REAL-TIME VISUALIZATION OF 3D CITY MODELS

DISSERTATION
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
DOCTOR RERUM NATURALIUM
(DR. RER. NAT.)
IN DER WISSENSCHAFTSDISZIPLIN INFORMATIK

EINGEREICHT AN DER
MATHEMATISCH-NATURWISSENSCHAFTLICHEN FAKULTÄT
DER UNIVERSITÄT POTSDAM

VON
HENRIK BUCHHOLZ
GEBOREN AM 22.11.1976 IN MÜNSTER

POTSDAM, DEN 29. SEPTEMBER 2006

ABSTRACT

An increasing number of applications requires user interfaces that facilitate the handling of large geodata sets. Using virtual 3D city models, complex geospatial information can be communicated visually in an intuitive way. Therefore, real-time visualization of virtual 3D city models represents a key functionality for interactive exploration, presentation, analysis, and manipulation of geospatial data.

This thesis concentrates on the development and implementation of concepts and techniques for real-time city model visualization. It discusses rendering algorithms as well as complementary modeling concepts and interaction techniques.

Particularly, the work introduces a new real-time rendering technique to handle city models of high complexity concerning texture size and number of textures. Such models are difficult to handle by current technology, primarily due to two problems:

- *Limited texture memory*: The amount of simultaneously usable texture data is limited by the memory of the graphics hardware.
- *Limited number of textures*: Using several thousand different textures simultaneously causes significant performance problems due to texture switch operations during rendering.

The *multiresolution texture atlases* approach, introduced in this thesis, overcomes both problems. During rendering, it permanently maintains a small set of textures that are sufficient for the current view and the screen resolution available.

The efficiency of multiresolution texture atlases is evaluated in performance tests. To summarize, the results demonstrate that the following goals have been achieved:

- Real-time rendering becomes possible for 3D scenes whose amount of texture data exceeds the main memory capacity.
- Overhead due to texture switches is kept permanently low, so that the number of different textures has no significant effect on the rendering frame rate.

Furthermore, this thesis introduces two new approaches for real-time city model visualization that use textures as core visualization elements:

- An approach for *visualization of thematic information*.
- An approach for *illustrative visualization* of 3D city models.

Both techniques demonstrate that multiresolution texture atlases provide a core functionality for the development of new applications and systems in the domain of city model visualization.

ZUSAMMENFASSUNG

Eine zunehmende Anzahl von Anwendungen benötigt Benutzungsschnittstellen, um den Umgang mit großen Geodatenmengen zu ermöglichen. Virtuelle 3D-Stadtmodelle bieten eine Möglichkeit, komplexe raumbezogene Informationen auf intuitive Art und Weise visuell erfassbar zu machen. Echtzeit-Visualisierung virtueller Stadtmodelle bildet daher eine Grundlage für die interaktive Exploration, Präsentation, Analyse und Bearbeitung raumbezogener Daten.

Diese Arbeit befasst sich mit der Entwicklung und Implementierung von Konzepten und Techniken für die Echtzeit-Visualisierung virtueller 3D-Stadtmodelle. Diese umfassen sowohl Rendering-Algorithmen als auch dazu komplementäre Modellierungskonzepte und Interaktionstechniken.

Insbesondere wird in dieser Arbeit eine neue Echtzeit-Rendering-Technik für Stadtmodelle hoher Komplexität hinsichtlich Texturgröße und Texturanzahl vorgestellt. Solche Modelle sind durch die derzeit zur Verfügung stehende Technologie schwierig zu bewältigen, vor allem aus zwei Gründen:

- *Begrenzter Textur-Speicher*: Die Menge an gleichzeitig nutzbaren Texturdaten ist beschränkt durch den Speicher der Grafik-Hardware.
- *Begrenzte Textur-Anzahl*: Die gleichzeitige Verwendung mehrerer tausend Texturen verursacht erhebliche Performance-Probleme aufgrund von Textur-Umschaltungs-Operationen während des Renderings.

Das in dieser Arbeit vorgestellte Verfahren, das Rendering mit *Multiresolutions-Texturatlanten* löst beide Probleme. Während der Darstellung wird dazu permanent eine kleine Textur-Menge verwaltet, die für die aktuelle Sichtperspektive und die zur Verfügung stehende Bildschirmauflösung hinreichend ist.

Die Effizienz des Verfahrens wird in Performance-Tests untersucht. Die Ergebnisse zeigen, dass die folgenden Ziele erreicht werden:

- Echtzeit-Darstellung wird für Modelle möglich, deren Texturdaten-Menge die Kapazität des Hauptspeichers übersteigt.
- Der Overhead durch Textur-Umschaltungs-Operationen wird permanent niedrig gehalten, so dass die Anzahl der unterschiedlichen Texturen keinen wesentlichen Einfluss auf die Bildrate der Darstellung hat.

Die Arbeit stellt außerdem zwei neue Ansätze zur 3D-Stadtmodell-Visualisierung vor, in denen Texturen als zentrale Visualisierungselemente eingesetzt werden:

- Ein Verfahren zur *Visualisierung thematischer Informationen*.
- Ein Verfahren zur *illustrativen Visualisierung* von 3D-Stadtmodellen.

Beide Ansätze zeigen, dass Rendering mit Multiresolutions-Texturatlanten eine Grundlage für die Entwicklung neuer Anwendungen und Systeme im Bereich der 3D-Stadtmodell-Visualisierung bietet.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Dr. Jürgen Döllner, for encouraging me to start my doctorate and for affording me the opportunity. His research impulses and innumerous pieces of valuable advice formed a steady stimulus for my work. I would also like to thank Prof. Dr. Hartmut Asche and Prof. Dr. Klaus Hinrichs for reviewing this thesis.

I am grateful to all public authorities and companies that provided their geodata for testing and illustrating purposes, particularly to the RSS GmbH for the city models of Munich and Berlin and to the IMK Automotive GmbH for the city model of Chemnitz. I would also like to express my gratitude to the Berlin Senate Department of Urban Development, particularly Takis Sgouros and Peter Jürgens, for the opportunity to obtain insights into visualization problems in urban planning and redevelopment in common visualization projects. In addition, I would like to thank the 3DGeo GmbH for their collaboration and for increasing the practical relevance of my work. Thanks are also due to the German environmental foundation Deutsche Bundesstiftung Umwelt for supporting the Lenné3D research project on interactive landscape visualization, and the Lenné3D GmbH for providing their 3D tree models.

Furthermore, I would like to express my gratitude to all collaborators in the work on automatic construction of urban terrain models, particularly to Lutz Ross and Birgit Kleinschmit for their collaboration in the concept development and for providing motivation and knowledge from the domain of landscape planning, and to the students Manuel Wellmann, Anselm Kegel, and Oleg Dedkow for their contributions to the editor implementation. In addition, I would like to thank Alexander Klimetschek, Tassilo Glander, Florian Brodersen, and Sascha Jütterschenke for the implementation of the CLOQ building editor.

My special thanks are due to Marc Nienhaus for the joint work on projects for non-photorealistic city model visualization and for developing the fragment shader for illustrative tree rendering. I am also grateful to Johannes Bohnet for the collaboration in the development of *smart navigation-strategies*, to my former colleague Florian Kirsch for providing his fading technique for the Spittelmarkt visualization project, and to Haik Lorenz for supporting the implementation of lighting-texture calculation in the CityGML tool. I would also like to express my gratitude to Konstantin Baumann for numerous inspiring discussions. For the helpfulness and the permanently positive working athmosphere in our workgroup, I would also like to thank Oliver Kersting, Stefan Maass, Benjamin Hagedorn, Sabine Biewendt, and Katrin Heinrich.

I would like to express my sincere thanks to my parents Eckhard and Ulrike Buchholz for always supporting me in my personal decisions, to my brother Rüdiger and my sister Angela for motivating comments, and to my brother Harald for frequent reports on technological advances in current computer games. Most of all, I would like to thank my wife Karin for her support in difficult times of my research period, for her helpful technical and linguistic remarks to my publications, and for her never-tiring patience in those ‘Sorry, dear, it will take longer again’ situations.

CONTENTS

Chapter 1

INTRODUCTION

This chapter describes thematic context, scientific contribution, and structure of this thesis. It outlines main characteristics of virtual 3D city models and visualization. In addition, it explains the key problem addressed in this work.

1.1 Virtual 3D City Models

Virtual 3D city models are applied in an increasing number of applications including urban planning and redevelopment, landscape planning, radio-network planning, real-estate management, disaster management, navigation systems, environmental simulation, location-based services, logistics, and tourism. Virtual 3D city models represent an important visualization paradigm for urban geospatial information and georeferenced thematic information. For instance, they enable the visual presentation of past [e.g., Suzuki and Chikatsu 2000] and existing [e.g., Schilling and Zipf 2003] cities and provide decision support in urban redevelopment projects [e.g., Chen and Knapp 2006]. Furthermore, the potential of virtual 3D city models is not restricted to visualization. They provide a data basis for spatial querying of thematic data [e.g., Dogan et al. 2004] as well as for computational models such as the simulation of dispersive contaminants for urban security [Qiu et al. 2004] or noise propagation models [Tsingos et al. 2004].

In this work, the term *virtual 3D city model* refers to a digitally storable representation of relevant entities of a city and their relationships such as geometric and topologic structures and semantic hierarchies. These entities comprise:

- terrain surface,
- buildings,
- vegetation objects such as trees or hedges,
- transportation objects such as railways or streets,
- land use classifications such as settlement areas or industrial areas,
- city furniture objects such as telephone boxes or traffic signs,
- water objects such as rivers or lakes,
- groups formed by other entities such as street districts or building complexes.

The term ‘city model’ will be used without ‘virtual’ and ‘3D’ in the following, because 2D models as well as non-virtual city models such as physical wood models used in urban planning are not addressed in this work. The list of entities is based on the top-level classes of the thematic model in CityGML, an upcoming international standard for the specification of 3D urban objects [Gröger et al. 2006]. CityGML is being developed by the Special Interest Group 3D (SIG 3D) of the initiative Geodata Infrastructure North-Rhine Westphalia (GDI NRW) and is currently being discussed by the Open GIS Consortium (OGC). All references to CityGML in this work are based on the currently available version 0.3.0 of the CityGML specification.

The classes of the thematic model in CityGML have been designed according to the requirements identified in various application areas. In future, they will prospectively be extended because new entities such as tunnels, excavations, walls, and embankments are already scheduled for integration.

The term ‘city model’ is frequently used to refer to generic 3D models for computer graphics applications. Such models allow, for instance, for flying or walking virtually through a city [e.g., De Leon and Berry 1998]. In general, however, a generic 3D model is frequently not sufficient to provide all relevant information of a city. Particularly, due to missing semantic information, it is unknown whether a certain 3D object represents a single building, a part of a building, a fraction of the terrain or something else.

Commonly, the representation of city models by generic 3D models leads to several limitations. For instance, for generic 3D models there are no standardized ways to integrate building-related information such as addresses or to update a single building model. Therefore, the term ‘city model’ cannot be generally restricted to 3D graphics representations. Accordingly, CityGML includes a *geometric model* to describe geometry, appearance, and topological relationships as well as a related *thematic model* to describe thematic information and semantics of urban objects.

The requirements on the information provided by a city model depend on the application domain in which the model is used.

> *‘Every second producer has requests to provide other objects or information than he is presently producing and three out of four users would like to have other city data than already available.’* [Förstner 1999]

For instance, wave propagation models for radio-network planning can be used with pure geometric building representations. In urban planning, photorealistic simulations of envisioned projects within an existing environment are required, so that visual details become essential. In urban redevelopment, thematic building information such as vacancy, number of floors, or state of repair are important aspects to be supported, whereas applications for tourism require other kinds of thematic information such as descriptions of historic landmarks or restaurants. Thus, the question, which information a city model must generally provide for current and future applications is not yet fully answered.

1.2 Real-Time Visualization

Real-time visualization combines both the disciplines of visualization and real-time rendering. Visualization of city models represents a special case of data visualization. According to Senay and Ignatius [1994],

> *‘The primary objective in data visualization is to gain insight into an information space by mapping data onto graphical primitives.’*

Foley [1994] defines visualization as follows:

> *'A useful definition of visualization might be the binding (or mapping) of data to a representation that can be perceived. The types of binding could be visual, auditory, tactile, etc. or a combination of these.'*

In Schumann and Müller [2000], visualization is, alternatively, described by (translated from German):

> *The task of scientific visualization is to create appropriate visual representations of a given data set to enable effective evaluation.*

Schumann and Müller also point out that *rendering*, i.e., the creation of synthetic images, forms part of the visualization process. In the context of computer graphics, the extended term *real-time rendering* means to 'create synthetic images fast enough so that the viewer can interact with a virtual environment' [Möller and Haines 1999]. Frame rates of 15 frames per second or more are desirable for real-time rendering, but a strictly defined frame rate limit separating real-time and non-realtime does not exist.

City models represent a special case of *geodata*. According to Bollmann and Koch [2002] (translated from German), geodata can be described as

> *data that contain spatial references identifying positions on the earth's surface. They describe real-world objects by means of geometric and thematic attributes based on an underlying geoobject model.* […]

Here, the term *geoobject model* refers to a conceptual model that defines a subdivision of the real world into objects according to semantic and logical criteria. For a more detailed characterization of geodata and geoobject models, see Bollmann and Koch [2002].

Visualization of city models belongs to the field of *geovisualization*, which can be characterized as follows:

> *'Rooted in traditional cartography, geovisualization refers particularly to the visualization of geospatial information and integrates knowledge and expertise from various related fields such as scientific visualization, information visualization, and virtual environments. Geovisualization on the other hand contributes significantly to other visualization fields. For instance, the map metaphor has been widely used to visualize non-geographic information in the domain knowledge visualization.'* [Jiang and Li 2005]

For city models containing explicit specifications of 3D geometry and appearance information, the terms 'visualization' and 'rendering' are frequently used synonymously. This, however, is incorrect in the general case for multiple reasons. First, the actual appearance of a city is not necessarily the primary aspect of interest. For instance, in the field of urban redevelopment, the geometry of a city model serves basically as a medium to visualize spatially referenced thematic building information which is not necessarily visible [Buchholz and Döllner 2005(i)]. Second, particularly for complex city models, direct rendering of a model might not necessarily provide actual insight. Therefore, it can be useful to simplify the resulting image perceptually by using non-photorealistic techniques [Döllner et al. 2005(i)] or to apply cartographic building generalization [Thiemann 2002, Forberg 2004]. Third, the effectiveness of interactive visualization is not determined by the rendering process only. It also depends strongly on the effectiveness of the navigation techniques available, which control the way in which a user steers the virtual camera within the virtual space. Therefore, city model visualization includes but is not restricted to 3D rendering.

Real-time visualization of city models can be implemented either on standard desktop PCs or on specialized high-end hardware systems. In both cases, hardware and software together form a special case of a *geovirtual environment* [MacEachren et al. 1999], i.e., a virtual environment

for geovisualization. Specialized systems provide high rendering performance as well as sophisticated visual devices such as head-mounted displays [Fisher et al. 1986] or cave projections [Cruz-Neira 1993]. As pointed out by Fuhrmann and MacEachren [2001], however, geovirtual environments running on standard PC hardware, called *desktop geovirtual environments*, '[…] are likely to have the most impact on science and society in the short run, due to the sheer numbers of people who can access them.' Consequently, this work concentrates on real-time visualization using standard PC hardware.

1.3 Graphics Representations of City Models

In the context of real-time computer graphics, the most common representation of a 3D object is a triangle-based boundary representation. It consists of one or more *triangle meshes*, which are composed of multiple triangles in 3D. Groups of adjacent triangles form contiguous surfaces that provide discrete approximations of the object's boundary surfaces.

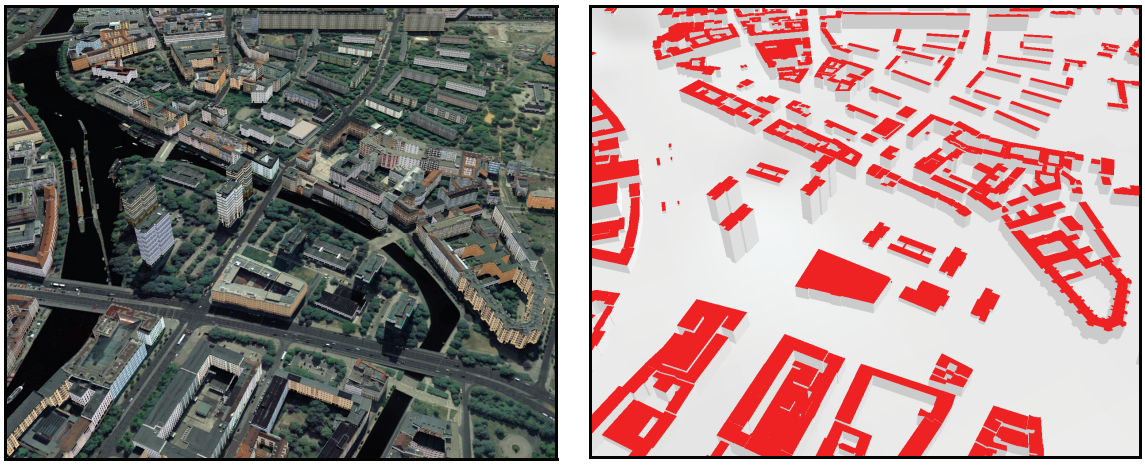
A *textured triangle mesh* is a triangle mesh with an assigned 2D image, called *texture* [Foley et al. 1994], which is mapped onto its triangles. In general, the term 'texture' does not necessarily refer to a 2D image, but 2D textures are most common, and other cases are not addressed in this work. The mapping of the images is controlled by *texture coordinates* that specify for each vertex of each triangle a corresponding location within a 2D image. In this way, the three 2D texture coordinate vectors of a triangle define a mapping from 2D texture space to the triangle surface in 3D. Multiple 3D objects are composed to *3D scenes*. Triangle-based boundary representations are supported by current 3D graphics hardware and, therefore, can be rendered quite efficiently.

The above description is simplified to the basic and most common form. In general, 3D scenes are expressed by *scene graphs*, i.e., directed acyclic graphs whose content is not necessarily restricted to textures and triangle meshes. A detailed discussion of scene graphs can be found in Kirsch [2005]. In addition, further vector-valued graphics attributes may be specified for each vertex such as colors and surface-normal vectors, and multiple textures may be assigned to the same triangle mesh [Segal and Akeley 2004].

In the context of city models, the geometry and appearance of 3D objects such as buildings or city furniture can be specified in several ways. For instance, a simple block building might be described by a horizontal footprint polygon and a height value, possibly extended by a 2D image that is mapped around the facade. In contrast, a detailed building in CityGML might be specified as a complex hierarchy of sub-objects, each of which being described by a set of polygons that form the object's surface. The different storage forms of objects in city models are useful to achieve a more compact data representation or to preserve structural information such as the subdivision of a building into roof, walls, windows, etc. Such representations, however, cannot be evaluated directly by graphics hardware. Therefore, the common method for real-time rendering of 3D objects in city models is to convert them into textured triangle meshes.

1.4 Role of Textures in City Models

'Texture mapping is a powerful and flexible low level graphics drawing primitive' [Haeberli 1993]. Textures provide means to store and associate discrete visual and thematic information for buildings, ground areas, plants, and other components of a city model. In particular, they allow for specifying the visual appearance of surfaces such as facades, roofs, and terrain. Therefore, they represent an established way to enrich city models with visual details. Figure 1 compares a textured and a non-textured city model of Berlin. Texture mapping is supported by CityGML as a fundamental part of the geometric model.



a) b)

Figure 1: Importance of textures for photorealistic rendering of city models: a) Snapshot of a city model of Berlin; b) The same view without textures.

Up to now, textures have primarily been considered to be essential for *photorealistic rendering*, which aims at creating images that resemble reality as close as possible. The potential of textures, however, is not restricted to the representation of photorealistic details. Alternative applications of textures are explained and demonstrated in this work.

1.5 Contribution

This thesis concentrates on the development and implementation of concepts and techniques for real-time city model visualization. It addresses rendering algorithms as well as complementary modeling concepts and interaction techniques.

In particular, this work makes the following main contributions:

1. It introduces *multiresolution texture atlases*, a new rendering technique to handle city models of high complexity concerning texture size and number of textures.
2. It proposes *new city model visualization approaches* that use textures as core visualization elements.

1. Multiresolution Texture Atlases

Real-time rendering of city models containing several thousand textured buildings represents a challenging task, primarily due to two demands that are typically made on the model textures:

- *High texture resolution*: Textures of city models must be provided at high resolutions, particularly if a city model is explored from a pedestrian point of view. Without sufficient texture resolution, textures appear blurry when seen from close distances.
- *Large number of textures*: Especially for photorealistic rendering, large numbers of textures are required to represent individual characteristics of buildings and other entities correctly. For instance, individual textures for building facades are necessary to give a reliable visual impression of a city. For imaginary city models such as in computer games, individual textures can be avoided by using a limited set of realistic-looking generic facade textures that are shared by several buildings instead. For real cities, however, generic facade textures make it difficult to recognize a known urban area in a 3D visualization. In addition, the visualization becomes misleading because real and artificial visual information is mixed.

The amount of textures needed to fit both requirements usually exceeds the capacity of even modern graphics hardware. For example, a city model of 10,000 buildings that defines a single 24-Bit RGB facade texture at a resolution of 512 × 512 pixels for each building requires a total texture amount of about 7.3 gigabyte for the facade textures.

The amount of textures that can be handled in real-time by current standard graphics hardware is limited due to two problems:

- *Limited graphics memory*: Each texture being accessed during rendering must be resident in the graphics memory. Since the transfer of texture data between main memory and graphics memory is time-consuming, all simultaneously visible textures should fit simultaneously into the graphics memory for efficient rendering. Currently, the size of the graphics memory is typically in the range of 64 MB to 512 MB.
- *Performance costs of texture switches:* To apply a texture to a triangle mesh, the texture has to be activated before rendering the triangle mesh. The activation of a new texture, possibly combined with the deactivation of a previous one, is called *texture switch*. Texture switches are a well-known performance bottleneck in real-time computer graphics [Wloka 2004]. Even if all textures of a city model fit into the graphics memory, the performance costs for several thousand texture switches lead to rather low frame rates.

The multiresolution texture atlases approach overcomes both problems. For this, it permanently maintains a small set of textures that are sufficient for the current view and the available screen resolution. In this way, the performance costs for texture handling are permanently low and independent of the total amount of texture data of the city model even if the full model is visible at a glance.

Multiresolution texture atlases have been implemented based on the VRS rendering library [Döllner and Hinrichs 2002]. Its rendering performance has been evaluated for two test city models. The implementation of the approach has been integrated into the LandXplorer geovisualization framework [Döllner et al. 2003].

2. New Approaches for Real-Time City Model Visualization

This thesis introduces two new approaches for real-time city model visualization that use textures as core visualization elements:

1. An approach for *thematic visualization*: Measurement results or results of computational models such as field strength within radio networks do not have a natural appearance or geometry. The proposed approach uses textures draped onto surfaces of affected city model entities to communicate such information.
2. An approach for *non-photorealistic visualization*: In contrast to photorealistic visualization, non-photorealistic visualization uses ‘concepts and techniques that deliberately abstract from reality using expressive, stylized, or illustrative rendering’ with the primary goals of ‘visual clarity, attractiveness, comprehensibility, and perceptibility’ [Nienhaus 2005]. The introduced approach uses non-photorealistic textures to synthesize illustrative city model depictions.

1.6 Structure of the Dissertation

The remainder of this thesis is structured as follows:

Chapter 2 provides technical background on real-time rendering of 3D scenes with large textures, a large number of textures, or both. First, texture atlases and multiresolution textures

are explained, which form a technical basis of multiresolution texture atlases. Second, the chapter gives a brief overview of further related approaches.

Chapter 3 discusses city models and their visualization. First, the components and creation of city models are addressed. Next, the role of navigation for real-time visualization is discussed, and a brief overview of existing techniques is given. Finally, several characteristic applications are described.

Chapter 4 gives a detailed description of the multiresolution texture atlases approach. It describes the core data structure, the related rendering algorithm, the preprocessing step as well as the memory management strategy to handle texture data that exceed the main memory capacity. The efficiency of the approach is demonstrated in performance tests. Finally, optimization aspects for efficient implementation are explained.

Chapter 5 demonstrates the application of textures in city models for visualization of thematic data as well as non-photorealistic visualization.

Chapter 6 discusses advantages and limitations of multiresolution texture atlases, summarizes the results of this work, and proposes potential future research.

Chapter 2

RELATED WORK

This chapter provides an overview of texture handling techniques in computer graphics and geovisualization. In particular, it explains texture atlases and multiresolution terrain textures. These techniques form a basis of the approach introduced in this work.

2.1 Texture Atlases

A *texture atlas* [Wloka 2004] is a texture image that combines multiple source textures into a single texture to simplify and accelerate their handling for real-time rendering (Figure 2). Texture atlases are an established method to reduce the number of texture switches per rendered frame. Given a set of geometric shapes $s_1,\ldots,s_n$ with individual textures $t_1,\ldots,t_n$, it, normally, requires n texture switches to render all shapes with their textures. If the texture images $t_1,\ldots,t_n$ are combined in a single large texture image T_{Atlas}, all shapes can be rendered with a single texture switch. This is, however, only possible if $t_1,\ldots,t_n$ fit into a single texture image. The maximum texture size is usually limited by the graphics hardware, typically to 4096×4096 pixels or less.

Texture atlases are not suitable to reduce the amount of texture data. On the contrary, the amount of texture data is increased by the use of texture atlases if they contain unused areas. This can also be seen in Figure 2. The percentage of the texture atlas image that is actually used, i.e., covered by sub-images, is called *atlas coverage* [Carr and Hart 2002]. An atlas coverage of 100% can only be achieved if it is possible to assemble all sub-images to a single rectangular area.

To create a texture atlas from a set of given source images of different sizes, a *packing algorithm* is needed, i.e., a strategy to arrange all source images as tightly as possible to fit into a single rectangular texture atlas image. For the implementation of the multiresolution texture atlases approach, a simple but effective heuristic algorithm described in Igarashi and Cosgrove [2001] has been used. It is illustrated in Figure 3: Given a maximum texture atlas size and a set of source images to be arranged (Figure 3a), the first step is to rotate all source images whose

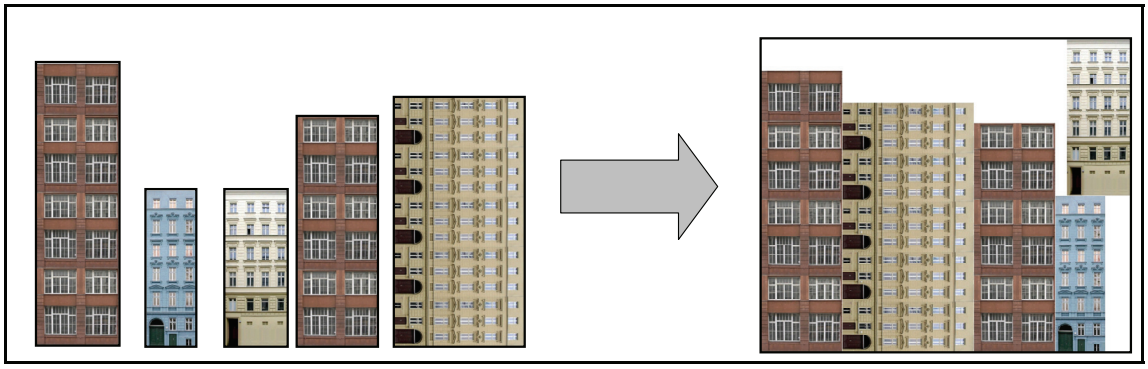


Figure 2: Example of a texture atlas created from multiple building-facade textures.

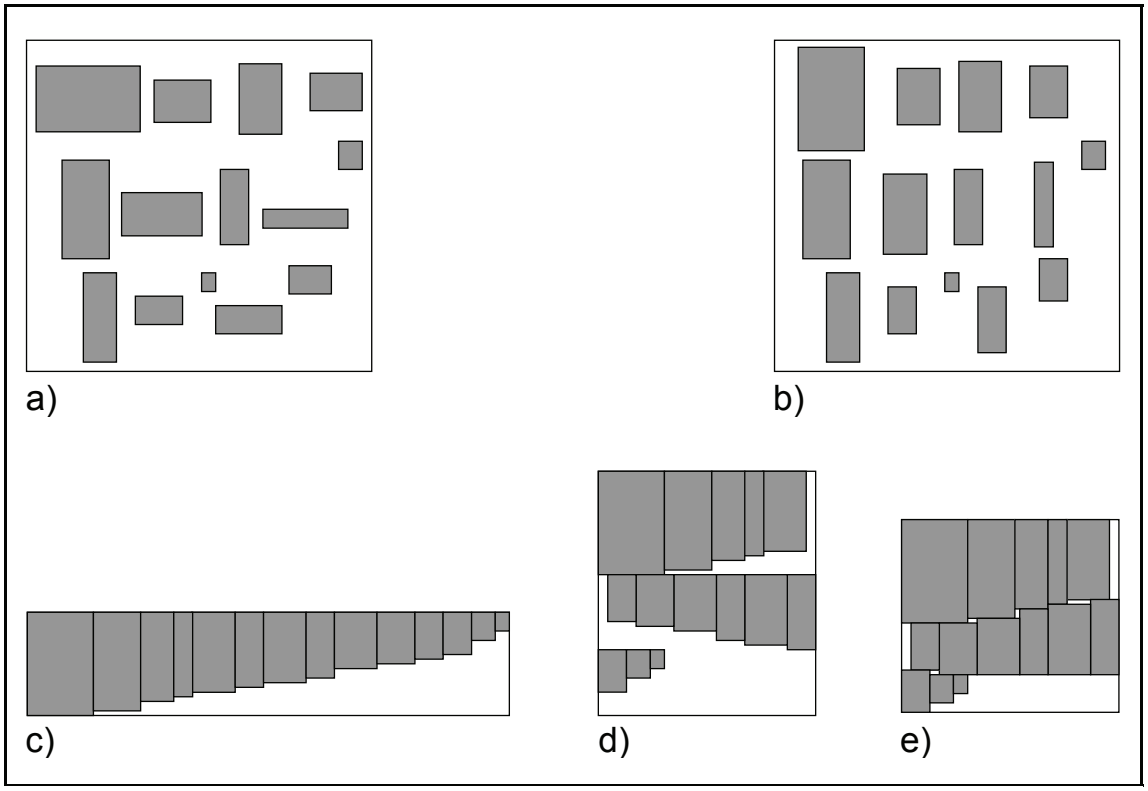


Figure 3: Illustration of the texture atlas packing algorithm in Igarashi and Cosgrove [2001].

height is smaller than their width through 90 degrees. Hence, all images are taller than wide (Figure 3b). Next, the images are sorted by decreasing height (Figure 3c) and lined up in rows (Figure 3d). Finally, each image is pushed upwards as far as possible without overlapping the row above (Figure 3e). This approach does only work if all textures fit into a single texture atlas, i.e., it does not include a strategy to share the source images among multiple target texture atlases. For the implementation of multiresolution texture atlases, however, such a strategy is not needed (see Section 4.5). An overview of existing packing algorithms that could be used alternatively is given in Preussner [2004].

After the creation of a texture atlas, it can replace all contained textures. For this, each texture coordinate vector referring to an original texture T must be remapped so that it refers to the corresponding point within the texture atlas region to which T has been copied. This is possible

as long as the texture coordinates of each triangle fit into the range $[0, 1]^2$. This area in texture space corresponds to the full extend of the underlying texture image. After remapping, texture coordinates outside $[0, 1]^2$ would, unintentionally, point to other regions in the texture atlas.

A given 3D scene may contain texture coordinates outside $[0, 1]^2$. Graphics hardware provides different *address modes* to control the way in which such texture coordinates are handled [Segal and Akeley 2004], i.e., to define which image data correspond to the texture space outside $[0, 1]^2$. A commonly used address mode is the *repeat mode*. It allows for repeating a texture image multiple times across a single triangle. For this, the hardware discards the integer part of the texture space coordinates for each texture access to a texture space location outside $[0, 1]^2$.

There are three methods to cope with repeated textures in texture atlases, but all methods involve specific drawbacks:

1. *Texture replication*: Repeated textures can be included into a texture atlas by replicating the textures multiple times in a texture atlas image. This means, however, if a texture is used, for instance, with texture coordinates covering the range $[0, 10]^2$, this texture has to be replicated 100 times in the texture atlas. Therefore, this method introduces considerable waste of texture memory and is impossible if the replicated texture exceeds the texture atlas size.
2. *Triangle subdivision*: The second method is to subdivide triangles recursively into smaller ones until for each triangle *t* the corresponding triangle *t*' formed by its texture coordinates is sufficiently small. If width and height of the bounding box of *t*' are 1.0 or less, at most two repetitions along both axes are necessary for each texture in the texture atlas. For initially large repetition counts, however, this methods leads to a critical increase of geometric complexity. For instance, for the city model of Chemnitz that has been used for performance tests (see Section 4.7.2) the necessary triangle splitting increased the number of triangles in the scene by a factor of 10.
3. *Fragment shader*: The third method is to implement a fragment shader that performs the address modes within texture atlases as proposed in Wloka [2004]. A *fragment shader* is a code section that is executed by the graphics hardware for each rasterized fragment [Rost 2004]. Using a fragment shader allows for texture repetition in texture atlases without involving additional texture data or increased geometric complexity. It requires, however, state-of-the-art shader support and is therefore restricted to recent graphics hardware. In addition, the fragment shader involves additional computational effort for the graphics hardware during rendering.

In contrast to conventional texture atlases, multiresolution texture atlases can be used in combination with texture replication without performance problems as is explained and demonstrated in Chapter 4.

2.2 Multiresolution Textures for Digital Terrain Models

In this work, the term *multiresolution texture* refers to a data structure that provides a single texture at different resolutions. Multiresolution textures form a core part of rendering algorithms for digital terrain models [Baumann 2000, Hwa 2004]. A general discussion of the term *digital terrain model* is given later in Section 3.1.1. In this section, it denotes a triangle mesh, whose projection to the horizontal plane has the following properties:

- Its vertices form a regular grid.
- Its triangles form a partition of a rectangle.

