Towards Interactive Landscape Visualization

vorgelegt von
Diplom-Informatiker
Malte Clasen
aus Berlin

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
Dr. ing.

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Olaf Hellwich
Berichter: Prof. Dr. Marc Alexa
Berichter: Hans-Christian Hege

Tag der wissenschaftlichen Aussprache: 13.10.2011

Berlin 2011

D 83

Zusammenfassung

In dieser Dissertation stellen wir die Komponenten eines interaktiven Landschaftsvisualisierungssystems mit Schwerpunkt auf Gelände und Vegetation vor. Zuerst beschreiben wir die Datenquellen eines typischen Landschaftsvisualisierungsszenarios, in dem Geländemodell und Luftbilder aus einem Geoinformationssystem (GIS) exportiert und um Pflanzenverteilungen ergänzt werden. Dieser Teil setzt die Rahmenbedingungen für folgenden Methoden, vorgestellt in Reihenfolge der Anwendung:

Level-of-Detail-Stufen (LoD) für Pflanzen müssen nur ein einziges mal pro Modell erzeugt werden, unabhängig vom jeweiligen Visualisierungsprojekt. Wir stellen eine Methode vor, die auf Linien und Ellipsoiden basiert. Mit Hilfe des Expectation-Maximization-Algorithmus auf einem Gaussian-Mixture-Model erstellen wir eine Hierarchie qualitativ hochwertiger Blatt-Cluster-Gruppen. Die Aststrukturen vereinfachen wir mit einem agglomerativen Clustering beginnend bei der höchsten Auflösung, um die Konnektivität zu erhalten. Die Vereinfachung erfolgt in einem Vorverarbeitungsschritt und erfordert keinerlei menschliche Eingriffe. Für einen Flug über und durch eine Szene aus 10 000 Bäumen erreichen wir mit unserem LoD eine Geschwindigkeit von durchschnittlich 40 ms pro Bild, was ungefähr sechs mal schneller ist als Billboard Clouds mit vergleichbaren Bildfehlern.

Als nächstes beschreiben wir, wie die räumlichen Daten eines Landschaftsvisualisierungsprojekts geladen und organisiert werden können. Wir zeigen eine konzeptionell einfache Verarbeitungskette, die Datendekompression und -synthese zur Laufzeit in einem vereinheitlichten Prozess handhabt. Als Geländedatenquellen kommen beispielsweise statische Satellitenbilder, texelweise Bildberarbeitungsschritte wie Überblendung oder auch leichtgewichtige Simulationen und Synthesen wie Texturgeneratoren in Frage. Punktdaten wie Pflanzeninstanzen und polygonale Formen wie Gebäudegrundrisse können in denselben Datenstukturen verarbeitet werden. Die Datenquellen werden parallel in Abhängigkeitsketten ausgewertet.

Aufbauend auf den sich daraus ergebenden Geländetexturen stellen wir

einen auf Clipmaps basierenden Rendering-Algorithmus für sphärische Gelände vor. Wir nutzen den hohen Geometriedurchsatz aktueller Grafikkarten um große Mengen statischer Dreiecke darzustellen. Die Vertices werden dabei über Höhentexturen verschoben. Unser Hauptbeitrag ist die Abbildung der Texturkoordinaten ausgehend von den statischen Vertex-Positionen und der variablen Ansicht auf die Höhentexturen.

Über das Gelände zeichnen wir als nächstes die vorbereiteten LoD-Stufen der Pflanzenmodelle. Dazu nutzen wir einen Raycaster für die Linien und Ellipsoide. Wir erweitern die Ellipsoide um Rauschtexturen für Alpha-Test und Normalenvektoren. Das erhöht den Realismus, ohne Aliasing durch Subpixel-Strukturen einzuführen. Weiterhin zeigen wir wie physikalisch basiertes Shading die Wahrnehmung der Tiefenkomplexität verbessert.

In einem letzten Schritt zeigen wir die Nachbearbeitung der erzeugten Bilder über Deferred Shading. Hier berechnen wir auch Schatten und atmosphärische Lichtstreuung.

Anschließend an die Methodenbeschreibungen beschreiben wir das sich daraus ergebende Landschaftsvisualisierungssystem aus Benutzerperspektive. Nach einem Überblick über die Funktionen stellen wir eien Evaluierung in einer Fallstudie für Klippenerosion vor.

Abstract

In this thesis we present the building blocks of an interactive landscape visualization system focussed on terrain and vegetation. First, we describe the data sources in a typical landscape visualization scenario, where terrain elevation and aerial images are exported from a geographic information system (GIS) and enriched with distributions of third-party plant models. This part sets the constraints for the following methods, presented in order of application:

Level of Detail (LoD) generation for the plants has to be done only once for each model, independent of the visualization project at hand. We present a method based on lines and ellipsoids. We leverage the Expectation Maximization algorithm with a Gaussian Mixture Model to create a hierarchy of high-quality leaf clusterings, while the branches are simplified using agglomerative bottom-up clustering to preserve the connectivity. The simplification runs in a preprocessing step and requires no human interaction. For a fly by over and through a scene of 10k trees, the resulting LoD can be rendered on average at 40 ms/frame, up to 6 times faster than billboard clouds with comparable artifacts.

Next we describe how to load and organize the spatial data for a landscape visualization project. We describe a conceptually simple pipeline that handles on-the-fly data decompression and synthesis in a unified process. Possible terrain data sources range from static satellite imagery over per-texel-processing such as image blending routines to light-weight simulations and synthesizers such as noise and filter based texture generators. Point data such as plant instances and polygonal shapes such as building outlines can be handled in the same data structures. The sources are evaluated in parallel based on dependency chains.

Given the resulting terrain textures, we describe a terrain rendering algorithm for spherical terrains based on clipmaps. It leverages the high geometry throughput of current GPU to render large static triangle sets. The vertices are displaced by a height map texture. Our main contribution is mapping of texture coordinates to calculate the height map sample position based on the static

vertex offset and the variable view position.

On top of the terrain, we render the preprocessed plant model LoD by ray-casting the line and ellipsoid primitives. We extend the ellipsoids by noise textures for alpha-test opacity and normal mapping. This yields a more realistic image, while still avoiding the aliasing artifacts of subpixel-sized primitives. We further show how physically based shading improves the perceived depth complexity.

As a last step rendering step, we postprocess the rendered image to apply deferred shading includig shadows and atmospheric scattering.

Following the methods, we describe the resulting landscape visualization system from a user's perspective. After an overview over the features, we present an evaluation in a case study of cliff erosion for climate change research.

Acknowledgments

This thesis would not have been possible without the invaluable input from various sources. Initial inspiration came from the Lenné3D project, funded by the Deutsche Bundesstiftung Umwelt (DBU, http://www.dbu.de/), where Liviu Coconu, Philip Paar and Hans-Christian Hege developed the landscape visualization tool Lenné3D Player. Liviu Coconu wrote the billboard cloud implementation I use as a reference. Philip Paar was always available for a comment on the user's perspective, which I hope to reflect in the overall design. Hans-Christian Hege had the clear vision that landscapes should definitely not be limited by an artificial end-of-the-world abyss, which was my motivation to write a spherical terrain renderer. The spin-off Lenné3D GmbH (http://www.lenne3d.com) and Jan Walter Schliep (http://www.wallis-eck.de/) provided me with plant models to experiment with. The plant file format has been developed by Carsten Colditz and Oliver Deussen.

Lenné3D was followed by the Silvisio project, funded by the Bundesministerium für Bildung und Forschung (BMBF, http://bmbf.de/, FKZ 0330560B). For this project, the Biosphere3D (http://www.biosphere3d.org) landscape visualization tool (the implementation of this thesis) has been developed. Wieland Röhricht (http://oik.de/) taught me quite a few things about vegetation, which resulted in the modular plant instance handling, and the general idea that millions of plant instances are required for a truly realistic image of a single view - from the ground. This motivated the development of the new level of detail method for plants for the Pergamon project, also funded by the BMBF (FKZ U809068). The Laubwerk GmbH (http://www.laubwerk.com) and Timm Dapper (http://www.timmdapper.de/) provided me with skeleton-based plant models and the accompanying loader and tesselator, which is the foundation of the primitive-specific simplification method.

Publishing the intermediate results was helped by Hans-Christian Hege, who co-authored [Clasen and Hege, 2005], [Clasen and Hege, 2006], and [Clasen and Hege, 2007]; Steffen Prohaska, who co-authored [Clasen and Prohaska, 2010]; Philip Paar with whom I worked on [Paar and Clasen, 2007], [Paar et al., 2008], and [Clasen and Paar, 2008]; Marc Alexa, who reminded me that

there's a bigger picture above the details of the current work at hand; and Irina Itschert, who carefully suggested that a thesis should be finished in finite time :)

While the core of Biosphere3D is described in this thesis, many features that make the user's and developer's lifes easier (including mine) were added by other people: Steffen Ernst is the lead developer of the user interface and the one who cares for our users. Ronny Günther helped writing the linear algebra code, Maria Gensel created the water surface. We use quite a few open source libraries which deserve credits: AGG, Boost, Bzip2, Cairo, CryptoPP, Curl, CurlPP, DevIL, Expat, Fontconfig, Freetype, GDAL, GLEW, FreeGlut, gSOAP, ICU, ImageDB, Jasper, JPEG/IJG, JsonC, LibECW, LibMNG, LibPNG, LittleCMS, Loki, LZMA, OpenEXR, OpenMesh, OpenNurbs, OpenSSL, StlSoft, Tiff, wxWidgets, Xerces, and Zlib.

The data we use for experiments includes the Landsat and Blue Marble texture sets and the Shuttle Radar Topography Mission data by NASA (http://earthobservatory.nasa.gov/) and the SRTM V3 by CGIAR-CSI (http://srtm.csi.cgiar.org/).

Contents

Chapter 1

Introduction

Landscape visualization is the transformation of a virtual model of a landscape into an image. This can be illustrated best by example: When a landscape planner works on a project, he usually uses a geographic information system (GIS) to design the landscape. This involves importing and editing the terrain, the vegetation and the man-made structures such as streets and buildings. The primary user interface metaphor of a GIS is a map, where all items are displayed as outlines or symbols. While this is an efficient concept for the landscape planners to work and to communicate with peers, it becomes an obstacle when laymens are involved, such as decision makers and interested citizens. They are usually not trained in reading maps, and even while they might understand the symbols, the map often does not transform into a mental image of the landscape as envisioned by the planner. This is where landscape visualization software excels: It allows to transform the map into an image, which minimizes the room left for interpretation and therefore misunderstandings [Haaren and Warren-Kretzschmar, 2006]. This has been pioneered as early as 200 years ago by Repton [1803], who drew the alternative scenarios by hand (Fig. 1.1).

Many users in science and engineering profit from landscape visualization. Landscape planners and landscape architects design modifications of existing terrain, and there is always a trade-off between the design goals. A retaining dam (Fig. 1.2) might be necessary to protect a village, but it also might reduce the recreational value of the valley. Whenever there is no obvious choice, landscape visualization can help getting an impression of the consequences. Climate researchers can calculate the movement of the shoreline due to erosion and draw it into a map, but the real impact is much more apparent in the comparison of images of now and then (Fig. 1.3). Archeologists, biologists, and forest researches encounter similar problems.

Depending on the use case, different kinds of landscape visualization are

(a) (b)

Figure 1.1: *Early landscape visualizations relied on drawn images to compare different scenarios.*

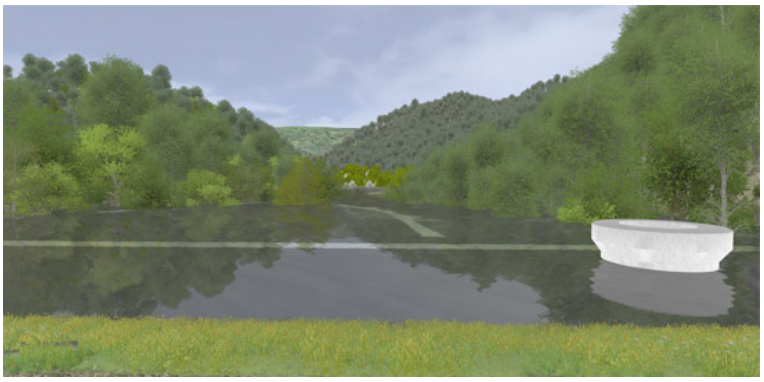


Figure 1.2: *Visualization of a planned flood control reservoir (source: Lenné3D).*

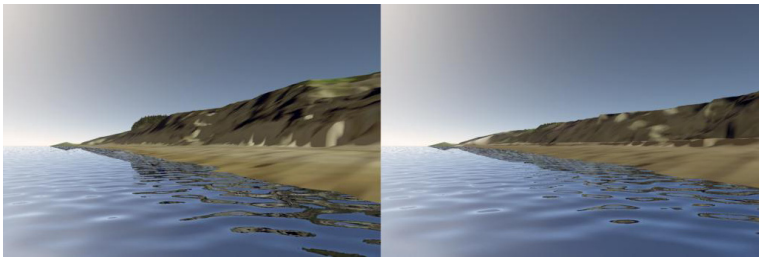


Figure 1.3: *Visualization of a coastal landscape change (source: [Paar et al., 2008]).*

appropriate. On the one hand, there are different design styles (see [Ervin, 2001]). Photorealistic images are best suited for direct comparisons to actual photographs of the current state of the landscape, for example to display the visual impact of a new powerplant. Non-photorealistic (NPR), sketchy styles convey the idea of work-in-progress scenarios or missing knowledge, for example due to archeological reconstructions based on few artifacts. On the other hand, there are different time budgets. First, there are landscape visualizations generated as the end result of a purely GIS-based workflow, where the final maps shall be displayed to decision makers. In this setting, offline renderers can take a up to a few hours for a single image of very high quality. This is the domain of many commercial solutions such as Eonsoftware Vue, Terragen, and vegetation add-ons for general purpose renderes like VRay. Second, there are pure viewers such as flight training simulators, which display static scenes at framerates matching the display system, usually 60 fps. They trade preprocessing time and limited image quality for the framerate constraint, which is often a hard requirement for the certification of the application. Third, there are workflow-oriented tools which are used by landscape planners while editing the project. They have to balance preprocessing time (to quickly load updated scenarios), interactive views (to set arbitrary view points at 15 fps or more), and image quality (to yield a reasonable realistic image, [Appleton and Lovett, 2003]). Even though the performance of commodity PCs steadily increased, this trade-off is still as important as in [Appleton et al., 2002]. Workflow-oriented tools allow landscape planners to work in a what-you-see-is-what-you-get environment, which reduces turn-around times for design decisions (see [Paar, 2006]).

In this thesis, we target workflow-oriented applications. In contrast to existing commercial solutions such as ESRI ArcGIS 3D Analyst (`http://www.esri.com/software/arcgis/extensions/3danalyst/index.html`), the Realtime Visualization module of 3D Nature's Visual Nature Studio (`http://3dnature.com/`), and Google Earth (`http://earth.google.com/`), we target a higher image quality, especially for vegetation as described by Paar [2006]. 3D Analyst (Fig. 1.4), Visual Nature Studio (Fig. 1.5), and Google Earth (Fig. 1.6) use billboards. In 3D Analyst and Visual Nature Studio, this is the only method to display plants. While it provides a hint of the planned vegetation, realism is quite limited. Google Earth employs a Level-of-Detail method which switches to coarse textured meshes in the foreground (Fig. 1.7). This is a step forward, but visual quality is way behind academic state-of-the-art methods such as used by Werner et al. [2005], over which Boudon et al. [2006] give an extensive overview.

