105] RWTH. OpenMesh, visualization and computer graphics library. `http://www.openmesh.org`.

[106] G. Schaufler. Dynamically generated impostors. *GI Workshop on Modeling, Virtual Worlds, Distributed Graphics, Bonn, Germany*, 1995.

[107] G. Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In Julie Dorsey and Philipp Slusallek, editors, *Eurographics Rendering Workshop 1997*, pages 151–162, New York City, NY, June 1997. Eurographics, Springer Wien. ISBN 3-211-83001-4.

[108] G. Schaufler. Per-object image warping with layered impostors. In George Drettakis and Nelson Max, editors, *Rendering Techniques '98*, Eurographics, pages 145–156. Springer-Verlag Wien New York, 1998.

[109] G. Schaufler and W. Sturzlinger. A three-dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):C227–C235, C471–C472, September 1996.

[110] J.W. Shade, S.J. Gortler, L.-W. He, and R. Szeliski. Layered depth images. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 231–242. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.

[111] F. Sillion, G. Drettakis, and B. Bodelet. Efficient impostor manipulation-for real-time visualization of urban scenery. *Computer Graphics Forum*, 16(3):207–218, August 1997. Proceedings of Eurographics '97. ISSN 1067-7055.

[112] Stanford Computer Graphics Laboratory. `http://graphics.stanford.edu/data/3Dscanrep/`, 2000.

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

[114] S. Teller, M. Antone, Z. Bodnar, M. Bosse, S. Coorg, M. Jethwa, , and N. Master. Calibrated, registered images of an extended urban area. *International Journal of Computer Vision*, 53(1):93–107, June 2003.

[115] P. Torguet, O. Balet, E. Gobbetti, J.-P. Jessel, J. Duchon, and E. Bouvier. Cavalcade: A system for collaborative prototyping. In Gérard Subsol, editor,

Proc. International Scientific Workshop on Virtual Reality and Prototyping, pages 161–170, June 1999. Conference held in Laval, France, June 3–4.

[116] VcgLib, visualization and computer graphics library. `http://vcg.sourceforge.net`.

[117] Maarten Vergauwen and Luc Van Gool. Web-based 3d reconstruction servic. *Machine Vision Applications*, 17:411–426, 2006.

[118] Roland Wahl, Manuel Massing, Patrick Degener, Michael Guthe, and Reinhard Klein. Scalable compression and rendering of textured terrain data. In *Journal of WSCG*, volume 12, pages 521–528, Plzen, Czech Republic, February 2004. UNION Agency/Science Press.

[119] Yahoo! Flickr - Photo Sharing. `www.flickr.com`.

[120] Sehoon Yea and William A. Pearlman. A wavelet-based two-stage near-lossless coder. In *Proc. ICIP*, pages 2503–2506, 2004.

[121] Christine Youngblut, Rob E. Johnson, Sarah H. Nash, Ruth A. Wienclaw, and Craig A. Will. Review of virtual environment interface technology. IDA Paper P-3186, Institute for Defense Analysesc, March 1995.

[122] Gernot Ziegler, Art Tevs, Christian Theobalt, and Hans-Peter Seidel. Gpu point list generation through histogram pyramids. In *Vision, Modeling and Visualization Workshop*, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, 2006.