Here, the *horizontal plane* refers to a plane spanned by two selected main coordinate axes, usually *x* and *y*. A *terrain texture* is a texture image that is orthogonally projected to the terrain

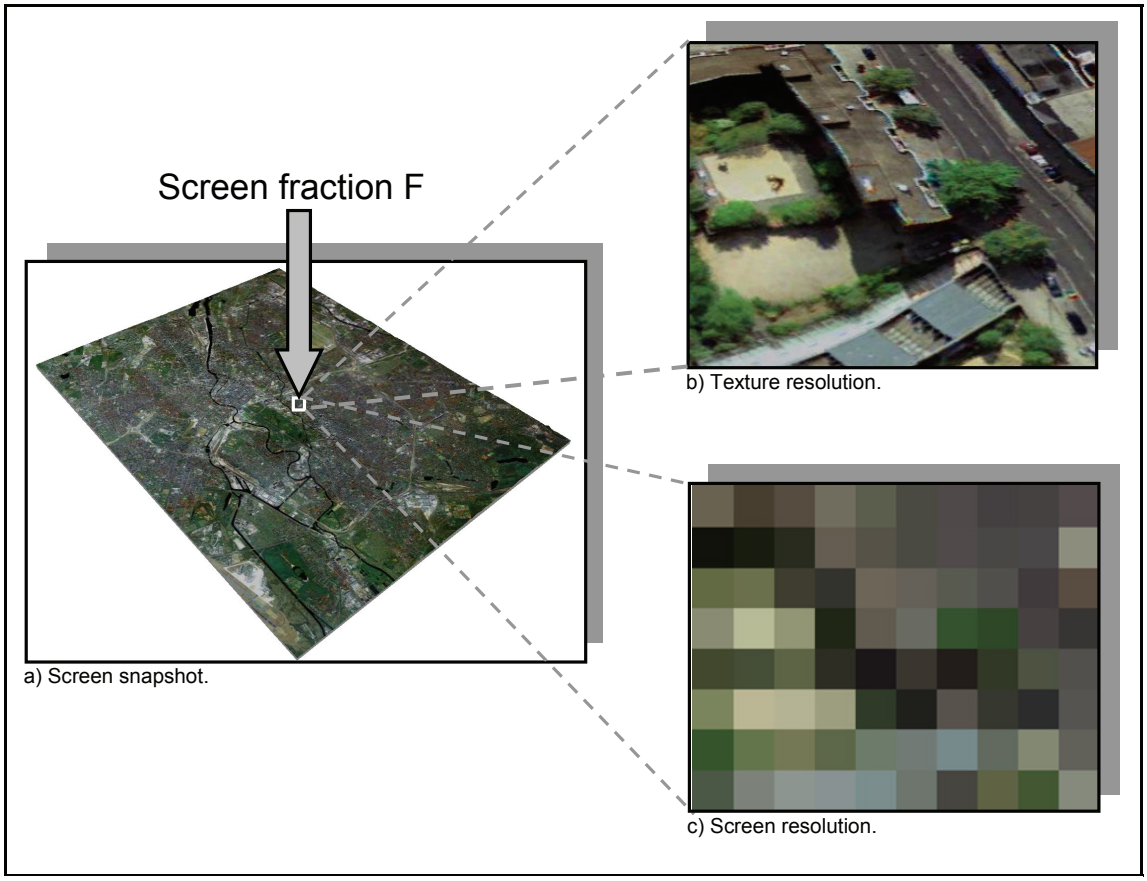


Figure 4: The required texture resolution is frequently considerably smaller than the maximum available texture resolution: a) Screen snapshot of a textured terrain with a highlighted screen fraction *F*. b) Texture region that is mapped to *F*, shown at full resolution. c) Enlarged view of *F* at the actual screen resolution.

model surface. Using multiresolution textures, it is possible to display terrain textures in real-time that are too large to be handled directly by the graphics hardware. The technique explained in this section has first been proposed in Lindstrom et al. [1995] and is also being used in a similar way by various later terrain rendering approaches. It is specifically used for terrain models and cannot be applied to arbitrary textured triangle meshes (see Section 0). The basic principle, however, has been partially adopted and extended for multiresolution texture atlases and, therefore, is explained in this section.

2.2.1 Purpose of Multiresolution Textures

Multiresolution textures make it possible to handle large terrain textures in real-time by exploiting the following observation, which is illustrated in Figure 4: In each frame, the amount of simultaneously visible texture data can be arbitrarily high at full resolution. In the worst case, a terrain texture consisting of multiple gigabytes might be completely visible (Figure 4a). However, texture regions that are viewed from large distances cover only very small fractions of the screen (Figure 4b), i.e., those texture regions are mapped to a small number of screen pixels (Figure 4c). Hence, it is not necessary to use high texture resolutions for such regions.

Based on this observation, the solution to handle large terrain textures is to use high texture resolutions only for terrain regions close to the camera and to decrease the texture resolution with increasing camera distance (Figure 5). Since at a given point in time only a small fraction

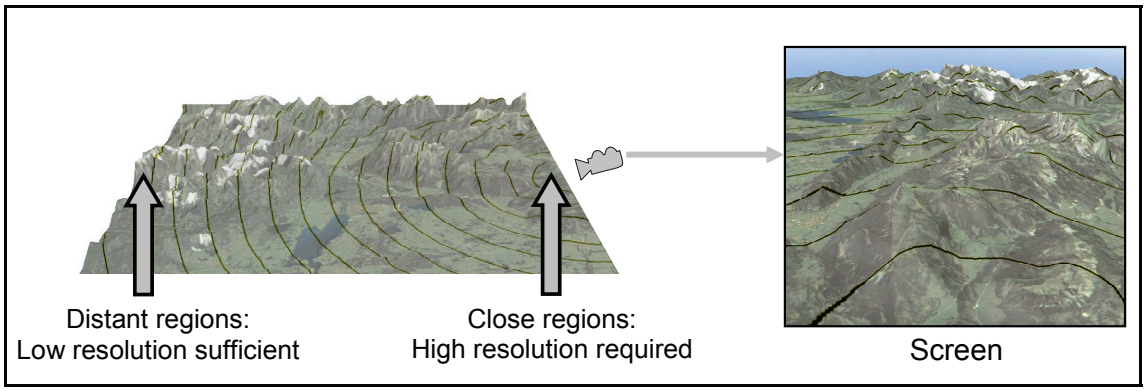


Figure 5: Principle of multiresolution texturing: Reducing texture resolution with increasing camera distance.

of the terrain can be close to the camera, the amount of actually needed texture data is approximately constant and independent of the full texture resolution.

Given a digital terrain model consisting of a set of triangles and a single large texture image T, the aim is to provide texture data at an appropriate resolution for each visible triangle at runtime.

For a given screen resolution, camera position, and viewing direction, the resolution of a texture can be considered to be *appropriate* if it is:

- *Sufficient*: No magnification occurs, i.e., the screen-space area onto which a single texture pixel is mapped is never larger than 1 for any texture pixel.
- *Minimal*: There is no smaller sufficient texture resolution.

In practice, the second criterion is only approximately fulfilled. Since the appropriate texture resolution of a triangle depends on the current camera settings, it is permanently changing during user navigation. Thus, it must be possible to access regions of T at different resolutions efficiently at runtime. This is provided by a multiresolution terrain texture.

2.2.2 Structure of Multiresolution Textures

A multiresolution texture is stored in the form of a quadtree [Samet 1984] structure as illustrated in Figure 6. The data structure is created in a preprocessing step and stored on hard disk. Each node N of the tree corresponds to a rectangular region $R(N)$ of the terrain texture T. For the root node, $R(N)$ corresponds to the full area of T. For each inner node N, the region of the four child nodes are obtained by splitting the region of N into four equally sized parts. Each leaf node stores its corresponding texture region at full resolution. The texture stored in each inner node is obtained by combining the textures of the four child nodes and scaling them down by a factor of two in both coordinate axes. The texture that is stored by a node is called a *texture tile*. For a texture tile F, the corresponding texture region is denoted by $R(F)$, i.e., $R(F) = R(N)$ for the node N containing F. The resulting tree structure has the following properties:

1. All texture tiles are of equal size.
2. For any tree level i, the texture tiles of all nodes at this tree level form a downsampled representation T_i of T. If k is the tree depth, T_i is obtained by downsampling T by the factor $2^{(k-i)}$.

The tree depth k is chosen sufficiently large, so that the texture tiles are small enough to be directly handled by the graphics hardware, e.g., 1024×1024 pixels.

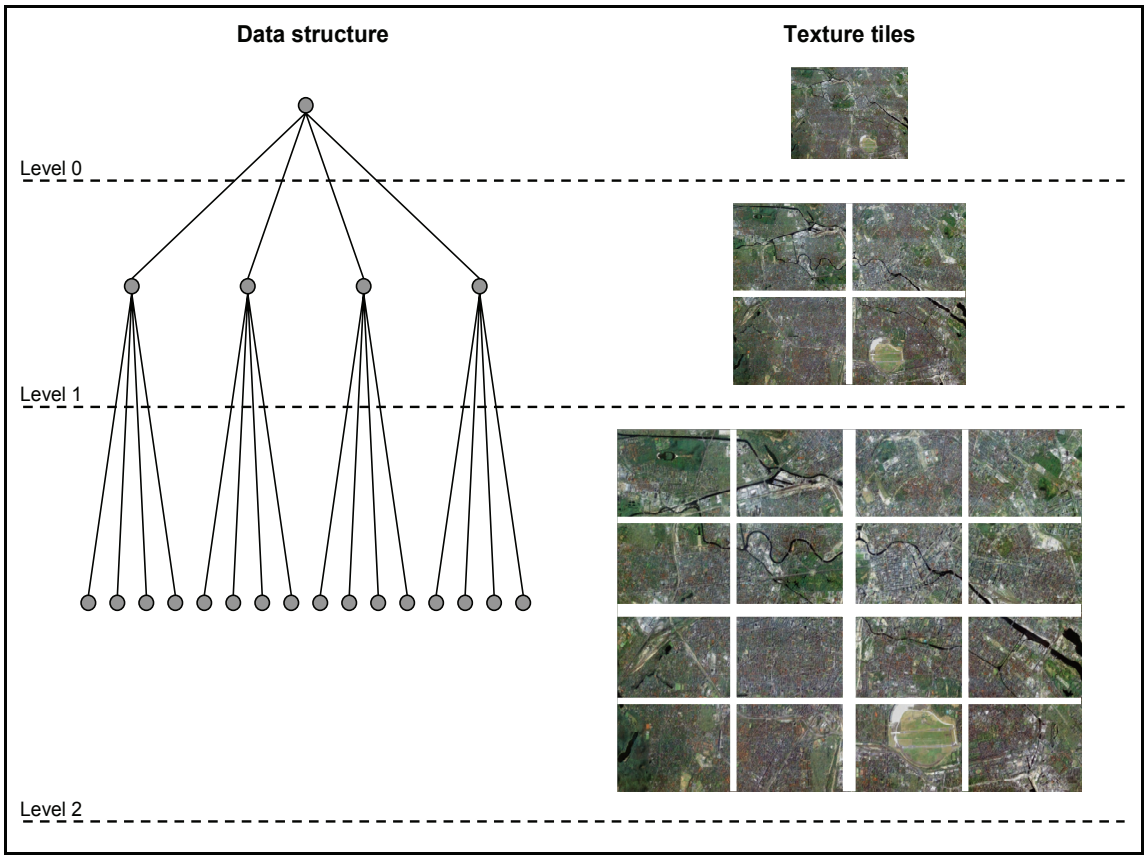


Figure 6: Data structure of a multiresolution terrain texture. Each node of the quadtree (left) stores a texture tile (right).

2.2.3 Rendering with Multiresolution Textures

During rendering, the multiresolution texture is used to obtain an appropriate texture tile for each triangle. To be appropriate for a triangle t, a texture tile F must provide an appropriate texture resolution according to the criteria defined in Section 2.2.1, and the texture region $R(t)$ of T that has to be mapped onto t must be fully contained in $R(F)$. Triangles must not intersect the borders between adjacent tiles because only one texture can be active for a single triangle.

Finding an appropriate tile for each triangle separately would cause considerable computational effort during rendering. Therefore, tile selection must be performed for several nearby triangles simultaneously. Here, a specific property of textured terrains can be exploited: Since the terrain texture is mapped orthogonally onto the terrain, the quadtree subdivision of T defines a directly corresponding quadtree subdivision of the terrain geometry, i.e., for each node N there is a corresponding rectangular terrain region $TR(N)$. The quadtree subdivision of geometry and textures is chosen in such a way that the borders of the quadtree cells correspond to the grid lines of the grid defined by the terrain vertices. In this way, each texture region $R(N)$ corresponds to a certain group of triangles that form a terrain region $TR(N)$ and do not exceed its borders. In addition to the texture tile, these triangles are also stored in N. Thus, the tree represents hierarchies of textures and geometry simultaneously. Using the geometric hierarchy, the texture tiles can be efficiently chosen by a depth-first traversal of the tree: If for a node N the texture resolution is appropriate for all triangles of $TR(N)$ or if N is a leaf node, the texture tile of N is applied to the whole terrain region $TR(N)$. Otherwise, the traversal is recursively continued for the four child nodes of N.

The above description of the selection process is slightly simplified and reduced to the aspects that are directly relevant for an explanation of multiresolution texture atlases. In fact, terrain rendering algorithms typically combine the texture tile selection with techniques for geometric level-of-detail control. In addition, the precise form of texture hierarchy and geometry hierarchy as well as the heuristics to determine whether a texture tile is considered to be appropriate or not vary between different approaches [Baumann 2000, Hwa 2004, Wahl et al. 2004].

2.3 Other Approaches

This section gives an overview of existing approaches that address texture complexity in terms of the number of textures or texture resolution. First, some standard methods are discussed that can be helpful to reduce texture workload or the number of texture switches. Second, a short overview of existing approaches based on multiresolution textures is given.

2.3.1 Standard Methods for Reducing Texture Workload and Texture Switches

Texture compression methods such as *S3 texture compression (S3TC)* represent an established method to reduce texture complexity. S3TC is a lossy compression scheme that achieves predictable compression ratios between 4:1 and 8:1, depending on the applied variant and the format of the input data (RGB or RGBA). S3 compression is supported by current graphics hardware, so that S3 compressed textures can be directly stored in the graphics memory and accessed for rendering without having to be fully decompressed before.

A simple approach to reduce texture complexity already in the modeling stage is to use texture repetition for surfaces with regular appearance. For instance, if a floor is textured with equally looking tiles, a single tile texture can be repeated several times. Texture repetition can only be used in special cases and does not reduce the number of texture switches as compared to using a single large texture.

A more sophisticated way of saving texture memory is provided by *procedural textures.* A procedural texture can be imagined as a texture that is not described explicitly but instead represented implicitly by program code. As discussed in Ebert et al. [2003], a formal definition of the term *procedural texture* is difficult to find. A procedural specification of a texture is usually much more compact than an equivalent explicit representation. Examples of procedural textures are given in Ebert et al. [2003]. Alternatively to specifying textures completely procedurally, there are also approaches to synthesize large seamless non-repetitive textures from small explicitly given example pattern images [Lefebvre and Hoppe 2005, Wei 2005]. Procedural textures are only useful to represent certain types of textures that exhibit certain known regularities or a homogenous appearance such as brick walls or wood surfaces. In addition, for explicit textures obtained from photographs, which are frequently used in city models, adequate procedural equivalents are rarely available.

If some textures of a scene are shared by multiple shapes, *state sorting* is an established technique to avoid unnecessary texture switches in each rendered frame. State sorting means to choose the rendering order of a 3D scene in such a way that all shapes sharing a single texture or, more generally, equal appearance attributes are rendered directly one after another. State sorting ensures that each texture is activated only once per frame.

2.3.2 Multiresolution Texturing Techniques

As stated in Section 2.2, multiresolution textures for terrain rendering have already been proposed in 1995 by Lindstrom et al. [1995]. To estimate an appropriate texture resolution for a rectangular terrain region, they use a heuristic that considers the camera distance and the

orientation relative to the camera. Cline and Egbert [1998] also use a quadtree, as described in Section 2.2, together with a texture caching framework that manages the dynamic transfer of textures between hard disk, main memory, and graphics memory. As stated by the authors, the described concept is actually not restricted to digital terrain models. The problem how to group triangles together to determine an appropriate texture tile efficiently, however, is only described for digital terrain models, and the approach is not demonstrated for general models. Baumann [2000] describes large-scale texture handling as a part of a terrain rendering algorithm. In contrast to previous approaches, it supports multiple terrain textures at different resolutions that can be mapped simultaneously to different regions of a single terrain. This allows, for instance, for locally refining a satellite image that covers a large region with an additional aerial image that provides higher texture resolution for a smaller area.

Shi et al. [2002] describe a multiresolution texturing approach for terrains and city models considering texture data for terrain and building facades. They divide the terrain into a regular grid and store textures of different resolutions in each grid cell. They assume relatively small facade textures and, hence, concentrate on terrain textures.

Wahl et al. [2004] also use multiresolution textures, but the primary novel aspects are concerned with efficient rendering of terrain geometry and are therefore not relevant in the context of multiresolution texturing. Hwa et al. [2004] introduce a new texture hierarchy for multiresolution terrain textures that provides an alternative to the usual quadtree subdivision scheme. Using this approach, switching between different texture resolutions for certain regions of the terrain due to camera movement is less noticeable than with previous approaches. Unfortunately, this approach is too tightly coupled with the structure of the digital terrain to be generalized to arbitrary scenes.

There are also a number of approaches that are not restricted to terrain models. Clipmaps [Tanner et al. 1998] and 3Dlabs' Virtual Textures are generic frameworks to handle large-scale textures that exceed the main memory capacity. They require, however, specialized hardware support that is not provided by standard graphics cards.

Lefebvre et al. [2004] propose a large-scale texture management approach for arbitrary 3D scenes. The key idea of their approach is to render all currently visible triangles into texture space, i.e., to render each triangle using its texture coordinates as vertex positions. In the resulting image, each pixel represents a region in texture space. A pixel is set if and only if the corresponding texture region is required for at least one currently visible triangle. As stated by the authors, however, the rendering part of the approach involves high performance costs for the graphics hardware and the whole geometry has to be processed multiple times per frame.

Lakhia [2004] described an approach for interactive rendering of detailed city models based on Hierarchical Levels of Detail (HLODs) [Erikson et al. 2001], a rendering acceleration technique that replaces distant scene parts by geometrically simplified versions to reduce the number of triangles to be rendered. To support texturing, they also store scene textures at different resolutions together with the geometry. Since the algorithm concentrates on geometric simplification, texture resolution is not explicitly considered as a factor for the scene subdivision and is not regarded in the level-of-detail selection heuristic, i.e., they do not provide explicit control of the texture quality. In addition, they do not address the problem of various separate textures either.

Sormann et al. [2003] described an approach to integrate texture support in the view-dependent simplification framework [Luebke and Erikson 1997]. They support multiple textures and combine them hierarchically into lower-resolution textures. Their approach reduces texture switches and texture memory requirement simultaneously. As stated in the paper, however, for scenes with thousands of textures the time spent in their runtime texture selection algorithm becomes a significant factor of the rendering performance.

Hesina et al. [2004] described a texture management for complex textured city models that dynamically loads textures for terrain and building facades. The paper proposes different policies to determine whether high or low texture resolution should be used for a part of the city model. Yet, it does not include a rendering algorithm for massively textured scenes.

Chapter 3

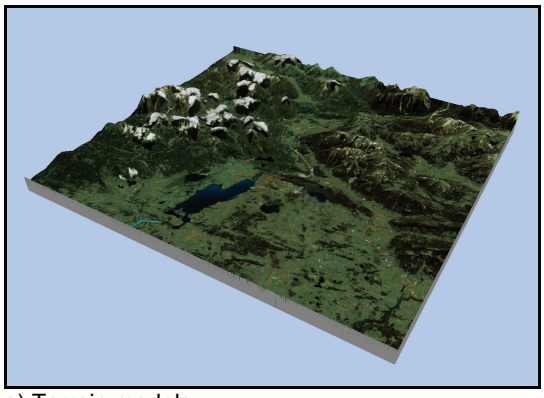
VISUALIZATION OF CITY MODELS

This chapter addresses fundamental aspects of city models and their visualization. First, it summarizes the components that form city models and methods to create them. Next, it discusses the role of navigation for real-time visualization and provides an overview of existing interaction techniques. Finally, it describes a selection of city model visualization applications.

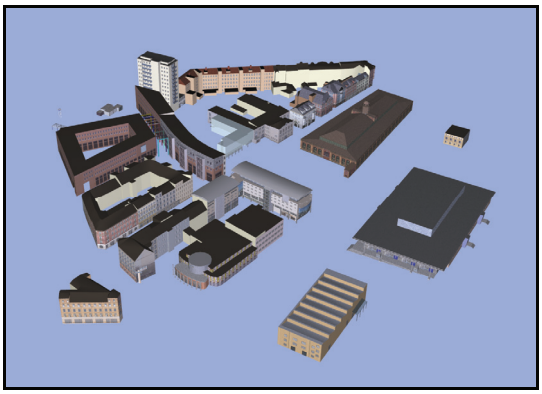
3.1 Components of City Models

This section describes the components city models are composed of and the commonly used specifications of these components. The components identified here correspond to the entities identified in the city model definition in Chapter 1. CityGML represents the only available open international standard for the complete specification of all city model components. Therefore, the specification of each component in CityGML is particularly considered in this section.

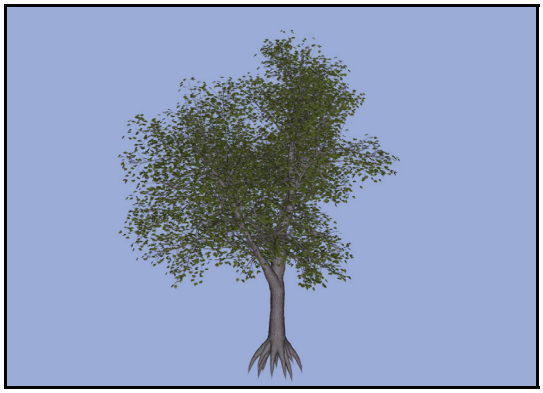
The description focuses on aspects of real-time rendering. Most components can be converted to textured triangle meshes for real-time rendering. For some components, such as digital terrain models, simplifying assumptions can be made and exploited by component-specific rendering techniques. Other components, such as models of buildings and city furniture, have to be handled as arbitrary textured triangle meshes.



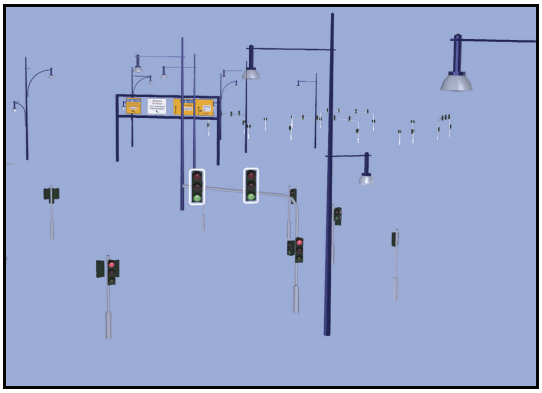
a) Terrain models.



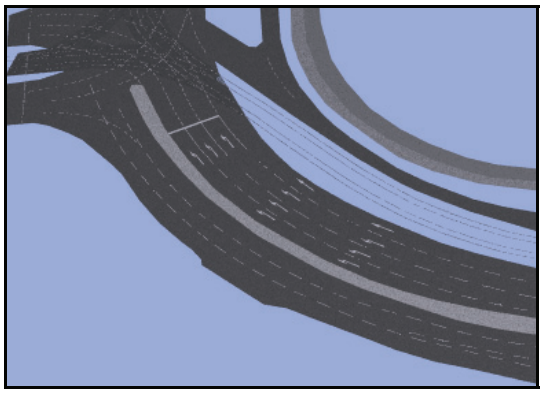
b) Building models.



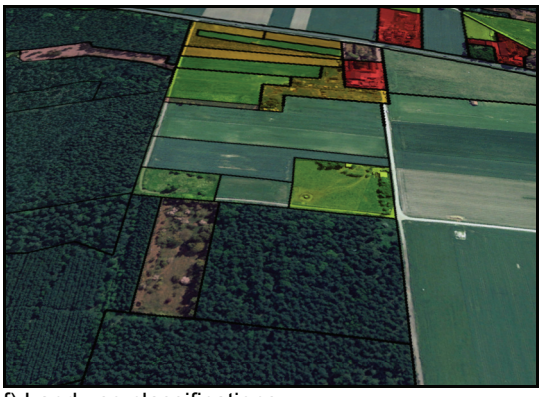
c) Vegetation models.



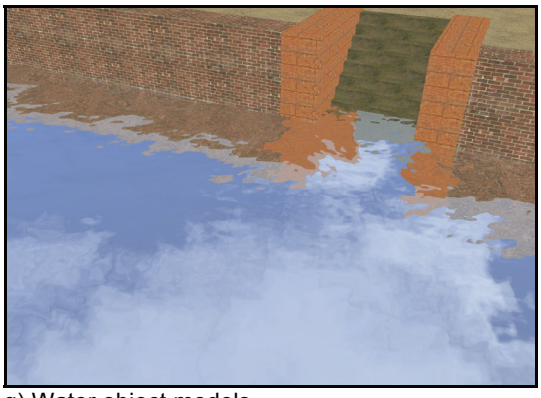
d) City furniture models.



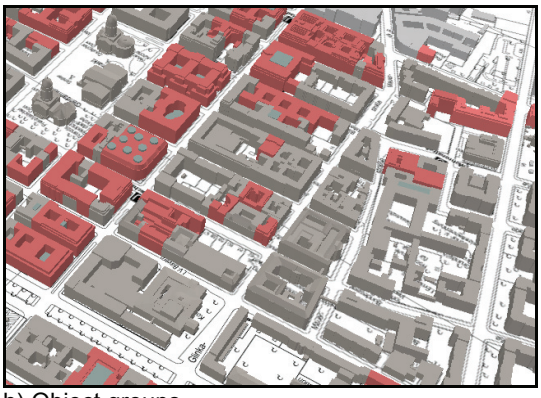
e) Transportation object models.



f) Land use classifications.



g) Water object models.



h) Object groups.

Figure 7: Components of city models.

Figure 7 provides an overview of examples for all components addressed. The distinction among these components is not always possible in a given city model. City models represented by generic 3D models, typically, contain a combination of multiple components for which there is no standardized way to distinguish them. In addition, an area or a 3D object might have different roles simultaneously. For instance, a road partially covered with grass can be interpreted as a plant-covered area as well as a traffic-related object. In CityGML, such conflicts are resolved by modeling such objects multiple times as different thematic objects that share a single geometric representation.

3.1.1 Terrain Models

A *digital terrain model* is a digital representation of a terrain surface by spatial coordinate triplets of a set of surface points [Bill and Zehner 2001] (translated from German). It provides a discrete approximation of the terrain surface geometry. The most common terrain representations are *grid terrain models* and *triangulated irregular networks* (TIN). In grid terrain models, the surface points are arranged in a horizontal regular grid. In TINs, they may be arbitrarily distributed provided that their projections to the horizontal plane are pairwise different. Each grid terrain model can be expressed as a TIN, but the simple structure of grid terrain models provides advantages for efficient rendering and storage.

For real-time rendering, the points of digital terrain models must be triangulated to obtain a 3D surface. As implied by their name, TINs typically provide a precomputed triangulation because the triangulation is a complex process for irregularly distributed points. The result of a triangulation consists of a triangle mesh that forms a contiguous surface and whose vertices correspond to the given terrain surface points. The terrain definition used in Section 2.2 corresponds to a triangulated grid terrain model. Due to their simple structure, grid terrain models are more common and are supported by the majority of terrain rendering algorithms, e.g., Lindstrom and Pascucci [2002] and Asirvatham and Hoppe [2005]. A terrain rendering approach for TINs is described in Hoppe [1998]. Baumann [2000] uses a hybrid terrain structure which supports the integration of local TINs into a grid terrain model.

Using terrain textures, high-resolution raster data can be displayed on terrain surfaces. Examples include satellite images, aerial images, topographic maps, and thematic maps. Figure 7a) shows an example of a textured grid terrain model.

CityGML supports the definition of terrains by arbitrary combinations of grid terrain models and TINs. Each of these terrain models may be restricted to a certain region of validity for which it provides a valid representation.

The classical digital terrain models introduced above provide only a rather rough approximation of urban terrain surfaces, because ground-structuring elements such as streets, pavements, or flower-beds are not represented as explicitly modeled 3D objects.

3.1.2 Building Models

Building models can be specified in various ways. Most representations explicitly support textures and, in contrast to terrain models, arbitrary shapes. Therefore, there are no established building-specific rendering approaches. The following paragraphs briefly describe different examples of building specifications.

Buildings in CityGML

CityGML supports the specification of city models at five levels-of-detail (LOD), which are primarily characterized by the way in which buildings are represented (Figure 8):

- *LOD0*: At the coarsest LOD, a city model consists essentially of a digital terrain model with overlaid raster images. Explicit 3D representations of buildings are not provided.

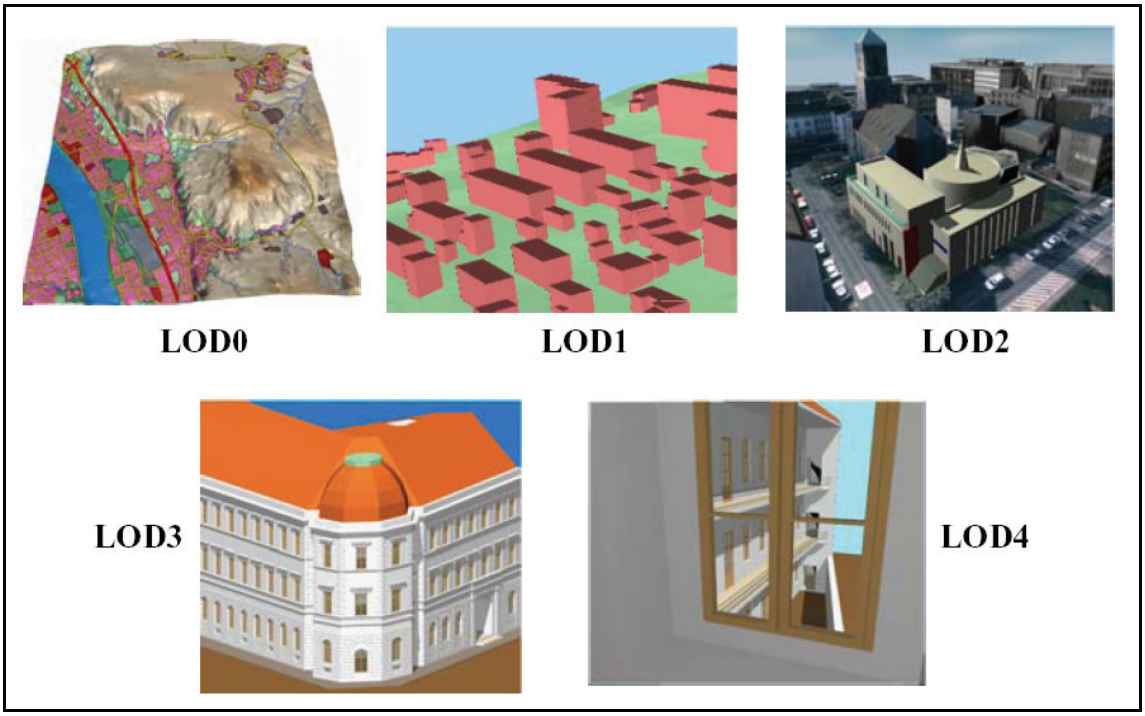


Figure 8: The five levels of detail in CityGML (Source: [Gröger et al. 2006]).

- *LOD1*: Buildings are represented as *block buildings*, i.e., simple 3D shapes that can be obtained by extruding horizontal footprint polygons along the vertical axis. Facades and roofs may be textured in all LODs.
- *LOD2*: Buildings may have an explicitly specified roof geometry, described by a set of polygons in 3D. In addition, the outer walls and the roof of a building are represented as separate thematic objects.
- *LOD3*: Buildings are represented as detailed architectural models. Openings such as windows and doors are accessible as separate thematic objects. All parts of a building may be textured.
- *LOD4*: In addition to LOD3, interior building structures such as interior walls, rooms, and furniture are provided as separate thematic objects.

A single building may contain representations for multiple LODs simultaneously. The geometry of buildings and sub-objects of buildings is specified using a general *geometric model* in CityGML, which is also used for other city model components. In the geometric model, a 3D object is described by hierarchical aggregations of polygons that represent the object's boundary surfaces. The model supports the specification of texture images and texture coordinates for all surfaces.

Continuous-Level-of-Quality Buildings

Continuous-level-of-quality (CLOQ) *buildings* [Döllner and Buchholz 2005] describe a building's geometry on a per-floor basis. Each floor refers to a floor prototype, which is defined by a ground plan, walls, and wall segments. To specify the appearance, projective textures across floors and textures per wall segment are supported. The floor-based construction allows for expressing most common building features such as courtyards, passages, multipart buildings, canopies, balconies, terraces, protrusions, and intrusions by a small set of construction components.

CLOQ buildings complement the CityGML building model by supporting refinement of buildings across different LODs using intuitive editing operations. Most frequent operations include adding and modifying floors, walls, and sections of walls. For instance, it is possible to add a penthouse to a CLOQ building by replicating its top floor and reshaping the ground plan of the new floor. A selected floor may also be enhanced by adding geometric details such as columns. Such operations cannot be defined for generic boundary representations of building geometry.

Industry Foundation Classes

The Industry Foundation Classes [IAI 2005] have been developed by the International Alliance for Interoperability (IAI) and provide a general standard to share building data among different applications such as tools for computer aided design (CAD) and technical analysis. In contrast to CityGML, the IFC standard is not intended to specify complete city models including multiple buildings and other city model components. Instead, its development is driven by specific requirements of applications that are related to the phases of design, construction, and maintenance of a single building. Therefore, it includes a comprehensive set of classes to describe single building elements such as water pipes, pumps, and air filters as well as their technical properties. CityGML and IFC data can be joined using the *external reference* feature of CityGML, which allows for specifying links to external data sources for all objects in a CityGML model.

Attributed Footprint Polygons

Collections of block buildings can be specified as 2D footprint polygons with attached attributes. These attributes must be interpreted by an application to derive 3D shape and appearance of each building. Examples of interpreted attributes include:

- *Height values*: To obtain 3D geometry from a footprint polygon, two height coordinates are required to define the extrusion along the vertical axis.
- *Roof type*: Some data sets contain roof type identifiers such as ‘hip roof’, ‘gable roof’, or ‘flat roof’. A roof type identifier can be interpreted to create an estimated roof geometry automatically [Laycock and Day 2003].
- *Facade texturing attributes*: A restricted form of texture mapping is possible by referencing texture images that are repeated around building facades. For each building, the corresponding image is referenced by a string-attribute that contains an image file name. In addition, a projection width must be specified by a floating-point attribute. The *projection width* specifies the width to which a single image is scaled due to the projection to the 3D facade. The vertical texture scaling is chosen in such a way that the image height corresponds to the height of a building.

Storage of georeferenced 2D polygons and related attributes is supported by established file formats such as ESRI shapefiles and GML. The advantage of the footprint-based storage form is its compactness. In contrast to block buildings in CityGML, however, this specification is not fully self-contained but requires dataset-specific knowledge for interpreting the data correctly. For instance, it must be known which attributes provide the above mentioned information. In addition, there is no common agreement how to specify the vertical extrusion parameters. For example, the minimum height value for the vertical extrusion might be specified by an attribute, by vertical coordinates in the polygon vertices, or, implicitly, by an underlying digital terrain model in another file.

Generic Graphics and Geometry Descriptions

Before the introduction of CityGML, the most common way to specify detailed building models was represented by generic 3D scene formats such as VRML [1997], X3D [2004], Collada, or

3D Studio Max. For instance, the buildings shown in Figure 7b) form a part of a city model of Chemnitz, which is stored as a collection of X3D files. Generic 3D scene formats, essentially, describe building shapes by sets of textured or non-textured polygons. GML and shapefiles also support 3D geometry, but do not support textures. Techniques to extend generic geometry descriptions of building models by structural and thematic information to obtain CityGML buildings at LOD3 or LOD4 currently represent an open research problem.

3.1.3 Vegetation Models

The term *vegetation* describes 'all the plants in a particular place' [Longman Dictionary 1987]. That is, the vegetation of a city is formed by a complex set of individual plants. Plant groups that constitute biotopes such as lawns or forests can be collectively identified as area-related entities. Correspondingly, the *vegetation model* of a city model comprises representations of individual plants as well as vegetation areas.

Representation of Individual Plants

A *plant instance* represents a single plant in a city model. It is described by a reference to a plant model and a related transformation matrix to determine position, orientation, and size of the plant instance. A *plant model* is a 3D graphics representation of a single plant (Figure 7c). Within a vegetation model, a plant model provides a prototype that is used by one or more plant instances. In this way, it represents a class of plants with similar properties such as species or age.

Theoretically, plant instances may also be specified as individual models. The prototypic use of plant models, however, is common practice, because the consideration of differences among similar individual plants is rarely required.

In CityGML, individual instances of plants are referred to as *solitary vegetation objects*. A solitary vegetation object may reference a prototypic plant model but may also define an individual 3D representation. The concept of sharing prototypic models by multiple instances is implemented as a general concept in CityGML and is also available for other components of a city model.

With respect to real-time rendering, plant models have some relevant characteristics: On the one hand, detailed plant models typically exhibit a high geometric complexity. For instance, the tree model in Figure 7c) comprises about 200,000 triangles. This is necessary to model complex geometric structures such as a treetop appropriately. On the other hand, the basic shape of a plant and small visual details are mostly perceived only approximately and are not explicitly remembered by an observer. Due to these properties, plant models are well-suited for rendering-acceleration techniques that replace parts of the model by textured quadrilaterals to reduce geometric complexity, e.g., Behrendt et al. [2005].