In the following, we show how state-of-the-art methods can be improved and leveraged to create landscape visualization system with the following design goals:

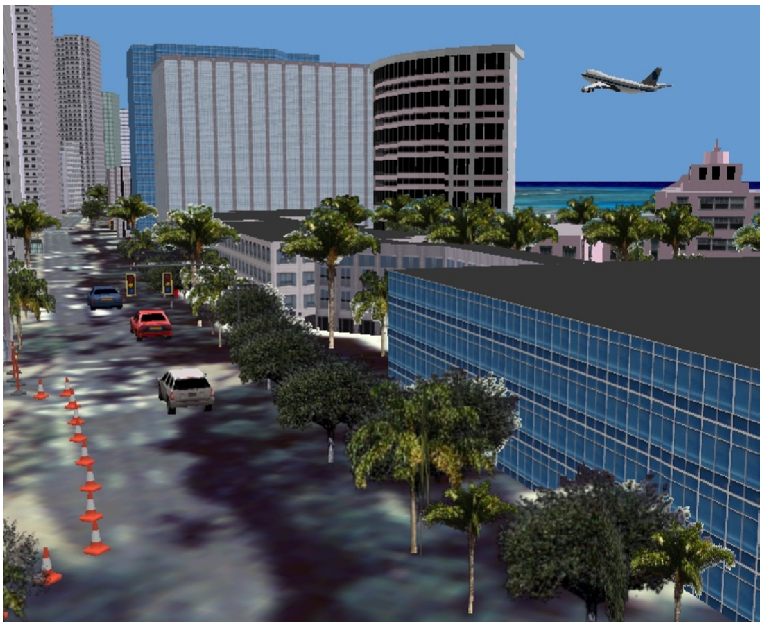


Figure 1.4: *ESRI ArcGIS 3D Analyst provides only rudimentary vegetation in form of few flat billboards (source: ESRI).*

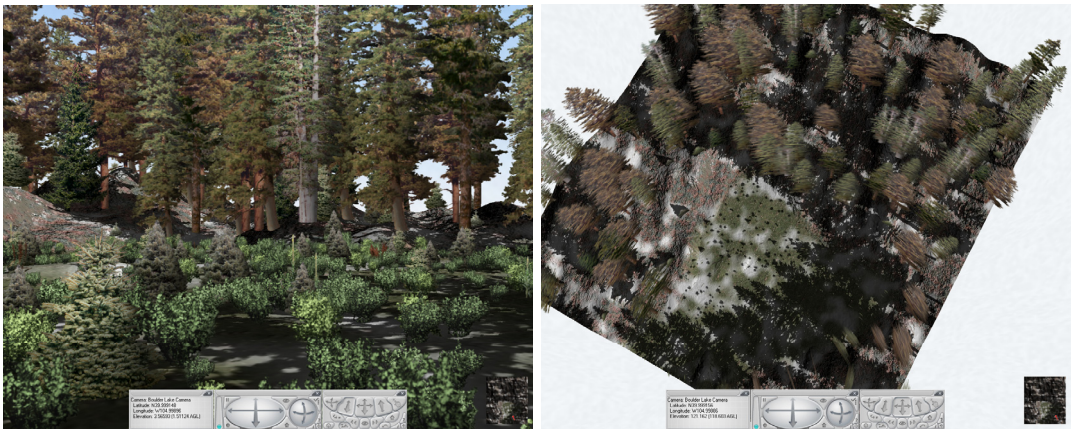


Figure 1.5: *Visual Nature Studio uses only flat up-axis-aligned billboards to display plants. While this works for horizontal perspectives (left), this approach does not work for top views (right).*

Figure 1.6: *Google Earth is capable of displaying hundreds of trees (left). To accomplish this, it relies on flat up-axis-aligned billboards for distant instances, which breaks the immersion for top views (right).*

Figure 1.7: *Google Earth switches to a coarse textured mesh in the foreground.*

- The image quality shall be reasonable high.
- The system shall enable interactive views.
- The system shall minimize the turn-around times for landscape planners. To accomplish this, precomputation of edited artifacts such as plant distributions shall be minimized, while it is allowed for rarely modified artifacts such as the plant models. This includes compatibility to common data formats such as terrain projections, where conversion is time consuming.
- The system shall run on commodity PCs.

Additionally we want a clean software architecture. This does not affect the user, but it increases the value of this thesis to the reader. Entangled algorithms and low-level optimizations can improve the performance, but at the end this thesis is not about the resulting binaries, but about the algorithms in a human readable form. We designed all methods to be used in a modular way. If you only need a Level-of-Detail method for trees, you can apply chapters 3 and probably 6. If you only need a terrain rendering front-end, read chapter 5. But if you want to build a landscape rendering from scratch, you still profit from the reuse of components, for example the unified tile handling for both terrain textures and plant instances in section 4.4. Following this design, we present the methods in closed form.

Chapter 2

Data Sources

Landscape visualizations consist of various elements with different properties. In this chapter, we present a set of typical elements, including terrain, 3D models and GIS features. We discuss each element both from a user's and from a developer's perspective. This is the foundation of the following chapters, where we present methods to render these elements interactively.

2.1 Terrain

Terrain is part of every landscape visualization. It determines the overall shape of the landscape, and most other elements are placed relative to the surface of the terrain. There are no inherent boundaries of the terrain due to the spherical topology of our planet. However, landscape planning is usually focussed on a bounded region. This is often reflected in the usage of multiple terrain models: A coarse model is used for the distant surroundings, while a high-resolution model is used for the focus region. The surroundings are usually left untouched, so precomputation time is not critical. Publicly available data such as the NASA Blue Marble color texture (Fig. 2.1) and the NASA Shuttle Radar Topography Mission (SRTM) digital elevation model (Fig. 2.2) is a common choice.

The focus region is usually not larger than a few dozen square kilometers. For this region, high-resolution terrain models for color and elevation are created or acquired (Fig. 2.3). These models are subject to change in the landscape planning workflow, so precomputation times can become relevant.

When the overlay of the various terrain data sources is computed in world space (in contrast to the view-dependent image space), a common coordinate system has to be used. For global data, unprojected latitude/longitude co-

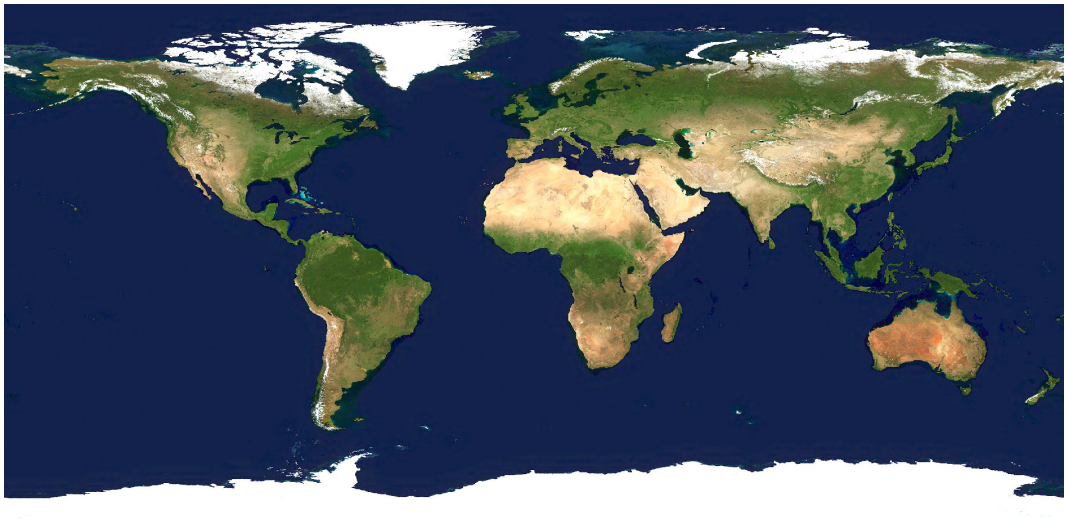


Figure 2.1: *The Blue Marble color texture covers the entire planet at a resolution of* 86400×43200*.*

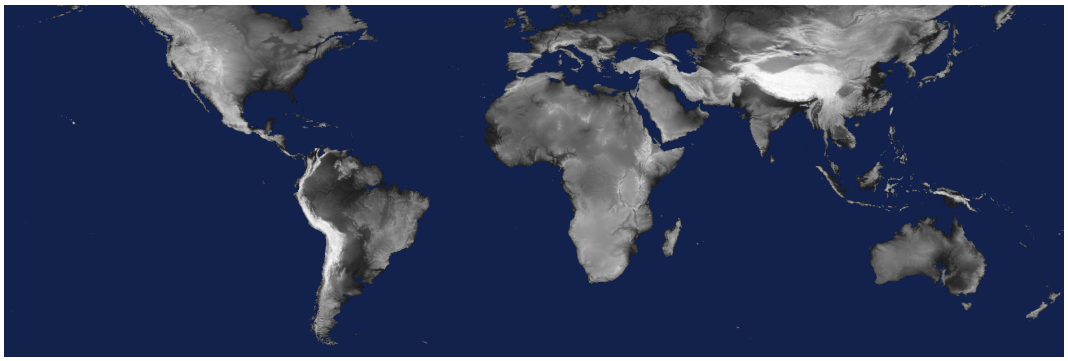


Figure 2.2: *The SRTM digital elevation model covers the latitudes between 60° North and 60° South at a resolution of* 432000×144000 *pixels.*

Figure 2.3: *Detailed images of the focus region enhance the surrounding terrain, up to 10 pixel per meter.*

ordinates based on the WGS84 reference ellipsoid are commonly used. Local data, on the other hand, is often projected to the local geographic coordinate system. In Germany, most official spatial data is available in the *Deutsches Hauptdreiecksnetz* (DHDN), while in Switzerland the thereto incompatible *Schweizer Landeskoordinaten* is used. When the amount of focus region data is small compared to the static surroundings, converting the focus region to latitude/longitude is feasible.

Apart from the coordinate systems, the primitive type is the second source of incompatibilities. On the one hand, there are raster formats, which organize color or elevation samples as images. This is common for satellite data, aerial photographies and laser scans. In this context, the term 2.5D denotes that for each 2D position, there is exactly one elevation value. When manually acquired data is used, it is often given as triangulated irregular network (TIN). This is the native format for manual measurements, and also widely supported in terrain editing tools. In contrast to 2.5D data, TINs can represent true 3D data including overhangs. Combining raster data and TINs requires the same decision as before, both have to be converted to a common format. Given that the static data is often available as raster data and much larger than the focus region data, a conversion to raster data is a common solution.

2.2 Models

Landscape visualization increasingly relies on 3D models of a multitude of objects [Ervin, 2001]. This includes man-made structures such as buildings and vegetation, but also features the underlying terrain data cannot represent, for example overhangs for 2.5D terrain models. We distinguish between models represented as general triangle meshes and plants.

2.2.1 Meshes

Triangle meshes are the most common interchange format for 3D models. All kinds of models relevant to landscape visualization can be represented as meshes (Fig. 1.4), and most tools in this context can display them. In practice, most meshes are used for buildings which are either numerous and simple (Fig. 2.5) or unique and complex (Fig. 2.4). The primary reason is that creating a detailed building is a laborious process, so buildings are created only as detailed as necessary. Stock models, where the overhead would be compensated over the frequent reuse, are often deliberately kept simple to avoid a false impression of accuracy for the project at hand. Triangle meshes have no inherent

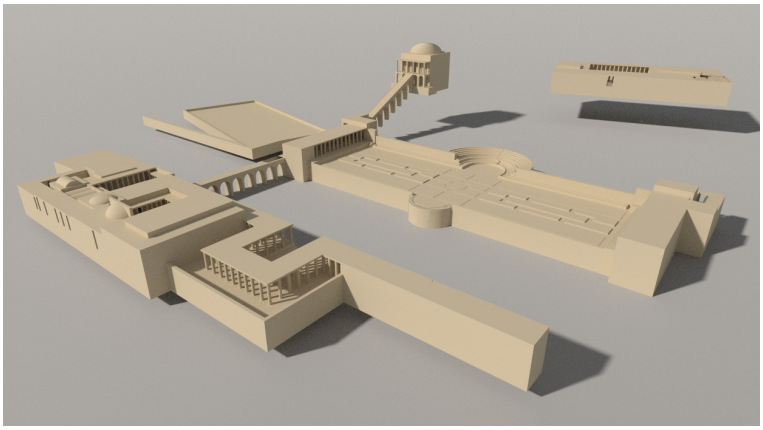


Figure 2.4: *This palace complex is modelled as a triangle mesh with 64 000 triangles (model: Jochen Mülder, Lenné3D).*

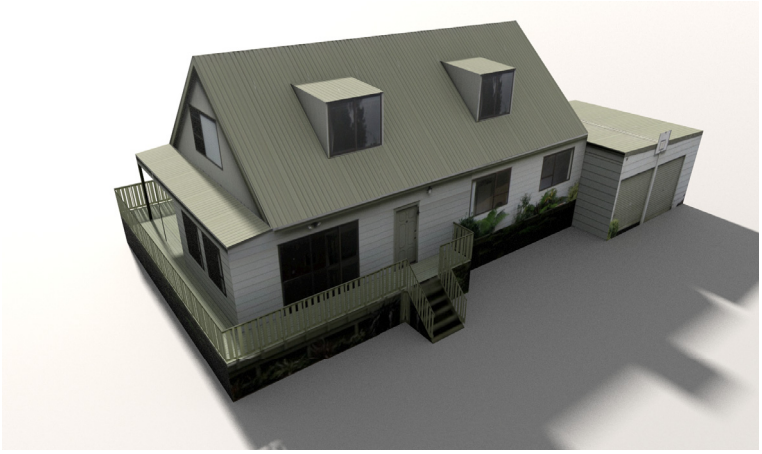


Figure 2.5: *This house consists only of 260 triangles (model: Google 3D Warehouse).*

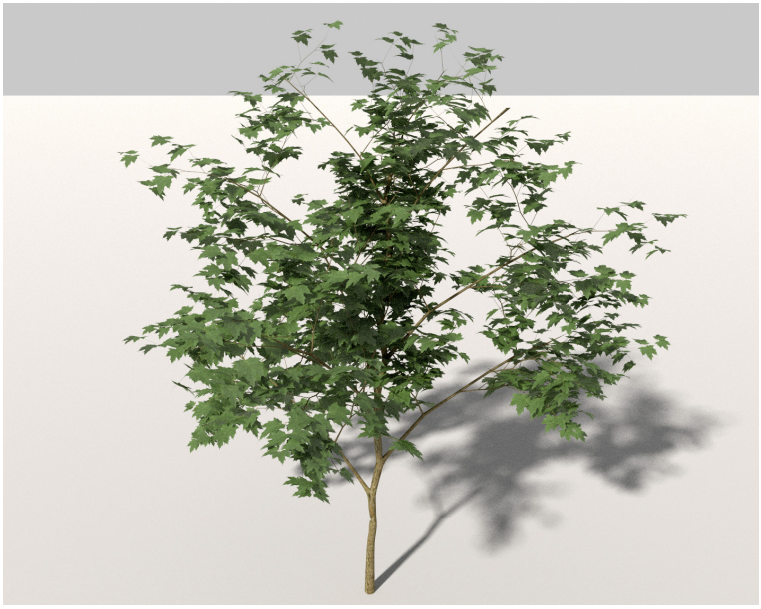


Figure 2.6: *In landscape visualization, the species of a plant can convey important information. This requires detailed models like this young Acer platanoides.*

level of detail mechanism. This is going to be a bottleneck when more efficient modeling tools become widely spread, but for now, GPU rendering performance is ahead of the modeling capacity of most landscape planners.

2.2.2 Plants

In contrast to buildings, plant models are effectively reused. Landscape planners usually rely on third party plant model libraries, such as Greenworks XfrogPlants (http://www.xfrog.com/) and Bionatics natFX (http://www.bionatics.com/natFX/). These libraries consist of ready-to-use 3D plant models, organized by species. In most cases it is unobtrusive to show the same model many times on screen, especially when rotated and slightly resized. Although depending on the requirements for botanical accuracy, it is relatively easy to create large amounts of plant instances when designing a landscape. But while the appearance of an individual instance is often insignificant, the chosen species shall be recognizable in sufficient detail by the interested viewer (Fig. 2.6). This is particularly important for trained audiences such as farmers and forest rangers. Large numbers of detailed plant models require a level of detail method to be displayed efficiently.

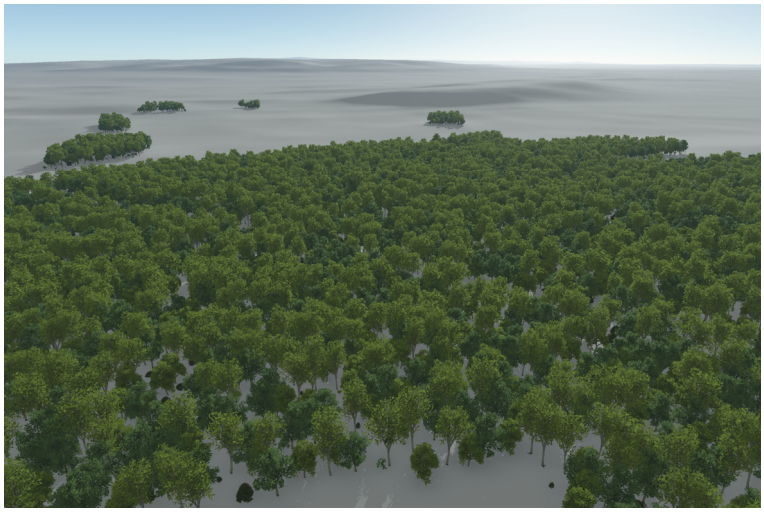


Figure 2.7: *Forests are created out of many instances of a few plant models.*

2.3 GIS Features

In the terminology of the Geography Markup Language (GML), a file format for geographic information systems (GIS), a feature represents a physical object, for example a building or a bridge. Features have no shape by themselves, they are associated with geometries. Points, lines, and polygons are common geometry types. We support GIS features as follows:

2.3.1 Instances

As suggested in section 2.2.2, plant models are usually instantiated multiple times, for example to create a forest (Fig. 2.7). A single tree instance can be stored as a GIS feature with an associated point geometry for the location. A link to the 3d model and transformations such as rotation and scaling can be stored as additional, non-spatial attributes. This allows managing large amounts of plants with common GIS applications, where point geometries are rendered as simple map symbols.

2.3.2 Building Outlines

When the buildings in a landscape shall remain vaguely specified, they can be represented as extruded outlines (Fig. 2.8). In this case, a house is given as a feature with an associated polygon geometry. Additional properties such as

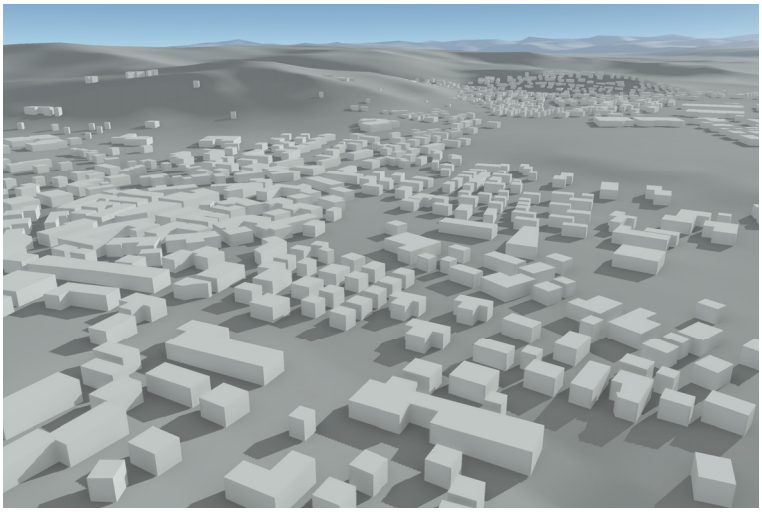


Figure 2.8: *When the details are not important, extruded shapes can be used to outline the buildings.*

height are optional. Using polygon shapes instead of point instances of unique triangle meshes reduces the overhead, since a list of polygon shapes can be batch processed in both the GIS application and the interactive viewer.

Chapter 3

Level of Detail for Vegetation

3.1 Introduction

Interactive plant visualization has many applications such as games, flight simulators, real-time preview for architectural modeling tools, and geovirtual visualization systems. The sheer numbers of plants of realistic densities of vegetation cover still pose a problem for interactive landscape rendering. In mainstream geographic information systems (GIS) used in landscape planning and in geo browsers such as Google Earth, vegetation cover can so far only be represented in a quite rudimentary way. In a survey by Paar [2006] on applications and requirements of 3d visualization software, landscape planners and landscape architects expressed their preoccupation with time-consuming rendering and insufficient visual representation of plants and habitats. Several studies such as [Lange, 2001] have demonstrated that the degree of detail, particularly for foreground features like vegetation, soil surface, or water, is a key factor on how people relate to such computer-generated visual simulations of landscape scenery. Particular difficulties occur in densely vegetated areas such as fields and forests, where specialized landscape rendering software such as E-on Software Vue or Planetside Software Terragen excel at visual quality, but at the cost of high turn-around times per image. Level of Detail (LoD) methods reduce the complexity of 3d models to limit both the resource usage (memory, time) and aliasing, enabling the rendering of visually rich scenes at interactive frame rates.

In this chapter, we present a LoD method especially suited for trees. Trees have a high geometric complexity with thousands of only loosely connected leaves. Therefore many common LoD methods, which focus on a reduction of surface complexity, cannot be applied. So while the simplification of buildings from sophisticated façades to flat rectangles is solved in numerous ways,

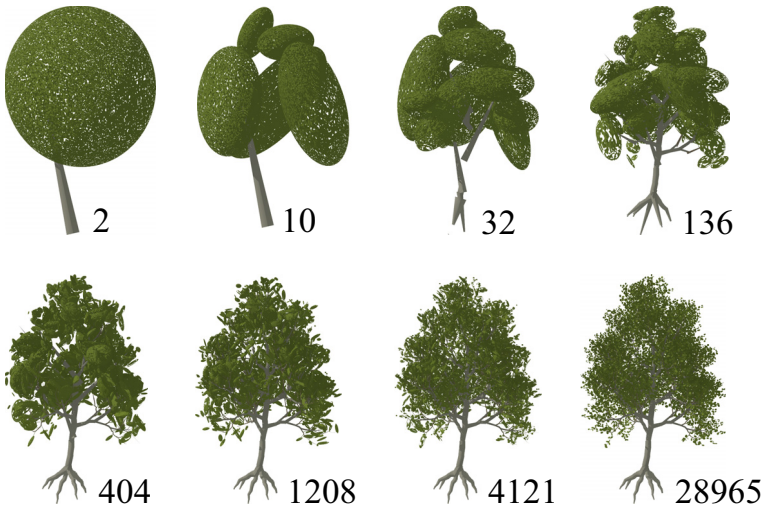


Figure 3.1: *Number of primitives for the levels of detail for a mesh with 122 566 triangles.*

rendering the vegetation surrounding the man-made structures is still a challenging task. There is still no method to render realistic (non-tiled) forests at interactive frame rates on commodity PCs. Our goal is to increase the performance of plant rendering as a step towards realistic vegetation density, without compromising image quality.

The following method is based on fuzzy clustering of unconnected leaf primitives, where each cluster is represented by a noise-textured ellipsoid. We implemented it on top of the solution proposed by Clasen and Prohaska [2010]. First, by making the clustering step explicit and choosing a better algorithm, we improve the accuracy of the lower LoD steps, which reduces the required number of primitives (Fig. 3.1) and therefore the GPU time. This is the main contribution of this chapter. Second, we apply a noise texture to the ellipsoids for surface normal and alpha-test opacity. This yields a rougher, more natural look than the perfect ellipsoids used by Clasen and Prohaska [2010]. Third, we propose a primitive-size-based LoD selection to reduce aliasing.

We start with a discussion of related work in section 3.2. In section 3.4 we present the clustering schemes for ellipsoids and lines, followed by the image error metric used in the line clustering step in section 3.5. We compare performance and image quality to previous methods in section 3.6 and discuss these results in section 3.7. The conclusion follows in section 3.8.

Parts of the content of this chapter have been published in [Clasen and Prohaska, 2010].

3.2 Related Work

Many plants have a high geometric complexity, with leaves loosely connected to a large number of twigs and branches. Boudon et al. [2006] classify a wide range of LoD methods developed for this special kind of 3d models. In their terms, our method is a multiscale approach (suitable for both near-field views and large scenes) using structural information with line primitives for the trunk and spatial information with ellipsoidal primitives for the leaves, similar to the work by Deussen et al. [2002], Gilet et al. [2005], and Clasen and Prohaska [2010].

The most popular techniques to render plants are based on the generic Billboard Cloud (BBC) method introduced by Décoret et al. [2003], such as the work of Fuhrmann et al. [2005] and Behrendt et al. [2005]. They use a set of textured impostors to transform the geometric complexity into planar images. Leveraging the GPU texture filtering capabilities results in low amounts of spatial and temporal aliasing, but at the cost of spatial deviations from the reference geometry. [Clasen and Prohaska, 2010], on the other hand, is optimized for a low measured image error compared to a reference image using the HDR-VDP metric introduced by Mantiuk et al. [2005]. Since this metric does not discriminate different noise patterns with similar properties, the draw-back is visible temporal aliasing (see Fig. 3.17 in the results section 3.6). While there are effective techniques to reduce temporal noise in videos, for example by Kim and Woods [1997], they usually introduce a lag of a few frames which limits the applicability to interactive applications. Given this trade-off, we tuned our method towards the behavior of billboard clouds, because measured image errors are usually less important in interactive applications than perceived artifacts. This can be done enforcing a lower limit on the point size in screen space, as proposed by Deussen et al. [2002] and Gilet et al. [2005].

Clustering has been used by various LoD methods in different ways. Gilet et al. [2005] and Clasen and Prohaska [2010] rely on hierarchical bottom-up clustering of point primitives. [Gilet et al., 2005] is driven solely by a spatial data structure, whereas [Clasen and Prohaska, 2010] uses an image error metric to merge the clusters. While the original billboard clouds in [Décoret et al., 2003] and its optimization for trees in [Fuhrmann et al., 2005] use ad-hoc clustering heuristics, [Behrendt et al., 2005] additionally maps the billboard cloud generation to the partitioning (non-hierarchical) k-means algorithm. We carry the idea of using a formalized clustering scheme over to point-based LoD and use the Expectation-Maximization algorithm first presented by Dempster et al. [1977] with a Gaussian Mixture Model for clustering.

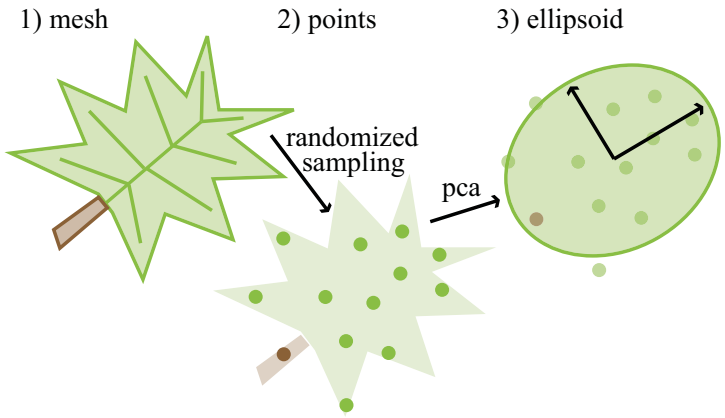


Figure 3.2: *We import ellipsoids by point-sampling the source triangles of a sub-object (leaf, fruit) and fitting the samples by a PCA.*

3.3 Import

Plant models for landscape visualization are usually acquired from third party sources (see section 2.2.2). We import the plant model branches and leaves separately for further processing.

3.3.1 Ellipsoids

We approximate the non-branch elements of plants with ellipsoids. Ellipsoids enable a good approximation of both flat structures (leaves) and voluminous structures (fruits). For coarse LoD, ellipsoids can approximate whole treetops quite accurately. To import the elements, we sample the textured triangles of the source model that belong to the respective element by uniformly distributed points (Fig. 3.2). For each point, we store the 3D position and material properties. If the alpha value of the texture is below 0.5, the point is discarded. In a second step, we run a principal component analysis (PCA) on the point positions and interpret the eigenvectors as coordinate frame for the ellipsoid. The radii r_i along the coordinate axes is given by the square root of the eigenvalues. In contrast to the lines, there is no connectivity between ellipsoids.

3.3.2 Lines

We use lines to approximate plant branches. The branch structure is either given by the modelling system as in Deussen and Lintermann [1997] or can

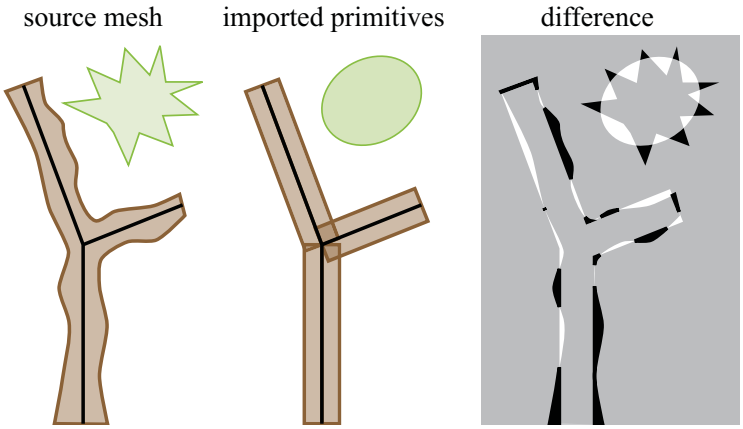


Figure 3.3: *Since the primitive import is not exact, we compare the coverage of the primitives to the source mesh to adjust the primitive size accordingly.*

be reconstructed from the source model. We interpret the branches as linear segments with 3D coordinates, radius and surface materials for both start and end points. We also retain the branch connectivity.

3.3.3 Calibration

Because the primitive import introduces approximation errors, we scale all primitives of the same type by a constant factor. This factor is determined automatically by comparing the coverage of the imported scaled primitives with the source model and selecting the factor with the lowest image error (Fig. 3.3).

3.4 Building the LoD Hierarchy

To build the LoD hierarchy, we employ two different methods for leaves and branches. Leaves are represented by sets of clusters of ellipsoids, while a successively simplified line hierarchy is used for branches.