Representation of Vegetation Areas

A vegetation area is characterized by a polygonal area and related information about the vegetation within this area such as species and average height. CityGML-representations of vegetation areas are referred to as *plant cover*. Plant cover objects may provide an explicit 3D representation that can be used for rendering. Without an explicit representation, there are two ways to obtain a corresponding rendering representation:

- If an appropriate plant model is available for the given species, instances of this model can be randomly distributed over the area for rendering.
- Certain types of vegetation can be rendered by specific rendering techniques such as the grass rendering technique by Shah et al. [2005] and the forest rendering technique by Decaudin and Neyret [2004].

3.1.4 City Furniture Models

The term *city furniture* comprises immovable objects such as lanterns, traffic signs, traffic lights, telephone boxes, benches, delimitation stakes, billboards, and advertising columns (Figure 7d). City furniture objects of the same type, typically, share a common appearance. Therefore, similar to plant models, using multiple instances that share a single prototype model is also useful for memory efficient representation of city furniture objects. A specialized rendering technique for city furniture objects, however, does not exist because the 3D representation of a city furniture object can be arbitrary, textured geometry. Therefore, city furniture objects are rendered using general rendering techniques for arbitrary scenes.

3.1.5 Transportation Object Models

Transportation objects are entities with a traffic-related meaning, e.g., streets, railways, squares, sidewalks, and tracks. They form, on the one hand, parts of network structures, and, on the other hand, individual objects with 3D shape and appearance. Correspondingly, transportation objects can be specified either in the form of 2D linear networks composed of nodes and edges or as textured 3D objects such as in Figure 7e).

In CityGML, linear network representations are used to specify transportation objects at LOD0 and explicit 3D representations for all higher LODs. The transportation object models in CityGML also include auxiliary traffic areas such as grass stripes or kerbs, which have no transportation-related meaning but are attached to traffic areas.

2D linear networks can be displayed as 2D vector data overlay on the terrain surface [Kersting and Döllner 2002]. 3D representations may consist of arbitrary textured geometry. Particularly, although they are close to the terrain, they cannot be rendered using terrain rendering approaches in general because they may contain vertical or down-facing triangles.

3.1.6 Land Use Classifications

Land use data specifies the way in which a certain area of land is utilized, for instance, as settlement area, industrial area, or farmland. They are represented by polygons in the horizontal plane and have assigned attributes providing descriptive information such as the classification and usage of an area.

In CityGML, land use areas can be represented as polygons, but may also provide explicit 3D representations consisting of textured surfaces for higher levels of detail. If they are provided as 2D polygons, they can be overlaid on the terrain surface for rendering as shown in Figure 7f).

3.1.7 Water Object Models

Water objects include rivers, lakes, canals, and basins. Visually, water objects are primarily characterized by their surface but, physically, they form volumes. Therefore, CityGML supports the specification of water objects as volume objects. The boundary surfaces of a water object are classified into the categories ‘air-to-water’, ‘water-to-ground’, and ‘water-to-water’, where the third category allows for specification of imaginary boundary surfaces to separate adjacent water objects from each other. In this way, the water volume can be fully specified, but the water surface is separately accessible for rendering.

For real-time rendering, water objects play a specific role because they are dynamic and cannot be appropriately represented by textured surfaces. Specific rendering techniques for water surfaces take into account dynamics as well as reflection and refraction of light as shown in Figure 7g) [Finch 2004, Claes 2004, Kryachko 2005].

3.1.8 Object Groups

Groups of objects may have own semantics and might, therefore, form discrete relevant entities of a city model. For instance, multiple buildings can form a building complex or different buildings of the same architect can be composed in a group.

In CityGML, groups can be explicitly stored as separate objects. These objects contain references to other objects that form the group. Each group may contain arbitrary objects of the thematic model such as buildings, single windows, vegetation objects, and city furniture. Particularly, object groups may contain other groups, so that they can be specified hierarchically. Each group may define attributes that describe its meaning as well as role names for each group member.

In contrast to the previously mentioned components, object groups are not visual. A single object group can be visualized by highlighting the contained objects. For instance, the red buildings in Figure 7h) form the group of all buildings in the region around 'Unter den Linden' in Berlin that have been built after 1990. Alternatively, a group can be visualized by replacing its members by a single generalized geometry, if available. For this, CityGML allows for assigning additional geometric representations for groups.

3.1.9 Other Components

The components introduced above represent the basic entities of a city. They are, however, not necessarily sufficient to describe all relevant aspects for a specific application. In the design of CityGML, the integration of information that is not explicitly supported is considered for two reasons:

- *Specialization*: CityGML aims at providing a general data model for sharing city model information among different applications. For this goal, however, it is not necessary to consider each possible requirement of any application. For instance, technical details of gas tanks or specific geological terrain properties are only relevant for specific application domains. Therefore, CityGML concentrates on aspects that are relevant for a broad range of different applications.
- *Development*: CityGML is still being developed further. For instance, thematic classes for certain site features such as bridges and tunnels are scheduled for integration but are not yet available.

CityGML provides the following ways to integrate domain-specific and application-specific information:

- *Generic city objects*: They are used in CityGML to represent objects that cannot be described by other classes of the thematic model. Examples of objects that are currently being modeled as generic city objects include bridges, tunnels, free-standing walls, pipelines, power lines, and dams.
- *Generic attributes*: The thematic classes in CityGML provide several attributes to store general properties such as addresses of buildings or species of plants. Generic attributes allow for assigning additional application-specific attributes to each object.
- *Domain-specific extensions*: CityGML is implemented as XML based schema. It forms an extension of GML. Similarly, further extensions of CityGML can be implemented for specific application domains. This allows, for instance, for extending existing classes by additional attributes, to introduce sub-classifications of city model components, and to add support for new components.

3.2 Creation of City Models

This section discusses methods to create 3D representations of various city model components. The acquisition of digital terrain models and building models has been extensively studied [e.g., Li et al. 2005, Brenner 2005]. Vegetation objects and city furniture objects are usually provided by libraries of prototypic 3D models.

Ground-structuring models, which complement building and vegetation models, however, have been less intensively studied, and almost no established methods for automatic construction are known so far. These models comprise manmade surface structures (e.g., roads, pavement, walls, stairs, or squares), natural terrain surfaces (e.g., agricultural land, grassland, or rocks), and water surfaces (e.g., rivers, canals, or lakes). A collection of these models will be referred to as *urban terrain model* in the following. In CityGML models, urban terrain models include several traffic objects, land use objects, water objects, and generic objects with detailed 3D representations.

Due to the lack of automatic construction methods, urban terrain models are rarely available. This represents a significant problem for visualization applications and systems using perspectives that are close to the ground level. Therefore, construction methods for urban terrain models are discussed in more detail in this section.

3.2.1 Building and Terrain Models

Acquisition of building and terrain models has been addressed by a variety of approaches. A comprehensive overview of existing techniques would exceed the scope of this work. See Foerstner [1999], Ribarsky and Wasilewski [2002] and Brenner [2005] for more detailed information. Many approaches are already available in automatic or semi-automatic tools such as inJECT, Cyber City Modeler, or MetropoGIS.

The most common approaches extract information from aerial images [e.g., Baillard and Zisserman 1999], from 3D point clouds obtained from airborne [e.g., Haala and Brenner 1999(i)] or terrestrial [e.g., Früh and Zakhor 2004] laser scanners, or from both. The primary difficulty is to identify and separate buildings, the underlying terrain surface, and other objects such as vegetation, people, or cars in the given raw data. If available, building footprints from cadastral databases can also be taken into account by some approaches. From aerial images, building shapes are constructed based on feature detection methods. If multiple images are available from different perspectives, this can be exploited to derive 3D information. Laser-scanning-based approaches construct 3D building shapes in such a way that their surfaces approximately matches a given point cloud. To achieve maximum reliable results, manual correction is still common practice. The result of most algorithms consists of block buildings, possibly extended either by explicit roof geometry or by roof types such as hip, gable, or flat roofs and parameters such as the slope of the roof surface. High-resolution facade textures are obtained from terrestrial photographs [Laycock and Day 2004]. Fully automated acquisition of detailed building models sufficient for LOD3 in CityGML is still an open problem, particularly if considering the semantic structure.

For applications in which a high degree of visual detail is more important than correctness of city models, e.g., computer games, techniques for procedural modeling of cities provide a possible alternative to building acquisition. Such techniques allow for randomly generating detailed city models based on a set of construction rules [Wonka et al. 2003, Müller et al. 2006].

3.2.2 Vegetation Models

To provide plant models for a city model, a catalogue of prototype plant models is needed as well as a specification where to place the instances and which models to choose. Detailed plant

a) b)

Figure 9: Aerial images are well suited for bird's-eye views (a), but fail for ground-level views as illustrated by the flattened trees (b).

models are usually created manually. Specialized plant modeling tools such as XFrog or Bionatics facilitate the modeling process.

Data for the plant placement can be derived from different data sources. Most city administrations are maintaining databases that provide positions and additional properties such as species, size, and age of trees. In addition, there are techniques for the acquisition of plants from aerial images. For instance, the approach of Haala and Brenner [1999(ii)] acquires information on vegetation using aerial images with infra-red channels, exploiting the fact that vegetation appears different from other parts of a city in infra-red images. Bacher and Mayer [2000] consider shadows of trees in aerial images to derive height information.

3.2.3 Urban Terrain Models

The majority of currently available city models represent terrain by a digital terrain model draped by aerial images to avoid the need for detailed urban terrain models. This approach is sufficient for visualization from a bird's-eye view (Figure 9a), but does not provide sufficient detail for visualization from a pedestrian's perspective (Figure 9b). In addition, the represented terrain elements cannot be analytically identified and modified because there is no concise object-oriented representation of these elements. This makes aerial images unsuitable, e.g., for planning tasks where explicit representations of these elements are required.

2D vector-based representations of urban terrain provide more structured information. 2D vector data are available from various administrative data sources such as ATKIS [ATKIS] or ALK [Bill and Zehner 2001], which are permanently maintained and updated. The integration of such data in an automatic workflow for the creation of 3D urban terrain models would significantly facilitate the city model creation process.

3.2.4 Traditional Approaches for Urban Terrain Modeling

There are three common approaches for creating urban terrain models: *manual modeling, triangulated irregular network* (TIN) *models*, and *attribute-based mapping of ground textures*.

Manual Modeling

Using standard 3D modeling tools or CAD tools for manual modeling of urban terrain provides maximum flexibility. It requires, however, high manual effort and a high degree of expertise. In addition, the result consists of purely graphical 3D representations. 2D data can be used as a starting point for modeling, but the resulting 3D models are detached from these data. Hence, the following limitations apply:

- *No automated updating*: The 3D models cannot be reused if underlying 2D geodata change.
- *No geodata context*: The 3D models have to be created separately from related geodata that could be helpful for editing. For instance, showing aerial images or topographic maps can be useful to validate a 3D model visually while editing. Since generic 3D editing tools are not aware of the geo-spatial context of the data, they do not provide such functionality in general.
- *No geodata linkage*: The 3D models do not preserve data semantics and thematic information contained in or associated with 2D plan elements.

TIN-based Modeling

TIN models can also be used to represent detailed terrain models. They support the integration of land use information and built surface structures by using line or polygon features as break lines and by assigning different colors or textures to certain triangles. This way, detailed terrain models can be created from GIS input data. The TIN-based approach, however, has two major disadvantages:

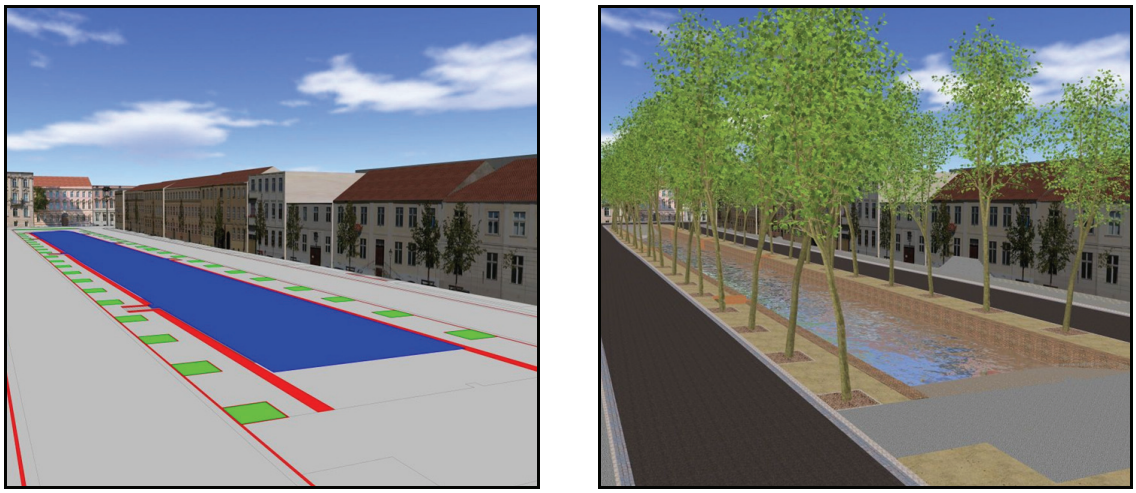
- *No linkage to 2D plan elements*: The underlying 2D plan elements are only implicitly represented by the TIN geometry, but a TIN contains no direct links to the original data from which it has been created. Therefore, thematic and semantic information is lost and changing the original data requires rebuilding the TIN.
- *Restricted geometry*: As an inherent limitation, vertical faces cannot be modeled by TINs.

Attribute-based Mapping of Ground Textures

Specialized modeling tools for virtual landscapes such as World Construction Set, ArcScene, or VirtualGIS facilitate the integration of GIS data by draping polygons onto the terrain and assigning colors or textures to these polygons. Such tools reduce the manual modeling effort and are able to create satisfying results at landscape scale [Appleton et al. 2002]. In landscape planning, they are primarily used to visualize terrain surfaces carrying vegetation by textures [Muhar 2001] and prototype 3D plant models. In addition, the tools mentioned above are suitable to visualize roads or other manmade terrain surfaces. In order to integrate manmade surface structures that include textured vertical faces, however, manual modeling effort and extensive data preparation is required.

3.2.5 Smart Terrain Models

The *smart terrain* approach [Buchholz et al. 2006] is a concept for generating detailed urban terrain models from 2D vector-based plans. Figure 10 shows an example. Figure 10a) shows a 2D vector data set describing a downtown area in Potsdam. Figure 10b) shows the resulting smart terrain model. The approach is based on the idea to keep the original 2D data and to add complementary information to enhance them to a complete 3D scene specification in an automated process. It is similar to the attribute-based texture mapping method but extends it by



a) b)

Figure 10: Snapshot of a 2D vector data set and superimposed building models describing a downtown area in Potsdam (a) and the resulting urban terrain model including vegetation (b).

defining specialized representations for typical terrain elements. This increases the modeling flexibility while providing significant advantages compared to general 3D graphics representations:

- *Accessibility for 2D GIS*: The underlying 2D vector data that form the core of the smart terrain model specification can still be accessed and edited by external 2D GIS tools and can be obtained and updated from 2D data sources.
- *Interactive queries in 3D*: Thematic information and data semantics can be directly queried within the 3D visualization.
- *Editing in 3D*: Due to the permanent link between 2D data and their 3D representation, interactive visualization of the generated 3D models can be directly combined with editing tools for the underlying 2D data. This way, the effect of modifications of the 2D data can be directly shown in the 3D model. Particularly, it allows for immediate corrections of errors in the underlying 2D data when they become apparent in the 3D visualization.
- *Specific rendering techniques*: Since the data semantics is preserved in the 3D representation, smart terrain models form a basis for the development of visualization tools that are aware of the data semantics and use it to apply specialized real-time rendering techniques for certain surface elements such as water and grass.

The principal workflow of the approach is illustrated in Figure 11. The specification of a smart terrain model provides the complete information for automatic generation of a 3D model representation and consists of two parts, the *base data* and the *generator configuration*.
The *base data* consist of polylines and polygons with attached attribute tables. They form a vector-based 2D plan of the urban terrain. In the following, single polygons or polylines of the base data are referred to as *base vector objects*. Each attribute table defines a set of numerical, textual, or Boolean values that can be accessed via certain attribute-table key strings. The base data can be obtained from existing geodatabases but can also be edited directly in the 3D-editor, e.g., based on a given aerial image. The base data can be specified in any format that allows for describing 2D vector data with associated attribute tables, e.g., ESRI shapefiles or GML.

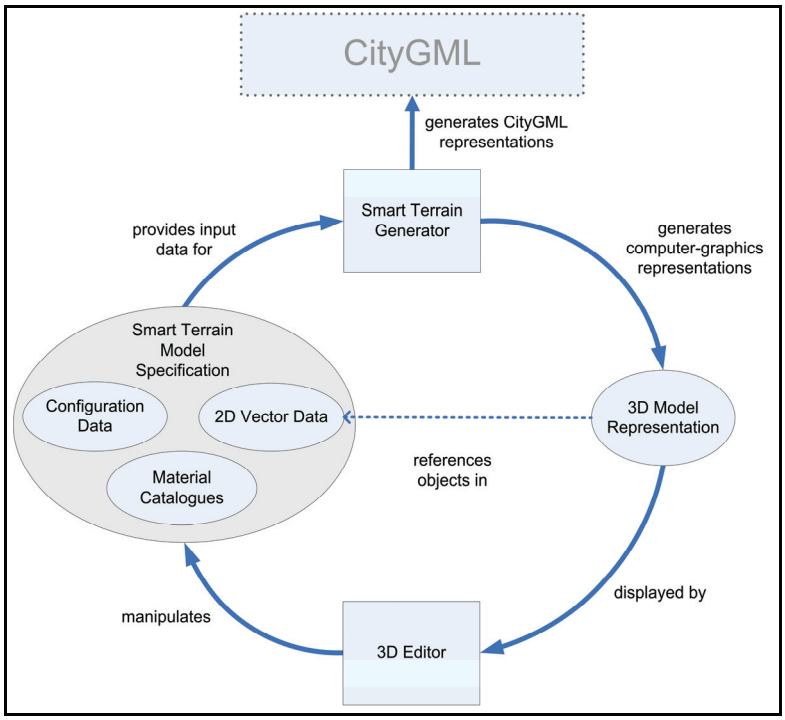


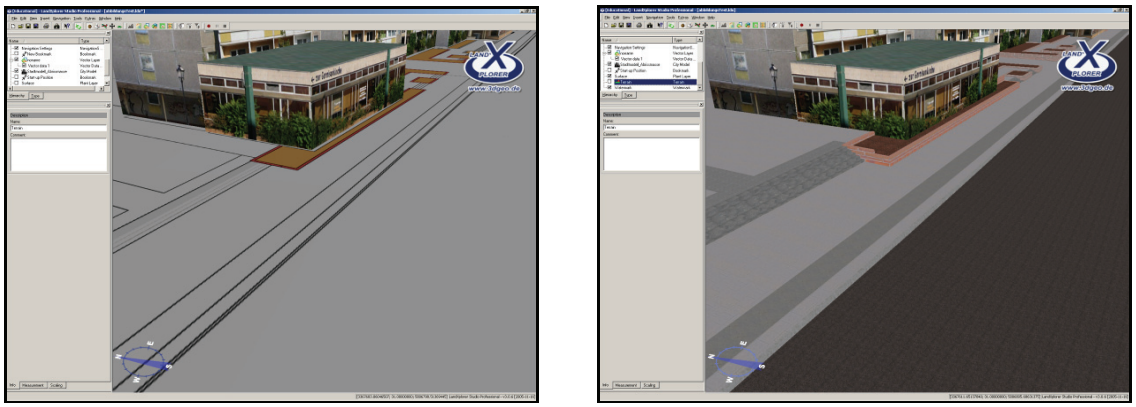
Figure 11: Workflow overview for the automated construction of urban terrain models.

The *generator configuration* specifies the way in which base data and related attributes are interpreted to generate the 3D terrain model. For instance, in a given base-data set polygons representing water areas may be indicated by defining the value 'Water' for the attribute-table key 'Area Type'. The generator configuration also defines one or more material catalogues.

A *material catalogue* provides a set of materials that can be referenced by the base vector objects via unique names. A material can be one out of two types: Color and texture. A *color material* defines an RGB color value. A *texture material* defines a reference to a texture image file and scaling parameters that define to which width and height the texture is stretched when it is applied to a surface. Each terrain element that references a texture material must define an anchor point onto which the texture origin is mapped in the 3D model and an orientation angle of the texture. Since these values are usually different even for objects of the same material, they are not stored as a part of the material itself. The generator configuration and the material catalogues are stored as separate XML files.

The *generator* takes the smart terrain model specification as input and creates a 3D representation of the terrain that is used for real-time rendering. The generator also maintains for each generated 3D object a reference to the underlying base vector object. This information is used by the editor to support selection and editing of surface-model elements directly in the 3D scene.

The *interactive 3D editor* is used to create and manage 2D base data, the generator configuration, and material catalogues. It provides real-time visualization of both the underlying base data and the resulting 3D representation (Figure 12). The 2D base data can be displayed on the surface of a grid terrain model using the technique described by Kersting and Döllner [2002]. If a terrain element is selected and edited by the user, the modification is applied to the underlying base data, and the generator immediately updates the corresponding parts of the 3D representation, so that changes of the base data are directly visible in the 3D environment. The 3D representation itself, however, is never changed by the user but always



a) Base data view. b) 3D view.

Figure 12: Snapshot of the editor system: The user can switch between a) the 2D view of the base data and b) the resulting 3D representation.

fully determined by the smart terrain model specification. Hence, editing effort is never lost when the 3D model has to be re-built by the generator.

The smart terrain model specification as well as the resulting 3D representation provides relevant information for an export to CityGML. The smart terrain model specification provides information on data semantics and thematic properties as well as a representation at low LOD. The 3D representation can be converted to the geometric model of CityGML to provide a representation for high LOD.

3.2.6 Smart Terrain Model Elements

The elements of a smart terrain model are organized in the classes `GroundArea`, `WaterArea`, `Stair`, `Wall`, `Kerb`, and `Barrier` (Figure 13). Instances of these classes are not explicitly part of the specification but are defined implicitly by the base vector objects and their related attributes. Most terrain elements correspond to exactly one base vector object. The only exception holds for irregular stairs, which require multiple polygons for specification. The attributes of a base vector object determine the class of the corresponding terrain element. Depending on the respective class, the required attributes for 3D shape and appearance of a terrain element are also taken from the attribute table of the base vector object.

The primary design goal was to make the refinement of pure 2D base vector data to a full smart surface-model specification as simple as possible for typical use cases. If the class model provided the unrestricted geometric flexibility of a generic 3D tool, it would not be possible anymore to specify the terrain elements completely via 2D base data, and the editing process would become rather complicated. Thus, the main advantages of the system would be lost. Therefore, the class model is restricted to cases that can be intuitively described by 2D polygons or polylines and supports optional refinement of individual objects by external 3D tools. For this, for each terrain element, the automatically generated 3D model can be exported, externally refined, and finally referenced by the terrain element. As a result, the effort for fitting an externally created 3D model correctly in the scene can be avoided and the 3D model is still related to the underlying base data.

Ground Areas

A `GroundArea` represents a polygonal surface on the ground that is displayed with a certain appearance. For instance, a ground area may represent a part of a street, a sidewalk, a lawn, or a wasteland area. Since the `GroundArea` class covers several kinds of terrain elements, it could

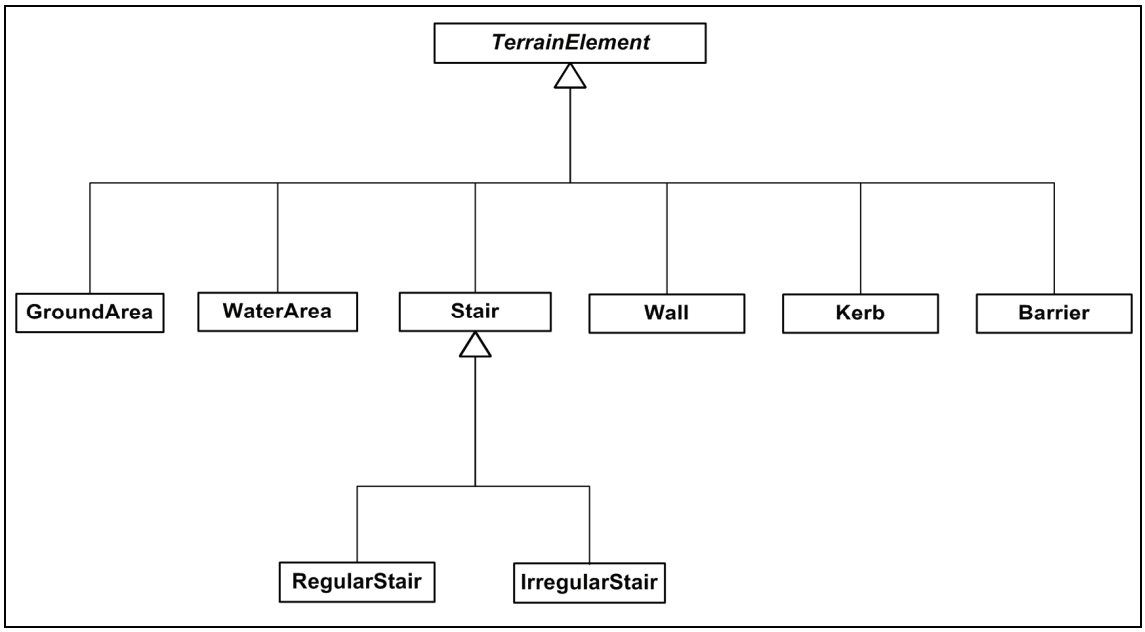


Figure 13: Overview of the classes of smart terrain models.

theoretically be split up into several subclasses but there are some reasons to use a single class instead:

- To enforce a generic sub-classification of ground areas can be impossible or ambiguous. For example, a single asphalted area might be interpreted as a part of a parking area, a part of a street, or a part of a square.
- From a technical point of view, a finer classification is not necessary because the created 3D representations differ only by material.
- From a user's point of view, a finer classification is not necessary because thematic classification can be obtained from the base vector objects' attribute tables, and, visually, thematic sub-classification can be achieved by assigning different materials.

Each `GroundArea` is defined either by a base-data polygon or by a polyline that is buffered to a certain width. If the data format used for the base data supports 2.5D vector data, i.e., if it allows for specifying per-vertex height values for each vector object, as in the case of shapefiles or GML, the surface geometry can be completely defined by the geometry of the underlying base vector object. If height values are not provided by the initial base data, they can either be automatically derived from a digital terrain model or edited manually. To assign height values by projecting the dataset onto a digital terrain model corresponds to the principle of image draping and is a good solution for continuous surfaces without vertical breaks.

In the general case, however, `GroundAreas` do not necessarily define identical height values at their borders. Therefore, some `GroundAreas` must be rendered with vertical border faces to avoid holes in the terrain model. For this, an optional extrusion depth can be specified, by which the surface is extruded downwards. If desired, a separate material can be specified for the vertical border faces.

Water Areas

A `WaterArea` represents a polygonal surface that appears as a water surface in 3D visualization. The geometry of a `WaterArea` is specified in the same way as a `GroundArea` but it must always be fully horizontal and allows for specifying a desired flow direction of the

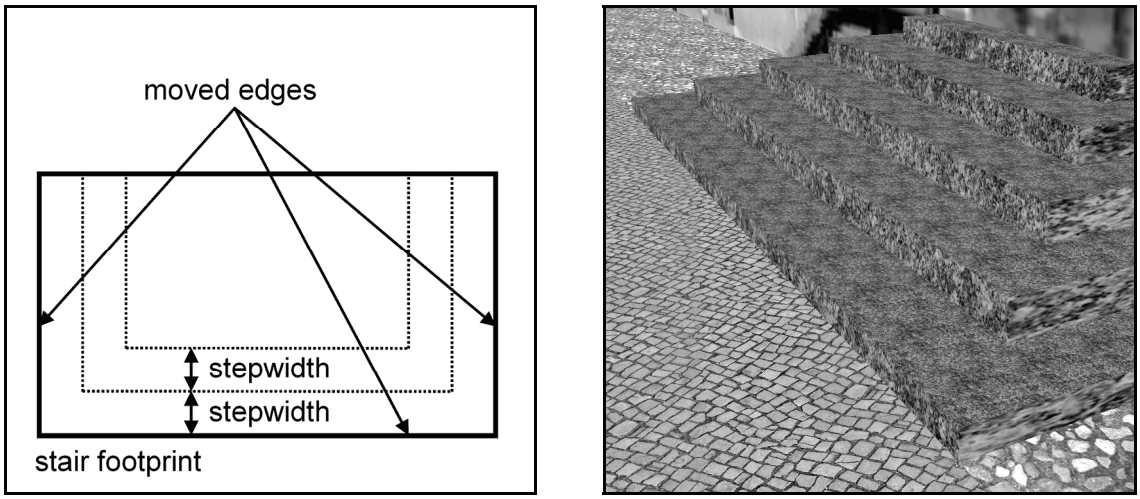


Figure 14: Construction of a simple stair: The footprints of the upper steps are defined by moving inwards certain edges of the full stair footprint (a). The 3D shape is then defined by extruding each step polygon downwards (b).

water. Water areas have been integrated as an own class to allow viewing applications for applying specific water-rendering techniques [Finch 2004, Claes 2004, Kryachko 2005].

Stairs

In the general case, a `Stair` is described by multiple base-data polygons, whereby each polygon represents a single step and is extruded downwards (`IrregularStair`) to obtain the 3D shape of the step. Many stairs can be described more simply via a single rectangular footprint by moving certain edges of the rectangle inwards to describe the footprints of the upper steps (`RegularStair`, Figure 14). A `RegularStair` can be specified by a single base-data polygon, whose attribute table defines number of steps, step height, step width, and the information which edges are to be moved inwards. Each stair defines a material for its surface and, optionally, an additional material for its vertical faces.

Walls

The class `Wall` represents freestanding walls and retaining walls. Many walls contain a separate top plate, which might differ from the rest of the wall by another material and by slightly exceeding the wall footprint. Therefore, the `Wall` class provides optional parameters to specify a separate top plate. Figure 15 illustrates the wall construction. Walls are represented by polygons or polylines of the base data and their attributes. By buffering lines to a width that is specified by the attribute table, a footprint polygon of the wall is obtained (Figure 15, left). The 3D shape is now determined by the extrusion parameters shown in Figure 15, center. Finally, materials are assigned to each side as illustrated in Figure 15, right.

Kerbs

The class `Kerb` represents boundary objects between two `GroundAreas` with a separately specified appearance, e.g., by an own kind of stone or by a different height (Figure 16). This corresponds to typical construction methods for, e.g., pavements and roads, where a kerbstone is necessary to stabilize the construction. A `Kerb` is defined by a polyline of the base-data set.

For the frequent case of two `Kerbs` along a single boundary line, both `Kerbs` should appear side by side and should not overlap. For this, each `Kerb` must specify one of the two adjacent

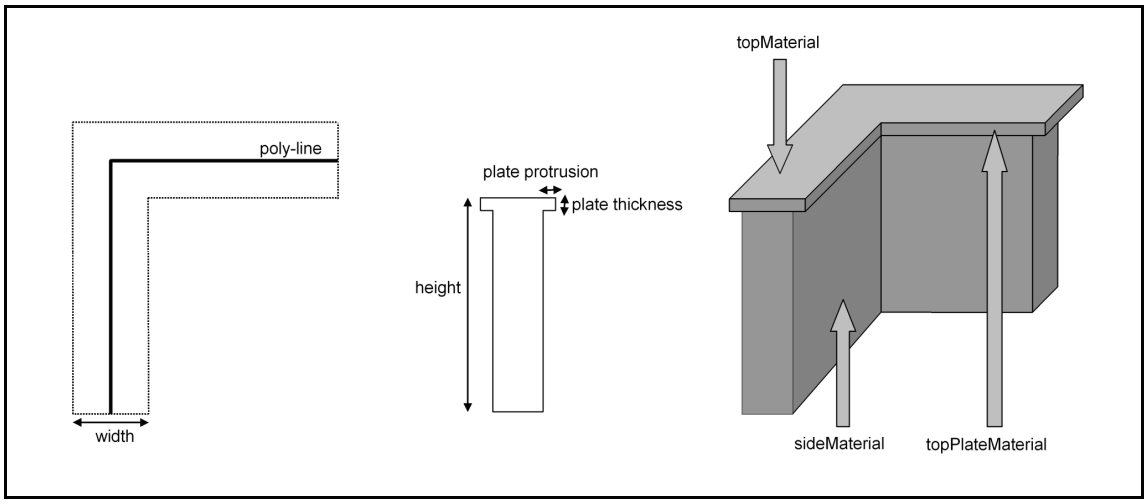


Figure 15: Construction of walls: Specification of the footprint (left); 3D extrusion parameters (center); materials for each surface (right).

`GroundAreas` as the *parent area* to which the `Kerb` belongs. This defines the order in which the `Kerbs` appear between the two adjacent `GroundAreas`. The parent area, i.e., its underlying base vector object, is referenced in the attribute table of the `Kerb`'s underlying polyline. To allow for referencing, the base-data vector objects must provide unique ID values in their attribute tables. In the rare case of three or more parallel boundary objects along a single boundary line, the center objects must be represented by additional `GroundAreas`.

The footprint of a `Kerb`'s 3D representation is obtained by buffering the `Kerb`'s underlying poly-line to a certain width. Since the kerb shall appear completely as a part of its parent area, buffering is performed only in the direction pointing inside the parent area. Finally, the 3D shape is obtained by extruding the footprint to a certain height. Buffering width, extrusion height, and material are specified via the attribute table.

Barriers

The class `Barrier` represents boundary objects such as fences or balustrades. In contrast to walls, barriers do not cover any area on the ground, but their footprint is only line-shaped. Each `Barrier` object is defined by a polyline in the base-data set which specifies the height of the `Barrier` in its attribute table. In the current implementation, `Barriers` are represented by extruded lines covered with partially transparent texture images. For instance, a balustrade can be modeled using a photograph of a balustrade and setting all image pixels to fully transparent at which the background of the balustrade is visible. If the standard type is not sufficient, the `Barrier` can be refined by an external 3D modeling tool.

3.2.7 Extension of the Smart Terrain Model Concept

The smart terrain concept provides means for later integration of additional features without changing the class model or existing smart terrain model specifications:

- *Shader materials*: In the current implementation, only colors and textures are supported. Homogenous materials such as asphalt or gravel can be easily specified by repeated photo textures. For heterogeneous structures such as stone mosaics, however, it is difficult or impossible to fit a repeated photo texture exactly onto an area in such a way that the surface pattern exactly follows the boundary of the area. Therefore, a third material type *shader material* supports integration of specific texture generation

Figure 16: Examples of kerbs along the border lines of streets and sidewalks.

algorithms. A shader material specifies the name of an XML file that defines name and configuration data of a specific texture generator. There are a small number of typical pavement types that cover a large amount of all streets and paths. This motivates the development of procedural-texture generators for these types, e.g., a texture generator that arranges and blends different stone images to obtain an individually fit image of a cobbled street for a given area. Procedural-texture generators can also add geometric surface detail using relief textures [Oliveira 2000].

- *Barrier types*: A barrier is currently represented by one or more textured quadrilaterals. It is more desirable to obtain true 3D geometry by procedurally constructing it from geometric parts that are automatically arranged along the underlying polyline. Algorithms for automatic creation of different barrier types are still to be investigated. They can later be integrated, identified by barrier types that reference XML files with type specifications.
- *Kerb types*: In the current implementation, the appearance of kerbs must be specified by assigning materials. An optional kerb type identifier allows for later integration of kerb-specific generator algorithms for geometry and texture of a kerb. In this way, for instance, a single stone row can be generated that follows the kerb line, and the surface can be modified by a height profile.

3.3 Navigation in 3D City Models

One of the most important aspects of human-computer interaction in virtual environments is the effectiveness and intuition of the underlying navigation tools controlled by the user. Particularly in geovirtual environments, navigation represents a key functionality because ‘the acquisition of spatial knowledge, essential for wayfinding, is primarily based on direct environmental experience, which is usually gained via movement’ [Gale et al. 1990].