3.4.1 Ellipsoids

We build a LoD hierarchy on top of the single leaf primitives. We use a fuzzy partitioning clustering algorithm for each level, subsequently reducing the target number of clusters until only a single cluster is generated for the coarsest

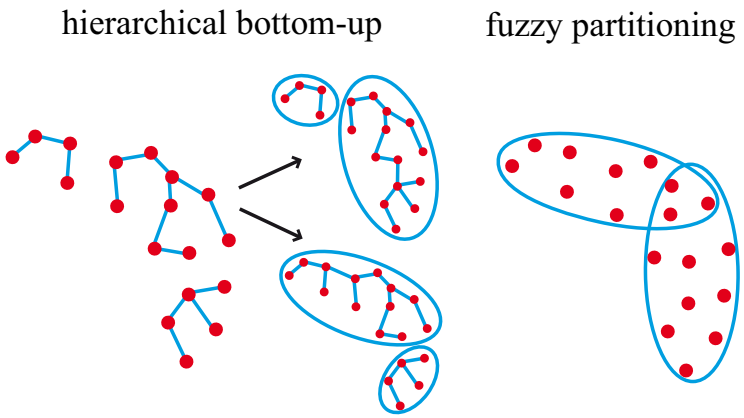


Figure 3.4: *Hierarchical bottom-up clustering tends to result in unbalanced clusters which depend on small variations in the input data, whereas fuzzy partitioning can find robust global optima.*

level. This yields more accurate representations for each level than the agglomerative bottom-up clustering used in previous methods such as Dachsbacher et al. [2003] and Clasen and Prohaska [2010] (Fig. 3.4).

To improve the approximation, we use the Expectation-Maximization algorithm first presented in Dempster et al. [1977] for clustering. It has a strong theoretical background and yields ellipsoidal clusters when combined with a Gaussian mixture model (GMM). Since the quality of the local optimum depends on the initialization values, we run the clustering and the subsequent primitive creation and error evaluation multiple times (Fig. 3.5, Alg. 1), where four showed to be sufficient. To get a LoD hierarchy, we run this loop for several steps, starting with a number of clusters of $\frac{1}{4}$ the number of leaves and further reducing this number by $\frac{1}{4}$ until only a single cluster is generated.

For each iteration, first we generate a set of random points in the bounding box of the leaves. The EM algorithm takes these as initial cluster center values. It then iteratively approximates the center points of the leaf ellipsoids with 3D normal distributions (Fig. 3.6). Once the EM loop has converged, we use the resulting fuzzy cluster assignments $w_{i,j}$ (probability that point i is represented by cluster j; $\sum_j w_{i,j} = 1$) as weights for a PCA of the center points of the leaves (Fig. 3.7). Although the EM algorithm internally generates ellipsoidal objects, we rebuild the ellipsoids to create a cleaner software architecture. This allows for varying implementations of the clustering method. Using the soft EM algorithm with fuzzy cluster assignments results in a more accurate representation of smaller details. Using EM as a hard clustering scheme by selecting the cluster assignments with the largest probabilities hides small features next to large clusters (Fig. 3.8), due to the impact of the cluster size on the probabilities.

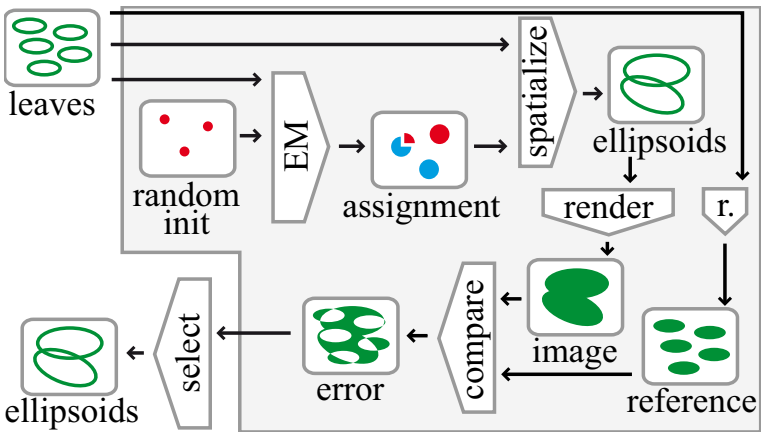


Figure 3.5: *To create a LoD, we first run the EM algorithm with a random initialization, convert the cluster weights to ellipsoids and measure the difference between the rendered ellipsoids and the source leaves. We repeat this several times and select the ellipsoids with the lowest image error.*

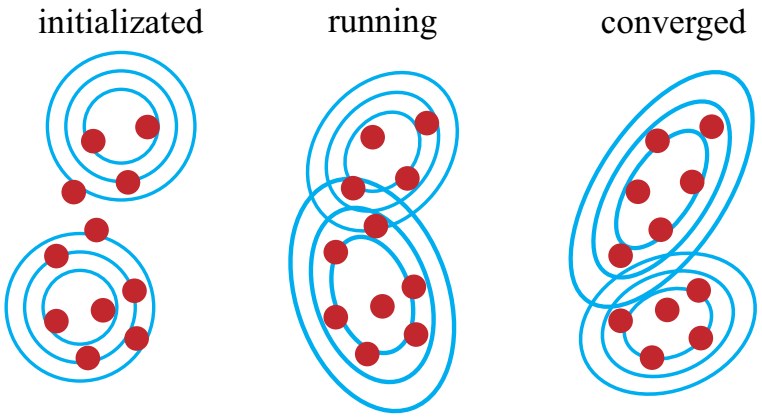


Figure 3.6: *The EM/GMM algorithm is initialized with random cluster positions and iteratively approximates the samples with normal distributions of arbitrary orientation.*

Algorithm 1: Building the ellipsoid cluster hierarchy H.

input : A set L of leaf ellipsoids; a number of tries t
output: A list H of sets of ellipsoids
$I_{ref} \leftarrow$ `RenderImage` (L);
$n \leftarrow \|L\| \cdot \frac{1}{4}$;
$H \leftarrow \emptyset$;
while $n \geq 1$ **do**
  $E \leftarrow \emptyset$;
  for $i \leftarrow 1$ **to** t **do**
    $P \leftarrow$ `CreateRandomPoints` (n);
    $W \leftarrow$ `FindEMClusterWeights` (P, L);
    $E_i \leftarrow$ `CreateEllipsoids` (W, L);
    $I \leftarrow$ `RenderImage` (E_i);
    $e \leftarrow$ `ComputeImageError` (I, I_{ref});
    $E \leftarrow E \cup (E_i, e)$;
  end
  $H \leftarrow H \cup$ `SelectOptimalEllipsoids` (E);
  $n \leftarrow n \cdot \frac{1}{4}$;
end

Once the ellipsoids are generated, we discard all those whose surface area is less than 1% of the largest. This reduces both rendering time and aliasing due to tiny primitives that hardly affect the resulting shape. The size of the remaining ellipsoids is calibrated using the same method as for the initial import.

To avoid the artificial look of perfect ellipsoids, we add a noise texture. We create two mip-mapped cube map textures filled with white noise: One grey-scale texture as alpha channel and one RGB as normal map. These noise textures can be re-used for all models. The alpha channel texture is used to create alpha-test holes in the surface based on a threshold value t shared among all clusters of a single level of detail, and a cluster specific mip-map lod level l_j (Fig. 3.9). The mip-map texture for level l has $2^l \times 2^l$ texels. For $t = 0.5$, half of the texels are discarded, so l_j represents about 2^{2l_j-1} features. The target number of features is given by the sum of leaf weights $\sum_i w_{i,j}$, so $l_j = 0.5(1+log_2 \sum_i w_{i,j})$. The threshold t is determined using the calibration routine, where the image error is computed for multiple values of t and corresponding scaling coefficients for the ellipsoid area to compensate the area loss due to the alpha test.

Mip-map lod level and blending coefficient of the normal map noise are computed in the same way. The blending coefficient is used to blend between the surface normal of the ellipsoid and a random unit vector from the normal

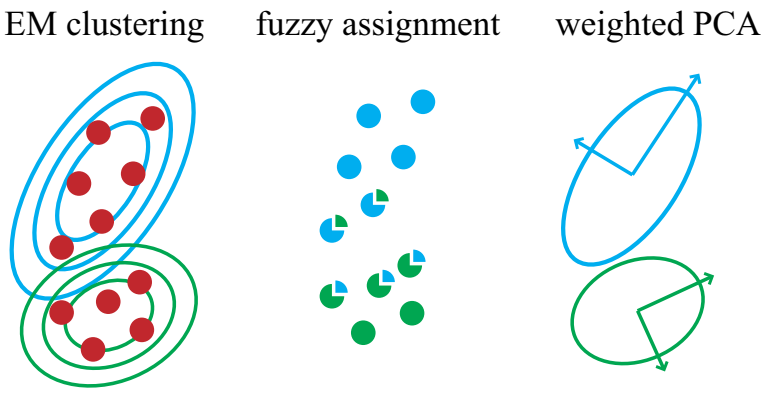


Figure 3.7: *The EM algorithm results in assignment probabilities for each sample. We use these as weights for a PCA to compute the cluster ellipsoids.*

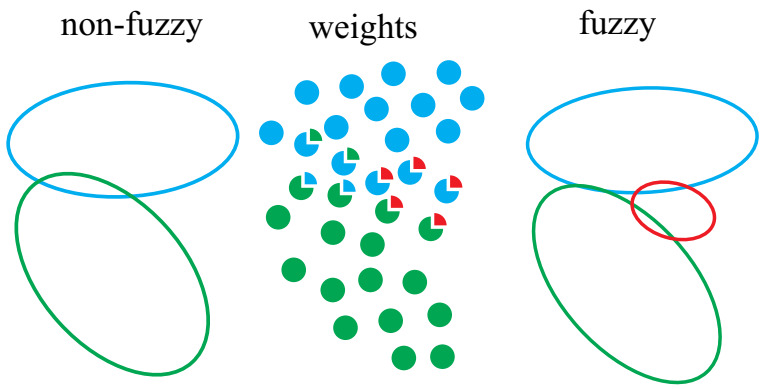


Figure 3.8: *Although the weights of the red cluster accumulate to 1.25, not a single point is assigned to it when using non-fuzzy EM due to the rounding step, while fuzzy EM preserves these details.*

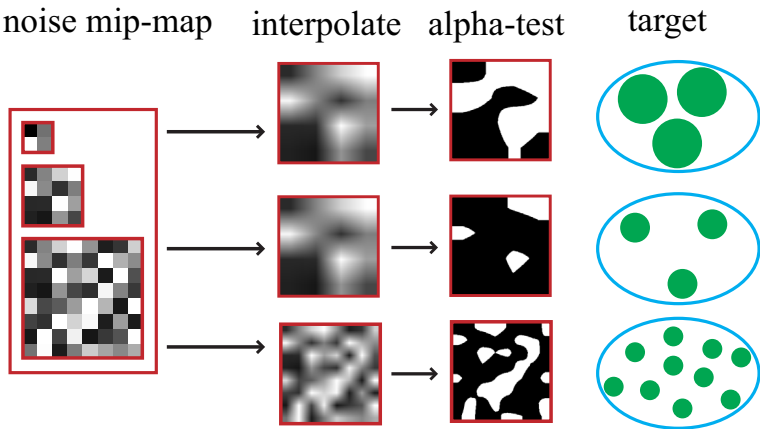


Figure 3.9: *We compute frequency and threshold for the ellipsoid noise texture based on the number and area of clustered leaves.*

map noise texture. To select the best clustering based on the multiple random initializations, we then render the generated ellipsoids and compute the image error relative to the rendered source leaves. While multiple runs of the EM algorithm are usually evaluated based on the log-likelihood value, we prefer the image-error-based comparison because it is a better estimate of the image quality at run-time. Since most of the time is spent on the EM algorithm itself, the additional image rendering and comparison steps are negligible.

3.4.2 Lines

Given the initially imported highest LoD, we successively merge two primitives until only a single primitive is left (Alg. 2). The simplification hierarchy is stored in a tree similar to Dachsbacher et al. [2003]. To choose the next two primitives to be merged, we first randomly gather $N_{new} \cdot N_{local}$ mergeable primitive pairs, called candidates. From these we select the N_{new} with the lowest local error estimate. We measure the image error resulting from the application of this merge step (relative to the source model), and insert candidate into a candidate heap based on Bischoff and Kobbelt [2002]. Then we choose the candidate with the lowest measured error from the heap. If the measurement is from an earlier simplification step, we measure it again, since the surrounding changes can affect the image error. Updating only the top of the heap can result in a suboptimal choice, but this is negligible compared to the cost of updating the full heap. If the best candidate is found, the two primitives are merged and the candidate is removed from the heap. In a last step, we limit the heap to the best N_{cache} candidates to avoid storing bad and outdated choices while retaining those that might be better than those gathered in the next iteration.

We don't use a pre-simplification step as proposed in Lindstrom [2000] due to the negative effect on the accuracy of lower levels of detail in this hierarchical scheme.

Algorithm 2: Successive merging

```
repeat
    gather N_new · N_local new merge candidates;
    select N_new with lowest local error estimate;
    foreach new candidate do
        measure the resulting error e;
        add to candidate heap;
    end
    repeat
        select candidate with the lowest error;
        if candidate error is outdated then
            measure error e again;
            add to candidate heap;
        else
            apply candidate;
        end
    until candidate is applied;
    prune heap to N_cache candidates;
until no candidates are left;
```

Based on this information, we can define merge steps and successively convert the geometric branch hierarchy to a LoD hierarchy (Fig. 3.10).

Two consecutive branches can only be straightened if the second branch has no other siblings to avoid losing visual connectivity. The resulting line is build from the start point of the first line and the end point of the second line. Two sibling branches can be combined if both don't have any following branches. Here all properties of the lines are interpolated, weighted with the weight attribute (Fig. 3.12). The weights of the lines of the source skeleton are initialized by their volumes. On each following operation, the resulting line gets the sum of the weights of the source lines. We estimate the local error of these operations based on their basic properties (Fig. 3.11): Straightening and combining have the least visual impact if the angle between the branches is small.