Navigation and real-time rendering techniques are strongly related to each other in two respects:

- *Usability*: Effective navigation tools are essential to make real-time rendering techniques usable. If the navigation fails, an application is not used any longer, independent of the underlying rendering technique. Navigation fails, for instance, if the user becomes disoriented in the virtual space, is unable to control the virtual camera to a desired viewing perspective, or is hampered by slow or intricate navigation controls.

- *Technical Requirements*: Freedom of movement provided to the user affects the demands on a real-time rendering technique. For instance:
  - If navigation is restricted to slow movements, the corresponding small changes in the resulting images can be exploited to distribute the computational effort over several subsequent frames [Schaufler 1995].
  - If navigation does only allow for movement on the ground, restricted visibility can be exploited to optimize rendering by occlusion culling algorithms [Wimmer and Bittner 2005].
  - If the navigation enforces a permanent minimum flying altitude, textures of a city model can be created at a lower resolution [Früh et al. 2004].
  - If the user can only navigate along predefined paths and can neither alter the viewing direction nor modify the environment, real-time rendering is not required at all, because everything can be provided as precomputed animations.

This section provides an overview of different ways to provide user navigation in city models. The variety of different existing navigation techniques motivates the development of a rendering technique that does not depend on restricting assumptions on the navigation.

3.3.1 Navigation Techniques

A *navigation technique* maps user inputs received from input devices such as mouse or keyboard to parameters of a virtual camera. Navigation techniques have to achieve conflicting goals: On the one hand, they must provide sufficient freedom of movement to allow a user to achieve all desired viewing perspectives. On the other hand, a navigation technique should be intuitive and, therefore, must not be too complicated to grasp and handle from the user's perspective. Controlling a virtual camera means to control at least five degrees of freedom simultaneously, taking only camera position and viewing direction into account. Therefore, a navigation technique must always provide an appropriate compromise between both goals. Correspondingly, a single navigation technique that is optimal for all application scenarios does not exist. 'The optimal navigation method depends on the exact nature of the task' [Ware 2000]. In addition, different user skills and experiences as well as individual preferences of users are relevant [Chen et al. 2000].

Examples of navigation techniques that are useful for city models are the following:

- *Pedestrian navigation* supports exploration of a virtual environment from the point of view of a virtual avatar. Forward, backward, and sideward movement is performed by keys, while the head rotation along with the walking direction is controlled by mouse movement. This technique has become quite popular due to its frequent use in computer games.
- *Flyer navigation* provides control of a virtual flying vehicle. In the variant provided by the LandXplorer platform, the user manipulates the speed of the forward movement by dragging the mouse along the y-axis. Negative speeds are possible to support backward movement. The x-axis controls direction and speed of the rotation around the vertical axis in the virtual space. In addition, the flying vehicle allows for changing the height or for tilting the viewing direction via keys. Flyer navigation is frequently used in geovirtual environments. The metaphor of the virtual flying vehicle also exists in various different variants, e.g., in the form of a flying saucer [Fuhrmann and McEachren 2001].
- *Trackball navigation* rotates the virtual camera around a focus point. For this, the mouse movement is mapped to a movement of the virtual camera along the surface of an invisible sphere. The focus point is determined by shooting a ray from the camera

a) b) c)

Figure 17: Image sequence to illustrate the adjustment strategy. a) No focus point is defined. The trackball is unable to work. b) The perspective has been modified by the adjustment strategy to obtain a valid focus point. c) The trackball can now be used for leaving the pedestrian perspective.

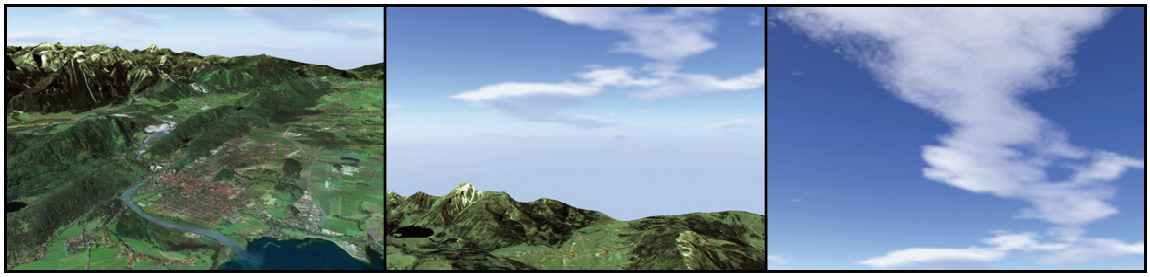
position towards the viewing direction and choosing the first intersection point with the environment.

- *Zoom navigation* is also based on a focus point and allows for controlling the distance between focus point and virtual camera, e.g., by mouse wheel or mouse movement.
- *Focus-point selection* enhances trackball and zoom navigation by the ability to select a new focus point. The technique rotates the camera in a way that the focus point moves smoothly to the center of the screen during a short animation period.
- *Panning navigation* moves the camera parallel to the horizontal plane at a constant height. By dragging an arbitrary point on the ground along the screen, the camera is moved in a way that the currently dragged point follows the mouse pointer.

Pedestrian and Flyer navigation are well suited for presentations and entertainment applications. A combination of trackball and zoom navigation is commonly used for the investigation of single objects. Extended by a focus-point selection, it becomes also suitable for the exploration of large-scale virtual environments.

In addition to the examples mentioned above, which map user input quite directly to parameters of the virtual camera, there are also different semi-automatic navigation techniques in which user navigation is interpreted to trigger on-demand camera animations:

- *Click-and-fly navigation* provides a directed flight via clicking on a distant visible target point by mouse [Mackinlay et al. 1990].
- *Landmark selection* invokes on-demand camera paths from the current camera perspective to certain pre-defined viewpoints [Helbing et al. 2000, Salomon et al. 2003].
- *Path-drawing* allows the user to specify camera paths spontaneously by drawing strokes onto the screen [Igarashi et al. 1998]. The strokes are projected back into 3D space, and a smooth camera path is invoked along the resulting curve.
- *Semantic-based navigation* [Döllner et al. 2005(ii)] comprises a collection of navigation techniques for city models which enhance the path-drawing approach. These techniques initiate camera paths based on user gestures taking into account the semantics of currently visible city model entities. For example, drawing a stroke along a street followed by clicking onto a building is evaluated by flying along the street and finally rotating the camera towards the selected building.



a) High. b) Low. c) Insufficient.

Figure 18: Sequence of snapshots with decreasing amount of orientation-supporting information in the image.

There is a large number of other useful navigation techniques, see Darken and Sibert [1993], Hand [1997], and Tan et al. [2001] for an overview. For efficient and intuitive navigation in a virtual city model, it is useful to provide multiple navigation techniques within a single application. The *smart navigation* approach [Buchholz et al. 2005] addresses two related problems:

- *Conflicts between different navigation techniques*: After switching between navigation techniques, the newly active navigation technique might be confronted with a view specification for which it cannot work. For example, when switching from pedestrian navigation to trackball navigation, the focus point might be undefined because only the sky is visible in the center of the screen (Figure 17a). In this case, an *adjustment strategy* invokes a short camera animation from the current view to a similar view (Figure 17b) for which the activated technique can work (Figure 17c).
- *Unsteady and discontinuous camera movements* are disturbing and hinder effective and comfortable user navigation. Ensuring smooth camera motion in all situations constitutes a significant portion of the implementation effort of navigation techniques. Therefore, a *physical motion system* is introduced that can be used by different navigation techniques. It treats the camera as an inertial 3D object that is only controlled indirectly via a virtual spring; the user controls only the spring's origin, whereas the camera itself follows to the desired position with a minimal time delay. Unsteady movements of the spring's origin are filtered due to the indirect power transmission and the camera's inertia.

3.3.2 Disorientation and Constrained Navigation

Disorientation poses a core problem for the usability of virtual environments: Users frequently get lost in virtual space, i.e., they lose their sense of position, relations, and orientation. As Fuhrmann and MacEachren [2001] point out, 'core problems for users of these desktop GeoVEs are to navigate through, and remain oriented in, the display space and to relate that display space to the geographic space it depicts.' Particularly, the 'end-of-world' problem is critical, i.e., if a user exceeds the virtual world's boundaries and reaches an undefined area.

Disorientation affects both inexperienced and experienced users. The former frequently need to restart or cancel ongoing activities, for the latter disorientation hampers effective work in virtual environments. Burtnyk et al. [2002] describes further problems such as 'a user may […] view the model from awkward angles that present it in poor light, miss seeing important features, experience frustration at controlling their navigation, etc.'

To address these problems, different approaches were proposed to systematically constrain navigation and to guide the user during navigation:

- In the *river analogy* [Galyean 1995], the user is guided by a predefined path and controls only the gaze direction and slight deviations away from the path.
- *Guide manifolds* [Hanson and Wernert 1997] generalize this concept to arbitrary 2D manifold surfaces within a 3D space. Using 2D input devices the user moves along a designer-provided surface. Viewing direction and up-vector of the camera are controlled by predefined guide fields according to the camera position. This concept has been extended later to *collaborative virtual environments*, i.e., virtual environments for multiple collaborating users [Wernert and Hanson 1999]. In this concept, the user guidance is affected by the navigation of another user, which holds a leader role. For instance, the navigation can be forced to keep within a certain distance from the leader.
- The approach used in *StyleCam* [Burtnyk et al. 2002], an authoring tool for camera animations, allows for seamless transition between the camera's spatial-control and its temporal-control of animation. Similar to the guide manifold approach, the user can move the camera along designer-provided surfaces. Starting from the boundary of such a surface, the user can invoke prepared camera animations to another surface.
- *Automatic view adjustment* presented by Kiss and Nijhold [2003] alters the viewing direction based on the terrain slope and some pre-defined objects of interest around the camera position. Each visible object of interest is shortly suggested to the user by guiding the view slightly towards it. The system also ensures that the user is always aware of obstacles that prevent him or her from moving.
- *Artificial force fields* [Xiao and Hubbold 1998] handle collisions between the camera and the virtual environment. To avoid undesirable blocking of user movement due to collisions, the user is directed parallel to the surface of an object if it comes to close.
- The *maintenance strategies,* which form a part of the smart navigation approach [Buchholz et al. 2005], extend existing navigation techniques such as flyer or pedestrian navigation by the ability to avoid disorienting views. A view is considered to be *disorienting* if it does not provide sufficient significant information for the user to keep oriented within the virtual environment. The principle is illustrated in the snapshot sequence in Figure 18 for the example of flying over a virtual terrain. Here, the visibility of the terrain is essential for orientation. During the forward movement, the amount of screen pixels at which terrain is visible is permanently decreasing. Finally, only sky is visible and it is impossible to identify the camera position relative to the terrain, particularly because forward movement does not change the image anymore. The maintenance strategy becomes active before a disorienting view occurs and guides the user away from the disorienting view.

3.4 Selected Applications of City Model Visualization

This section describes application examples of real-time city model visualization and proposes improvements based on multiresolution texture atlases. All examples have been developed based on the LandXplorer [Döllner and Kersting 2000] geovisualization platform, whose city model support constitutes a practical result of this work.

3.4.1 LandXplorer CityGML Tool

The LandXplorer CityGML tool is a freely available software for interactive visualization, editing, and conversion of city models in CityGML format (Figure 19a). It provides different combinations of navigation techniques for interactive exploration:

- Panning and Zooming
- Examine Navigation
- Walkthrough Navigation
- Flyer Navigation

These navigation techniques are used in all application examples described in this section. The implementation of the navigation techniques is based on the navigation framework introduced by Buchholz et al. [2005]. In addition to spatial navigation, the CityGML tool supports browsing the semantic structure of a CityGML model via a tree view, in which semantically relevant objects such as buildings, terrain, and land use objects can be selected and modified. The tool as well as a complete list of the currently provided features is available at [CityGMLTool].

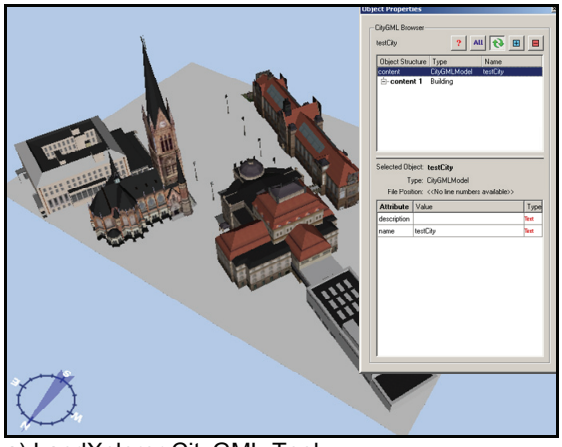
The implementation of the multiresolution texture atlases approach proposed in this work has currently not yet been integrated in the tool but is planned to be in future extended versions. In this way, the following two restrictions can be overcome:

1. *Restriction of model textures*: Currently, city models can only be rendered efficiently if the amount of specified textures in the model is sufficiently small.
2. *Restriction of lighting-texture quality*: The viewer provides automatic creation of lighting textures to improve the rendering quality compared to standard real-time lighting (Figure 20). Currently, these textures are rendered in a conventional way, so that the quality of the lighting is restricted by the available texture resolution. In addition, the required texture memory restricts its application to small models.

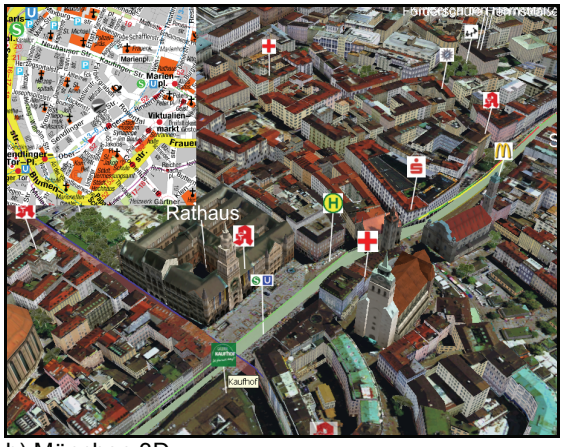
3.4.2 Virtual Munich

The software product *München 3D*, which aims at being a '3D city map', forms a combination of entertainment product and 3D geoinformation system for tourism and private use (Figure 19b). It is based on the viewing software *LandXplorer Xpress*, implemented by 3D Geo GmbH. München 3D provides an interactive visualization of the inner city of Munich, in which the user can freely navigate. A similar software is available for the inner city of Berlin (Figure 19c). The available navigation techniques are the same as in the CityGML tool. Besides navigating through the model, the software supports

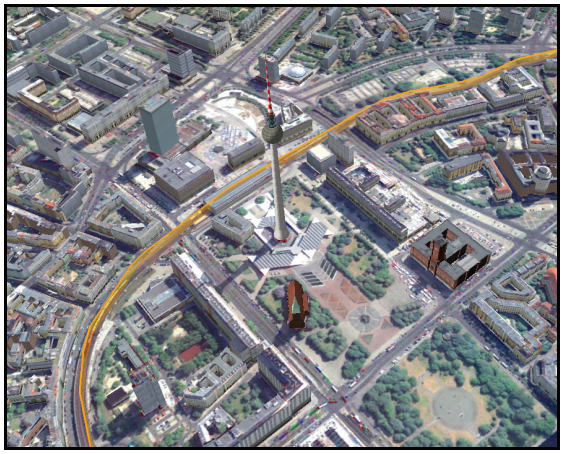
- Initiating automatic camera flights to selected targets, e.g., points of interest such as 'Frauenkirche' or 'Feldherrenhalle' or stations of both underground and interurban trains.
- Measuring distances within the model.
- Display a 2D city map that indicates the current position within the 3D model.
- Display text labels to show street names, lines on the terrain surface to show the public-transport network, and map symbols to indicate positions of facilities such as banks, drugstores, and restaurants.



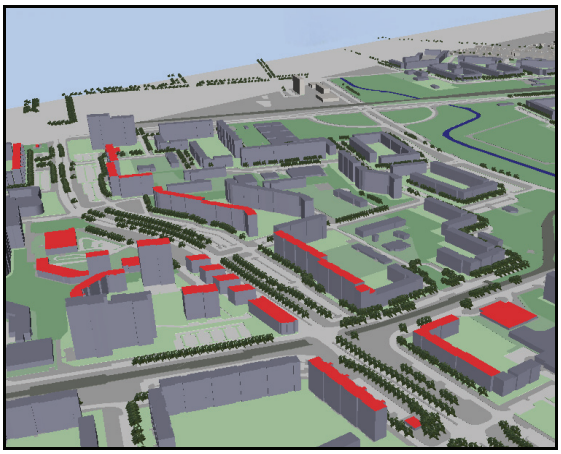
a) LandXplorer CityGML-Tool.



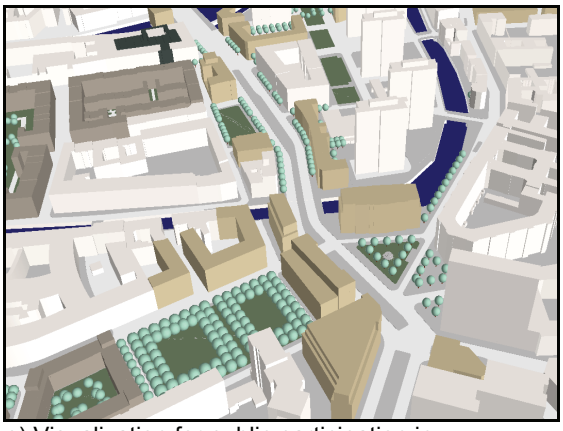
b) München 3D.



c) Berlin 3D.



d) Decision-support tool for urban redevelopment.



e) Visualization for public participation in city planning

f) 'Land of Ideas' marketing movies.

Figure 19: Applications of city model visualization.

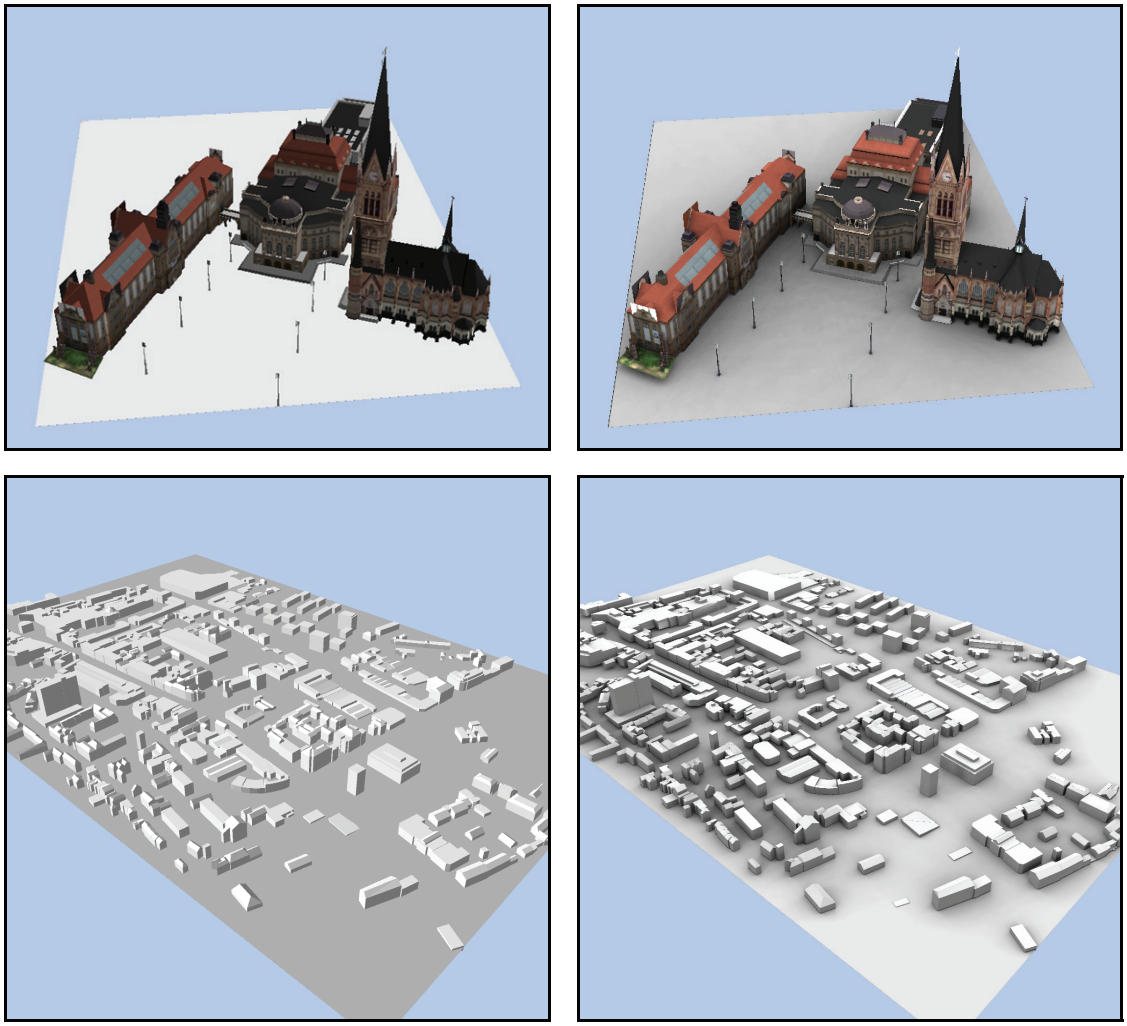


Figure 20: Comparison of standard real-time lighting (left) and precomputed lighting textures (right).

The underlying city model consists of:

- A grid terrain model.
- A satellite image at a resolution of 15 meters per pixel.
- An aerial image at a resolution of 20 centimeters per pixel that locally refines the satellite image in the inner part of the city.
- A set of 55,000 textured block building models, specified by a shapefile that defines footprint polygons, heights of buildings, and references to facade image files.
- 63 more detailed, manually created models of important landmark buildings.
- Shapefiles with attributed lines representing the public-transport networks and streets.
- Shapefiles with attributed points providing positions and descriptions of all displayable point symbols and flying targets.

The terrain texture is not only applied to the terrain but also to roofs of buildings. For this, an extension of the terrain rendering approach of Baumann [2000] has been developed that is explained in Section 4.8.2.

The underlying city model of the München 3D software represents a primary motivation for the applications of multiresolution texture atlases. The contained amount of texture data represented a main problem for the application development. The model contains about 4,000 different facade textures. Therefore, it was necessary to reduce the texture resolution of facade textures by uniformly downscaling them, so that all textures had a fixed width of 128 pixels. The user navigation had to be restricted to ensure that the virtual camera always keeps a certain minimum distance from the buildings to avoid the low texture resolution from becoming too obvious. Nevertheless, the textures were still a primary performance bottleneck. Furthermore, there are still many buildings that are decorated with shared textures instead of original facade photographs. Consequently, RSS GmbH, who provided the data set for München 3D, has been permanently improving the model, so that the number of available textures is steadily increasing.

3.4.3 Decision Support Tool for Urban Redevelopment

Since 2002, the Berlin Senate Department of Urban Development has been constructing and maintaining an official Berlin city model. This model is used to create perspective images of the city, in which the appearance of buildings is utilized to reflect certain thematic properties such as year of construction, vacancy, state of repair, number of floors, or owner of a building. The images are used as a communication and decision support for discussions between urban planners and political decision makers. The images are manually designed and created using CAD tools, and there is no possibility to interact or directly reconfigure the information display.

As supposed by the senate department, it is more desirable to provide thematic views within an interactive city model visualization, so that different kinds of thematic building data can be explored immediately. Therefore, an application prototype has been developed based on a part of the Berlin city model (Figure 19d). The primary goal was to get an impression whether an interactive tool is helpful for this application.

The prototype was created and tested within the context of an urban planning program called ‘Stadtumbau Ost’ [StadtumbauOst]. The primary aim of this program is to reduce vacancy in East German cities to make them more attractive, support the real estate market, and to avoid the fragmentation of cities. In a part of this project, called ‘Ahrensfelder Terrassen’, a specific method was applied in which buildings were reduced on a per-floor basis while the remaining floors were renovated.

The tool provides an interactive visualization, which communicates thematic building data as well as planning proposals such as to reduce a certain building by three floors and remove another one completely. For this, floors that were scheduled for removal were highlighted in transparent blue and single floors were indicated by separation lines. The underlying city model consisted of block building models, thematic building information, and a 3D representation of the ground geometry. After testing the tool in a discussion, the senate department confirmed that an interactive visualization does provide advantages compared to static images.

The underlying rendering framework [Buchholz and Döllner 2005(i)] works for large numbers of buildings in the range of 100,000 and supports additional graphics attributes for reflecting thematic data such as color and style of lines along the primary edges of building shapes [Döllner and Walther 2003]. Photographic facade textures for the 3D model were neither available nor relevant in this application.

Nevertheless, the ability to apply large amounts of texture data offers a potential for future improvements:

1. The standard lighting model of OpenGL [Segal and Akeley 2004] works very efficiently but provides only poor spatial impression. Individual precomputed lighting textures can significantly improve the visual quality as shown in Figure 20.
2. Using basic graphics attributes such as roof color and facade color provides only a restricted flexibility to communicate thematic information. Creating individual high-resolution building textures based on thematic information can strongly increase this flexibility. For instance, the facades can contain text elements or symbolic information.

3.4.4 Visualization Tool for Public Participation

In September 2003, a public discussion took place, in which two planning proposals for the development of the region around Spittelmarkt in Berlin were compared. In collaboration with the Berlin Senate Department of Urban Development, based on the official Berlin city model, both planning proposals were interactively visualized to support the discussion. The visualization contained block buildings, generalized tree representations, and a detailed geometric representation of the ground. The 3D application provided navigation at overview and pedestrian perspectives and allowed for specifying and playing camera flights through the model (Figure 19e). During the visualization, the user could cross-fade between both planning proposals. Facade textures were not provided for the visualization.

Following the presentation, discussions with participants led to the conclusion that the visualization was helpful when viewing from an overview perspective. The large uniformly colored surfaces due to missing textures, however, were a significant problem, particularly for pedestrian perspectives. Therefore, at least for the ground and for facades of already existing buildings, detail textures should be provided in visualizations for public participation.

3.4.5 Land-of-Ideas Movies

The initiative '*Land of Ideas*' [LDI] was founded on the occasion of the Soccer World Championship 2006 in Berlin. The goal of the initiative is to point out significant German achievement in science and arts. As a part of the program, six sculptures were created and exposed in the inner city of Berlin. The sculptures were also available as 3D models. Using these 3D models and a city model of Berlin, different videos were created that show the models and their positions within Berlin (Figure 19f). The slightly reflecting sculpture surfaces have been rendered using environment mapping [Möller and Haines 1999]. The videos are available at the Land of Ideas web site. To create the videos, the sculptures were integrated into an interactive visualization, which made it possible to create and test different camera animations spontaneously.

Chapter 4

MULTIRESOLUTION TEXTURE ATLASES

This chapter introduces multiresolution texture atlases, an approach for real-time rendering of city models with high texture complexity in terms of number of textures and texture size. It describes the texture atlas tree, a data structure that forms the core of the approach, as well as the related construction process, real-time rendering algorithm, and memory management strategy. The efficiency of the approach is demonstrated in performance tests. Finally, additional methods are described to optimize the implementation.

4.1 Differences to Terrain Multiresolution Textures

In the following sections, the graphics representation of a city model is assumed to be given as a 3D scene, which represents the most common form for real-time rendering. The quadtree hierarchy for terrain textures described in Section 2.2.2 cannot be used for general 3D scenes. This is primarily due to two reasons:

1. *Number of textures*: For textured terrain models, the texture hierarchy is built for a single large texture. General 3D scenes, typically, contain a large number of separate textures.
2. *Texture distribution*: Terrain textures are usually orthogonally projected onto the terrain surface. Hence, for each rectangular region of the terrain, there is a corresponding rectangular texture region. Particularly, the texture workload is evenly distributed over the terrain, i.e., the amount of necessary texture data for two equally-sized regions of the terrain is always equal. This cannot be assumed for more general 3D scenes such as city models (Figure 21). Generally, the texture workload might be arbitrarily distributed in the scene. For instance, one area of a city model might contain several large textured buildings, while another area of the same size might contain no buildings at all.

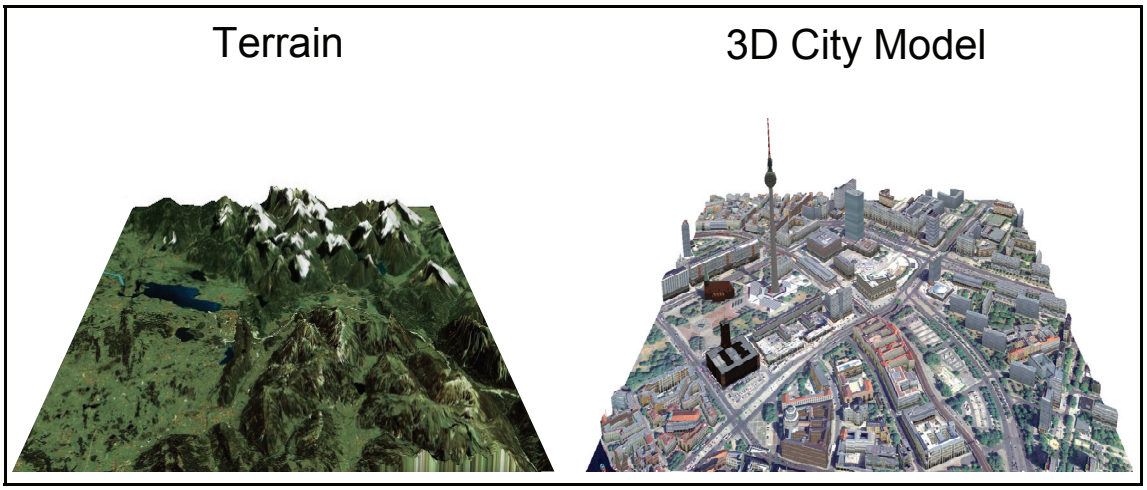


Figure 21: Differences between terrain textures and textures of city models: A terrain (left) is covered by a single texture that is orthogonally projected, so that the texture workload is equally distributed over the terrain area. In a city model (right), the texture workload might be arbitrarily distributed due to different building heights, building density, and building texture resolution.

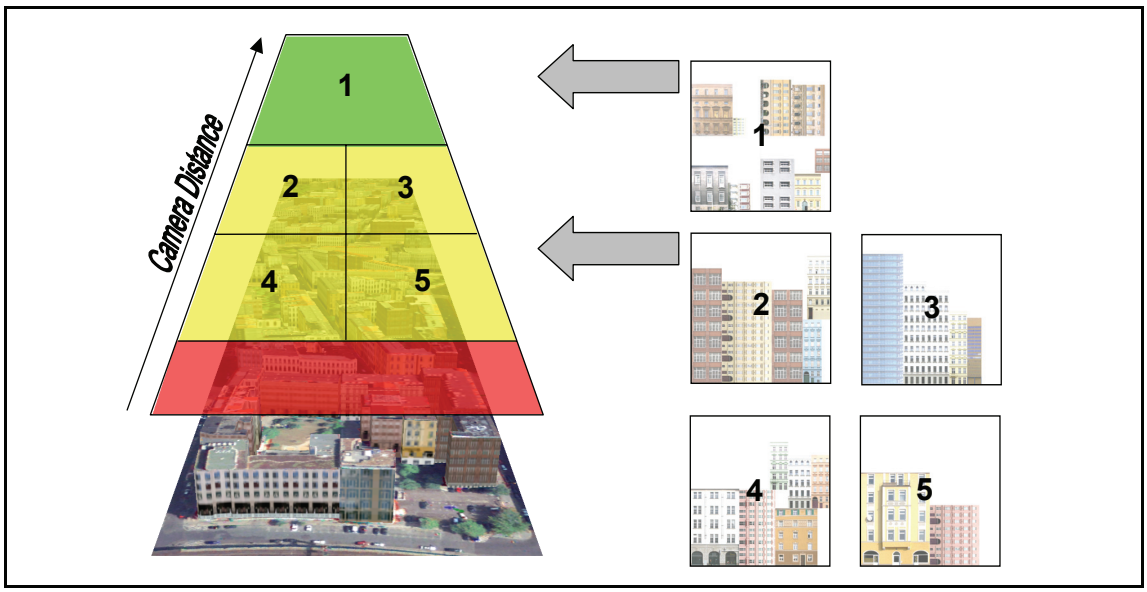
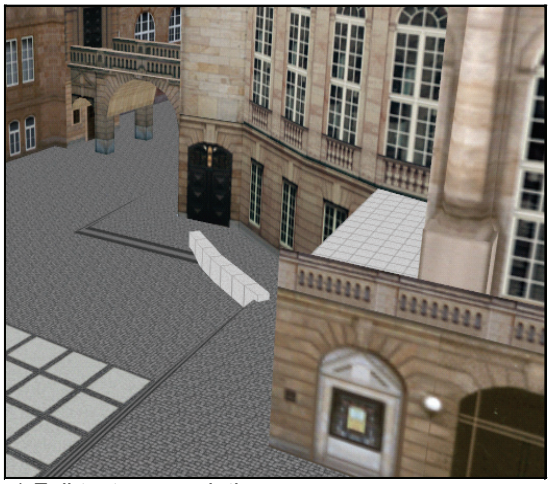


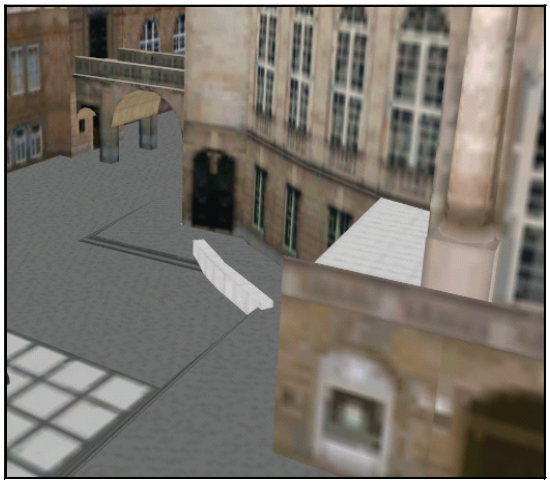
Figure 22: Hierarchical texture atlas composition: Original textures are used for the near area (red), Four texture atlases are used for the center area (yellow), and a single combined texture atlas is used for the far area (green).

The first problem is addressed by a hierarchical composition of several scene textures into texture atlases. The principle is illustrated in Figure 22:

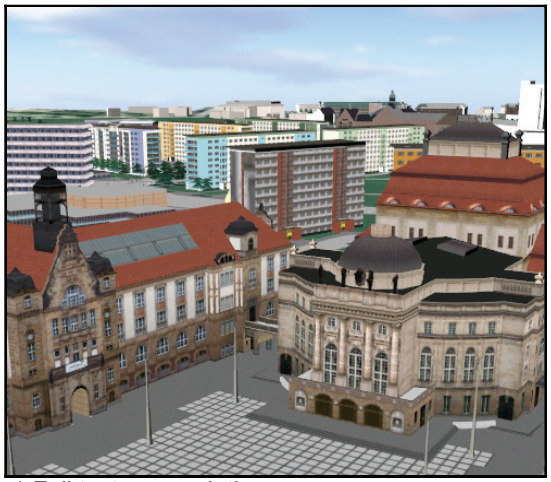
- *Red Area*: For scene parts very close to the camera, the original scene textures are used at full resolution. This does not cause performance problems, because only a limited number of textures can be close to the camera simultaneously.
- *Yellow Areas*: For scene parts whose distance from the camera exceeds a predefined distance threshold d_{min}, textures are combined into texture atlases. The resolution, in which each texture is represented in a texture atlas is chosen appropriately for



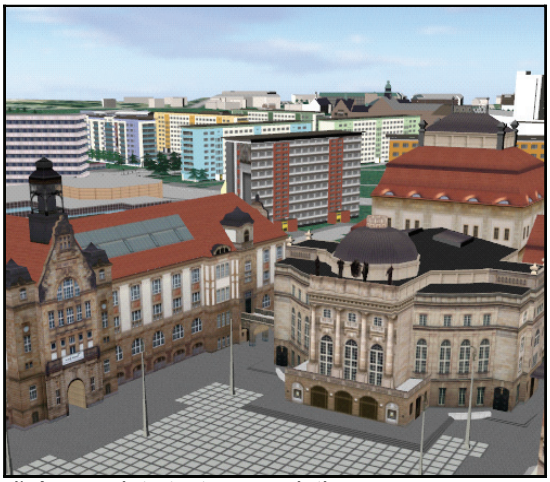
a) Full texture resolution.



b) Insufficient texture resolution.



c) Full texture resolution.



d) Appropriate texture resolution.