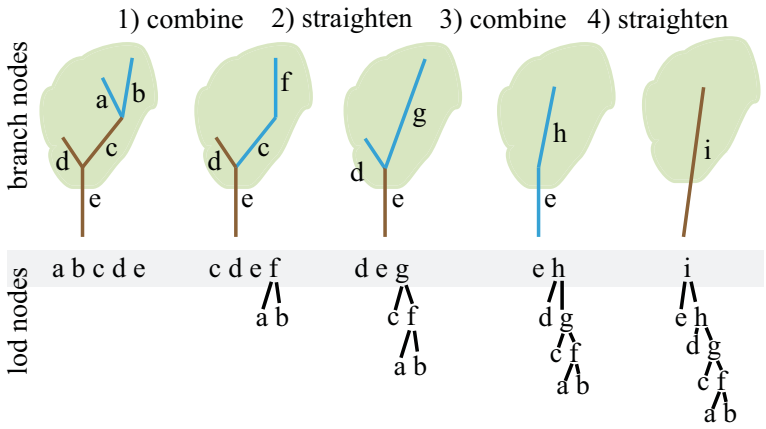


Figure 3.10: *The source line skeleton is given as a hierarchy of branch nodes which represents the geometric connectivity. Two adjacent nodes can be replaced by a parent LoD node, until only a single branch node is left. This is the root of the LoD hierarchy.*

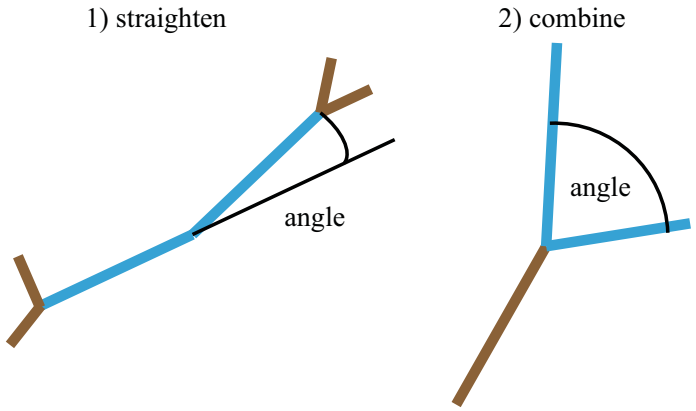


Figure 3.11: *We estimate the error of the line combine and straighten operations based on the angle between the branches.*

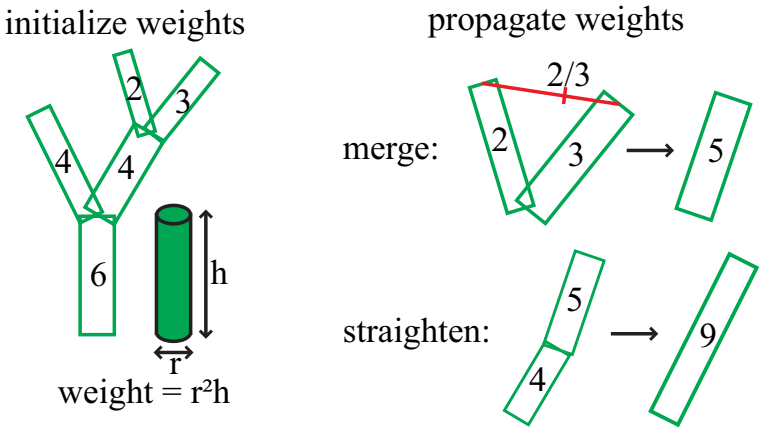


Figure 3.12: *We initialize line weights by primitive volume. We use the weights to interpolate in merge steps.*

3.5 Image Error Metric

To measure the difference between two images, we use a root mean square error (RMSE) metric. RMSE is only sensitive to differences in single pixels. Since the perceived quality is also affected by large scale artifacts, we run the RMSE metric for multiple image resolutions. While perception-based error metrics seem to be a natural choice for comparing image quality, we found RMSE better suited in our case. Perception-based metrics such as HDR-VDP by Mantiuk et al. [2005] and MS-SSIM by Wang et al. [2003] are designed to ignore global differences in contrast and brightness, and emphasize the difference between correlated and uncorrelated noise. These kinds of errors do not appear in our controlled environment. On the contrary, it is a significant difference in the calibration step whether the primitives are invisibly small or cover the entire frame buffer, even though MS-SSIM could detect no structural difference between these cases. Therefore we rely on RMSE, which is also an order of magnitude faster (996 comparisons per second for 512^2 images on a Radeon HD4850, 88 cps for MS-SSIM in Clasen and Prohaska [2010]). Our GPU implementation of multi-scale RMSE runs in a single pass on the GPU by scanning the difference image (pixel-wise $img_{sample} - img_{reference}$) along a space-filling z-curve (Fig. 3.13). This way we can accumulate the errors for each resolution band with a single value per band, taking $log_2(n)$ values, where n is the width of the image.

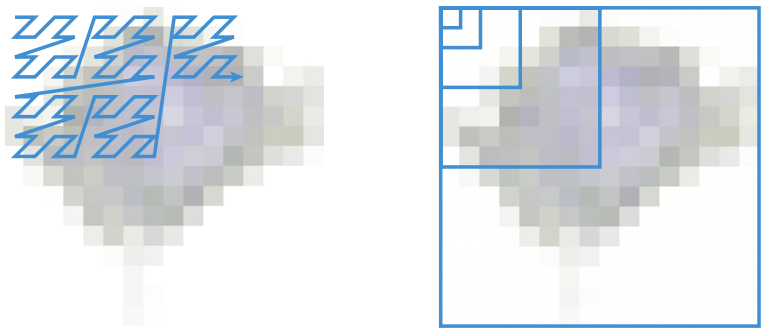


Figure 3.13: *To compute the RMS in multiple resolutions in a single pass, we use a space filling z-curve.*

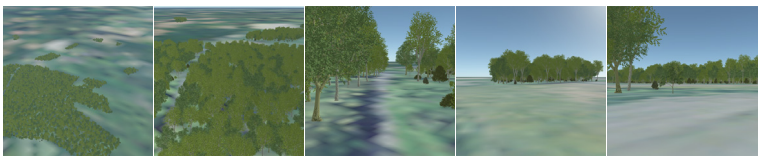


Figure 3.14: *We measured the performance along a camera path through a scene of 10,000 trees.*

3.6 Results

To evaluate our new LoD method, we compared it to Clasen and Prohaska [2010] and the billboard cloud (BBC) implementation used in Coconu [2008]. We calibrated the level of detail selection for Clasen and Prohaska [2010] and BBC similar to the method described in section 6.3, so that the feature size is in the order of a few pixels. Shading is limited to local illumination to avoid image differences due to the varying lighting terms in the implementations. We target interactive applications, so we built a scene of 10 000 plants in forest groups that could be rendered at interactive frame-rates with all three candidates. We measured the performance along a camera path of 1440 frames of 1536×864 pixels (Fig. 3.14) on an Intel Core 2 Duo at 3 GHz and an ATI Radeon HD5870. It starts with an overview, followed by a descent down to human perspective inside the main forest. Then it heads towards a smaller group of trees and turns back to the main forest.

On each frame, we measured the time for the plant rendering only by taking the difference to a run without trees. On average, our new method took 40 ms

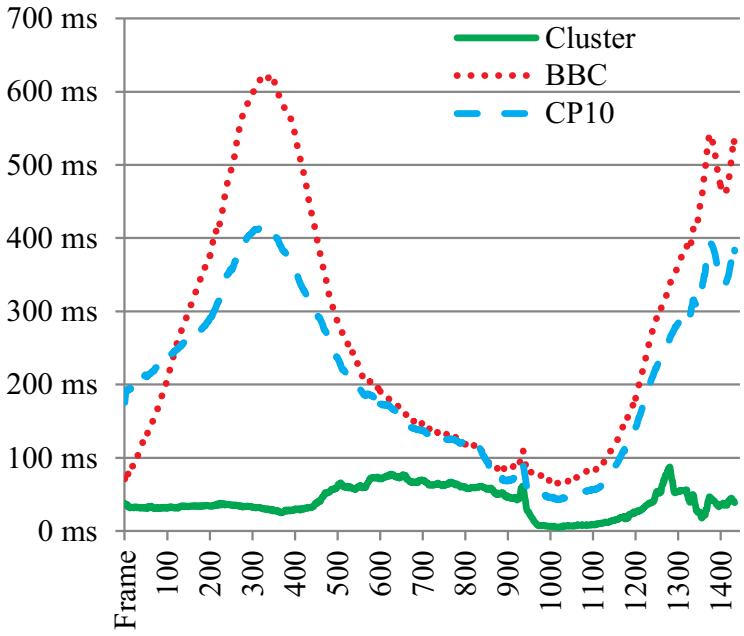


Figure 3.15: *Time per frame to render the plants at the default resolution for all three methods.*

per frame, BBC 260 ms, Clasen and Prohaska [2010] 207 ms, and the mesh without LoD 925 ms. The exact frame times are shown in Fig. 3.15 (we omitted the mesh for improved clarity).

Since most applications allow reducing the image quality in favor of performance, we measured the same path with the LoD selection set to $\frac{1}{2}$ and $\frac{1}{4}$ of the actual image resolution. We chose three distinct frame groups to illustrate the performance behavior: In the first 200 frames, all trees cover only a few pixels of the screen. From frame 250 to 450, the trees in the foreground are a few dozen pixels tall. From frame 600 to 850, the camera is inside the forest and trees cover the whole LoD range from mesh in the foreground to a few pixels in the background. Fig. 3.16 illustrates the performance for the three methods in the three frame groups for the three resolutions. Our new method shows the best overall performance. At lower resolutions and for distant trees, the billboard clouds marginally outperform it.

We initially configured the LoD selection based on the feature sizes. Based on the captured frames, we also measured the actual RMS image errors compared to a super-sampled reference rendering of the source mesh, and additionally to the next frame in the sequence to evaluate temporal noise (Fig. 3.17).

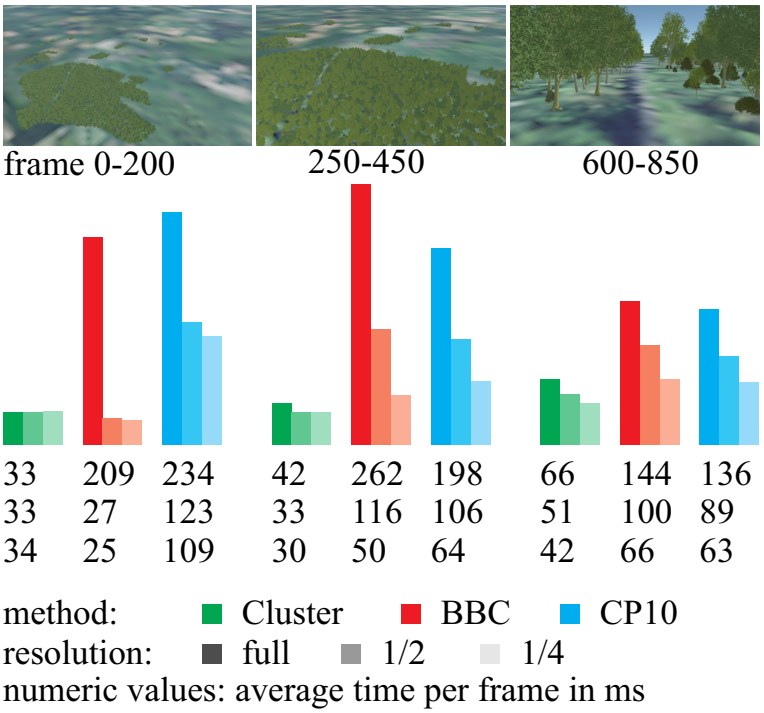


Figure 3.16: *Performance depends on the camera view and the target resolution for the LoD selection. The rendered image resolution remained constant at* 1536×864 *pixels.*

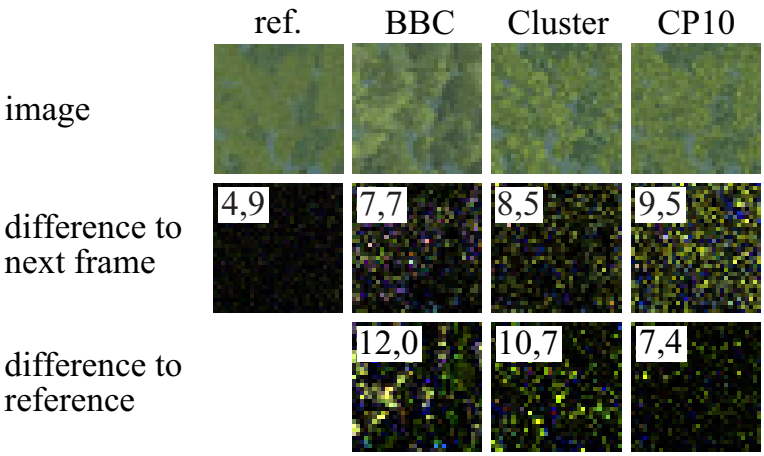


Figure 3.17: *Our new clustering method has an artifact intensity between BBC and Clasen and Prohaska [2010], where BBC is less accurate compared to the reference image and Clasen and Prohaska [2010] exhibits more temporal noise.*

Temporal noise increased from BBC over our new clustering method to Clasen and Prohaska [2010], while the difference to the reference image decreased in the same order.

We also measured the difference between a single tree model at the highest LoD and the respective source mesh (Fig. 3.18). This difference causes popping artifacts in the near field. Our new method uses the same high level representation as Clasen and Prohaska [2010]. Both show only small deviations in the order of single pixels, while BBC shows larger artifacts due to the plane alignments.

To analyze the behavior at low resolutions, we first captured images of all methods at 19 kb memory usage and second at about 130 primitives (Fig. 3.19). 19 kb corresponds to the lowest BBC resolution with three textured quads using GPU texture compression. Ellipsoids take 140 bytes each, lines 48 bytes, so we took a conservative assumption of 130 primitives for the LoD steps of the new clustering method and Clasen and Prohaska [2010]. To have comparable vertex shader load, we also added a BBC LoD step at 125 primitives, which uses 240 KB. The resulting image quality of the 19 KB clustering is comparable to the 240 KB BBC, while the 19 KB BBC shows considerable artifacts. The 19 KB Clasen and Prohaska [2010] has hardly visible resemblance at the rendered resolution and is only usable at the target resolution of a few pixels.

The precomputation times depend on the number of primitives. On average, our implementation took 0.9 s per source leaf and 0.3 s per source branch to generate the complete LoD hierarchy, resulting in the absolute times shown in

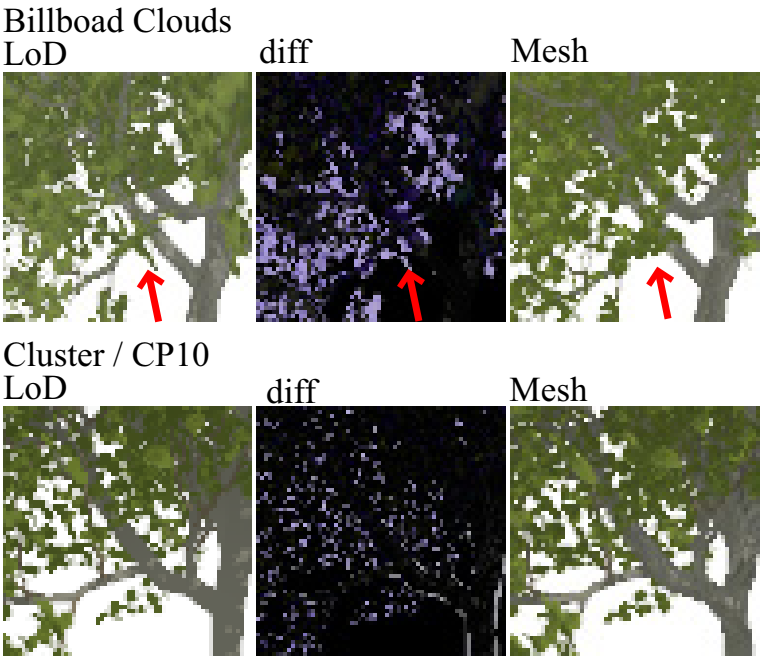


Figure 3.18: *The difference between the highest LoD and the source mesh is larger for BBC, where the texture planes change the leaf positions. For this LoD, BBC uses the source mesh for the trunk.*

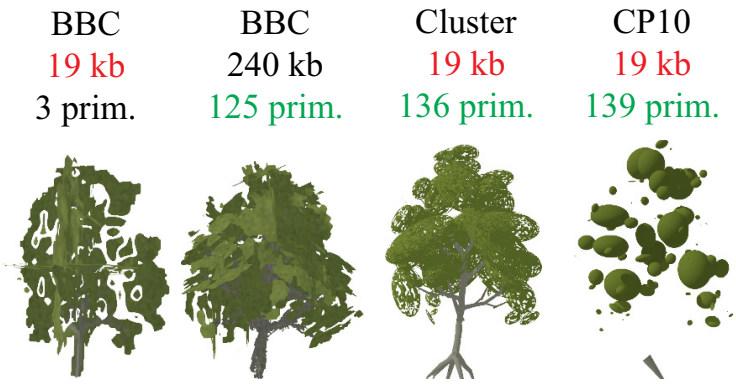


Figure 3.19: *Comparison of the methods at 19 KB (equal memory usage) and about 130 primitives (equal vertex shader load).*

#primitives	1 k	2.7 k	5 k	10 k	27 k
time[min]	7	12	23	91	396

Table 3.1: *Precomputation times in minutes by number of source primitives*

table 3.1. For our models, ellipsoid clustering accounted for 83% of the time.