Figure 23: Full texture resolution is only required for close views: a) Close view rendered using full texture resolution. b) Close view rendered using inappropriately low texture resolution. c) Distant view using full texture resolution. d) Distant view using the same texture resolution as in b).

distances $d > d_{min}$. In this way, a single texture atlas can replace multiple textures, even if the original textures are too large to be combined.

- *Green Area*: For increasing camera distance, the required texture resolution decreases further. Hence, more and more textures can be replaced by a single texture atlas.

In practice, there are, usually, more texture resolution levels than shown in Figure 22. The hierarchical composition of textures is continued with increasing distance until all textures of the whole scene fit into a single texture atlas.

Figure 23 demonstrates the principle for an example. In Figure 23a), all visible scene parts are closer than d_{min}, so that the resolution of the texture atlases is insufficient as shown in Figure 23b). Therefore, such views are rendered using full texture resolution. For the perspective in Figure 23c), more textures are visible, but no scene part is closer than d_{min}. Hence, the whole scene can be rendered using a small set of texture atlases as shown in Figure 23d).

The second problem mentioned at the beginning of this section was the possibly uneven texture distribution in a 3D scene. The actual distribution of texture workload should be considered by the hierarchical subdivision of the scene geometry. Therefore, instead of a regular 2D quadtree subdivision, an irregular 3D subdivision based on texture workload is used as described later in greater detail.

4.2 Input Data and Parameters

The input data from which a texture atlas tree is built consists of an arbitrary 3D scene, given in the form of a set G of triangles, a set T of related textures, and an assignment of each triangle in G to a texture in T. Each triangle vertex is given by at least a position vector and a 2D texture coordinate vector. In practice, each vertex usually contains additional vertex attributes such as normals or colors, but these are not relevant for the description. Usually, each texture is specified in the form of a 24-Bit-RGB image. If a scene contains textures with alpha channel, i.e., 32-Bit-RGBA images, these textures and the related triangles are handled in a separate texture atlas tree that is simultaneously used during rendering. The following parameters are required for the construction of a texture atlas tree:

1. A *minimum distance* parameter $d_{min} \in \mathbb{R}^+$. d_{min} should be chosen small enough so that for each possible camera position within the scene, the geometry within a distance of d_{min} around the camera can be efficiently rendered with original scene textures. The effect of different choices for d_{min} is explained in greater detail in Section 4.5.2.
2. A desired *pixel-per-texel ratio* $\tau \in \mathbb{R}^+$, which is usually set to 1.0. The pixel-per-texel ratio provides an explicit control to trade image quality for performance if desired. It specifies the maximum area in image space to which a single texture pixel, called *texel*, is allowed to be stretched during rendering. A higher pixel-per-texel ratio has the effect that textures at lower resolutions are used during rendering. In contrast to the other input parameters, the pixel-per-texel ratio can be changed later during rendering, as explained in Section 4.5.2.
3. A fixed *texture atlas size* $A \in \mathbb{N}$, which indicates width and height of the generated squared texture atlases. The texture atlas size should be chosen as large as possible provided that texture atlases of this size can be efficiently loaded and rendered at runtime without causing noticeable delays. In the experiments, $A = 512$ turned out to be an appropriate choice.

4.3 Data Structure

The basic structure of the texture atlas tree is illustrated in Figure 24. A texture atlas tree defines a hierarchical subdivision of the initially given 3D scene and a corresponding hierarchy of texture atlases. The node data structure is summarized in Figure 25. To simplify the description, inner nodes and leaf nodes are combined in a single node data structure.

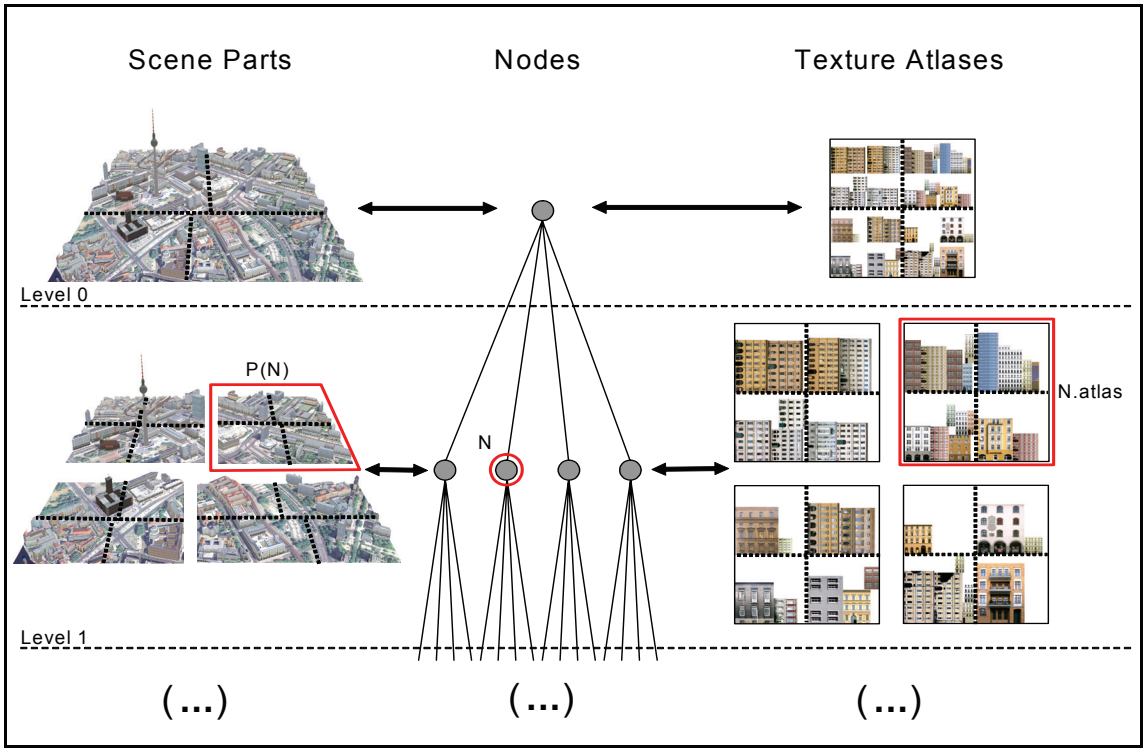


Figure 24: Structure of a texture atlas tree. Each node *N* (center) represents a scene part *P*(*N*) (left) and provides a texture atlas (right) for *P*(*N*).

```
struct Node {

   // Bounding box of the related scene part P(N)
   BoundingBox bounds;

   // Texture atlas for P(N)
   Texture atlas;

   // Matrix for atlas texture coordinate transformation
   Matrix2x2 textureMatrix;

   // Minimum distance for which the texture atlas is appropriate
   float minDistance;

   // Contains 4 child nodes or is empty (for leaf nodes)
   Array<Node> children;

   // --- The following fields are used only for leaf nodes ---

   // Triangles of P(N)
   Array<Vector3> triangles;

   // Full-resolution textures and texture coordinates
   TextureData fullResolutionTextures;

   // Texture coordinates for texture atlas
   Array<Vector2> atlasTextureCoordinates;
};
```

Figure 25: Pseudo-code summary of the node data structure.

Each node **N** represents a part **P(N)** of the scene geometry and stores the following values:

N.bounds: The bounding box of the related scene part **P(N)**.

N.atlas: A single texture atlas containing downsampled versions of all scene textures that are mapped onto triangles of **P(N)**. The atlas size is specified by the input parameter *A*, i.e., all texture atlases are of equal size.

N.textureMatrix: A texture matrix that transforms the texture coordinates of **P(N)** for texture atlases. Without this texture matrix, separate texture coordinates would have to be stored for the texture atlases at each level of the tree. Purpose and computation of the texture matrix are explained in greater detail in Section 4.5.4.

N.minDistance: A floating point value indicating the minimum required distance between camera and the node's bounding box **N.bounds** to ensure that the node's texture atlas resolution is sufficient.

Geometry and related texture coordinates are stored in the leaf nodes. That is, a leaf node **L** explicitly stores the related scene part **P(L)** while an inner node **N** contains **P(N)** only indirectly via the leaf nodes of the subtree rooted at **N**. Therefore, a leaf node **L** contains the following additional fields:

L.triangles: The triangles of the related scene part **P(L)**, represented by an array of 3D floating point vectors. Each triangle is represented by a triplet of three subsequent vectors.

L.fullResolutionTextures: The **TextureData** type is used here to summarize all data that define the full-resolution textures for all triangles of **P(L)**. This includes the full-resolution texture images, the related texture coordinates for each triangle, and the assignment, which texture is applied to each triangle. This information is directly taken from the input data.

L.atlasTextureCoordinates: An array of alternative texture coordinates that determine the texture mapping when using texture atlases. If according to the input data a triangle *t* is covered by a region *R* of an original texture, the atlas texture coordinates of *t* indicate the region in **L.atlas** that contains a downscaled copy of *R*. These texture coordinates are computed during the texture atlas computation in the preprocessing step as explained in Section 4.5.4.

4.4 Rendering Algorithm

The rendering algorithm for the texture atlas tree is similar to the one for textured terrains outlined in Section 2.2.3. For the moment, the tree is assumed to be residing completely in the main memory. The necessary changes for dynamic loading and deletion of textures are explained in Section 4.6. The main parts of the rendering algorithm are summarized in Figure 26.

For each traversed node **N**, the bounding box **N.bounds** is checked against the current view frustum of the virtual camera. If **N.bounds** is completely outside the view frustum, the node is skipped because it is currently invisible. Otherwise, the distance *d* between the camera and **N.bounds** is computed because it provides a lower bound for the distance of each triangle of **P(N)** from the camera. In the case $d \geq$ **N.minDistance**, the node **N** is rendered.

```
void renderTree() {

   // Depth-first traversal of the tree
   Stack<Node> nodeStack;
   nodeStack.push(rootNode);
   while (!nodeStack.isEmpty()) {

      Node N = nodeStack.top();
      nodeStack.pop();

      // Skip invisible nodes
      if (outsideViewFrustum(N.bounds)) {
         continue;
      }

      // Compute lower bound for the camera distance of P(N)
      float d = computeCameraDistance(N.bounds);

      // Determine whether to use N.atlas for rendering
      bool useNodeAtlas = (d >= N.minDistance);

      if (useNodeAtlas) {
         // The resolution of N.atlas is sufficent
         renderNode(N);
      }
      else {
         // The resolution of N.atlas is not sufficient.

         if (isLeaf(N)) {
            // The highest atlas resolution is still insufficient.
            // => Render using full-resolution textures.
            renderScenePart(N);
         }
         else {
            // Continue traversal
            pushChildren(nodeStack, N);
         }
      }
   }
}

// Render node using its texture atlas
void renderNode(Node N) {

   // Apply texture and texture matrix
   activateTexture(N.atlas, N.textureMatrix);

   // Find leafs of the subtree with root N
   // by a traversal of the subtree.
   Array<Node> leafs = findLeafs(N);

   for_each Node L in leafs {

      // Render geometry of all visible nodes
      if (!outsideViewFrustum(L.bounds))
         renderTriangles(L.triangles, L.atlasTextureCoordinates);
      }
   }

   // revert last texture activation
   deactivateTexture();
}
```

Figure 26: Pseudo-code of the texture atlas tree rendering algorithm.

Rendering a node means:

1. The texture **`N.atlas`** and the texture matrix **`N.textureMatrix`** are activated, so that they are applied to the subsequently rendered geometry.
2. The subtree rooted at **`N`** is traversed. For each leaf **`L`** of the subtree, the triangles **`L.triangles`** are rendered using **`L.atlasTextureCoordinates`** for texture mapping. Leaf nodes outside the view frustum are skipped.

If $d <$ **`N.minDistance`**, the resolution of **`N.atlas`** is not sufficient. Hence, if **`N`** is an inner node, the traversal is continued with the child nodes of **`N`**. If **`N`** is a leaf, the triangles of **`N`** are rendered using **`N.fullResolutionTextures`**, i.e., the full-resolution scene textures and the related texture coordinates. In this case, the texture coordinate matrix is set to the unit matrix.

4.5 Construction of a Texture Atlas Tree

This section describes an algorithm to construct a texture atlas tree. First, an overview of the construction process is given, followed by a description of the most important steps.

4.5.1 Overview

The preprocessing starts with a tree data structure consisting of a single root node that contains all triangles of the scene together with their related original texture coordinates. For each triangle, a reference is stored to the related scene texture. The tree structure is built by recursive node splitting:

1. For a given Node **`N`**, it is tried to create a single texture atlas that provides texture data for all triangles of **`N`** at appropriate resolution according to the minimum distance value d_{min}. This step is described in greater detail in Section 4.5.2.
2. If the atlas creation fails, **`N`** is split into four new child nodes. The triangles of **`N`** are shared among these child nodes. The subdivision algorithm for the triangles of a node to be split is described in Section 4.5.3. After a node split, step 1 and 2. are recursively repeated for the new child nodes until the atlas creation is successful. As explained in Section 4.5.3, the subdivision algorithm ensures that the recursive node splitting terminates.

If the atlas creation for a node is successful, the node, finally, becomes a leaf of the tree. In this case, the original scene data, i.e., **`N.triangles`** and **`N.fullResolutionTextures`** can be directly written into the new leaf node. In addition, the newly created texture atlas and its related texture coordinates are stored as **`N.atlas`** and **`N.atlasTextureCoordinates`**. The minimum distance value **`N.minDistance`** of each leaf node is set to the minimum distance value d_{min}.

At this point, the tree structure has been created and all leaf nodes have been filled with data. Next, the texture atlases for the inner nodes are computed bottom up. To obtain the texture atlas of an inner node, the four texture atlases of the child nodes are downsampled by a factor of 2 in both coordinate axes, and the results are combined in a single texture image. If d is a valid minimum distance value for a given texture atlas T and T' results from downsampling T by a factor of 2, $d' = 2d$ is a valid minimum distance value for T' (see Section 4.5.2). Therefore, an appropriate minimum distance value for an inner node is the maximum of all minimum distance values of its child nodes multiplied by 2. It should be noted that the minimum distance values of the child nodes are not necessarily equal because leaf nodes might be located at different tree levels.

The final step is the computation of the texture matrix **`N.textureMatrix`** for each node and the modification of the atlas texture coordinates **`N.atlasTextureCoordinates`** stored in the leaf nodes. Without this step, valid texture coordinates for all triangles would only be available for the texture atlases of the leaf nodes, but not for the combined atlases of inner nodes. The computation of the texture matrices for inner nodes and the corresponding modification of the leaf atlas texture coordinates are explained in greater detail in Section 4.5.4.

4.5.2 Node Atlas Creation

This subsection describes the creation of a texture atlas for the triangles of a node **`N`**. The atlas is created in such a way that for a given global parameter, d_{min}, it is guaranteed that no magnification occurs for any part of the texture atlas, provided that the camera has a distance of at least d_{min} from all triangles. d_{min} should be chosen with respect to two criteria:

- d_{min} should be set small enough, so that from each viewpoint in the scene all triangles within a distance of d_{min} can be rendered quickly with original textures. For instance, in a city model, the complexity of the surrounding area within a distance of 10 meters around any viewpoint is usually uncritical for rendering. For larger d_{min} values, leaf nodes contain more full-resolution textures. This leads to slow rendering performance for inappropriately large d_{min} values.
- d_{min} should not be chosen significantly smaller than necessary, because smaller values for d_{min} increase the size of the tree data structure, finally resulting in higher preprocessing time and a larger swap file.

The atlas is created in the predefined size A, e.g., 512 × 512 pixels. For the calculations, a perspective camera projection with a previously known field of view angle φ is assumed. The atlas creation is performed in three steps:

1. For each texture, calculate the minimum resolution at which the texture must be represented in the texture atlas to avoid magnification at a distance of d_{min}.
2. Based on step 1, compute for each texture the size of the required area within the texture atlas and the corresponding source area in texture space.
3. If possible, create a single texture atlas of the predefined size A for all textures, based on the results of step 2.

Texture Size Estimation

Given a texture and a set of triangles using the texture, the aim is to compute the minimum required texture resolution to avoid magnification at a camera distance of d_{min}. For this, the required values for texture width and texture height are separately computed for each triangle. The maxima of all calculated minimum widths respectively all calculated minimum heights yield the required texture resolution.

The texture coordinates and the vertex positions of a triangle define a mapping from 2D texture space to 3D world space. Combined with the 3D perspective projection defined by the settings of the virtual camera at a fixed point in time during rendering, this mapping finally defines a mapping from 2D texture space to 2D screen space. For a single triangle T at a camera distance of $d \geq d_{min}$, the required texture dimensions are computed based on the largest possible screen-space projections s_{screen} and t_{screen} of the unit vectors (1, 0) and (0, 1) in texture space (Figure 27). If s_{screen} has a length of m pixels, the texture must have a width of at least m texels. The same applies to t_{screen} and the height of the texture. Therefore, an upper bound is needed for the lengths of s_{screen} and t_{screen}.

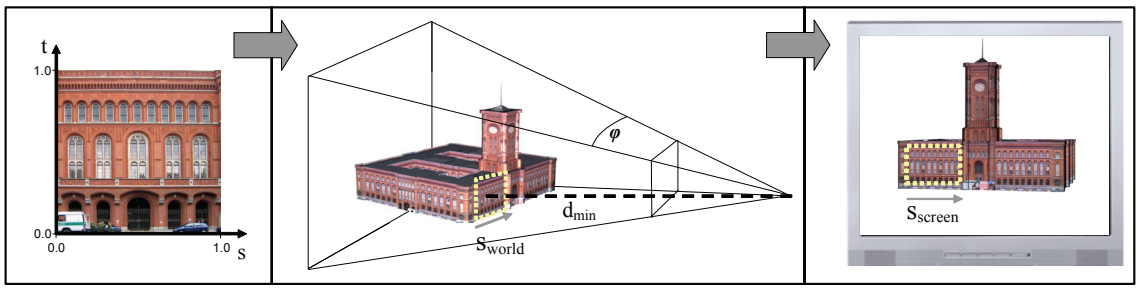


Figure 27: Computation of the largest possible screen projection of the unit vector (1,0) in texture space. The length of s_{screen} is chosen as required texture width.

Let $T = (v_0, v_1, v_2, t_0, t_1, t_2)$ with $v_i \in \mathbb{R}^3$ and $t_i \in \mathbb{R}^2$ for $i = 0, 1, 2$ be a triangle defined by vertex positions and texture coordinates. (v_0, v_1, v_2) can be assumed to be non-degenerate because triangles with degenerate vertex positions are not relevant for rendering. In the following, the description is restricted to the calculation of the minimum width, because the minimum height calculation is analogous. The upper bound for the length of s_{screen} is obtained in two steps:

1. Computing the length of s_{world}, the projection of the texture space unit vector (1, 0) to world space coordinates.
2. Computing an upper bound for the scale factor involved by the perspective projection from world space to screen space at a camera distance $d \geq d_{min}$.

If (t_0, t_1, t_2) is degenerate, s_{world} is not properly defined. For the case $t_0 = t_1 = t_2$, the triangle is skipped and a fixed minimum size of 1×1 is set, because only a single color is mapped to the triangle. If (t_0, t_1, t_2) is degenerate with a non-degenerate longest edge $e = (t_i, t_j)$, step 1 is skipped and the length of s_{world} is estimated by

$$s_{world} = \frac{\| v_i - v_j \|}{\| t_i - t_j \|},$$

i.e., the factor by which e is stretched due to the projection into world space. Therefore, in the following, (t_0, t_1, t_2) can be assumed to be non-degenerate as well. Without loss of generality, it can be assumed that $v_0 = 0$ and $t_0 = 0$ because the computation is translation-invariant in texture space as well as in world space. Let P_T be defined by

$$P_T(a,b) = a \cdot v_1 + b \cdot v_2 .$$

P_T maps a parameterization $(a, b) \in \mathbb{R}^2$ to a point $p \in \mathbb{R}^3$ on the plane spanned by T. Let further be

$$Tex_T(a,b) = a \cdot t_1 + b \cdot t_2 .$$

Tex_T maps a parameterization $(a, b) \in \mathbb{R}^2$ to a point $p \in \mathbb{R}^2$ in texture space.

Since (t_0, t_1, t_2) is non-degenerate, t_1 and t_2 are linearly independent, i.e., Tex_T is invertible. Therefore, the result of step 1 can be calculated by:

$$s_{world} = P_T(Tex_T^{-1}((1,0))) .$$

To compute the scale factor for step 2, it will be assumed for the moment that the screen resolution is fixed and previously known. As explained below, it will, nevertheless, be possible to change the screen resolution and the window size later during rendering. The scale factor will be estimated based on the given minimum camera distance d_{min}, the predefined field-of-view angle φ, and the expected horizontal resolution w_{screen} of the window to which the scene will be rendered. Usually, the scale factor is identical along the horizontal and the vertical screen axis, so that the vertical resolution does not need to be considered separately. The screen projection

of a line segment of the length s_{world} viewed from a distance d has maximum size if it is oriented orthogonally to the viewing direction. Hence, the resulting projection factor is:

$$f_{proj} = \frac{w_{screen}}{2d \cdot \tan(\frac{\varphi}{2})}.$$

Since d is estimated by the lower bound d_{min} for all triangles, f_{proj} has only to be computed once for the texture atlas creation, while s_{world} has to be computed for each triangle separately. Finally, the appropriate texture resolution for a single textured triangle T is obtained by:

$$w_{tex}(T) = \| s_{world} \| \cdot f_{proj}$$
$$h_{tex}(T) = \| t_{world} \| \cdot f_{proj}$$

The projection factor is inversely proportional to the viewing distance. This explains the statement in Section 4.5.1 on the minimum distance value for downsampled textures. Based on the per-triangle formula given above, the appropriate resolution for a given texture that is referenced by a set M of triangles is obtained by:

$$w_{tex} = \max_{T \in M}(w_{tex}(T))$$
$$h_{tex} = \max_{T \in M}(h_{tex}(T))$$

The above computations of texture dimensions assumed a pixel-per-texel ratio of 1.0 and a fixed and previously known screen width of w_{screen}. Since the values for the required texture width and texture height are inversely proportional to the camera distance, a pixel-per-texel ratio of τ can be achieved by dividing all minimum distance values by τ in the rendering procedure. Similarly, rescaling the screen width w_{screen} can be compensated by scaling the minimum distance values by the same factor during rendering.

Computation of the Required Atlas Area

The previously computed values w_{tex} and h_{tex} represent suitable upper bounds for minimum width and minimum height of each texture to avoid magnification. These values apply to a single texture repetition, i.e., w_{tex} and h_{tex} represent the required texture size to provide the texture space area [0, 0] × [1, 1] at an appropriate resolution. To find the final size of the required texture atlas area, the area in texture space that is actually needed for the triangles of the node must also be taken into account. This is necessary for two reasons:

1. If only a fraction of a texture is needed by the triangles of a node, only this fraction should be copied to its texture atlas.
2. If a single triangle uses a texture repeatedly, the texture must also be repeated multiple times in the texture atlas. For instance, if the texture coordinates of a triangle are (0, 0), (3, 0), and (3, 2), the texture must be repeated six times in the texture atlas to allow for correct texturing of the triangle.

At this point it is important to note that high repetition counts, e.g., a triangle with the texture coordinates (0, 0), (1000, 0), (1000, 1000) causes no problems: If the texture coordinate triangle (t_0, t_1, t_2) of a triangle T is scaled by a factor of x, the projected screen size of a single repetition shrinks by the same factor, so that the required texture area size is not affected.

The texture space area that is needed for a texture can be expressed as the axis-aligned bounding rectangle A_{tex} formed by the texture coordinates of all triangles of the node that reference the texture. Scaling w_{tex}, and h_{tex} with width $w(A_{tex})$ and height $h(A_{tex})$ of A_{tex}, respectively, results in the required texture size to represent the full required texture space area at sufficient resolution.

Finally, an additional texture border area is considered for each texture. This border is necessary to avoid *texture pollution*, i.e., visual artifacts caused by the effect that adjacent border pixels of

distinct textures are merged due to downsampling of a texture atlas. For the implementation, a constant border width $b = 8$ pixels has been used. Although, theoretically, this guarantees the avoidance of texture pollution only for the first three downsampling levels, visual artifacts were hardly noticeable (see example screenshots in Figure 23). To fill the border area in the texture atlas with valid content of the original texture, the bounding rectangle A_{tex} is scaled around its center by corresponding scale factors in horizontal and vertical direction:

$$f_{hor} = \frac{w_{tex} \cdot w(A_{tex}) + 2 \cdot b}{w_{tex} \cdot w(A_{tex})}$$

$$f_{vert} = \frac{h_{tex} \cdot h(A_{tex}) + 2 \cdot b}{h_{tex} \cdot h(A_{tex})}$$

Finally, after enlarging of A_{tex}, height and width of the required area TA_{tex} that has to be reserved within a texture atlas for a single scene texture is obtained by:

$$w_{TA_{tex}} = w_{tex} \cdot w(A_{tex})$$
$$h_{TA_{tex}} = h_{tex} \cdot h(A_{tex})$$

Texture Atlas Generation

The results of the previous paragraphs are, firstly, for each texture the required width and height within the texture atlas and, secondly, the source area A_{tex} in texture space from which the original texture content has to be copied. Using the atlas-packing algorithm described in Section 2.1, it can now be attempted to find suitable positions for all textures to fit them in a texture atlas of fixed predefined size. If the texture areas do not fit completely in the atlas image, the atlas creation fails. In this case, the atlas creation is retried after a node split for a smaller subset of the triangles (see Section 4.5.1).

If all textures fit in the atlas, a texture atlas image is created to be finally stored in a leaf node **`L`**. The atlas creation can be efficiently performed by exploiting the graphics hardware: Starting with a blank screen, a textured quad is rendered into the atlas image for each texture. The position of the quad is chosen according to the location of the texture within the atlas, while the corners of A_{tex} are used as texture coordinates. Once all textures have been rendered, the resulting screen content is copied into a texture.

Since for each texture the corresponding texture atlas area is known, to which the texture has been copied, the texture coordinates for the texture atlas can, finally, be computed from the original texture coordinates and stored in **`L.atlasTextureCoordinates`**.

4.5.3 Triangle Set Subdivision of a Node

If in the procedure described in Section 4.5.1 the atlas creation for a node fails, the triangles of the node have to be subdivided into four groups, one for each child node. A well-chosen subdivision should meet two conditions:

a) The overlap of the group's bounding boxes should be small.

b) The summed number of texture pixels for all required texture areas (see Section 4.5.2) of a group should be approximately equal for all groups.

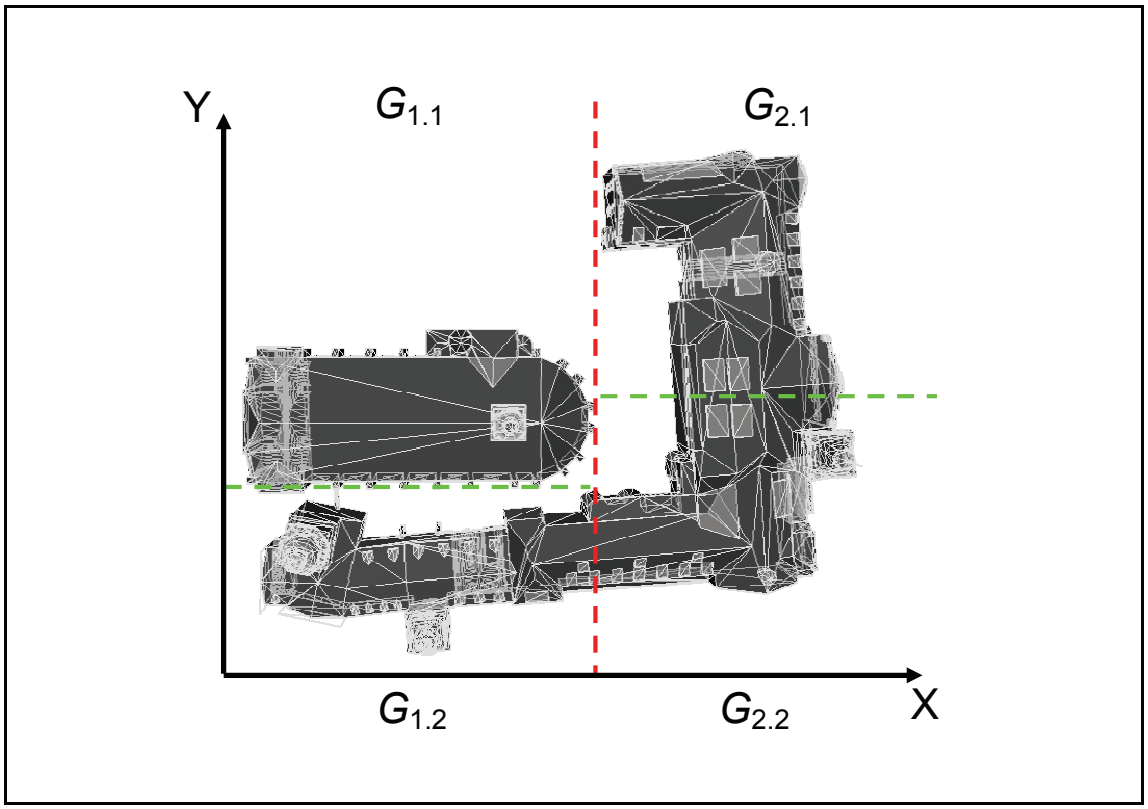


Figure 28: Triangle set subdivision according to equal summed triangle surface area: Firstly, all triangles are split along the x-axis into two groups 1 and 2. Secondly, both groups are again split along the y-axis into two subgroups.

To simplify the problem, it is assumed that the texture mapping is unique. That is, for each texture each point in texture space is mapped to at most one point in world space. As a consequence, if for a single texture the same area is mapped to two different triangles, this is not considered for the subdivision. Since multiresolution texture atlases can cope with scenes with completely individually textured triangles, it is not strictly necessary to exploit the fact that some texture areas are possible referenced multiple times. Given the assumption of unique texture mapping, the summed number of texture pixels for all required texture areas is approximately proportional to the summed areas of all triangles in world space. Therefore, condition b) can be substituted by the condition:

b') The summed surface area of all triangles of a group should be approximately equal for all groups.

The subdivision algorithm chooses those two of the three main coordinate axes along which B has the largest extension, where B is the bounding box formed by the triangle set to be subdivided. To simplify the description, it is assumed in the following that $d_z < d_y < d_x$, whereby (d_x, d_y, d_z) represents the diagonal vector of B. That is, the algorithm chooses the x- and y-axes for subdivision. To find a proper subdivision to meet both criteria mentioned above, a simple greedy algorithm is used that is illustrated in Figure 28: First, the complete set of triangles is sorted according to increasing x-coordinates of triangle midpoints. Next, the ordered sequence is split into two groups G_1 and G_2 (separated by the red line in Figure 28). G_1 contains the first k triangles of the sequence, G_2 the remaining ones. k is chosen in a way that the summed area of the first k triangles is as close as possible to $F / 2$, where F denotes the summed surface area of all triangles. Finally, both groups are split in the same way along the y-axis, i.e., G_1 into

subgroups $G_{1.1}$ and $G_{1.2}$ and G_2 into $G_{2.1}$ and $G_{2.2}$ respectively (indicated by the green lines in Figure 28). Assuming single triangles to be sufficiently small compared to the bounding box of the triangle set to be subdivided, the resulting four groups have approximately equal summed areas and the overlap among their bounding boxes is very small.

Finally, it has to be ensured that the recursive node splitting (see overview in Section 4.5.1) terminates after a finite number of steps. That is, after a sufficient number of subdivisions for an initially given large triangle set, the texture atlas creation must be successful for each resulting subset. By enforcing that after each split of a triangle set G with $|G|>1$ both subgroups are non-empty, it is sufficient to guarantee that there is no *critical triangle*, i.e., no triangle whose required texture atlas area alone exceeds the predefined maximum texture atlas size. Critical triangles can only occur for an inappropriately small chosen d_{min} or for scenes that contain triangles at extremely varying sizes. To prevent these situations, critical triangles are detected before the recursive node splitting starts. Since enlarging the texture atlas size A might slow-down the rendering process, there are two ways to deal with critical triangles:

1. If only a small amount of the triangles is critical, this can be handled by splitting them into smaller triangles. This involves a slight increase of the geometric scene complexity.
2. If a considerable amount of triangles is critical, it is more useful to choose a larger value for d_{min}: Since the required texture atlas area depends on the size in which a triangle appears on the screen, critical triangles seen at a camera distance $d > d_{min}$ are per definition still large enough to fill a considerable amount of the screen. That is, a few critical triangles are already sufficient to cover the screen completely. Therefore, it is more appropriate to use the original scene textures for those triangles, i.e., to enlarge d_{min}.

To make these considerations more concrete, the following preparation steps are performed to cope with critical triangles:

- If the amount of critical triangles exceeds a predefined threshold value c, e.g. 10 %, d_{min} is enlarged, so that the amount of critical triangles is reduced to c. The width and the height of the required texture atlas area of each critical triangle are inversely proportional to d_{min}, except for the constant texture border width. Hence, the new larger value for d_{min} can be computed based on the required texture atlas areas that have been computed for the previous d_{min} value.
- The remaining triangles are recursively split into smaller ones until no critical triangles are left.

At the cost of a slight increase of geometric complexity, this preparation step, finally, guarantees the termination of the recursive node splitting.

4.5.4 Computation of Texture Coordinates

This subsection describes the computation of the texture coordinates for texture atlases as well as the corresponding texture matrices. As mentioned in Section 4.5.1, this step is taken after the tree structure has been built and texture atlases have been created for all nodes. In addition, texture coordinates for texture atlases of the leaf nodes have already been computed as a part of the atlas creation process (Section 4.5.2). These texture coordinates, however, are only valid for the texture atlases stored in the leaf nodes, but not for the combined atlases of inner nodes.

Storing additional texture coordinates for each inner node would mean to store l additional texture coordinate vectors for all triangles of a leaf node L, whereby l denotes the tree level of L. To avoid this, texture matrices are used to allow for using the same set of atlas texture coordinates for all levels of the tree. The principle is illustrated in Figure 29: The atlas of a node N corresponds to a squared area $R(N)$ of the root node atlas. If k is the tree level of N (where

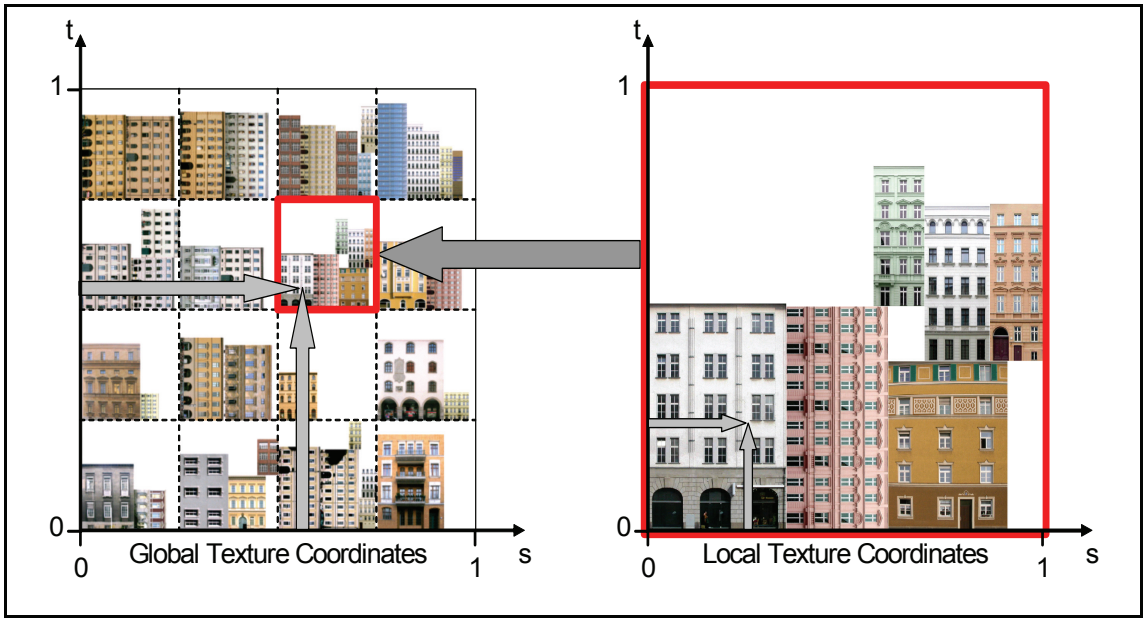


Figure 29: The local texture coordinates for the atlas at tree level k = 2 are obtained by scaling the root atlas coordinates by a factor of $4 = 2^k$ and discarding the integer part.

level 0 indicates the root level), $R(N)$ has a width of $1/2^k$ and both coordinates of the lower left corner are multiples of $1/2^k$. That is, if t is a texture coordinate vector for the texture atlas of N and $r(t)$ the corresponding texture coordinate vector for the root atlas, t can be obtained from $r(t)$ by multiplying with 2^k and discarding the integer part of the result. Fortunately, both operations can be performed during rendering without any additional performance costs: Multiplying by $f = 2^k$ is achieved by setting the texture matrix to a scaling matrix for uniform scaling by the factor f, and discarding the integer part of texture coordinates corresponds to rendering textures in repeat mode. Therefore, by storing the texture coordinates for the root node atlas in the leaf nodes and setting the appropriate texture matrix for each node, the same set of texture coordinates can be used for all tree levels.