3.7 Discussion

In most cases, our method is faster than the previous methods BBC and Clasen and Prohaska [2010]. The single exception is rendering of distant trees at reduced LoD resolution. In this case, BBC has the lower draw call overhead. It uses only a single primitive type, textured quads, and requires only a single draw call for a model, while our methods requires two for ellipsoids and lines. The peak in the BBC results for the highest resolution in the first frame group is caused by the relatively large gap between the lowest LoD used by the two reduced resolutions (essentially a simple cross-billboard) and the next step. So while the lowest BBC LoD is more efficient than the lowest LoD of our method, the latter has a smoother transition, avoiding sudden peaks. This yields a more predictable performance behavior.

The noise analysis shows that there is a trade-off between the difference to the reference frame and the difference to the next frame. BBC has larger deviations from the reference, but less temporal noise, while Clasen and Prohaska [2010] shows the opposite behavior. We consider our method well balanced in between. While contrast preservation as described in Cook et al. [2007] happens automatically by averaging the properties of merged primitives, the increased contrast in Fig. 3.17 visible for both our method and BBC is the result of the lower limit on the primitive size to avoid aliasing. The only practical way around this trade-off is supersampling, because per primitive filters would result in translucent pixels which require a costly depth-sort for proper blending.

We attained our goal of an increased performance over the state-of-the-art billboard clouds at a comparable image quality, as measured in section 3.6. However, as the accompanying video shows, 10 000 trees are still not enough for larger forests. Distant views, where each instance takes only a few pixels on screen, could profit from LoD schemes for groups. Taking a random subset of primitives as proposed in Deussen et al. [2002] is fast and easy to implement but prone to aliasing. Our method could be extended by interpreting coarse LoDs of multiple models as a new model and running the simplification process as described. Due to our self-contained primitives, this could be applied to

groups of different models. However, in many applications users want to modify scenes. Although the precomputation times are acceptable for static models, this step would have to be accelerated at least by an order of magnitude.

3.8 Conclusion

We presented a novel LoD method for rendering trees. It combines the rendering efficiency of the ellipsoid and line primitives and the sequential point tree data structure with the accuracy of approximation of fuzzy partitional clustering. As a result, it outperforms previous methods for interactive rendering of forests (40 ms on average for a scene 10 000 trees).

We expect future work on the precomputation step to improve the clustering performance, either by using parallelized EM implementations or by applying a different clustering algorithm, which is relatively easy due to the clean interface of this part. Additionally, given the achieved efficiency of rendering instances at low LoD, transforming, culling, and LoD selection are probably becoming the next bottlenecks in scenes with a larger amount of tree instances. Furthermore, we consider extending the LoD hierarchy beyond single instances the most important next step towards realistic landscapes.

Chapter 4

Tiling

4.1 Introduction

Terrain rendering applications can be found in many different contexts from cartography over landscape planning to virtual outdoor environments. Satellite images at resolutions below 1m and aerial photography data exceed the resolution of the display devices by many orders of magnitude for even moderate sized terrain areas. Efficient level-of-detail algorithms are required to simplify the data sets before rendering without introducing errors. Data should only be processed or synthesized only if it contributes to the final image. Clipmaps were first introduced by Tanner et al. [1998] to solve this for stored images. They consist of a stack of images similar to a mipmap. However, whereas each mipmap level covers the whole texture with images of increasing size, the clipmap uses fixed size levels that cover a decreasing area around an arbirtary focus point. Therefore the memory usage is linear to the level of detail, not exponential.

Since clipmap based rendering front-ends have recently become capable of displaying data at display resolution on commodity PCs, the new bottleneck is the preparation of the clipmap. We present an update strategy that feeds very large terrain data to clipmap based rendering front-ends, thereby exploiting the special properties of the rendering algorithm.

This data can be static satellite imagery, but since only small regions of interest are processed, almost interactive manipulation of large terrain data (height and color) becomes possible. You can for example adjust brightness and contrast to match color maps of different sources, overlay the digital elevation model (DEM) with modifications of different planning scenarios or add vector data layers that are rasterized just in time. This enhances clipmap based terrain

rendering to a powerful interactive visualization technology.

We focus deliberately on well-known simple and robust techniques. Efficient terrain rendering does actually not require complex algorithms and can be implemented using a few building blocks such as clipmaps, acyclic filter graphs, and manager/worker-multithreading.

The content of this chapter has been published in [Clasen and Hege, 2007].

4.2 Related Work

Losasso and Hoppe [2004] presented a terrain rendering algorithm based on clipmaps. The clipmap focus follows the position of the viewer. Therefore the area near the viewer can be rendered at high levels of detail while the regions further away are displayed in a lower resolution. This matches the distortion of the perspective projection, so each area is rendered at a resolution that roughly results in a fixed primitive size in screen space. Asirvatham and Hoppe [2005] improved the performance of this method by moving nearly all rendering operations to the GPU, leaving only decompression and clipmap updating to the CPU. Clasen and Hege [2006] then extended it to spherical domains, showing that this rendering front-end is actually well capable of handling planet-sized datasets at high resolutions.

Tanner et al. [1998] already described a multithreaded clipmap update system. We tailored their general strategy to fit the needs of terrain rendering on commodity PCs without specialized hard- or middleware support. In the following we describe the complete system and explicitly emphasize the differences to [Tanner et al., 1998]. The main difference for visualization applications is the extension from static source images to an interactive framework.

Systems similar to [Tanner et al., 1998] have been developed in commercial contexts, for example the MultiGen-Paradigm Virtual Texture. The technical documentation by Ephanov [2006] describes this approach for the power user, giving a general impression of the design of their implementation. It is also focussed on static imagery and provides no information on the data processing pipeline.

Several terrain rendering front-ends based on an explicit triangulation of the height data have been developed, such as [Lindstrom and Pascucci, 2002], [Cignoni et al., 2003a], [Bao et al., 2004], [Wahl et al., 2004], and [Gobbetti et al., 2006]. Although these systems are highly optimized and provide high quality output and interactive framerates, they are harder to implement since height data and color textures cannot share the same algorithms. This results

not only in two distinct level-of-detail systems but also imposes restrictions on the the triangulation due to the texture tile boundaries (see [Wahl et al., 2004]).

Geometry clipmap based systems can use a unified codepath for height maps and color textures. They require no preprocessing such as triangulation, therefore enabling efficient real-time synthesis. By leveraging the high processing power of current GPU, geometry clipmap systems can reach a relatively high performance without obscuring the source code with special case handling and elaborated optimizations. While view frustum culling for a single camera with 6 degrees of freedom is already non-trivial on any terrain rendering front-end, doing so for a shadow mapped scene becomes quite involved. Brute force geometry clipmap rendering might not outperform specialized triangulation based systems, but it is much easier to get to an usable performance level even without culling.

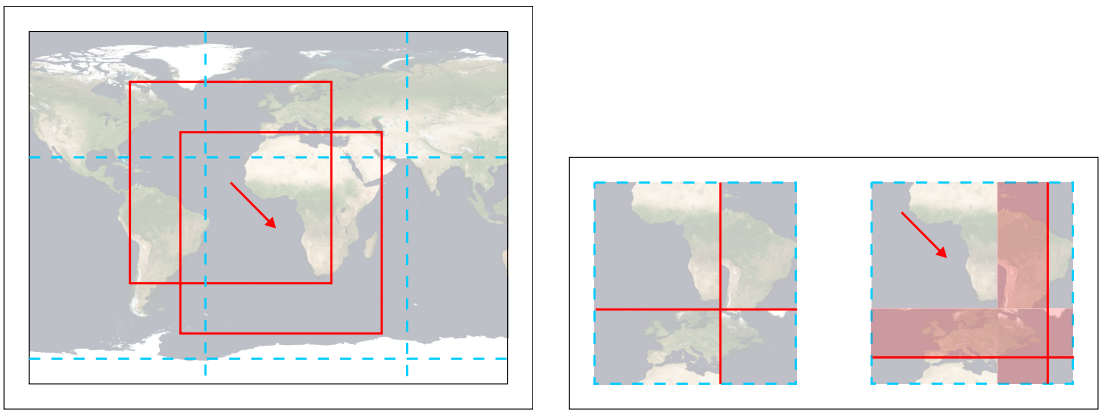
4.3 Rendering Front-End

The proposed method is designed to feed the rendering front-end described in detail in chapter 5. It basically works as follows: The static mesh for each level of detail is initialized once on program startup and stored on the GPU. For each frame the current position of the camera (projected straight down to the surface of the planet) is used to set the new clipmap focus. The clipmaps for height map and color map are updated accordingly. The height clipmap is used to displace the vertices of the mesh of the corresponding level of detail.

4.3.1 Clipmap

The original clipmap by Tanner et al. has been implemented as part of IRIS Performer, a visualization middleware. It provides an interface that hides modes of the internal workings to be used as a direct mipmap replacement. Although this greatly simplified the usage by application developers, terrain rendering front-ends like [Asirvatham and Hoppe, 2005] can benefit from knowledge about the underlying clipmap since the front-end can skip entire levels instead of rendering them with an artificially limited texture resolution. This motivates an interface that provides more control about the update process.

Current consumer GPU do not support clipmaps directly (e. g. through the OpenGL extension `GL_SGIX_clipmap`), so we have to manage them explicitly using a stack of textures and the usual texture update functions such as



(a) (b)

Figure 4.1: *(a) Since the clipmap contains only a subset of the corresponding mipmap, it has to be updated when the focus (e. g. camera position) changes. (b) Toroidal indexing results in relatively small update stripes instead of a full clipmap rebuild.*

`glTexSubImage` and the pixel buffer object extension.

The first initialization of a clipmap is straight forward: Each level is centered around the focus point of the clipmap. The first texture covers the entire terrain and each following level covers a fourth of the previous level (half length in both dimensions). When the focus point moves (see Fig. 4.1(a), the following cases exist:

1. The focus point moves less than one pixel, so no update is required.

2. The focus point moves less then the edge length of the covered area in both dimensions. In this case some parts of the previous contents can be reused.

3. The focus point moves further away, so the entire level has to be updated.

The second case is the most interesting one since clever reuse of existing data can reduce the bandwidth requirements significantly. Tanner et al. solved this by toroidal indexing: The focus point is not set to the center of the texture but to its world position modulo the size of the current level. Therefore the content has a fixed position in the texture and does not have to be moved (see Fig. 4.1(b)).

4.4 Tile Generation

When the camera moves smoothly through the scene, most clipmap updates change only a few texels. Given an efficient rendering front-end, this can go down to $1 \times N$ (for texture size $N \times N$) texel small update stripes that have to be calculated on every frame. This results in highly inefficient texture generation for almost anything more complicated than a simple copy operation. If, for example, a jpeg is used as image source, only full DCT blocks of 8×8 pixels could be decoded. Many image synthesis algorithms also perform better on larger blocks because higher code and data locality improves branch prediction and cache usage.

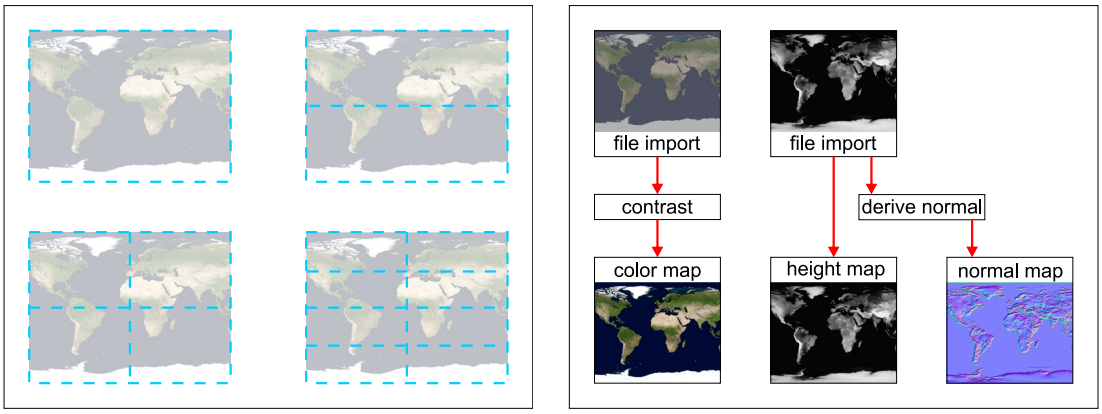
Tanner et al. divide the map into tiles with a size in the same order of magnitude as the clipmap texture. Smaller tiles improve the paging latency, but larger tiles are processed more effiently. These tiles are generated by image sources and stored in tile caches.

4.4.1 Image Sources

Image sources load and process the terrain raster data as presented in section 2.1. An image source returns a section of a map when given the section size and offset. Since we operate on a fixed tile grid, we pass the grid size and the grid coordinates. We use a grid hierarchy that corresponds to the mipmap levels: A grid subdivides the map into $2^m \cdot 2^n$ tiles (see Fig. 4.2(a)), with $m = n$ for planar clipmaps and $m > n$ for spherical clipmaps (see chapter 5).

Tanner et al. have a fixed source design: Since their approach tries to mimic a conventional mipmap as close as possible, they have one large mipmap stored on the disc that is paged into the clipmap transparently. However, this simple cache logic can be easily replaced by a texture synthesis framework: We use a generalized image source system that generates the tiles on-the-fly (Fig. 4.2(b)). Similar to the usual clipmap usage, our main image source is a file loader based on the ECW file format (see [Ueffing, 2001]). This wavelet format is optimized for streaming and region-of-interest decoding. We used it for a Landsat texture with about $1.4 \cdot 10^6 \times 0.6 \cdot 10^6$ RGB pixels compressed to 26 GB in our tests ($97 : 1$). Losasso and Hoppe [2004] used likewise a hierarchical file format with spatially localized bases to enable efficient region-of-interest decoding.

One of the main advantages of this flexible clipmap back-end is the ability to reuse sources in different clipmaps. For example normal maps are usually derived from height maps. Many common operations can be implemented based



(a) (b)

Figure 4.2: *(a) The map is subdivided into different grid resolutions that can be anisotropic when using spherical clipmaps. (b) Image sources form a dependency tree: In this example, the color clipmap receives the tiles from a contrast filter which depends on a file loader. The height clipmap and the normal clipmap share the height map file loader source as the normal map is generated on-the-fly.*

on this image source framework:

1. Map Resampling
   Most georeferenced maps do not cover the whole planet. A map loader source can therefore realign and possibly resample the map on the fly to hide this fact from the following rendering steps. Missing texels are filled with an arbitrary value, e. g. transparent or black.

2. Vector Rasterizer
   Apart from raster data, vector shapes are quite important in terrain rendering applications. These should be rasterized in map space and rendered together with the underlying height map to avoid visible accuracy errors as described by Kersting and Döllner [2002]. They use an *on-demand texture pyramid* which fits conceptionally well to our clipmap source system.

3. Overlay
   Several maps can be overlaid on-the-fly. This is useful for example when only parts of a map are available in high resolution or to implement the thematic lens effect introduced by Döllner et al. [2000].