According to the principle described above, the texture coordinates that are valid for the leaf-node atlases have to be converted into the corresponding texture coordinates for the root node atlas. For each leaf node L, the position of the corresponding area $R(L)$ depends on the path from the root node to L. Let the child nodes of each inner node N be ordered by (0 = *bottom-left*, *1 = bottom-right*, *2 = top-left*, *3 = top-right*), i.e., child 0 of N corresponds to the bottom-left quadrant of the atlas of N, the second one corresponds to the bottom-right quadrant etc. For a leaf node L let be $P = (P_1,\ldots,P_k)$ a sequence of integer values encoding the child indices on the path from the root node to L. For instance, in Figure 29 the node L whose atlas is shown on the right would be encoded by $P = (3, 0)$. If $t = (t_s, t_t)$ is a texture coordinate vector for the texture atlas of L, the corresponding root atlas texture coordinate vector $r(t) = (r_s, r_t)$ is obtained by:

$$r_s = t_s \cdot 2^{-k} + \sum_{i=1}^{k} (P_i \bmod 2) \cdot 2^{-i}$$

$$r_t = t_t \cdot 2^{-k} + \sum_{i=1}^{k} (P_i \text{ and } 2) \cdot 2^{-i}$$

The *mod* denotes the modulus operator, and the *and* operator denotes the bitwise and operation.

4.6 Memory Management

In the rendering algorithm in Section 4.4, the texture atlas tree data structure was assumed to be completely available in main memory. Since the texture atlas tree is intended for large amounts of texture data, this assumption will, usually, not hold in practice. This section describes a memory management strategy that allows for using multiresolution texture atlases for scenes whose complete amount of textures exceeds the main memory capacity. During rendering, the strategy performs dynamic loading and deleting of texture data that is streamed from hard disk or via a network connection. The memory management is applied to texture atlases and to full-resolution textures of leaf nodes. All other data, including the geometry, is comparatively small and, therefore, is permanently kept in memory.

As a first step, the rendering procedure has to be slightly modified to consider missing textures. The necessary modification of the rendering procedure is shown in Figure 30: The only modification of **`renderTree()`** is the determination of **`useNodeAtlas`**: If the resolution of a texture atlas is not sufficient for a node **`N`** but refinement is currently impossible, all triangles of the missing child nodes, nevertheless, are rendered immediately using the texture atlas of **`N`**. Refinement is possible if and only if:

- **`N`** is an inner node and the texture atlases of all child nodes are currently in memory, or
- **`N`** is a leaf node and the full-resolution textures **`N.textures`** are currently in memory.

The root node atlas is always kept in memory. Thus, in the worst case some triangles are temporarily rendered with too coarse textures, but it is always guaranteed that texture data is available for each triangle.

The memory management strategy handles texture atlases and full-resolution textures simultaneously in the form of memory blocks. Each block represents either a texture atlas or an original texture of a leaf node. Large original textures, i.e., textures whose memory consumption exceeds that of texture atlases by a factor greater than two, are split into multiple blocks. The memory management strategy is based on the following considerations:

- *Explicit control of memory usage*: The summed size in bytes of all blocks in memory must not exceed a predefined maximum capacity *k*.
- *Preference of immediately needed textures*: At each point in time, a certain set of blocks is needed to provide textures at appropriate resolution for all currently visible scene parts. If all needed blocks are in memory, the modification of the rendering procedure shown in Figure 30 has no effect. Otherwise, i.e., whenever a node atlas at insufficient resolution must be used, the blocks necessary to provide the appropriate texture resolution should be loaded with high priority.
- *Consideration of camera distance*: Appropriate texture resolution is most important for scene parts close to the camera. That is, among all textures that are needed for the current view, those nearest to the camera should be preferred.

```
bool refinementPossible(Node N) {

   if (isLeafNode(N)) {
      // The next higher resolution is only provided by
      // the full-resolution textures.
      return inMemory(N.fullResolutionTextures);
   }
   else {
      // Refinement is possible if and only if
      // all child nodes of N are either in memory
      // or invisible
      for each Node C in N.children {
         if (!inMemory(C) && !ousideViewFrustum(C.bounds)) {
            return false;
         }
      }
      return true;
   }
}

void renderTree() {

   ... (code section remains unchanged)

   while (!nodeStack.isEmpty()) {

      ... (code section remains unchanged)

      // Determine whether to use N.atlas for rendering
      bool useNodeAtlas = (d >= N.minDistance);

      // ------------ Added Code Section: -------------
      // Enforce atlas usage if textures at higher
      // resolution are not available.
      if (!refinementPossible(N)) {
         useNodeAtlas = true;
      }
      // ----------------------------------------------

      ... (code section remains unchanged)
   }
}
```

Figure 30: Modification of the rendering code in Figure 26 to consider missing textures.

- *Avoidance of abrupt texture resolution changes*: Abruptly changing texture resolution can be disturbing. Abrupt changes occur, for instance, if full-resolution textures of a leaf node L are removed from memory while all texture atlases on the path from the root to L are also not in memory. In this case, full-resolution textures are immediately replaced by the lowest resolution texture of the root node. To avoid this, for each texture in memory, the corresponding lower-resolution textures should also be kept in memory. That is, if the atlas of a node N is in memory, the atlases for all nodes on the path to N should also be in memory. Similarly, if the full-resolution textures of a leaf node L are in memory, the atlas of L should be in memory as well. In this way, textures can be quickly removed from memory if necessary without causing abrupt resolution changes.
- *Prefetching of textures that might suddenly become visible*: Scene parts outside the view-frustum are excluded from rendering, independent of their camera distance. Close but invisible scene parts might become visible very quickly due to a rotation of the viewing direction. Therefore, textures for such scene parts should be prefetched if possible.

- *Prefetching of textures for the next refinement step*: If the texture atlas of a node N is currently used for rendering, textures at the next higher resolution should be prefetched if possible. Therefore, for an inner node N, the texture atlases of the child nodes of N should be prefetched. Similarly, if N is a leaf node, the original textures for the corresponding scene part $P(N)$ should be prefetched.

For a given camera position and viewing direction, the priority of a block is defined based on the criteria *usage*, *distance* and *hierarchy level*. The usage categories, ordered by decreasing priority, are:

`Needed`: This category contains the blocks for all textures that are needed to run the rendering procedure without using a texture atlas at insufficient resolution. Since the rendering traversal runs in depth-first order and aborts at missing textures, it follows that for each node N in the **`Needed`** category each node on the path from the root to N is in the **`Needed`** category as well.

`Culled`: If a node N is outside the view frustum, but there is at least one sibling whose texture is currently in the **`Needed`** category, the texture atlas of N is in the **`Culled`** category.

`Parent needed`: For the texture atlas of a node, this usage value indicates that the texture atlas of the parent node is currently in the **`Needed`** category. For blocks containing full-resolution textures of a leaf node L, the usage value **`Parent needed`** indicates that the texture atlas of L is currently in the **`Needed`** category. If all nodes of this category have been prefetched, the next refinement step is possible for all visible textures.

`Unused`: This category contains all remaining textures.

If a block fulfils the criteria for multiple categories, the category of highest priority is chosen. The usage category of each node is determined in a separate tree traversal that simulates the rendering process assuming that all nodes are currently in memory. Instead of rendering, this traversal does only mark blocks according to their category. By using an additional tree traversal, needed nodes are also recognized if their parent nodes are currently not in memory.

Among blocks of the same usage category, the priority is determined by the distance between the bounding box of the corresponding node and the current camera position. If usage and distance are equal for two blocks, the block with the lower tree level is preferred. In this way it is ensured that the parent node of a needed node always has higher priority.

Based on the priority order introduced above, a set of blocks is identified that should currently be in memory, called the *ideal block set*. The ideal block set is updated for each frame and contains references to all blocks that have been identified for the categories **`Needed`**, **`Culled`**, and **`Parent needed`**. If the summed sizes of all blocks in the ideal block set exceed the maximum capacity k, it is reduced to the blocks of highest priority. That is, if $b_1,\ldots,b_n$ denote the elements of the ideal block set sorted by decreasing block priority, the ideal block set is reduced to $b_1,\ldots,b_l$, where l is chosen maximal with the property that the summed size of the reduced set is smaller than k.

References to all blocks that are currently in memory are maintained in a list ordered by decreasing priority. If a requested block of the ideal block set cannot be loaded without exceeding the maximum summed size k of all blocks in memory, the block of lowest priority is removed from memory.

Since loading of textures must be done during rendering, it must not consume too much computation time at once. If each texture was immediately loaded on demand, this would

sometimes interrupt the rendering process for considerable time intervals. Thus, texture loading is done gradually and distributed over several successive frames. The resulting slightly delayed loading of textures has no visible consequences as long as it affects only textures in the categories **`Culled`** and **`Parent needed`**. Only if textures in the **`Needed`** category are missing, some scene parts might temporarily appear with insufficient texture resolution and will then be gradually refined. Due to the prefetching involved by the categories **`Culled`** and **`Parent needed`**, this is mostly avoided if the camera movement is not too fast. In addition, a gradual refinement of the texture resolution is less disturbing than an instable frame rate.

The requested blocks are maintained in a priority queue. In each frame, a fixed time slot is reserved to continue the loading process of the first blocks in the queue. A block is removed from the queue if it has been completely loaded or if it is no longer in the ideal block set. As long as the queue does not exceed a predefined maximum length, newly requested blocks are added to the queue. To determine the next block to be added, the ideal block set is traversed in the order of decreasing priority. The most important block that is currently neither in memory nor queued is added to the queue. The performance overhead for the management of the ideal block set is not critical, because its size is usually at a scale of 100 entries or lower.

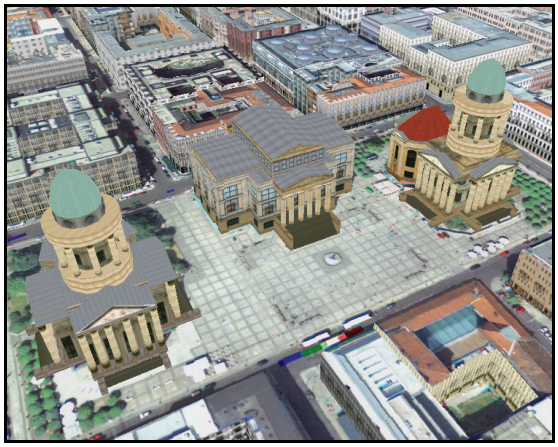
4.7 Performance Tests

To evaluate the efficiency of multiresolution texture atlases, the approach has been tested for two city models. This section describes the test configurations and the achieved results. The texture atlas resolution was set to 512 × 512 pixels. S3 texture compression was used to reduce the disk-space requirements of the texture atlas tree and to speed-up dynamic texture loading. According to the memory requirements of a texture atlas of 512 × 512 pixels size compressed by a factor 6, the size of a single block was about 130 KB. The summed size of all blocks in memory was limited to 100 MB. All tests were performed at a screen resolution of 800 × 600 pixels with a pixel-per-texel ratio of 1.0 on a Dell Inspiron 8600 laptop with Pentium-M processor at 1.5 GHz, 1 GB main memory, and ATI Radeon 9600 Mobility graphics card with 128 MB graphics memory.

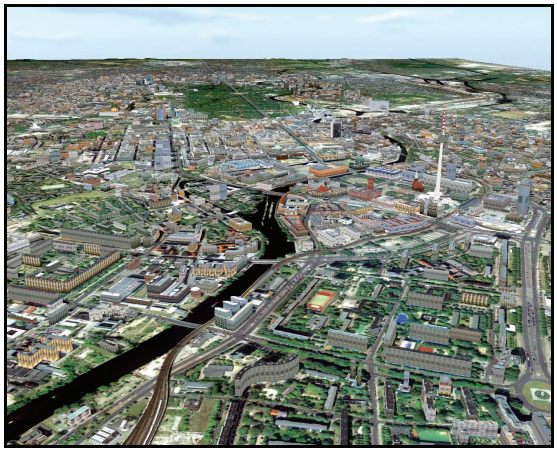
4.7.1 City Model of Berlin

The first test was made using a city model of a part of Berlin, consisting of 10,000 block buildings and 40 manually created models of selected landmark buildings. Snapshots of the model are shown in Figure 31a)-c). For each block building, an individual texture of 512 × 512 pixels resolution was repeated around the facade. Since only 300 different facade photographs were available for the test, 10,000 individual textures were created by labeling each texture with a unique number. The uncompressed size of all block-building facade textures was about 7.32 GB (1.22 GB with S3 compression). The additional, manually created models contained 369 textures with a total amount of 160 MB (27 MB with S3 compression) texture data. The global minimum distance parameter d_{min} was chosen as $D/72$, where D denotes the length of the model's bounding-box diagonal-vector. The aerial image that was used for both terrain and roofs of buildings in Figure 31 was rendered using the technique described by Baumann [2000]. The performance test has been carried out only for those scene parts rendered with multiresolution texture atlases. That is, roofs and terrain are neither considered in the specification of texture complexity mentioned above nor in the following frame rate measurements.

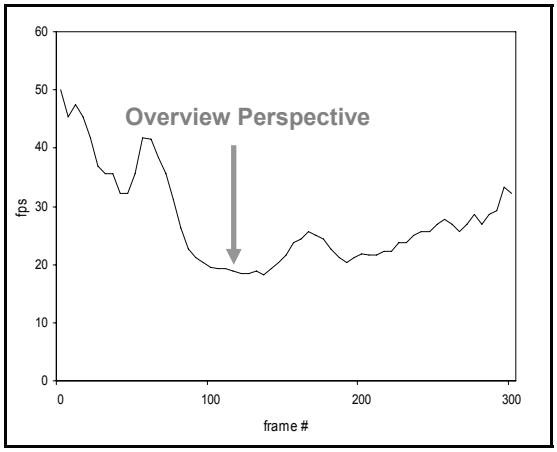
a) Close view at a 512 x 512 facade texture.



b) Flythrough perspective: 34 atlas nodes.



c) Overview perspective: 47 atlas nodes.



d) Frame rate measurements.

Figure 31: a)-c) Snapshots of the Berlin city model used in the first test. d) Rendering speed in frames per second measured for the Berlin city model using a 300 frame camera-path animation with rapid camera movements and strong changes in visibility.

Figure 31d) shows the rendering speeds in frames per second (fps) that have been measured for a test camera animation consisting of 300 frames. To test the ability of multiresolution texture atlases to cope with unrestricted user navigation, the camera path has been chosen in a way that the visibility is rapidly changing. That is, the camera path performs rapid changes between different perspectives ranging from a pedestrian view to a full overview perspective. The average frame rate was 28 fps. The frame-rate minimum of 18 fps occurred at an overview perspective. Remarkably, particularly the overview perspective is uncritical from the point-of-view of textures because since no building is close to the camera, only a small number of texture atlases is required for rendering. For instance, when the frame rate minimum occurred, the number of used texture atlases was 34, while in the beginning of the camera path, where the frame rate was between 40 and 50 fps, the number of texture atlases was in the range of 60. This indicates that the bottleneck of texture switches has been successfully overcome, and the frame rate drop is caused by other factors such as geometric complexity or fill rate, i.e., the number of triangles or the number of fragments that have to be processed by the graphics hardware.

The preprocessing time for the atlas tree creation was about two hours. Since the scene did not completely fit in main memory, a considerable part of the preprocessing time was spent in

loading and saving image files. The tree consisted of 1773 nodes. That is, the hard disk requirements for all generated texture atlases excluding full-resolution textures were only 221.6 MB.

Differences in performance measurements compared to a similar test previously presented in [Buchholz and Döllner 2005(ii)] are primarily caused by two reasons: First, in the camera path used in the test described above, the amount of simultaneously visible triangles is higher, leading to a lower average frame rate than in the previous test. Second, the original test-data contained a large amount of redundantly high tessellated geometry, i.e., rectangular walls that were represented by several triangles instead of just two. For the new test described above, these redundancies had been removed from the model. Therefore, the frame rate minimum was higher in the new test (18 fps instead of 12 fps) although twice as many block buildings and additional, manually modeled buildings were rendered.

4.7.2 City Model of Chemnitz

The second test was performed on a detailed city model of downtown Chemnitz. The primary aim of the test was to check the ability of the multiresolution texture atlases approach to cope with scenes in which the texture workload is more irregularly distributed over the scene than in the first test. Snapshots of the model are shown in Figure 23. The geometric complexity of the model was 300,000 triangles, which is not critical for current graphics cards. The total amount of texture data is only about 36 million texture pixels, which could, theoretically, be handled directly by the graphics hardware. The model contains, however, about 2,000 different textures, which are all visible at once from an overview perspective. Therefore, the primary performance problem when rendering the scene in a conventional way is caused by the number of texture switches. In addition, the scene contains several repeated textures with high repetition counts such as the tile texture on the ground in the lower part of Figure 23c) and d). This makes it difficult to reduce the number of texture switches by applying conventional texture atlases (see Section 2.1).

Repeated textures must also be considered when using multiresolution texture atlases. Therefore, the method of texture replication introduced in Section 2.1 has been applied for the Chemnitz model. In contrast to conventional texture atlases, multiresolution texture atlases allow for using texture replication without problems even for large repetition counts because the texture resolution does only depend on the screen-space area covered by each triangle at a distance $d > d_{min}$. That is, it is not relevant for the required texture space whether a triangle contains 100 repetitions of a tile texture or a single one because the repeated tiles appear correspondingly smaller. This made it uncritical to apply texture repetition to the Chemnitz model. In contrast, using conventional texture atlases for the model, texture replication would have increased the total amount of texture data to multiple billion texture pixels, leading to an unacceptably large number of necessary texture atlases.

Since the Chemnitz city model contains transparent parts, two texture atlas trees were used, one for RGB and one for RGBA textures. The global minimum distance parameter d_{min} was chosen as $D/62$, where D is again defined by the diagonal of the model's bounding box. Both trees required a total preprocessing time of 8 minutes and a total hard disk space of 223 MB. Figure 32 shows the time measurements for a test camera animation.

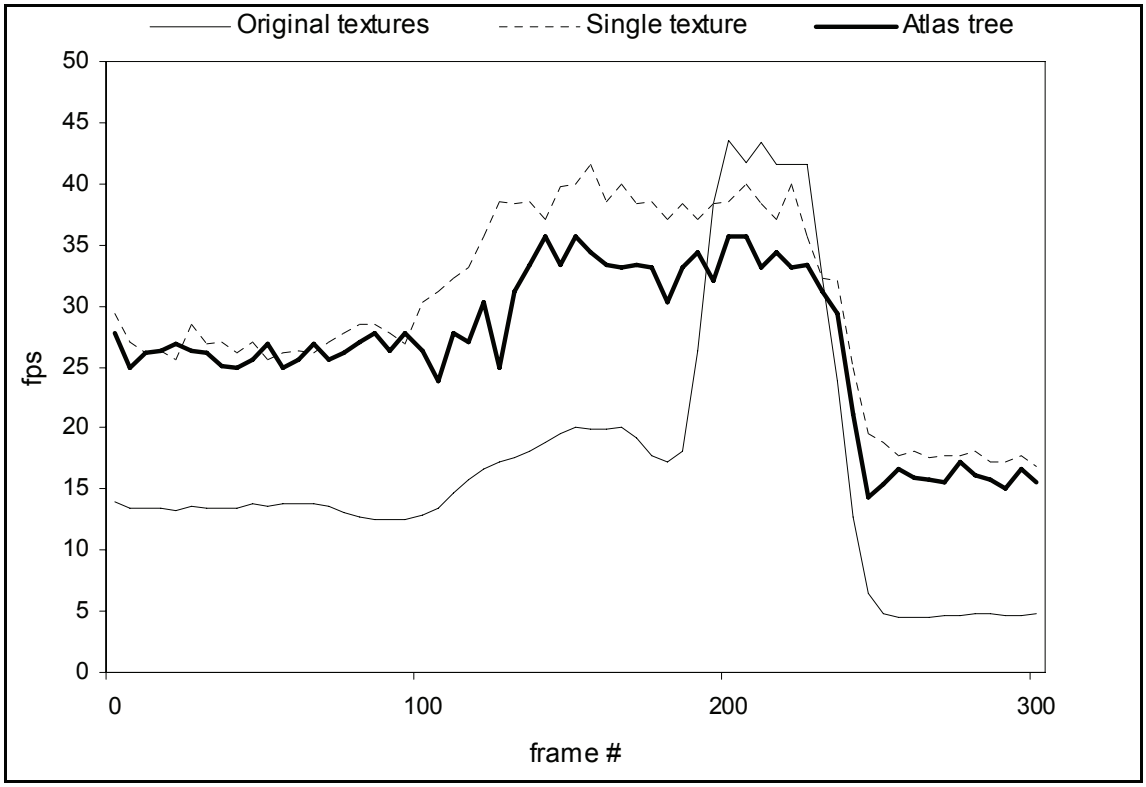


Figure 32: Frame rate comparison of different rendering configurations applied to the Chemnitz city model.

The frame rates for the camera animation were measured three times:

1. In the first run, the scene was rendered using the original scene textures. The results are indicated by the thin solid line. In contrast to the Berlin city model, the Chemnitz model completely fitted in main memory so that the frame rate measurements could be done for comparison.
2. In the second run, the scene was rendered completely without texture switches, i.e., instead of several textures a single white texture has been used for the complete scene. In this run the model did not appear correctly. The measured frame rates, however, provide an approximate upper bound for the rendering performance that could theoretically be achieved by reducing texture switches and texture workload. The result is indicated by the dashed line.
3. In the final run, the scene was rendered using two texture atlas trees. The results are indicated by the thick line.

For the comparison, the scene has been rendered without alpha tests and alpha blending because considering the alpha values would have caused a dependency between texture content and frame rate. This dependency would primarily have degraded the rendering speeds for the white texture in the second test so that the dashed line could not provide an upper bound for rendering performance.

The average frame rates were 17 fps for the original textures, 30 fps without texture switches and 27 fps for the atlas tree. At the overview perspective the original-texture rendering dropped below 5 fps, while the atlas tree was permanently over 14 fps.

For a short moment during the test, around frame 200, the second run surprisingly yielded the maximum performance. This effect can be explained as follows: When rendering with a single

white texture or rendering with multiresolution texture atlases several small objects share a single texture atlas. Therefore, they can be combined to larger objects and rendered with a single rendering call. This improves the rendering performance for most perspectives. Around frame 200, however, the camera was in a perspective for which only a very small number of triangles and textures were inside the view frustum. In this special case, the rendering of more separate small objects became slightly advantageous because all objects outside the view frustum could be excluded from rendering.

Finally, the test has been repeated with activated blending of alpha-textures to test which frame rate can be finally achieved. For this final test, the achieved average frame rates were 21 fps using multiresolution texture atlases and 13 fps when rendering with original textures. The number of atlas-tree nodes that had to be rendered per frame was about 30 on average and about 60 in the worst case.

4.8 Optimizations

This section describes optimization methods for the implementation of the texture atlas tree.

4.8.1 Geometry Batching

Theoretically, a set of triangles can be rendered using a separate rendering call for each triangle. It is, however, considerably more efficient to *batch* the geometry, i.e., to combine a large number of triangles into a single vertex array because triangles that are stored successively in a vertex array can be efficiently rendered using a single rendering call to the graphics subsystem. For the rendering algorithm of the texture atlas tree, this means that if a cell of an inner node N is completely inside the view-frustum and visible at a sufficiently large camera distance to be rendered with a single texture atlas, it would be desirable to provide all triangles of the node cell in a single vertex array that is stored in N. In this way, the child nodes of N would not have to be traversed and a single rendering call would be sufficient to render the whole node cell of N. This variant, however, would involve a waste of memory because all scene triangles would have to be stored once per tree-level. Thus, the aim is to store triangles in a way that the following conditions are met:

1. If a node cell is completely visible and can be rendered with a single texture atlas, all triangles can be rendered with a single rendering call.
2. For each inner node, the triangles of each child node can be separately rendered in order to apply different textures or to skip nodes whose cells are outside the view frustum.
3. Each triangle is stored only once.

The above criteria can be achieved by storing all triangles of the tree in a single vertex array that is appropriately ordered. The basic idea is to sort the triangles in a way that for each node cell all triangles correspond to a single contiguous range within the vertex array. Each node stores start index and length of its corresponding range in the vertex array. The appropriate order of the vertex array is exemplified in Figure 33: Each numbered quad represents the cell of a leaf node L_i of the texture atlas tree. L_i stores a set of triangles $\{L_i(1),\ldots,L_i(n_i)\}$. To simplify the illustration, the scene subdivision is shown as a quadtree with equally subdivided cells, because the irregularity of the cell subdivision is not relevant for the explanation. If the triangles are copied into a single vertex array in the order $\{L_1(1),\ldots,L_1(n_1), \ldots ,L_{22}(1),\ldots,L_{22}(n_{22})\}$, the triangles of each node correspond to a contiguous range in the vertex array. Thus, all three conditions mentioned above are met.

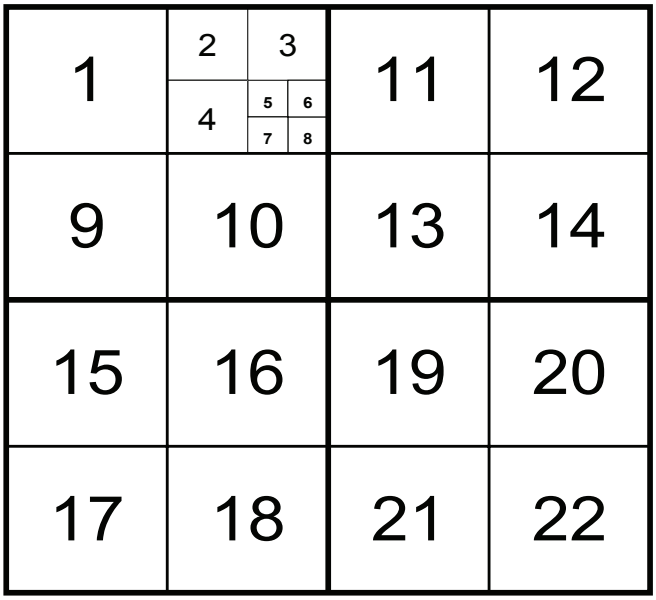


Figure 33: Ordering of triangles in a way that the triangles of each node cell correspond to a contiguous range in the vertex array.

For a given texture atlas tree containing all scene triangles in its leaf nodes and an initially empty target vertex-array *VA*, the appropriate order is computed as follows: The tree is traversed in depth-first order. For each visited node *N*, the current length of *VA* is stored as range start index. If *N* is a leaf node, all triangles stored in *N* are appended to *VA*. Otherwise, the method is recursively called for the child nodes of *N*. When all triangles of the sub-tree with root *N* have been processed, the length of the range for *N* in *VA* is obtained by the difference between the current length of *VA* and the range start index that has been previously stored in *N*.

4.8.2 Specialized Rendering of Roofs

3D building models are frequently available in combination with a digital terrain model and additional terrain textures such as aerial images or satellite images. In this case, the texture atlas tree can be used for rendering textured building models while a specialized terrain rendering technique is used for rendering the underlying terrain and its textures. Although the texture atlas tree could, theoretically, be used for all parts of the scene, terrain-specific rendering approaches are preferable for terrain rendering, because they include geometric level-of-detail management and exploit the regular structure of the terrain texture for more compact disk storage. Therefore, the most useful combination is to use a terrain rendering technique and the texture atlas tree simultaneously. This involves, however, some redundancies concerning the handling of roof textures: For existing buildings, roof textures are typically provided implicitly as a part of an aerial image. That is, texture data for roofs are provided as a by-product of the terrain rendering technique. If roofs are handled by the texture atlas tree, this cannot be exploited. The roof textures have to be extracted from the aerial image and managed by the texture atlas tree in the same way as other textures. That is, the roof textures would unnecessarily increase the disk space requirements of the texture atlas tree, require additional preprocessing time, and lead to additional performance costs for the memory management strategy.

Redundant storing and loading of roof textures can be avoided by shifting the rendering of roof triangles to the terrain rendering technique. For this, an extension of the technique of Baumann [2000] has been implemented that allows for exploiting the terrain texture for building

roofs as well. For each frame, the terrain rendering technique provides a set of textures, whereby each texture T of this set provides the terrain texture data for a corresponding terrain region R_T at appropriate resolution. Each such region R_T can be described by an axis-aligned rectangle in the horizontal plane. A detailed description of the algorithm is given in Baumann [2000] but is not necessary for this explanation.

The main problem for the integration of roof rendering in this terrain rendering technique is that the terrain regions for which textures are provided are not known in advance but are determined at runtime by the terrain rendering technique. That is, to apply a texture provided for a region R to the corresponding roof triangles within R, the following problem must be solved rather frequently during rendering:

Given a terrain region R, render all triangles and triangle parts that overlap R.

Simply checking for each triangle whether it has to be rendered or not would be very slow due to the large number of overlapping tests. In addition, triangles intersecting the borders of R would be rendered partially incorrectly outside the borders of R.

To reduce the number of overlapping tests, the roof triangles are organized in a quadtree structure in the way as explained in Section 4.8.1: The roof triangles are sorted in a vertex array according to the quadtree structure. In this way, for each quadtree node whose node cell is completely inside a terrain region R and completely visible, all triangles can be rendered by a single rendering call. In this way, the number of overlapping tests is significantly reduced and rendering is performed efficiently by processing contiguous ranges of triangles using a single rendering call to the graphics hardware.

The final problem is to handle roof triangles that intersect the boundary between two adjacent terrain regions R_1 and R_2. These triangles are rendered twice, once for R_1 and once for R_2. If such a triangle t is rendered the first time using the terrain texture for R_1, valid texture data is only available for the part of t that overlaps R_1. Since the exceeding part must not be rendered without valid texture data, a fully transparent texture border color is used. That is, an alpha value of 0.0 is mapped to each fragment outside the valid texture area. In this way, both parts of t are rendered separately using different textures. On the finally rendered triangle the seam between both textures is completely invisible.

4.8.3 Efficient Rendering of Geometry Close to the Camera

Rendering scene parts very close to the camera at full texture resolution frequently consumes a significant amount of the overall rendering time. Therefore, it is useful to accelerate rendering of leaf nodes whose corresponding cells are close enough to require rendering with full-resolution textures by using sate-sorting and conventional texture atlases as long as the textures are small enough and non-repetitive.

Chapter 5

Applications of Multiresolution Texture Atlases

This chapter addresses the potential of multiresolution texture atlases for real-time city model visualization. As a key advantage, they facilitate the assignment of individual textures to all parts of a city model without regard to the total amount of texture data. This provides a basis for developing new visualization approaches in which the role of textures is not restricted to the representation of photorealistic detail. Two visualization approaches are introduced that apply individual textures for visualization of thematic data and illustrative visualization.

5.1 Computed Textures for Thematic Visualization

As a key advantage, multiresolution texture atlases facilitate the rendering of large city models that are completely covered with individual textures. Each triangle can be textured independently. This provides a technical basis to use surfaces of city model objects to visualize various kinds of thematic data that result from computational models. For this, these data are encoded into *surface textures*, i.e., textures that are draped onto the surfaces of city model objects. Examples of thematic data that can be visualized using surface textures include:

- *Visibility of objects*: Surface textures can be used to encode from which points a certain object of interest, for instance, a planned building, is visible or partially visible. This is

demonstrated in an example given later in this section. Alternatively, if vegetation data are separately available for a given city model, the color of a surface point can be used to indicate the amount of vegetation visible from that point.

- *Lighting*: Lighting calculations can be used to increase the degree of photorealism in city model visualization but can also be used for the estimation of lighting conditions. For instance, for real-estate management it might be of interest where sun-lit apartments can be found. Using precomputed textures, bright and dark regions of a city at a fixed date and time can be visualized. Alternatively, a computational model can be used to estimate the annual average number of hours per day during which the sun is visible from a certain point. By visualizing the results of such a model via surface textures, the average brightness of an apartment, a garden or a place becomes directly visible in an interactive 3D visualization. Although lighting can, theoretically, be also computed in real-time, simple real-time enabled illumination models are not suitable to provide reliable simulations of real lighting conditions.
- *Noise exposure*: Based on a computational model and basis information such as location and noise level of main roads, the resulting noise propagation in a city can be simulated. The results of such a computation can be visualized by coloring the surfaces of city model objects according to the computed noise level. In this way, it becomes directly visible which facades are affected to which degree.
- *Wave propagation in radio-network planning*: In radio-network planning, complex computational models are used to simulate the radio-network resulting from certain antenna positions, antenna orientations, and other configuration parameters. Surface textures can be used to visualize radio-network properties such as field strength or the best-suited antenna to supply a certain surface point of the city.
- *Exhaust-gas pollution*: The estimated degree of exhaust-gas pollution for places, streets, and locations can also be conveyed via surface textures. In contrast to 2D visualizations, this makes it possible to visualize differing values at different heights.

To demonstrate the potential of individual surface textures in city models for visualization of precomputed thematic data, a texture creation step has been implemented. It allows for integrating an arbitrary computational model to visualize its precomputed results. The extension will be explained and illustrated for the example of visibility calculations. The result is shown in Figure 34, in which surface textures are used to visualize the visibility of a church tower.

5.1.1 Specification of a Computational Model

The term *computational model* here refers to a function f that determines a color $f(p)$ for a given surface point p on an arbitrary triangle t of the initially given 3D scene. A simple way to specify the parameter p is to define p as a vector in $\mathbb{R}^3$. However, a computational model might require additional information such as the surface normal vector of p obtained by interpolation of the three vertex normals specified for the vertices of t. A more flexible approach is to define the parameter p as a tuple consisting of a unique identifier (e.g., a number) of the triangle t and the 2-dimensional barycentric coordinates that identify the relative position of p on t. In this way, an implementation of f can derive not only the position in $\mathbb{R}^3$ of p but also any other available vertex parameters of t by accessing the original scene data.