4. Detail Synthesizer
   When a few representative high resolution sample textures are given for a low resolution map, they can be used to give a visual impression of the

missing details using the constrained texture synthesized by Wang and Mueller [2004]. Although this method adds guessed information to an otherwise reliable data set, it is a powerful tool when applied judiciously. If only the general appearance of a terrain should be visualized, pure synthesis algorithms can be used. For example Berger [2003] sketched a real-time texture synthesis technique suitable for terrain rendering that used spectral noise to generate the different levels of detail.

5. Normal Map
   Normal maps are usually derived from height maps and not sampled directly. This requires only one fast additional filter in the source tree instead of a whole precalculated map which would quadruple the required disk space.

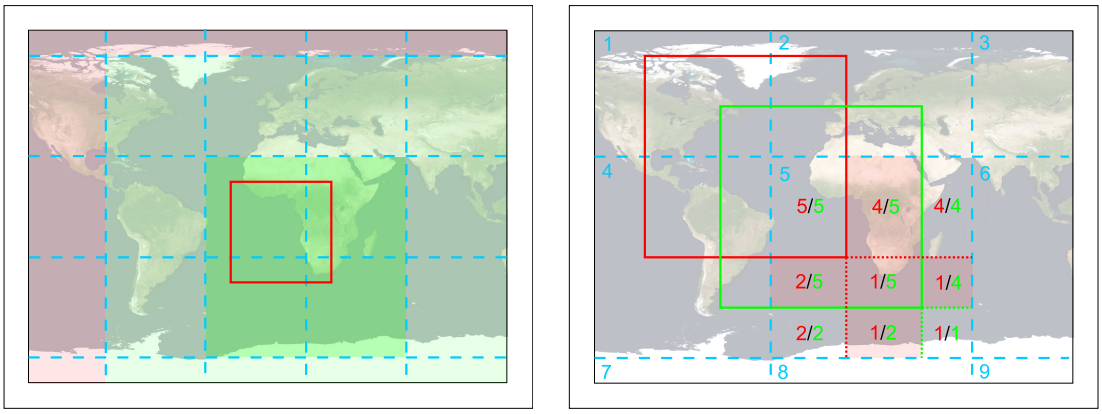
6. Filters
   Adjusting texels without context is one of the fastest ways to modify the image. Changing hue, brightness, contrast, saturation, gamma curves etc. adds virtually no overhead (texture decompression or synthesis is noticable slower). These operations are nonetheless invaluable because they work without expensive precalculation or overhead in the rendering front-end. Since only the visible tiles are processed, changing the parameters of these sources is almost interactive.

4.4.2 Feature Sources

Feature sources load and process the GIS features as presented in section 2.3. A feature tile contains all features where the intersection of the associated geometry and the tile is not empty. For point data, we evaluate the points directly, not the shapes of the model instances they represent, for two reasons: It is faster, and later in the pipeline, we can simply render all points as instances without rendering a single instance near a border twice. Feature sources integrate into the same processing pipeline as image source (Fig. 4.2(b)) and can be used as source for the vector rasterizer proposed in section 4.4.1. In the following, we describe the rest of the pipeline for image sources, but the same applies for feature sources.

4.4.3 Tile Cache

The tile cache decouples the clipmap update algorithm from the image sources. Image sources store their output in a source-specific tile cache and the clipmap update routine gets it from there. Tanner et al. used a fixed cache size of n^2



(a) (b)

Figure 4.3: *(a) While tiles at the current clipmap position should be kept to avoid regeneration on the next clipmap update, the tiles that are not directly adjacent to these can usually be discarded to free memory. Keeping the adjacent ring avoids cache thrashing when the camera moves back and forth over a tile border. (b) The texture update depends on exactly four tiles. The two wrap-around positions split the texture (displayed over tile 5) into nine update regions (red number: previous tile index; green number: new tile index)*

tiles that are accessed in the same toroidal way as the clipmap textures. We implemented a more flexible solution based on an associative array: The tile locator that identifies a tile unambigously is used to index the tiles. There is no fixed limit on the size of the cache, instead it is regularly pruned according to an arbitraty criterion. We currently use the distance to the focus point (see Fig. 4.3(a)), although more intelligent strategies would model the basic idea better: Those tiles that are least likely to be used again should be removed. The distance correlates with this, but camera movement and view direction could provide valuable hints.

4.5 Clipmap Update

We divide the clipmap updates into two steps: Gathering the necessary map tiles and copying the data to the clipmap texture.

4.5.1 Quads

The number of tiles required for a clipmap update depends on the ratio between tile size and clipmap texture size. Instead of the arbirary ratio of Tanner et. al, we prefer using tiles of the same size as the clipmap texture. This reduces the updating process to a small and efficient algorithm:

Using a size ratio of $1 : 1$, a clipmap update requires exactly four map tiles (ignoring the degenerate case when the clipmap contains exactly one map tile). These map tiles form a so-called quad. By the time of the clipmap update, the tile cache should already contain the necessary tiles. Therefore we can just retrieve them and pass the quad to the following step.

Fig. 4.3(b) illustrates this with a simple example: The previous clipmap location is represented by the red square, the new one is drawn in green. The new clipmap intersects the map tiles 1, 2, 4 and 5 which are passed to the update step.

4.5.2 Update Regions

If the focus point has been moved by more than one tile, the complete clipmap texture has to be updated and no further optimizations can be applied. However, the common case in terrain rendering is a relatively slow moving camera, so we can reuse parts of the clipmap texture.

We describe this process by means of the example in Fig. 4.3(b): Since the clipmap texture content is placed modulo the tile size, we draw the clipmap texture over map tile 5. As Fig. 4.1(b) illustrated, the toroidal indexing splits the clipmap texture into four regions, each corresponding to a different map tile. Splitting the clipmap texture by both the old and the new wrap-around position, we end up with nine regions. Now we can determine the corresponding map tile for each part for both the old and the new focus point. These two tile indices are compared for each region, and only those regions with a differing tile index have to be updated.

The map tile indices are then used to get the correct tile from the quad. Now the region can be copied directly from the tile to the clipmap texture without translation.

4.6 Multithreading

Terrain rendering as presented by Asirvatham and Hoppe [2005] is almost completely done on the GPU which runs asynchronously to the CPU. The bottleneck becomes visible at the clipmap update routine: At high framerates only small stripes of the clipmap texture have to be replaced on every frame which can be interleaved nicely with the rendering routines. But, depending on the camera movement, every now and then a new map tile has to be generated. This relatively large task stalls the update process and in turn the rendering, so passing over the tile grid lines results in noticable stutter.

We move the tile generation task to separate threads to avoid this. Although the total workload remains the same, the tile generation can be spread in time over multiple frames on single CPU systems, avoiding the occasional stutter. Multicore CPUs and systems with multiple CPUs can run the tile generation in parallel with the rendering thread.

Tiles are enqueued according to their priority. It depends on the visibility (pre-fetched tiles are less important than visible ones) and level (lower (coarser) levels are more important as they are rendered first). Tiles within one priority class have no special order.

We employ the well-known manager-worker pattern (see [Freisleben and Kielmann, 1997]) to handle the parallel generation. The list of required tiles is determined by the render thread and sent to the manager thread. The render thread can now continue its main task while the manager distributes the workload among the worker threads and deals with tile dependencies. Tile dependencies can be resolved by a depth-first traversal of the source tree starting at the final soures. This yields the correct order in a single pass.

Thread synchronisation is limited to three interfaces: The two tile queues between renderer and manager, and manager and worker are common message queues. The third synchronization is required inside the tile cache because multiple workers can write to the same cache simultaneously while the render thread tries to read. We use one mutex per cache which appears to be sufficiently fine grained. Since the tiles themselves are immutable, no further synchronization is required there.

The render thread runs with the highest relative priority, the manager thread below that and the worker threads at the lowest level. This way the render thread is not blocked by the tile generation process and can continue updating the display at a coarser resolution to improve interactivity.

4.7 Algorithm

When the application is initialized, a fixed number of worker threads is spawned. They simply block inside the message queue when no tasks are available and take no CPU time. No further setup is required.

Clipmap updates and rendering are handled in a single loop over the visible levels for each frame (Alg. 3). These are determined by the front-end (see chapter 5). The other two input parameters are the current clipmap focus and the time limit. The latter is used to target a fixed frame-rate.

Algorithm 3: Per frame update and rendering.

```
Input: minLevel, maxLevel, timeLimit, focus
begin
    if focus changed then
        foreach clipmap do                          // height, color, etc.
            set focus;
            gather new tile tasks → taskList;
        end
        cancel previous tile tasks;
        enqueue taskList;
    end
    for i ← minLevel to maxLevel do
        // prepare level
        foreach clipmap do
            get quad from tile cache;                       // can block
            update clipmap texture;
        end
        // render level
        nextLevel ← i +1;
        if nextLevel ≤ maxLevel
        ∧ IsReady(nextLevel)
        ∧ GetCurrentTime() < timeLimit then
            render ring level;
        else
            render full level;
        end
    end
end
```

If the clipmap focus has changed since the previous frame, the required tiles might have changed. Therefore we iterate over all clipmaps and gather

the new tile generation tasks. The worker threads are still running during this step since the probability of generating usable tiles is quite high in the common case of slow camera movements. When all tasks are known, the current tasks are cancelled and the new ones are enqueued. Cancelling the tasks does not interrupt the worker threads, it affects only the message queue of the manager. This is also because of the high probability of generating usable tiles.

The following loop over the rendering levels starts with the level that has the lowest resolution and proceeds with increasing resolution. This order ensures that we can interrupt the rendering process and still cover the whole terrain, just with less detail than expected. Note that this order is different from [Losasso and Hoppe, 2004]: Losasso renders from fine to coarse to exploit hardware occlusion culling. This results in higher framerates for a given level of detail, but does not allow a fixed framerate.

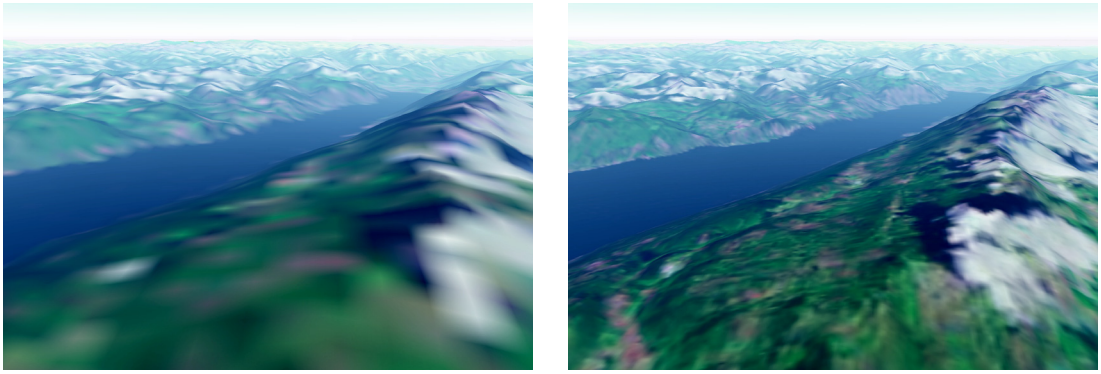
At first the current level of each clipmap is prepared. This requires the tiles that are generated asynchrously, so getting the quad can block if they are not ready. The following update just copies the appropriate regions of the quad to the texture. When all clipmaps are ready, we can actually render the terrain. There are two alternatives: If further levels of detail follow, we render a ring, otherwise we render a disk (see [Asirvatham and Hoppe, 2005]). It depends on three conditions whether a finer level follows:

1. The level visibility calculation of the rendering front-end limits the maximum resolution. Since image quality is also limited by the output device, increasing the terrain resolution to more than one primitive per screen pixel is rarely useful.

2. If preparing one of the clipmaps of the next level would block, we stop the rendering process at the current level. This keeps the framerate predictable, and there's a good chance that this level can be rendered on one of the next few frames due to the asynchronous worker threads.

3. If the time limit is reached, no further levels should follow. This is a rare case since the operations in this loop usually do not take much CPU time. It can actually happen due to a blocking update of the lowest level when the camera altitude is increased rapidly. However, most of the time it's just a safety check against unfortunate thread scheduling and other external influences.

Tanner et al. handle this differently because of they target full transparency for the user: Instead of explicitly communicating the availability of clipmap levels, they simply limit the *MaxTextureLOD* value, enforcing clipmap lookups



(a) (b)

Figure 4.4: *(a) The full area is rendered at a low level of detail if the more detailed clipmap levels are missing. (b) The same scene rendered in full quality*

in the lower levels only. Since our terrain rendering front-end also generates the geometry based on the clipmap, there's no benefit in rendering highly detailed meshes of coarse raster data. Therefore we prefer using an application specific method to deal with level availability.

4.8 Results

4.8.1 Visuals

The image quality depends primarily on the rendering front-end. The only effect introduced by our method is the omission of higher levels of detail: Fig. 4.4(a) shows a view of the Alps where the rendering has been stopped a few levels early because of delayed clipmap updates.

Fig. 4.4(b) shows the same scene a few frames later, when the required data is available.

4.8.2 Resources

The following results were produced on a dual CPU Xeon (3.2GHz) with 2 GB RAM and a NVidia Geforce 7900GTX (512 MB RAM), running Windows XP x64. The color map is a Landsat satellite image with $1.4 \cdot 10^6 \times 0.6 \cdot 10^6$ 8 Bit RGB pixels (2 TB compressed to 26 GB). The height map is a SRTM data set with $432 \cdot 10^3 \times 216 \cdot 10^3$ 16 Bit greyscale pixels (180 GB compressed to 5 GB). The

altitude	levels	velocity	framerate
800 km	1-5	640 km/s	64 fps
80 km	3-9	64 km/s	60 fps
50 km	3-10	40 km/s	50 fps
20 km	4-11	16 km/s	38 fps
6000 m	5-13	4800 m/s	36 fps
2000 m	5-14	1600 m/s	36 fps
800 m	6-16	640 m/s	32 fps
200 m	7-17	160 m/s	32 fps
50 m	8-20	40 m/s	28 fps

Table 4.1: *The maximum velocity at which the back-end can keep up and the corresponding framerates*

color clipmap uses a texture size of 256^2, the height clipmap uses 64^2 texels. The normal map is derived on the fly from the height map at full color texture resolution. About 800 MB RAM were used.

Due to framework issues, the maximum framerate is limited to 64 frames per second (independent of the vertical refresh rate). We chose front-end settings that allow rendering at this speed for a static camera when all clipmap data is ready. The framerate is therefore only limited by date processing and clipmap updates. Table 4.1 shows the maximum velocity at which the back-end could keep up with the front-end and the corresponding framerates. At velocities higher than about $altitude \cdot \frac{4}{5s}$, the highest levels of detail are occasionally ommitted while the framerate stays quite constant. Moving slower also does not affect the framerate significantly due to the overhead in the clipmap update routines (more levels are considered visible at lower altitudes).

A second test using a height generator based on 3D simplex noise and a derived color map showed basically the same behaviour, although it was about 5 times slower.

4.9 Conclusion

While terrain rendering using clipmaps depends on low latencies but requires little CPU time, preparing the clipmaps takes high CPU bandwidth but is of limited urgency. We presented a way to decouple these two tasks to enable terrain rendering with a constant high framerate and an adaptive level of detail. Most details are ommited when the camera moves fast, which is exactly the situation in which the user would not perceive it anyway. As soon as the camera

stops, the resolution converges to the maximum for the given terrain rendering front-end. This allows terrain visualization applications with state of the art rendering front-ends that do not have to rely on explicit coarse replacement geometry or other placeholders to remain highly interactive in a professional visualization setting, including on-the-fly data manipulation.