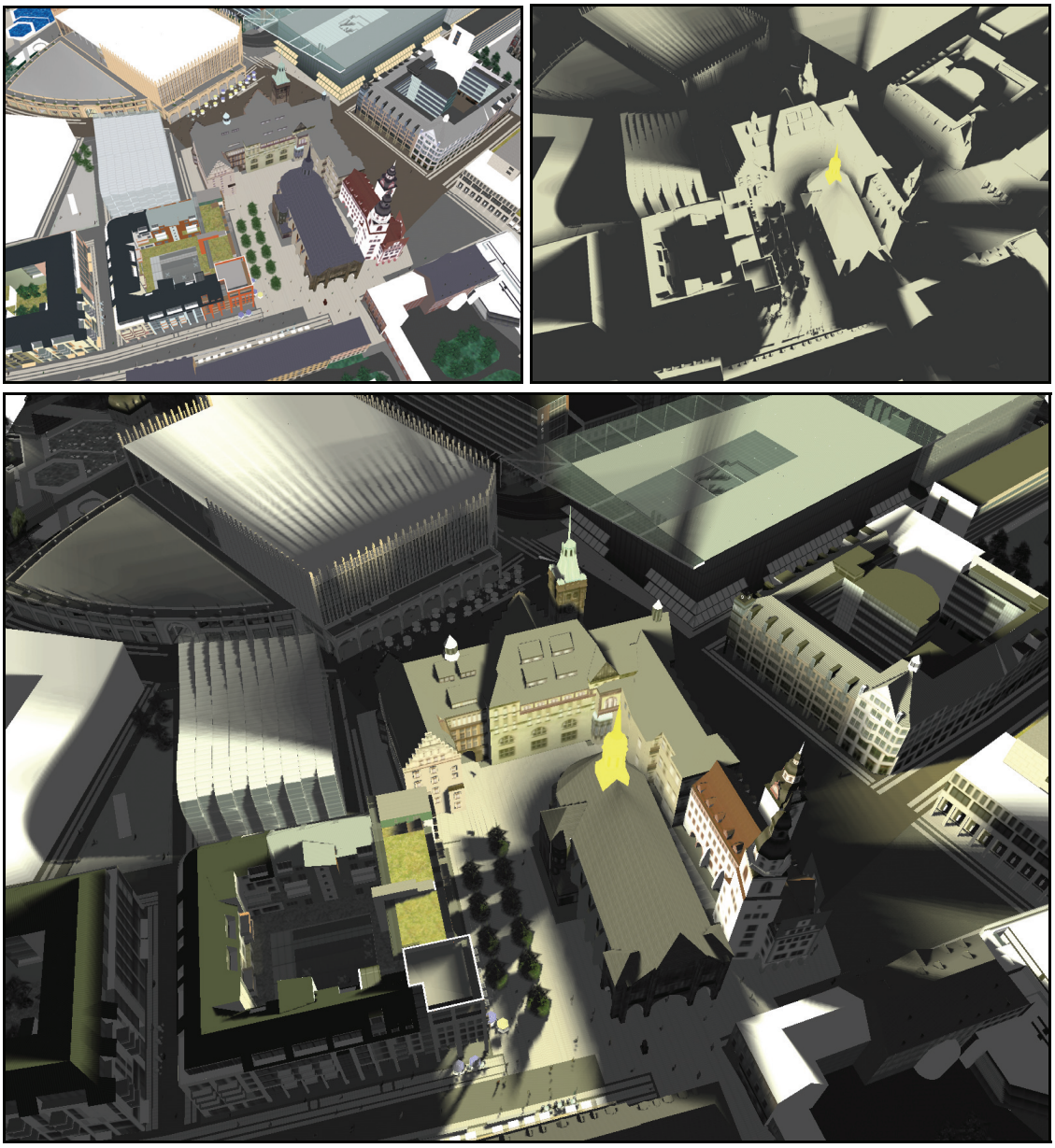


Figure 34: Visualization of visibility using surface textures. Where is the church tower (highlighted in yellow) visible? The upper left image shows the original surface textures of the scene. The upper right image shows the scene using precomputed visibility textures. The brightness of a surface point is computed according to the occlusion of the church tower when seen from that point. The lower image shows the result obtained from combining both texture layers.

For the visibility visualization example introduced above, the aim is to compute for each surface point p a greyscale value $f_{vis}(p)$ whose brightness $b \in [0,1]$ indicates the degree to which an object of interest O is visible when looking from p towards O. The principle is illustrated in Figure 35 for three points with different visibility values. If O is completely unoccluded, b is set to 1.0, while $b = 0$ is set for full occlusion. More generally, b is set to the quotient $A_{unoccluded} / A_{total}$. Here, $A_{unoccluded}$ denotes the image area on which O is not occluded by other objects, and A_{total} denotes the image area that is covered by O if it is rendered without environment. The resulting color value can, finally, be combined with the surface color defined

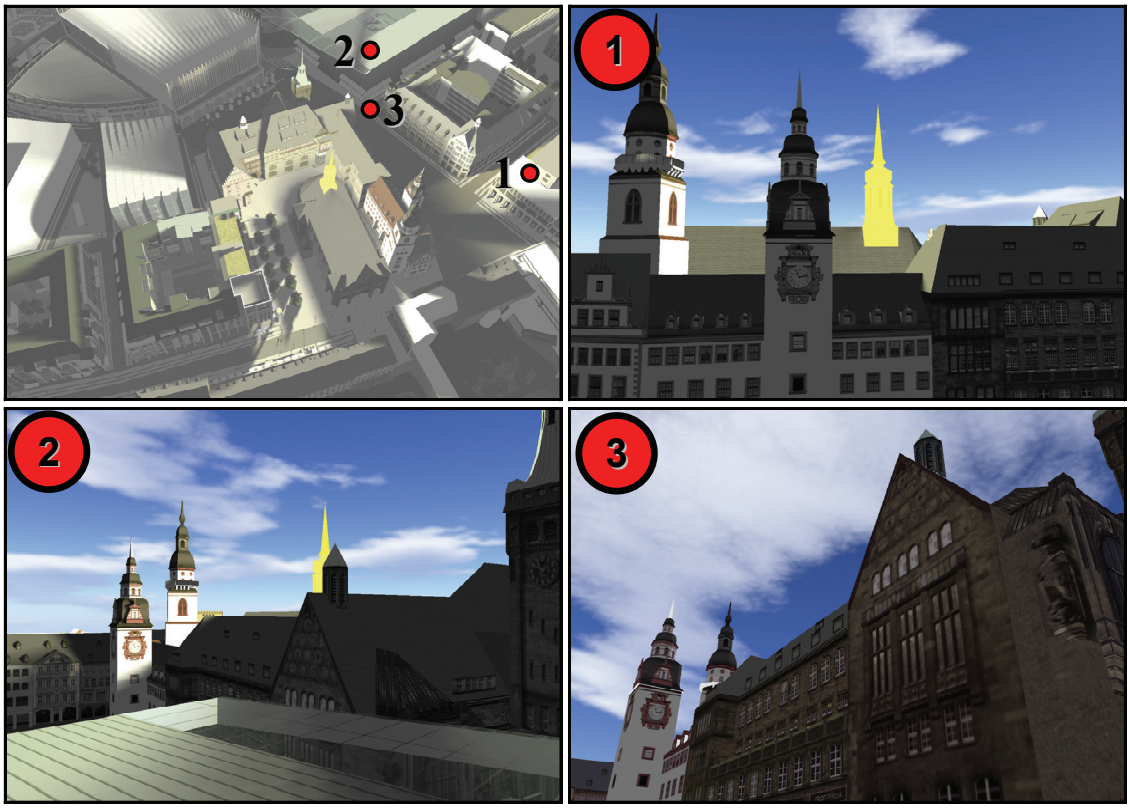


Figure 35: Computation of a visibility values for three different example points.

by the original surface textures as explained in Section 5.1.3. To compute $f(p)$, the scene is rendered in a conventional way using the graphics hardware but without displaying the result on the screen. For this, the rendering system is set up in the following way:

- The position of the virtual camera is set to p.
- The viewing direction is set to $(m\text{-}p)/||m\text{-}p||$, where m denotes the midpoint of the axis-aligned bounding box $B(O)$ of O.
- The resolution of the temporary synthesized image is set to a small predefined value, for instance, 10×10 pixels. Width and height are chosen equally.
- The field of view angle φ is chosen equally for vertical and horizontal direction. φ as well as the far-plane distance, i.e., the distance of the back end of the view frustum, are chosen minimally with the property that all vertices of $B(O)$ are inside the view frustum.

When rendering a 3D object in OpenGL, occlusion queries [Segal and Akeley 2004, Kilgard 2004] provide efficient and hardware-supported counting of the pixels at which a previously rendered object is visible. Based on occlusion queries, the following steps are performed:

- The target object is rendered alone. The number of pixels covered by the object is determined by an occlusion query and considered as A_{total}.
- The environment, i.e., the scene without O, is rendered so that the depth-values are written to the z-buffer.
- The target object is rendered separately. Now, an additional occlusion query counts only those pixels of O that are not occluded by other objects. The result is considered as $A_{unoccluded}$.

The resolution of the temporary, synthesized image determines the number of different degrees of visibility that can be distinguished. For the example shown in Figure 34, a resolution of 16×16 has been used.

The implementation of f_{vis} can also be used for the precomputation of lighting textures by choosing a large quad above the city, representing the sky as the object of interest. In this configuration, the visibility computation works similarly to *ambient occlusion* [Landis 2002], a technique for lighting calculation in computer graphics. The effect of this technique can be seen in Figure 20. Since the visibility calculation described above is introduced primarily as an example to demonstrate the potential of individual textures for city model visualization, it is kept simple and, therefore, is not optimized in terms of precomputation time. For lighting textures, the computation can be significantly accelerated by using 'outside-in' approaches [Sattler et al. 2004], i.e., by determining the visibility of the environment from multiple points of the light source. Meanwhile, there are first real-time approaches, but since they work with limited precision and add significant computational effort during rendering, precomputed textures still have advantages, particularly for complex city models.

5.1.2 Texture Generation

Based on a given computational model f, the aim is to create individual textures for all triangles of a given 3D scene so that the color of each texture pixel c is computed by $f(p_c)$, where p_c is the unique surface point to which c will be finally mapped. In addition, the total set of surface points for which f is evaluated should be approximately uniformly distributed over the surface formed by all triangles of the scene. That is, since f will be evaluated once for each texture pixel, the number of texture pixels that are mapped to a triangle should be proportional to its area. Similar to the resolution of the leaf-node texture atlases of a texture atlas tree, the desired texture resolution can be specified by a minimum distance value d_{min}, for which the texture resolution shall be sufficient assuming both screen resolution and field of view angle to be previously known.

Before the textures can be created, a unique texture parameterization is needed, i.e., a set of initially blank textures and related texture coordinates that define an injective mapping from the texture images to the surface formed by the triangles of the scene. In general, finding a well-chosen texture parameterisation that does not waste texture space and does not introduce any distortions is a difficult problem [Wang et al. 2004]. For the implementation, a simple solution has been used that creates a separate texture image for each triangle. The resulting very large number of separate textures is uncritical because the textures are finally converted to a texture atlas tree. Nevertheless, more complex texture parameterization methods provide some advantages. Particularly, they try to guarantee that no discontinuities appear at shared edges of adjacent triangles.

Given a triangle $t = (v_0, v_1, v_2)$, a texture image and related texture coordinates (t_0, t_1, t_2) are computed as follows: The texture coordinates are set to $t_0 = (0, 0)$, $t_1 = (1, 0)$, and $t_2 = (1, 1)$. Knowing vertex positions and texture coordinates of t, the required width w and height h of the texture to be created for a given minimum distance can be computed in the same way as described for the leaf node atlases of the texture atlas tree in Section 4.5.2.

To avoid visual artifacts at the triangle borders, an additional texture border is introduced. That is, the texture resolution is increased by $2 \cdot b$ for a predefined border width, e.g., 8 pixels, and the texture coordinates of t are slightly shifted inwards. Hence, the new texture coordinates of t are $t_0 = (o_x, o_y)$, $t_1 = (1\text{-}o_x, o_y)$, and $t_2 = (1\text{-}o_x, 1\text{-}o_y)$. The offsets o_x and o_y are obtained by $o_x = b/(w+2b)$ and $o_y = b/(h+2b)$.

Finally, the midpoint of each pixel in the texture image, i.e., $(x + 0.5, y + 0.5)$ for each pixel $c = (x, y)$, is transformed to barycentric coordinates to obtain p_c, so that $f(p_c)$ can be evaluated to determine the color of c.

5.1.3 Multiresolution Texture Atlases with Multiple Texture Layers

The result of the texture creation step described in the previous paragraph is a large set of textures where each texture has been computed for an individual triangle. If the computed textures are the only textures to be considered, a texture atlas-tree can be created as described in Section 4.5. If the scene, however, contains additional detail textures, or if multiple texture layers are to be computed, a method is needed to apply multiple texture layers during rendering. A *texture layer* here denotes a set of textures for the given 3D scene and a set of related texture coordinates, so that for each triangle t of a subset of the scene (some triangles may be untextured) a single texture image and a single texture coordinate vector for its three vertices are defined. By extending the texture atlas tree in a way that multiple texture layers are simultaneously available for rendering, the texture layers can be switched or arbitrarily modulated at runtime, such as in Figure 34 where the brightness of the original scene textures is modulated according to the computed textures.

Creation of a Texture Atlas Tree for Multiple Texture Layers

Creating a separate texture atlas tree for each texture layer would be problematic because the geometry would be managed by each tree separately. The hierarchical scene subdivision would be different for each tree due to different texture parameterizations. Therefore, the texture atlas tree is only created for a single texture layer, and additional node atlases are created for each additional texture layer. The additional node atlases are created in such a way that the same texture atlas coordinates can be used for all texture layers.

Let be $L_0, \ldots, L_k$ a given set of texture layers. The first step is to compute a texture atlas tree only considering L_0. For rendering triangles at maximum resolution, the corresponding full-resolution texture images and texture coordinates are explicitly stored for each texture layer in each leaf-node of the tree. At this time, a node N contains only a texture atlas A_0 providing texture information from L_0. Therefore, the aim is to create texture atlases $A_1,\ldots,A_k$ for N that contain the corresponding texture information from $L_1,\ldots,L_k$.

For the following explanation, the following notations are needed:

- For a triangle t and a texture layer L_i, $s_i(t)$ denotes the source texture that is assigned to t by L_i.
- For a texture s in L_0, $R_0(s)$ denotes the corresponding rectangular region in A_0 to which s has been copied including texture border. For a texture atlas A_i for $i \in \{1,\ldots,k\}$ that has to be filled, the corresponding region is denoted by $R_i(s)$.
- For a triangle t, $R_i(t)$ denotes the corresponding triangular region in a texture atlas A_i that is mapped to t according to the given texture atlas coordinates.

To provide valid texture data for a triangle t, $R_i(t)$ must be filled with texture data from a corresponding triangular region R' in $s_i(t)$. The atlas texture coordinates and the texture coordinates of L_i for t define a linear mapping m_t from $R_i(t)$ to R'. If t and the texture atlas coordinates of t are non-degenerate, m_t is well-defined. Otherwise, a constant mapping to an arbitrary point in R' can be chosen as m_t. Computing m_t for each triangle and filling A_i on a per-triangle basis would lead to two problems:

1. The texture mapping defined by L_0 might not be injective. In this case, a single texture atlas region might be mapped to multiple triangles. If two triangles t_1 and t_2 are sharing

a single texture atlas region R_i in A_i, but are differently textured in L_i, copying the texture data for t_2 to R_i would overwrite the texture data previously copied for t_1.

2. The texture borders in A_i would not be filled.

Both problems can be overcome by choosing a computed texture layer as L_0. The computed texture layers define a separate texture for each triangle. Therefore, they define an injective texture mapping except for degenerate triangles, which may be ignored. Moreover, for each texture s_0 in L_0, there is a unique triangle t, whose texture coordinates determine the texture mapping of s_0. That is, $R_i(s_0)$ is exclusively mapped to t for $i \in \{0,\ldots,k\}$. Hence, the linear mapping m_t can be extended to a linear mapping m_s that maps the whole region $R_i(s_0)$ to a region R' within $s_i(t)$. Using m_s, $R_i(s_0)$ can be filled with texture data copied from R'. In this way, the texture borders are also filled with valid texture data.

Having created additional texture atlases $A_1,\ldots,A_k$ for each leaf node, the corresponding lower-resolution texture atlases of an inner node can be computed more simply by downscaling and combining the texture atlases of its child nodes.

Rendering of a Texture Atlas Tree for Multiple Texture Layers

The result of the previously described extended preprocessing step is a texture atlas tree providing original textures and texture atlases for multiple texture layers $L_0,\ldots,L_k$. The way in which the additional texture layers are used for rendering can be varied at runtime. For instance, only considering textures of one of the layers enables switching between different texture layers at run-time.

Alternatively, multiple layers can be applied simultaneously, i.e., instead of determining the surface color based on a single texture, color values of multiple textures can be combined such as in Figure 20 (upper right image) and Figure 34 (lower image). Using fragment shaders [Rost 2004], the way in which multiple textures are combined to obtain the final surface color is freely programmable on current graphics hardware. Simple standard operations such as component-wise multiplication or additive blending of textures are also supported on older graphics cards that do not support fragment shaders yet. For terrain textures, the potential of combining multiple texture layers has been demonstrated by Döllner et al. [2000].

The fragment shader used in the example shown in Figure 20 (upper right image) modulates the texture layers L_0 (for the computed textures) and L_1 (for the original scene textures) in the following way:

$$c(i) = \text{scale}(l_0) \cdot l_1(i) \qquad \text{with} \quad \text{scale}(x) := \min(1.0, (b_{min} + x \cdot (b_{max} - b_{min}))\,,$$

where l_0 and l_1 denote the colors taken from the textures of L_0 and L_1, c denotes the final fragment color, , and the '( )'-operator the access to the i-th RGB-component for $i = 0,\ldots,2$. l_0 is treated as a single luminance value because f_{vis} does only produce gray-scale values. Since color components and luminance values are in the range [0, 1], multiplying l_0 and $l_1(i)$ directly would lead to a relatively dark overall appearance because the colors of the original texture could only be darkened or left unchanged but never brightened. The scale function rescales the values of the l_0 to a configurable range $[b_{min}, b_{max}]$.

In the example shown in Figure 34, a fragment shader has been implemented that affects the surface color by the following steps according to the computed visibility value:

- The surface brightness is varied in the same way as in Figure 20.
- The color saturation is varied according to the visibility values: Regions of visibility near zero are rendered in black and white, while regions of high visibility appear with unreduced color saturation.

- The surface color is slightly blended to yellow according to the visibility value to emphasize the relation to the fully yellow highlighted object of interest.

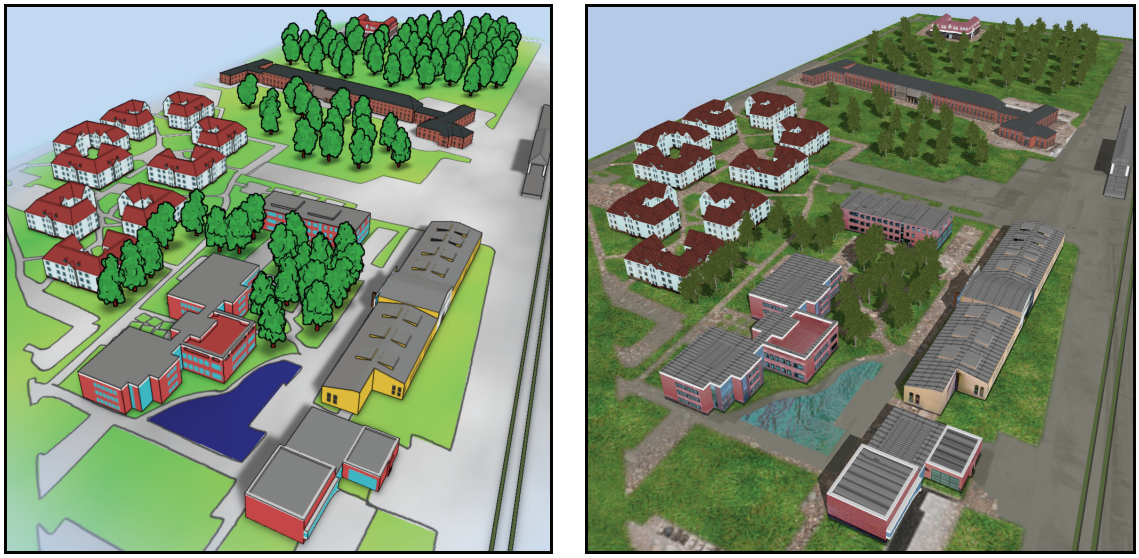
The hierarchical evaluation of the tree when rendering with multiple texture layers remains, basically, the same as for a single texture layer: Since the texture atlases $A_0,\ldots,A_k$ are sharing the same texture parameterization and have the same size, the resolution of A_0 is sufficient for a given view if and only if the resolution of A_i is sufficient as well for any $i \in \{1,\ldots,k\}$. Therefore, only slight changes are necessary to support multiple texture layers:

- *For node rendering with texture atlases*: The **`renderNode`** method in Figure 26, which renders the geometry of a node using its texture atlas, has to be slightly changed: Instead of applying a single texture atlas, the texture atlases for all active layers must be applied simultaneously using different texture units of the graphics hardware. The *active* texture layers are those that are to be currently considered for the determination of the surface color.
- *For node rendering with full texture resolution*: The method for rendering the geometry of a leaf node with full texture resolution must also activate the textures of all active texture layers. In addition, it becomes essential to optimize the rendering process for full-resolution textures by conventional texture atlases, due to two reasons:
  - If multiple texture layers are active, the number of necessary texture switches increases accordingly.
  - Since the computed texture layers define one texture per triangle, the resulting number of texture switches might become performance critical, even though it is only necessary for a small number of leaf nodes.
- *Memory Management*: The memory management must consider the additional texture layers. For this, full-resolution textures and texture atlases of additional texture layers are managed as independent blocks with their own priorities. All textures of inactive texture layers, i.e., those that are currently not activated for rendering, are assigned to the **`Unused`** category. To all textures of active texture layers the same priority is assigned, which is determined as for a single texture layer. Finally, during rendering, a node is considered to be in memory if and only if the textures of all active texture layers are currently in memory.

5.2 Illustrative Textures for Non-Photorealistic Visualization

Textures also provide a potential for non-photorealistic visualization. For instance, non-photorealistic facade textures can be obtained from photorealistic textures by applying image-based stylization techniques [DeCarlo and Santella 2002]. In this case, the requirements on the amount of texture data are the same as for photorealistic rendering. Alternatively, illustrative facade textures can be created during rendering [Döllner et al. 2005(i)]. Texture creation at runtime, however, involves additional performance costs and is restricted to real-time enabled algorithms. Using multiresolution texture atlases, an arbitrary amount of textures can be precomputed or manually created in advance.

To demonstrate the potential of textures for non-photorealistic rendering, this section describes a method to create an interactive virtual bird's-eye-view map, in which illustrative textures are combined with additional non-photorealistic elements. The approach will be demonstrated for a campus map of the Hasso-Plattner-Institute. Figure 36 compares the virtual bird's-eye-view map (a) with the 3D model from which it has been created (b).



a) b)

Figure 36: Comparison of a virtual bird's-eye-view map (a) and the 3D model used as input data (b).

The required input data for the approach consists of:

- *A digital terrain model*, either in the form of a grid or a TIN.
- *A vector-based plan*, consisting of a set of 2D polygons in the horizontal plane that form a partition of the ground into different cells such as streets, lawns, or water areas.
- *Textured 3D building models* with photographic facade textures, for instance, given in the form of a CityGML building model in LOD-2 or LOD-3.
- *3D Vegetation models,* defined via a set of georeferenced points and a set of photorealistic 3D vegetation models that are referenced by these points.

The technique can be subdivided into the following parts:

- Precomputation of an illustrative ground texture.
- Creation of building facades.
- Precomputation of smooth shadows.
- Non-photorealistic rendering of trees.
- Enhancement of building edges.

For the enhancement of building edges, the approach of Nienhaus and Döllner [2003] has been used. The other steps are described in the following.

5.2.1 Non-Photorealistic Ground Textures

The aim of this step is to create a high-resolution ground texture that shows a vector-based plan in an illustrative way. Serving as stylistic elements, edge enhancement, geometric smoothing, and color variations are used as illustrated in Figure 37. First, vector objects are directly rasterized into a texture (Figure 37a). Next, the polygon outlines are rasterized as anti-aliased lines (Figure 37b). The strictly straight lines and the precisely visible creases at sharp polygon corners make the result still inappropriate for the use as an illustrative texture. To avoid this, each line segment is replaced by a Bezier curve that connects its endpoints (Figure 37c) so that each line loop is transformed into a C^1-continuous closed curve that roughly approximates the line loop. For this, for each vertex v_i of a line loop, the tangent-vector direction is chosen as

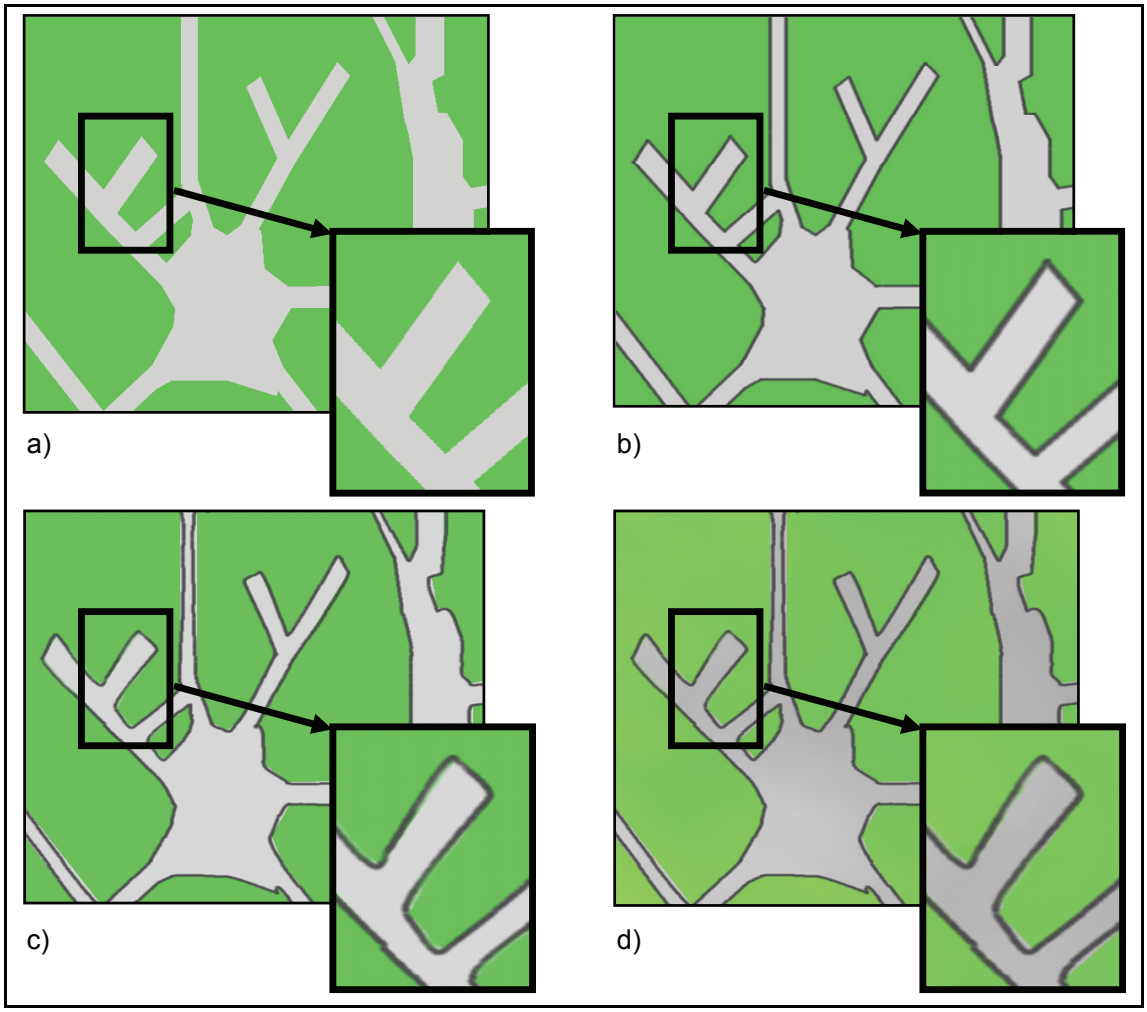


Figure 37: Generation of non-photorealistic ground textures. a) Rasterized vector data. b) Polygon boundaries as anti-aliased lines. c) Lines represented by Bezier segments. d) Introduced color irregularities.

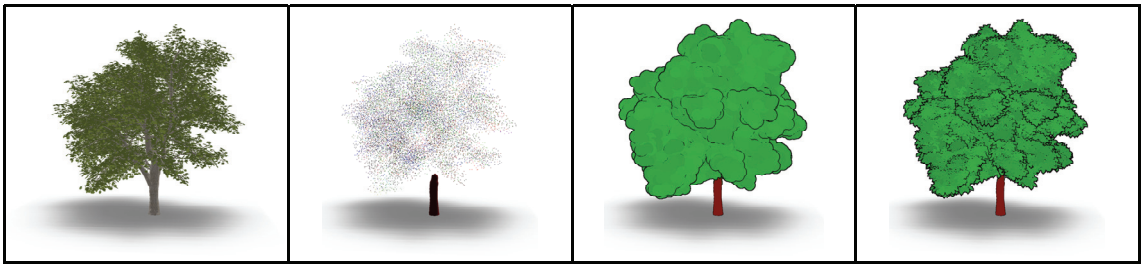
(v_{i+1}-v_i) for both connected segments. The tangent lengths control the deviation of the Bezier curve from the original lines. Finally, the polygons are filled with slightly varying colors (Figure 37d) created using Perlin-noise [Perlin 1985].

5.2.2 Building Facades

In non-photorealistic depictions, facade textures are used to convey the primary visual properties of building facades. In contrast to photorealistic visualization, small visual details of building facades are not desirable, because they increase the visual complexity of the rendered image without contributing important information. Therefore, the facade textures in a visual bird's-eye-view map are reduced to three primary visual properties:

- Average facade color
- Average window color
- Arrangement and size of windows.

If the building models are given as CLOQ buildings [Döllner and Buchholz 2005] or as CityGML buildings in LOD3 or LOD4, information about windows is explicitly available.



a) b) c) d)

Figure 38: Non-photorealistic rendering of trees. a) Original photorealistic model. b) Point-cloud derived from treetop. c) Result for circular point sprites. d) Result for leaf-shaped point sprites.

Otherwise, the generalized facade textures must be created based on photographic facade textures. For the campus model in Figure 36, this step has been done manually. For large city models, it becomes necessary to investigate semi-automatic or fully-automatic approaches to convert photographic facade textures to illustrative images, for instance, based on rectangle detection [Jung and Schramm 2004]. Extraction of structural information from facade photographs, however, is a general problem that is beyond the scope of this work.

5.2.3 Smooth Shadows

Shadows represent an important depth cue for the visual perception of 3D scenes. Therefore, they are not only useful for photorealistic visualization, but constitute relevant visual elements in non-photorealistic depictions as well. For the virtual bird's-eye-view map, shadows of buildings and trees are geometrically computed by projecting all shadow-casting shapes to the corresponding shadow receiving shapes. The shadows are then rasterized into surface textures. Finally, the rasterized shadows are smoothened by a Gaussian image filter. Using shadows as a depth cue does not require precise shadow geometry. Due to the filter, shadows appear only as slightly adumbrated without drawing too much attention to their precise shape.

5.2.4 Non-Photorealistic Tree Rendering

Trees in the virtual bird's-eye-view map are derived from 3D tree models, which, generally, are designed for photorealistic display, e.g., Figure 38a). There are various tools available for modeling organic structures, e.g., Xfrog or Bionatics. Using these models as a basis for non-photorealistic vegetation allows for systematically deriving a variety of different styles and variants for each vegetation object.

The technique for illustrative rendering of vegetation described here has been developed and implemented in collaboration with Marc Nienhaus and is based on his previous work on edge enhancement [Nienhaus and Döllner 2003]. A requirement for a 3D tree model to be processed is the ability to separate stem and branches from foliage; most modeling tools support this separation. Alternatively, triangles of the foliage can be heuristically identified based on the average colors of assigned textures.

The non-photorealistic rendering of tree models preserves the rough shape of the treetop while discarding small details such as singular leafs. In a preprocessing step, each triangle of the treetop is replaced by its midpoint, resulting in a point cloud (Figure 38b).

Tree models created for photorealistic rendering, typically, have high geometric complexity, so that the point cloud is quite dense. For instance, the treetop of the model in Figure 38a) contains 85,000 triangles.

The density of the point cloud affects the final rendering performance for the trees. Therefore, the geometry is reduced by two preprocessing steps:

- Clusters of nearby vertices are unified.
- Inner vertices of the treetop are removed, since they do not contribute to the primary shape of the tree.

To determine which points are considered to be inner points, the point cloud is rendered from multiple viewing directions into a temporary image. By applying hardware occlusion queries for each point, it can be efficiently determined which points are never visible, independent of the viewing direction. Those points are removed. The resolution of the temporary image controls the degree of simplification.

The run-time part of the algorithm that evaluates the point cloud will not be explained in greater detail here, because it was, primarily, a contribution of Marc Nienhaus: At run-time, all points of the simplified treetop point-cloud are rendered as point sprites, i.e., 2-dimensional quads of constant size on the screen. The fragments produced for these quads are processed by a fragment shader that creates color variations and emphasizes depth discontinuities by edges. Different transparency textures for the quads allow for creating different appearances as shown in Figure 38c) and d).

Chapter 6

CONCLUSIONS

This chapter discusses advantages and limitations of the multiresolution texture atlases approach, summarizes the results of this thesis, and offers proposals for future work.

6.1 Discussion

In this section, advantages of multiresolution texture atlases are pointed out, followed by a discussion of technical limitations and proposals for future improvements.

6.1.1 Advantages of Multiresolution Texture Atlases

The key advantages of multiresolution texture atlases can be summarized as follows:

- Using the memory management strategy described in Section 4.6, multiresolution texture atlases make it possible to render scenes in real-time for which the amount of simultaneously visible textures exceeds the main memory capacity.
- It keeps the number of texture switches per frame permanently low, even for 'worst case perspectives', for instance, if the whole scene is visible at once, but at the same time several parts of the scene are near to the camera. As shown by the comparison in Section 4.7.2, the overhead for texture switches is almost completely eliminated.
- By the pixel-per-texel parameter, the approach provides a runtime-controllable trade-off between required image quality and rendering performance. This makes it useful for a wider range of different graphics hardware.
- Since the major complexity of the approach is located in the preprocessing part, the runtime overhead for the rendering algorithm is small.
- It does not make any assumptions about the way the user acts within the virtual environment, i.e., it does not favor or restrict any specific navigation techniques.
- It does not make any assumptions about the structure of the given 3D scene. Particularly, in contrast to previous approaches, it does explicitly address the problem that texture workload is possibly irregularly distributed in a 3D scene.
- Due to the priorities defined in Section 4.6, it provides the most urgently needed textures for a given view very quickly. That is, only a small amount of texture data has to be loaded at the startup of an application.

6.1.2 Limitations and Future Improvements

This section discusses technical aspects that could be investigated in future works to achieve further improvements of the proposed rendering approach..

Reducing Geometric Workload

The assumption that the geometric complexity of a scene can be handled by the graphics hardware is acceptable for a wide range of scenes: If geometry is processed in an optimized way by using vertex-buffer objects of OpenGL, current graphics hardware is capable of rendering scenes of a geometric complexity of multiple million triangles at interactive frame rates. In addition, for textured scenes the memory requirement of geometry is, typically, considerably smaller than that required for textures. Nevertheless, probably the most relevant limitation of multiresolution texture atlases is the fact that the approach is only scalable in terms of texture complexity. A promising direction for future works is to investigate how to combine multiresolution texture atlases with techniques to handle geometrically complex scenes. For this, the following proposals could be investigated:

- *Geometric Simplification*: Several approaches have been proposed to simplify a complex triangle mesh in a way that the visual result is similar to the original mesh according to certain heuristic quality criteria such as geometric deviation [Hoppe 1996], texture stretch [Sander et al. 2001], or lighting conditions [Williams et al. 2003]. Generally, geometric simplification could be used in combination with multiresolution texture atlases: Instead of using the original geometry for all tree levels, simplified versions of the scene geometry could be stored in the inner nodes of the texture atlas tree. For the case of city models, however, geometric simplification is probably not appropriate: Geometric simplification works best for smooth continuous surfaces. Detailed city models usually consist of large numbers of small disjoint geometric objects, so that drastic simplification frequently causes visual artifacts.
- *Billboard Clouds*: A *billboard cloud* [Décoret et al. 2003] is a set of partially transparent textured quads that is used to replace a geometrically complex object at large camera distances. The key idea of billboard clouds is to replace several nearby and approximately coplanar triangles by a single quad whose texture is obtained by projecting all triangles onto the quad. There are different approaches to determine how to choose these quads in order to keep the number of necessary quads as well as the amount of necessary texture data as small as possible, proposed for instance, by Meseth and Klein [2004] and Huang et al. [2004]. As a main advantage compared to geometric simplification, billboard clouds do not assume any topological properties of the 3D input model, which makes them more suitable for city models. A key problem of billboard clouds is the property that the required texture data of a billboard cloud is, usually, considerably higher than that of the original model. In future work, billboard clouds could be integrated into the multiresolution texture atlases by storing billboard cloud representations for the geometry of a node. Billboard clouds can be computed in such a way that the geometric distance between a triangle and its projection is always smaller than a predefined geometric error threshold ε. Given a minimum camera distance, ε could be computed in a way that the corresponding projected screen projection of ε is always smaller than one pixel. For this, some open questions have to be investigated. For instance, it must be determined how to integrate the billboard clouds into the hierarchy and how to adapt the scene subdivision with respect to both, texture and geometry. Particularly, for some models, billboard cloud representations tend to cause some visual artifacts such as small holes in the model. Therefore, it would

be useful to search for specialized billboard-cloud techniques that work well for building models.

Reducing the Texture Atlas File Size

The precomputed file that stores the texture hierarchy as well as the original textures of a texture atlas tree, usually, comprises several gigabytes. A possible way to reduce the file size is to increase the atlas coverage, i.e., to reduce the amount of unused texture atlas areas by investigating how to arrange the textures more efficiently. As demonstrated in Section 4.7.1, however, the size of the file is largely caused by the original scene textures because even the resolution of the leaf-node atlases is considerably smaller, so that the optimization potential of improved atlas-packing schemes is rather limited.

Compared to the full-resolution textures, the texture atlases constitute a considerable fraction of the file size only if the represented scene contains several textures with high repetition counts such as the Chemnitz city model in Section 4.7.2. Although the scene can, nevertheless, be interactively rendered, it is desirable to avoid the need for replicating textures multiple times in the texture atlas. As described in Section 2.1, the combination of texture atlases and repeated textures is a general problem. Instead of texture replication, repeated textures in a texture atlas could be handled by a fragment shader as proposed by Wloka [2004]. Integrating such a shader in the rendering process of multiresolution texture atlases would reduce file size and preprocessing time at the cost of additional computational effort involved by the fragment shader and the restriction to recent graphics hardware.

Smoothing Image Transitions

Full-resolution textures are, usually, rendered with activated mipmapping. Hardware-generated mipmap textures do not exactly correspond to the texture atlases provided by the texture atlas tree. Mipmaps as well as atlases are created from the same full-resolution textures, but are resampled in different ways. Concerning the image quality, this is not necessarily a problem. As shown by the comparison in Figure 23c) and d), it is even possible that some scene parts appear sharper with multiresolution texture atlases. A slight disadvantage, however, is caused by the difference between original textures and leaf-node texture atlases. Due to this difference, transitions between full-resolution textures and leaf-node texture atlases might become noticeable. Therefore, it should be investigated how to make the transition completely invisible, e.g., by developing a fragment shader that introduces a continuous blending between the different texture resolutions.

Avoiding Texture Pollution

The insertion of texture borders of 8 pixels width to avoid texture pollution, as mentioned in Section 4.5.2, provides a practical solution and avoids obvious visual artifacts involved by texture pollution. This method, however, represents a compromise between wasted texture space and image quality: If downsampling of textures is performed by a simple 2×2 box filter, the borders avoid merging of texels of adjacent textures for three downsampling steps.

A future work could address the problem how to achieve a definitely full avoidance of texture pollution without the need of considerable additional texture atlas space. The problem of texture pollution is not restricted to multiresolution texture atlases, but applies to texture atlases in general, as discussed in [Wloka 2004]. This is due to reasons:

1. Since mipmapping is usually applied during rendering, downsampled versions of texture atlases are created for the lower mipmap-levels.
2. Bilinear filtering determines the color of a fragment from up to four texels. Thus, if a single texture of the resolution $m \times n$ was originally used with texture coordinates

outside the range $[0.5/m,\ 1\text{-}0.5/m] \times [0.5/n,\ 1\text{-}0.5/n]$, the corresponding texture coordinates for a texture atlas cause access to texels outside the area to which the texture has been copied. Moreover, if lower mipmap levels are accessed, i.e., downsampled by a factor 2^k, the problem becomes even worse: To avoid texture pollution completely, the original texture coordinates of a single texture would have to be completely within a range of $[2^k \cdot 0.5/m,\ 1\text{-}2^k \cdot 0.5/m] \times [2^k \cdot 0.5/n,\ 1\text{-}2^k \cdot 0.5/n]$ because the footprint of the bilinear filter, i.e., the texture area from which pixels are merged, is scaled by a factor of 2 for each mipmap level in horizontal and vertical direction.

If a single texture is located at an area $[x, x + w] \times [y, y + h]$, and x, y, w, h are divisable by a factor 2^k, downsampling is possible k times without texture pollution. In the ideal case, all textures are of equal size and the size is a power of 2. Unfortunately, this restriction contradicts the requirement of the computed appropriate texture sizes which form part of the multiresolution texture atlases approach. Increasing all textures to achieve sufficient resolution as well as to meet the above requirement would strongly increase the required texture atlas size. Therefore, this method cannot be directly applied to multiresolution texture atlases.

As stated in Wloka [2004], texture pollution caused by bilinear filtering can possibly be addressed by a fragment shader that shifts texture coordinates slightly inwards for each texture according to the currently used mipmap-level. As stated by the author, however, the proposal is only a theoretical one, it has not yet been implemented, and would probably involve high performance costs during rendering due to the complexity of the fragment shader. Therefore, whether at all and in which way it is useful to integrate such a shader into the multiresolution texture atlases approach remains to be investigated.

Computing the Minimum Distance Value

Finding an appropriate choice of a suitable minimum distance parameter d_{min} manually is, usually, not a fundamental problem because d_{min} is simply a distance within the 3D model and no abstract value. Nevertheless, it would be desirable to compute a useful value for d_{min} automatically taking into account the following criteria:

- The estimated maximum time required for rendering all objects within a distance of d_{min} from a given viewpoint in the scene when using full-resolution textures. This time increases for increasing d_{min} values.
- The number of necessary texture atlases, which increases for decreasing d_{min} values.
- The number of necessary triangle splits to avoid critical triangles (see Section 4.5.3), which increases for decreasing d_{min} values.

Since it does not make sense to store all texture atlases at a higher resolution than the original textures, a lower bound for the range of useful d_{min} values can be computed automatically: By setting w_{tex} in the texture size calculation in Section 4.5.2 to the original width of a texture image T, the formulas in Section 4.5.2 can be resolved to obtain the d_{min} value d_T, for which the horizontal texture atlas resolution is equal to the original texture resolution. The same applies for h_{tex} and the vertical texture atlas resolution. Computing the minimum over all d_T values for all textures and for height and width respectively obtains a value d_{full} so that for $d_{min} = d_{full}$ each original texture is represented at its original resolution or higher in a leaf-node texture atlas. Hence, d_{min} values below d_{full} are not useful, and if it is possible to choose $d_{min} = d_{full}$, the original textures are no longer needed. Always setting d_{min} automatically to d_{full}, however, would be an unstable approach: If only a single small triangle is covered with a texture at inappropriately high resolution in the original scene, this triangle determines d_{full}, forcing all other textures to be stored at the same resolution. Therefore, to achieve a fully automatic computation of an appropriate choice for d_{min}, more robust approaches should be investigated.

Handling of Critical Triangles

For both test models in Section 4.7, the approach to handle critical triangles described in Section 4.5.3 did not significantly increase the geometric complexity of the model. Theoretically, however, a 3D scene could be constructed in such a way that the handling of critical triangles became problematic. Strongly varying sizes of triangles alone do not necessarily cause a problem: If only a few triangles are drastically larger than the remaining ones, they can be split; if only a few triangles are drastically smaller than the remaining ones, they do not need to be considered for the choice of d_{min}. If a scene contains, however, two or more large groups of triangles at strongly different scales, the resulting splitting of triangles would involve a significant increase of geometric complexity. Such special cases could be addressed by investigating more sophisticated ways for the avoidance of critical triangles. For instance, two or more groups of triangles at approximately the same scale could be detected in advance and assigned to separate texture atlas trees with different d_{min} values.

6.2 Summary

The presented work has developed concepts and techniques for real-time city model visualization. In particular, it has introduced and evaluated multiresolution texture atlases, a new real-time rendering technique to handle city models of high complexity concerning texture size and number of textures. The test results show that two general restrictions of current graphics hardware have been overcome:

- *Limited texture memory*: As shown in Section 4.7.1, multiresolution texture atlases make it possible to render a city model containing 1.25 GB texture data at interactive frame rates using a graphics card with 128 MB graphics memory, whereby the main memory consumption of the required texture data can be restricted to 100 MB during rendering.
- *Texture switch overhead*: As shown by the comparison in Section 4.7.2, overhead due to texture switches is almost completely eliminated by multiresolution texture atlases.

Furthermore, two new approaches for real-time city model visualization have been presented:

- *Visualization of thematic information*: The technique described in Section 5.1 enables visual communication of complex thematic information in an intuitive way. Its application has been demonstrated for the example of precomputed visibility information. By integrating different computational models into the texture creation step, various other kinds of thematic data such as noise levels or average brightness values can be visualized.
- *Illustrative visualization of city models*: The technique described in Section 5.2 facilitates the automatic creation of illustrative, interactive bird's-eye-view maps. Compared to photorealistic display, interactive bird's-eye-view maps simplify the perception, comprehension, and communication of geospatial information. As a main characteristic, they expose high scalability, in particular, they can be reduced to thumbnail images without dramatic loss of information while still being clear and comprehensible. This makes them particularly useful for applications on mobile devices.

Complementing the main contributions described above, the work has also addressed additional related aspects of real-time city model visualization. It has introduced a concept to simplify the construction of complex urban terrain models and has discussed techniques to support intuitive user navigation in real-time city model visualization applications.

6.3 Outlook

Multiresolution texture atlases overcome a fundamental restriction in real-time visualization of city models. They represent a core functionality to use the whole surface areas of city model entities such as buildings and traffic objects as a kind of projection canvas for arbitrary multi-layered high-resolution raster data. As a consequence, multiresolution texture atlases enable a number of innovative visualization techniques that allow for new applications and systems in the domain of city model visualization. Three application domains have been discussed:

- Photorealistic visualization;
- Visualization of thematic data resulting from computational models;
- Illustrative visualization.

For the application domain of urban redevelopment, the potential of high-resolution textures for visualization of thematic building information is still to be explored. Presenting thematic building information based on surface textures is possible in a variety of ways. Therefore, it should be investigated how to achieve advantages compared to simple approaches such as visualization using building colors.

Multiresolution texture atlases also have certain properties that make them useful for streaming textured 3D scenes via Internet: First, the memory management strategy reduces the amount of transferred texture data, because high-resolution textures are only loaded when they are actually needed at full resolution. Second, texture resolution is incrementally refined. That is, if textures are streamed over a slow network connection, a small amount of low-resolution textures is provided very quickly, and the resolution of the currently most important textures is then improved stepwise.

Multiresolution texture atlases have been developed for real-time rendering of static models. The approach, however, has also an important property that could be exploited in applications in which textures are interactively changed: For given camera settings, the texture atlas tree provides a small set of texture atlases that provide texture data for all visible scene parts while the original textures at full resolution might require several gigabytes. That is, if a large amount of texture data is about to be changed at once, this change can be immediately previewed by applying the changes only to the texture atlases that are used for the current view. To make texture changes permanent, they could, then, be propagated to the remaining texture atlases and, finally, applied to the original textures. The potential of multiresolution texture atlases for interactive texture modification is still to be investigated.

REFERENCES

1. Amtliches Topographisch-Kartographisches Informationssystem (ATKIS). https://www.atkis.de
2. Appleton, K., Lovett, A., Sünnenberg, G., Dockerty, D.: Rural Landscape Visualization from GIS: A Comparison of Approaches, Options, and Problems. *Computer, Environment and Urban Systems*, 26, pp. 141-162, Elsevier, 2002.
3. Asirvatham, A., Hoppe, H.: Terrain Rendering Using GPU-Based Geometry Clipmaps. *GPU Gems 2*, pp. 27-45, Addison-Wesley, 2005.
4. Bacher, U., Mayer, H.: Automatic Extraction of Trees in Urban Areas from Aerial Imagery. *International Archives of Remote Sensing and Photogrammetry*, Vol. 33, Part B 3/1, pp. 51-57, GITC, 2000.
5. Baillard, C., Zisserman, A.: Automatic Reconstruction of Piecewise Planar Models from Multiple Views. *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, pp. 559-565, IEEE Computer Society Press, 1999.
6. Baumann, K.: *Modellierung, Texturierung und Rendering digitaler Geländemodelle*. Ph. D. Thesis, University of Münster, 2000.
7. Behrendt, S., Colditz, C., Franzke, O., Kopf, J., Deussen, O.: Realistic Real-Time Rendering of Landscapes Using Billboard Clouds. *Proceedings of Eurographics 2005*, 10, pp. 507-516, Blackwell Ltd., 2005.
8. Bill, R., Zehner, M.L.: *Lexikon der Geoinformatik.* Wichmann, 2001.
9. Bollmann, J., Koch, W.G. (eds.): *Lexikon der Kartographie und Geomatik*. Spektrum, 2002.
10. Brenner, C.: Building Reconstruction from Images and Laser Scanning. *International Journal of Applied Earth Observation and Geoinformation*, 6(3-4), pp. 187-198, Elsevier, 2005.
11. Buchholz, H., Bohnet, J., Döllner, J.: Smart and Physically-Based Navigation in 3D Geovirtual Environments. *Proceedings of the 9th International Conference on Information Visualization (IV'05)*, pp. 629-635, IEEE Computer Society Press, 2005.
12. Buchholz, H., Döllner, J.: *Visual Data Mining in Large-Scale 3D City Models*. GIS Planet 2005 - International Conference on Geographic Information, published on CD, 2005 (i).
13. Buchholz, H., Döllner, J.: View-Dependent Rendering of Multiresolution Texture-Atlases. *Proceedings of IEEE Visualization 2005*, pp. 215-222, IEEE Computer Society Press, 2005 (ii).
14. Buchholz, H., Döllner, J., Ross, L., Kleinschmit, B.: Automated Construction of Urban Terrain Models. *Proceedings of the 12th International Symposium on Spatial Data Handling (SDH 2006)*, pp. 547-562, Springer-Verlag, 2006.
15. Burtnyk, N., Khan, A., Fitzmaurice, G., Balakrishnan, R., Kurtenbach, G.: StyleCam: Interactive Stylized 3D Navigation using Integrated Spatial & Temporal Controls.

Proceedings of the 15th Annual ACM Symposium on User Interface and Technology (UIST), pp. 101-110, ACM Press, 2002.

16. Carr, N.A., Hart, J.C.: Meshed Atlases for Real-Time Procedural Solid-Texturing. *ACM Transactions on Graphics*, 21(2), pp. 106-131, ACM Press, 2002.
17. Chen, C., Czerwinski, M., Macredie, R.: Individual Differences in Virtual Environments - Introduction and Overview. *Journal of the American Society for Information Science*, 51(6), pp. 499-506, Wiley InterScience, 2000.
18. Chen, Y., Knapp, S.: VEPS - Virtual Environmental Planning System - First Steps towards a Web-based 3D-Planning and Participation Tool. *Proceedings of CORP2006 & Geomultimedia06*, pp. 275-285, 2006.
19. Claes, J.: *Real-Time Water Rendering*. Master Thesis, Lund University, 2004.
20. Cline, D., Egbert, P.K.: Interactive Display of Very Large Textures. *Proceedings of IEEE Visualization 1998*, pp. 343-350, IEEE Computer Society Press, 1998.
21. Cruz-Neira, C., Sandin, D.J., DeFanti, T.A.: Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 135-142, ACM Press, 1993.
22. Darken, R.P., Sibert, J.L.: A Toolset for Navigation in Virtual Environments. *Proceedings of the 6th ACM Symposium on User Interface Software and Technology (UIST)*, pp. 157-165, ACM Press, 1993.
23. DeCarlo, D., Santella, A.: Stylization and Abstraction of Photographs. *ACM Transactions on Graphics*, 21(3), pp. 769-776, ACM Press, 2002.
24. Decaudin, P., Neyret, F.: Rendering Forest-Scenes in Real-Time. *Proceedings of the 15th Eurographics Workshop on Rendering Techniques*, pp. 93-102, Eurographics Association, 2004.
25. Décoret, X., Durand, F., Sillion, F., Dorsey, J.: Billboard Clouds for Extreme Model Simplification. *Proceedings of ACM SIGGRAPH 2003*, pp. 689-696, ACM Press, 2003.
26. DeLeon, V.J., Berry, H.R.: Virtual Florida Everglades. *Proceedings of Virtual Systems and Multimedia (VSMM98)*, pp. 458-463, IEEE Computer Society Press, 1998.
27. Döllner, J., Baumann, K., Hinrichs, K.: Texturing Techniques for Terrain Visualization. *Proceedings of IEEE Visualization 2000*, pp. 207-234, IEEE Computer Society Press, 2000.
28. Döllner, J., Hinrichs, K.: A Generic Rendering System. *IEEE Transactions on Visualization and Computer Graphics*, 8(2), pp. 99-118, IEEE Computer Society Press, 2002.
29. Döllner, J., Baumann, K., Kersting, O.: LandExplorer - Ein System für interaktive 3D-Karten. *Kartographische Schriften*, 7, pp. 67-76, Kirschbaum-Verlag, 2003.
30. Döllner, J., Walther, M.: Real-Time Expressive Rendering of City Models. *Proceedings of the 7th International Conference on Information Visualization (IV'03)*, pp. 245-250, IEEE Computer Society Press, 2003.
31. Döllner, J., Buchholz, H., Nienhaus, M., Kirsch, F.: Illustrative Visualization of 3D City Models, *Proceedings of SPIE - Visualization and Data Analysis 2005 (VDA 05)*, pp. 42-51, SPIE / IS&T, 2005 (i).

32. Döllner, J., Buchholz, H.: Continuous Level-of-Detail Modeling of Buildings in Virtual 3D City Models. *Proceedings of ACM GIS 2005*, pp. 173-181, ACM Press, 2005.

33. Döllner, J., Hagedorn, B., Schmidt, S.: An Approach towards Semantics-Based Navigation in 3D City Models on Mobile Devices. *Proceedings of the 3rd Symposium on Location-Based Services and TeleCartography*, Geowissenschaftliche Mitteilungen, Heft 74, pp. 171-176, TU Wien, 2005 (ii).

34. Dogan, R., Dogan, S., Altan M.O.: 3D Visualization and Query Tool for 3D City Models, *Proceedings of the 20th ISPRS Congress*, Volume XXXV, Part B3, pp. 559-563, GITC, 2004.

35. Ebert, D.S., Musgrave, F.K., Peachey, D., Perlin, K., Worley, S.: *Texturing and Modeling*, Third edition, Morgan Kaufman, 2003.

36. Erison, C., Manocha, D., Baxter, W.V. III: HLODs for Faster Display of Static and Dynamic Environments. *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pp. 111-120, ACM Press, 2001.

37. Finch, M.: Effective Water Simulation from Physical Models. *GPU Gems*, pp. 5-29, Addision-Wesley, 2004.

38. Fisher, S.S., McGreevy, M., Humphries, J., Robinett, W.: Virtual Environment Display System. *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, pp. 77-87, ACM Press, 1986.

39. Förstner, W.: 3D-City Models: Automatic and Semiautomatic Acquisition Methods. *Proceedings Photogrammetric Week '99*, pp. 291-303, Wichmann-Verlag, 1999.

40. Foley, J.D., Ribarsky, W.: Next-Generation Data Visualization Tools. *Scientific Visualization: Advances and Challenges*, pp. 103-127, Academic Press, 1994.

41. Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F., Phillips, R.L.: *Introduction to Computer Graphics*. Addison-Wesley, 1994.

42. Forberg, A.: Generalization of 3D Building Data Based on a Scale-Space Approach. *Proceedings of the 20th ISPRS Congress*, Vol. XXXV, Part B4, pp. 194-199, GITC, 2004.

43. Früh, C., Sammon, R., Zakhor, A.: Automated Texture Mapping of 3D City Models with Oblique Aerial Imagery. *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization, and Transmission*, pp. 396-403, IEEE Computer Society Press, 2004.

44. Früh, C., Zakhor, A.: An Automated Method for Large-Scale Ground-Based City Model Acquisition. *International Journal of Computer Vision*, 60, pp. 5-24, Kluwer Academic, 2004.

45. Fuhrmann, S., MacEachren, A.M.: Navigation in Desktop Geovirtual Environments: Usability Assessment. *Proceedings of the 20th ICA/ACI International Cartographic Conference*, pp. 2444-2453, 2001.

46. Gale, N., Golledge, R., Pellegrino, J.W., Doherty, S.: The Acquisition and Integration of Route Knowledge in an Unfamiliar Neighborhood. *Journal of Environmental Psychology*, 10, pp. 3-25, Elsevier, 1990.

47. Galyean, T.A.: Guided Navigation of Virtual Environments. *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pp. 103-104, ACM Press, 1995.

48. Gröger, G., Kolbe, T.H., Czerwinski, A.: *OpenGIS CityGML Implementation Specification (City Geography Markup Language)*. OGC Discussion Paper, online published, http://www.opengeospatial.org/standards/dp, 2006.

49. Haala, N., Brenner, C.: Virtual City Models from Laser Altimeter and 2D Map Data. *Journal of Photogrammetric Engineering and Remote Sensing*, 65(7), pp. 787-795, ASPRS, 1999 (i).

50. Haala, N., Brenner, C.: Extraction of Buildings and Trees in Urban Environments. *ISPRS Journal of Photogrammetry and Remote Sensing*, 54(2-3), pp. 130-137, Elsevier, 1999 (ii).

51. Haeberli, P., Segal, M.: Texture Mapping as a Fundamental Drawing Primitive. *Proceedings of 4th Eurographics Workshop on Rendering*, pp. 259-266, 1993.

52. Hand, C.: Survey of 3D Interaction Techniques. *Computer Graphics Forum*, 16(5), pp. 269-281, 1997.

53. Hanson, A.J., Wernert, E.A.: Constrained 3D Navigation with 2D Controllers. *Proceedings of IEEE Visualization*, pp. 175-182, IEEE Computer Society Press, 1997.

54. Helbing, R., Strothotte, T.: Quick Camera Path Planning for Interactive 3D Environments. Smart Graphics Demo Session, 2000.

55. Hesina, G., Maierhofer, S., Tobler, R.F.: *Texture Management for High-Quality City Walk-Throughs*. Technical Report No. 25, 2004.

56. Hoppe, H.: Progressive Meshes. *Proceedings of ACM SIGGRAPH 1996*, pp. 99-108, ACM Press, 1996.

57. Hoppe, H.: Smooth view-dependent Level-of-Detail Control and its Application to Terrain Rendering. *Proceedings of IEEE Visualization 1998*, pp. 35-42, IEEE Computer Society Press, 1998.

58. Huang, I.T., Novins, K.L., Wünsche, B.C.: Improved Billboard Clouds for Extreme Model Simplification. *Proceedings of Image and Vision Computing '04 New Zealand (IVCNZ '04)*, pp. 255-260, 2004.

59. Hwa, L.M., Duchainau, M.A., Joy, K.I.: Adaptive 4-8 Texture Hierarchies. *Proceedings of IEEE Visualization 2004*, pp. 219-226, IEEE Computer Society Press, 2004.

60. International Alliance for Interoperability (IAI): *IFC 2x2 - Industry Foundation Classes*. online published, http://www.iai-ev.de/spezifikation/Ifc2x2/index.htm, 2005.

61. Igarashi, T., Kadobayashi, R., Mase, K., Tanaka, H.: Path Drawing for 3D Walkthrough. *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology (UIST)*, pp. 173-174, ACM Press, 1998.

62. Igarashi, T., Cosgrove, D.: Adaptive Unwrapping for Interactive Texture Painting. *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pp. 209-216, ACM Press, 2001.

63. Jiang, B., Li, Z.: Geovisualization: Design, Enhanced Visual Tools and Applications. *The Cartographic Journal*, 42(1), pp. 3-4, 2005.

64. Jung, C.R., Schramm, R.: Rectangle Detection based on a Windowed Hugh Transform. *Proceedings of the 17th Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, pp. 113-120, IEEE Computer Society Press, 2004.

65. Kersting, O., Döllner, J.: Interactive Visualization of 3D Vector Data in GIS. *Proceedings of ACM GIS 2002*, pp. 107-112, 2002.

66. Kilgard, M.J. (Editor): *NVIDIA OpenGL Extension Specifications*, NVIDIA Corporation, online published, 2004.

67. Kirsch, F.: *Entwurf und Implementierung eines computergraphischen Systems zur Integration komplexer, echtzeitfähiger 3D-Renderingverfahren*. PH.D. Thesis, University of Potsdam, 2005.

68. Kiss, S., Nijholt, A.: Viewpoint Adaption during Navigation based on Stimuli from the Virtual Environment. *Proceedings of the 8th International Conference on 3D Web Technology*, pp. 19-26, ACM Press, 2003.

69. Kryachko, Y.: Using Vertex Texture Diplacement Mapping for Realistic Water-Rendering. *GPU Gems 2*, pp. 283- 294, Addision-Wesley, 2005.

70. Lakhia, A.: Efficient Interactive Rendering of Detailed Models with Hierarchical Levels of Detail. *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization, and Transmission*, pp. 275-278, IEEE Computer Society Press, 2004.

71. Landis, H.: Production-Ready Global Illumination. *ACM SIGGRAPH 2002 Course Notes 16*, pp. 87-101, ACM Press, 2002.

72. *Land of Ideas Initivative*. http://www.land-of-ideas.de

73. LandXplorer CityGML Tool, http://www.hpi.uni-potsdam.de/~doellner/projekte/landxplorer_citygml_viewer.html

74. Laycock, R.G., Day, A.M.: Automatically Generating Roof Models from Building Footprints. *Proceedings of WSCG*, Poster Presentation, 2003.

75. Laycock, R.G., Day, A.M.: Automatic Techniques for Texture Mapping in Virtual Urban Environments. *Proceedings of Computer Graphics International 2004*, pp. 148-155, 2004.

76. Li, Z., Zhu, Q., Gold, C.: *Digital Terrain Modeling: Principles and Methodology*. CRC Press, 2004.

77. Lindstrom, P., Koller, D., Hodges, L.F., Ribarsky, W., Faust, N., Turner, G.: *Level-of-Detail Management for Real-time Rendering of Phototextured Terrain*. Technical report GIT-GVU-95-06, 1995.

78. Lindstrom, P., Pascucci, V.: Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3), pp. 239-254, IEEE Computer Society Press, 2002.

79. Lefebvre, S., Darbon, J., Neyret, F.: *Unified Texture Management for Arbitrary Meshes*. INRIA Research Report No. 5210, 2004.

80. Lefebvre, S., Hoppe, H.: Parallel Controllable Texture Synthesis. *Proceedings of ACM SIGGRAPH 2005*, pp. 777-786, ACM Press, 2005.

81. *Longman Dictionary of Contemporary English*. Langenscheidt / Longman Group, 1987.

82. Luebke, D., Erikson, C.: View-Dependent Simplification of Arbitrary Polygonal Environments. *Proceedings of ACM SIGGRAPH 1997*, pp. 199-208, ACM Press, 1997.

83. MacEachren, A.M., Edsall, R., Haug, D., Baxter, R., Otto, G., Masters, R., Fuhrmann, S., Qian, L.: Virtual Environments for Geographic Visualization: Potential and Challenges.

Proceedings of the 1999 Workshop on new Paradigms in Information Visualization and Manipulation, pp. 35-40, 1999.

84. Mackinlay, J.D., Card, S.K., Robertson, G.G.: Rapid Controlled Movement through a Virtual 3D Workspace. *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 171-176, ACM Press, 1990.

85. Meseth, J., Klein, R.: Memory Efficient Billboard Clouds for BTF Textured Objects. *Proceedings of the Vision, Modeling, and Visualization Conference 2004 (VMV 2004)*, pp. 167-174, Aka GmbH, 2004.

86. Möller, T., Haines, E.: *Real-Time Rendering*. A. K. Peters, 1999.

87. Muhar, A.: Three-Dimensional Modeling and Visualization of Vegetation for Landscape Simulation. *Landscape and Urban Planning*, 54(1-4), pp. 5-17, Elsevier, 2001.

88. Müller, P., Wonka, P., Haegler, S., Ulmer, A., Van Gool, L.: Procedural Modeling of Buildings. *Proceedings of ACM SIGGRAPH 2003*, pp. 614-623, ACM Press, 2006.

89. Nienhaus, M., Döllner, J.: Edge-Enhancement - An Algorithm for Real-Time Non-Photorealistic Rendering. *Journal of WSCG*, 11(2), pp. 346-353, 2003.

90. Nienhaus, M.: *Real-Time Non-Photorealistic Rendering Techniques for Illustrating 3D Scenes and Their Dynamics*. PH.D. Thesis, University of Potsdam, 2005.

91. Oliveira, M., Bishop, G., McAllister, D.: Relief Texture Mapping. *Proceedings of SIGGRAPH 2000*, pp. 359-368, ACM Press, 2000.

92. Perlin, K.: An Image Synthesizer. *Proceedings of ACM SIGGRAPH 1985*, 19(3), pp. 287-296, ACM Press, 1985.

93. Preussner, G.: *Effiziente Speicherung von Impostern für das Echtzeit-Rendering komplexer 3D-Szenen*. Master Thesis, University of Rostock, 2004.

94. Qiu, F., Zhao, Y., Fan, Z., Wei, X., Lorenz, H., Wan, J., Yoakum-Stover, S., Kaufman, A., Mueller, K.: Dispersion Simulation and Visualization for Urban Security. *Proceedings of IEEE Visualization '04*, pp. 553-560, 2004.

95. Ribarsky, W., Wasilewski, T., Fast, N.: From Urban Terrain Models to Visible Cities. *IEEE Computer Graphics and Applications*, 22(4), pp. 10-15, IEEE Computer Society Press, 2002.

96. Rost R.J.: *OpenGL Shading Language*. Addison-Wesley, 2004.

97. Salomon, B., Garber, M., Lin, M.C., Manocha, D.: Interactive Navigation in Complex Environments using Path Planning. *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics*, pp. 41-50, ACM Press, 2003.

98. Samet, H.: The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16, pp. 187-260, ACM Press, 1984.

99. Sander, P., Snyder, J., Gortler, S., Hoppe, H.: Texture Mapping Progressive Meshes. *Proceedings of ACM SIGGRAPH 2001*, pp. 409-416, ACM Press, 2001.

100. Sattler, M., Sarlette, R., Zachmann, G., Klein, R.: Hardware-Accelerated Ambient Occlusion Computation. *Proceedings of the Vision, Modeling, and Visualization Conference 2004 (VMV 2004)*, pp. 331-338, Aka GmbH, 2004.

101. Schaufler, G.: Dynamically Generated Impostors. *Proceedings of the Workshop Modeling - Virtual Worlds - Distributed Graphics (MVD'95)*, pp. 129-136, 1995.

102. Schilling, A., Zipf, A.: Generation of VRML City Models for Focus Based Tour Animations. Integration, Modeling and Presentation of Heterogeneous Geo-Data Sources. *Proceedings of 8th International Conference on 3D Web Technology Web3D'2003*, pp. 39-47, ACM Press, 2003.

103. Schumann, H., Müller, W.: *Visualisierung - Grundlagen und allgmeine Methoden.* Springer-Verlag, 2000.

104. Segal, M., Akeley, K.: *The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004)*, Silicon Graphics Inc., 2004.

105. Senay, H., Ignatius, E.: A Knowledge-Based System for Visualization Design. *IEEE Computer Graphics and Applications*, 14(6), pp. 36-37, IEEE Computer Society Press, 1994.

106. Shah, M.A., Kontinnen, J., Pattanaik, S.: Real-Time Rendering of Realistic-Looking Grass. *Proceedings of GRAPHITE 2005*, pp. 77-82, 2005.

107. Shi, W., Yang, B., Li, Q.: Integrated Dynamic Model for Multi-Resolution Data in Three Dimensional GIS. *Proceedings of the Joint International Symposium on Geospatial Theory, Processing, and Applications*, 2002.

108. Sormann, M., Zach, C., Karner, K.F.: Texture Mapping for View-Dependent Rendering. *Proceedings of the 19th Spring Conference on Computer Graphics*, pp. 131-148, ACM Press, 2003.

109. *Stadtumbau Ost*. http://www.stadtumbau-ost.info

110. Suzuki, S., Chikatsu, H.: Recreating the Past City Model of Historical Town Kawagoe from Antique Map. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, Vol. XXXIV, Part 5/W10, 2003.

111. Tan, D.S., Robertson, G.G., Czerwinski, M.: Exploring 3D Navigation: Combining Speed-Coupled Flying with Orbiting. *Proceedings of the CHI 2001 Conference on Human Factors in Computing Systems*, pp. 418-425, ACM Press, 2001.

112. Tanner, C.C., Midgal, C.J., Jones, M.T.: The Clipmap: A Virtual Mipmap. *Proceedings of ACM SIGGRAPH 1998*, pp. 151-158, ACM Press, 1998.

113. Thiemann, F.: Generalization of 3D Building Data. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, Vol. XXXIV, Part 4, 2002.

114. Tsingos, N., Gallo, E., Drettakis, G.: Perceptual Audio Rendering of Complex Virtual Environments. *ACM Transactions on Graphics*, 23, pp. 249-258, 2004.

115. *The Virtual Reality Modeling Language (VRML)*. International Standard ISO/IEC 14772-1:1997 and ISO/IEC 14772-2:2002.

116. Wahl, R., Massing, M., Degener, P., Guthe, M., Klein, R.: Scalable Compression and Rendering of Textured Terrain Data. *Journal of WSCG*, 12(3), pp. 521-528, UNION Agency - Science Press, 2004.

117. Ware, C.: *Information Visualization*. Morgan Kaufman, 2000.

118. Wei, L.: Tile-Based Texture-Mapping. *GPU Gems 2*, pp. 189-199, Addison-Wesley, 2005.

119. Wernert, E.A., Hanson, A.J.: A Framework for Assisted Exploration with Collaboration. *Proceedings of IEEE Visualization*, pp. 241-248, IEEE Computer Society Press, 1999.

120. Williams, N., Luebke, D., Cohen, J.D., Kelley, M., Schubert, B.: Perceptually Guided Simplification of Lit, Textured Meshes. *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, pp. 113-121, ACM Press, 2003.

121. Wimmer, M., Bittner, J.: Hardware Occlusion Queries Made Useful. *GPU Gems 2*, pp. 91-108, Addison-Wesley, 2005.

122. Wloka, M.: Improved Batching via Texture Atlases. *ShaderX3*, pp. 155-167, Charles River Media, 2005.

123. Wonka, P., Wimmer, M., Sillion, F., Ribarsky, W.: Instant Architecture. *Proceedings of ACM SIGGRAPH 2003*, pp. 669-677, ACM Press, 2003.

124. *Extensible 3D (X3D)*. International Standard ISO/IEC FDIS 19775-1:2004.

125. Xiao, D., Hubbold, R.: Navigation Guided by Artificial Force Fields. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 179-186, ACM Press/Addison-Wesley, 1998.

EHRENWÖRTLICHE ERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Dissertation ohne Hilfe Dritter und ohne Zuhilfenahme anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Potsdam, den 29. September 2006

_____

Henrik Buchholz