4.10 Future Work

Although the current solution already works fine, we found a few points that could need further methological improvement:

1. Prefetch
   The cache prefetch strategy currently only takes the clipmap focus into account. However, prefetching the tiles lying on an extrapolated camera path would be preferable as those behind the camera are less likely needed for the following frames. It could also be desirable to allow explicit prefetching for non-interactive visualizations where the application knows in advance which regions are visited next.

2. Clipmap Mipmaps
   There's a general problem with manual clipmaps as introduced by Asirvatham in Asirvatham and Hoppe [2005]: When only small portions of a texture are updated, the cost of regenerating a mipmap for this texture are relatively high. However, mipmaps with anisotropic filtering would improve both rendering performance and quality significantly when the camera aims at the horizon and not the geocenter. However, avoiding the explicit clipmap texture and passing the quads to the GPU would circumvent this issue, but at a cost of four times more texture samplers and no interpolation between tiles which is equivalently undesirable.

3. Scheduling
   Currently we do not employ any clever scheduling strategy to feed the worker threads. This might result in a bottleneck when using complex source trees on systems with many CPUs, but we didn't do any further research in this direction, yet.

Chapter 5

Terrain Rendering

5.1 Introduction

Terrain rendering has a broad range of applications from science, e.g. cartography and landscape planning, to entertainment, e.g. outdoor games and movies. We focus on the serious applications that usually don't allow artistic tricks to hide technological deficiencies. The target is quite simple to state: We want to visualize spherical terrains (whole planets) on many scales (from space flight to sunday afternoon walk) on commodity hardware. This imposes two major challenges: The size of the data exceeds the capabilities of current PCs by far and numerical errors of 32 bit floating point numbers, the maximum accuracy of current GPU, become relevant.

The content of this chapter has been published in [Clasen and Hege, 2006].

5.2 Existing technology

Previous publications and applications can be divided into two parts: Those with planar terrain and those with sperical terrains. Both converge to the same solution with increasing scale, and there are many cases where a planar terrain is absolutely sufficient. But in the real world, you just cannot see from Lisbon to New York.

5.2.1 Planar terrain

Many popular terrain rendering algorithms deal with planar terrain. Losasso and Hoppe [2004] categorize them as follows:

- Irregular meshes (a.k.a. triangulated irregular networks)
- Bin-tree hierarchies (a.k.a. longest-edge bisection, restricted quadtree, hierarchies of right triangles)
- Bin-tree regions (coarser than Bin-tree hierarchies)
- Tiled blocks (square patches that are tessellated at different resolutions)

The error for a given number of triangles increases with each category. Irregular meshes result in the best possible approximation but requires a large computational overhead. In practice, tiled block algorithms can take advantage of the huge geometry bandwidth of current GPU most effectively and overcompensate their deficiencies in accuracy. Losasso and Hoppe [2004] introduces the Geometry Clipmaps algorithm which is especially designed for this bandwidth. Asirvatham and Hoppe [2005] further improve it to handle most of the computations on the GPU.

5.2.2 Spherical terrain

Although the same categorization is valid for spherical terrain, most research seems to focus on planar terrain. O'Neil [2001] and Hill [20002] tried to extend the ROAM algorithm by Duchaineau et al. [1997] (bin-tree hierarchy) to handle spherical surfaces, but Hill dropped this approches in favor of a tiled block solution in the same publication. Cignoni et al. [2003b] introduce a bin-tree region type algorithm, they extend the BDAM algorithm to planets (P-BDAM) in [Cignoni et al., 2003a]. All solutions have in common that they partition the planet into square regions, using a cube as base geometry.

The popular terrain viewers Google Earth (`http://earth.google.com/`) and NASA World Wind (`http://worldwind.arc.nasa.gov/`) apparently use tiled block approaches, but these solutions are not published.

5.3 Spherical clipmaps

We chose to extend the GPU-based geometry clipmaps by Asirvatham to spherical terrains because of the following reasons:

- The rendering speed depends on the screen resolution, not on the size of the digital elevation model (DEM) and the corresponding surface color texture. This basic feature of each LOD algorithm is handled exceptionally well by the underlying clipmap. Image resampling is a thoroughly researched domain and this knowledge can be applied directly in the construction of the clip map.
- Different levels of detail can be blended smoothly even when they are more than one level apart.
- Rendering can be limited on-the-fly to the coarsest n levels without overhead in case streaming data is late or the framerate does not meet the requirements.
- The implementation is simple because the geometry is static and the only image operation is copying regions between buffers.
- The technique is quite fast and the current bottleneck, vertex texture lookups, is expected to disappear with unified shaders.

The following changes to the original algorithm enable spherical terrains:

5.3.1 Clipmaps

The original Clipmap by Tanner et al. [1998] is a texture representation that can be used to display textures of virtually unlimited size with maximum detail around a variable focus point. It resembles a mipmap pyramid where each level is clipped to a fixed number of samples around the focus point (fig. 5.1). When a level is sampled, it is first tested whether the sampling point lies in the clipped region. If not, the next higher level is searched, which covers an area four times as large. This results in a memory requirement of $O(log\, n)$ for a base texture of size n.

Losasso and Hoppe used this representation for height maps. This effectively enables the usage of arbitrary height map sizes independent of run-time memory requirements and provides an inherent level of detail representation that reduces rendering time similar to memory. Each ring is rendered using the

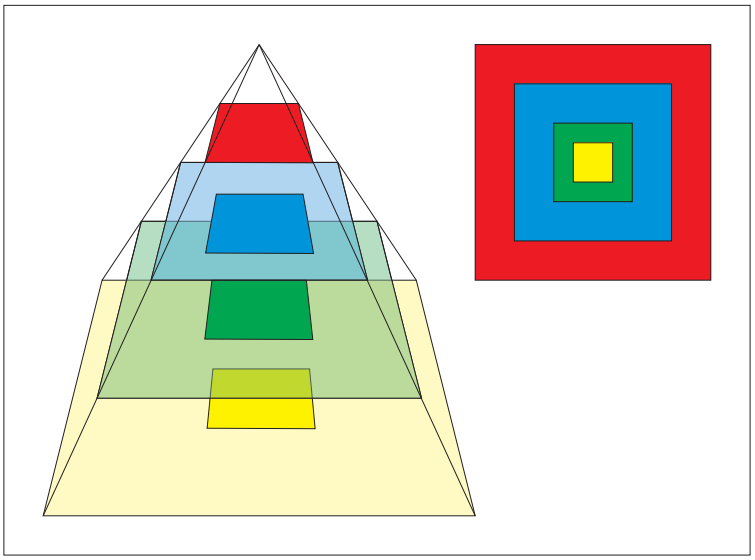


Figure 5.1: *The clipmap contains a fixed-size segment of each mipmap level around an arbitrary focus point.*

same number of vertices just as each ring contains the same number of image samples.

Since the main feature of Geometry Clipmaps is the static geometry relative to the viewer (plus some minor translation), this support geometry had to be changed to accommodate our parametrisation: Any rectangular grid aligned to the underlying parametrisation changes its shape with the distance to the poles of the planet. The problems becomes inevitably visible when the viewer is close to the pole: The support geometry becomes infinitely thin towards the pole and stops there as spherical coordinates do not wrap around in θ direction.

In the following we replace the underlying geometry with one that maps better to the sphere. No matter how far away the viewer is relative to the planet, he cannot see more of it than one hemisphere. This led to the idea of using concentric rings instead of rectangles. The resulting spherical Geometry Clipmap is displayed in fig. 5.2.

5.3.2 Map parametrisation

The trivial parametrisation of the plane, (x, y), cannot be transfered directly to the sphere. However, an equally simple parametrisation exists: Spherical coordinates, denoted by $(\phi, \theta) \in [0, 2\pi) \times [0, \pi)$ (fig. 5.3). Given a coordinate system with the axes (x, y, z), a point p on the unit sphere can be parametrized

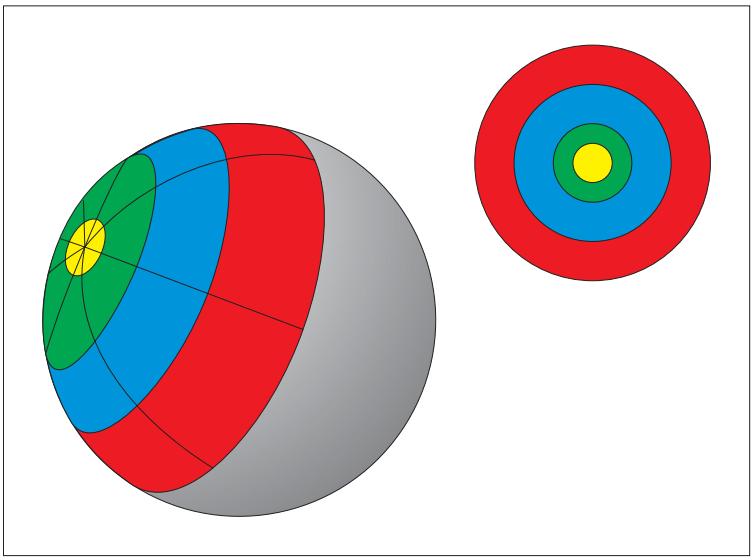


Figure 5.2: *We use circular instead of rectangular rings to cover the hemisphere.*

by its angle $theta$ to the z-axis, and the angle phi from p projected to the x, y-plane to the x axis. $(0, 0, 1)$ and $(0, 0, -1)$ can be denoted as north and south pole respectively. All points with the same phi belong to a meridian, the 0-meridian intersects the positive x-axis.

5.3.3 Map transform

Since we want to focus the clipmap around the viewer, we have two different spaces: The world space (x, y, z) that provides an absolute orientation of the spherical terrain and the view space $(\tilde{x}, \tilde{y}, \tilde{z})$ that locates the viewer at the north pole. The introduction of the view space enables a static geometry (vertices plus connectivity) that has to be calculated and transferred to the GPU only once. The hemisphere around the viewer is parametrized by $(\tilde{\phi}, \tilde{\theta})$ whereas the terrain is parametrized by (ϕ, θ). The mapping between both spaces (fig. 5.4) depends on the position of the viewer (in world space), $(\phi_{\mathbf{v}}, \theta_{\mathbf{v}})$.

We can assume without loss of generality that the viewer v is located exactly above the 0-meridian at $(0, \theta_{\mathbf{v}})$ since any deviation in $\phi_{\mathbf{v}}$ translates directly to a simple ϕ-offset in the height map. Thus we need a mapping

$$f(\theta_{\mathbf{v}}, \tilde{\phi}, \tilde{\theta}) \rightarrow (\phi, \theta). \tag{5.1}$$

This mapping is a rotation around the y-axis as we chose the 0-meridian to intersect the positive x-axis (fig. 5.5).

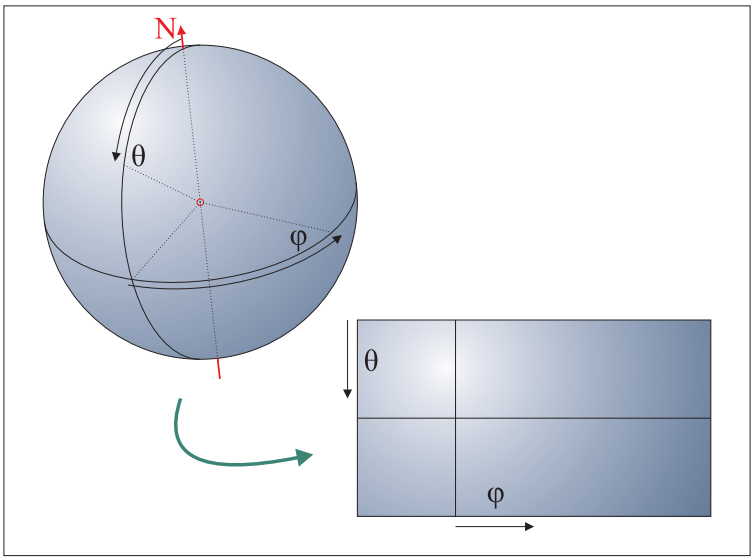


Figure 5.3: *The sphere can be parametrized by (ϕ, θ) which map directly to a planar rectangle.*

A point $\tilde{p}$ on the hemisphere with the local spherical coordinates $(\tilde{\phi}, \tilde{\theta})$ has the coordinates

$$\tilde{p} = \begin{pmatrix} cos\tilde{\phi} \cdot sin\tilde{\theta} \\ sin\tilde{\phi} \cdot sin\tilde{\theta} \\ cos\tilde{\theta} \end{pmatrix} \tag{5.2}$$

in view space. The rotation affects only the x and z coordinates, resulting in:

$$p = \begin{pmatrix} cos\theta_{\mathbf{v}} \cdot \tilde{p}_x - sin\theta_{\mathbf{v}} \cdot \tilde{p}_z \\ p_y \\ -sin\theta_{\mathbf{v}} \cdot \tilde{p}_x + cos\theta_{\mathbf{v}} \cdot \tilde{p}_z \end{pmatrix} \tag{5.3}$$

This point is converted back into spherical coordinates by:

$$\begin{pmatrix} \phi \\ \theta \end{pmatrix} = \begin{pmatrix} tan^{-1}\frac{p_y}{p_x} \\ cos^{-1}(p_z) - \theta_{\mathbf{v}} \end{pmatrix} \tag{5.4}$$

Note that we subtract $\theta_{\mathbf{v}}$ from θ to set the origin of the transformed coordinate system to the position of the viewer. This offset and the previously fixed $\phi_{\mathbf{v}} = 0$ define the focus point of the clip map. tan^{-1} has to take into account the quadrant in which (p_y, p_x) lies, similar to `atan2()` in C.

For a vertex on the hemisphere, we precalculate $\tilde{p}$ on the CPU and pass it as vertex attribute to the vertex shader where the next two steps are performed. The per-frame-constants $cos\theta_{\mathbf{v}}$ and $sin\theta_{\mathbf{v}}$ can also be calculated on the CPU and passed as uniforms to the shader.

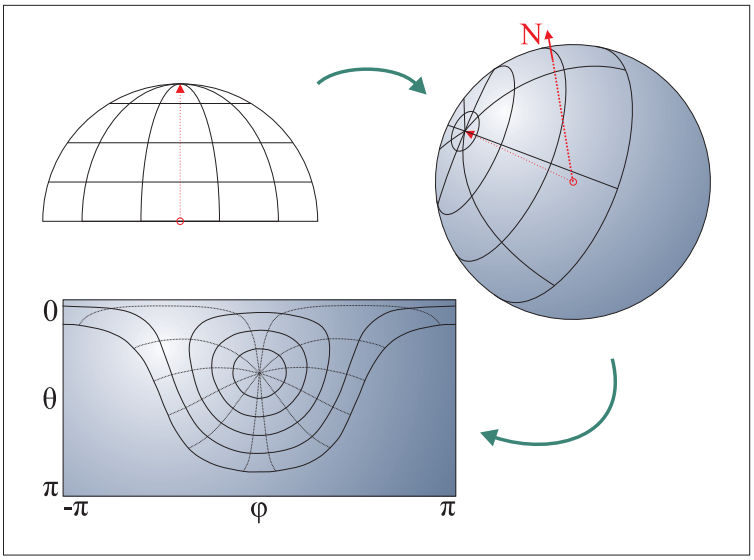


Figure 5.4: *Points on the view hemisphere are transformed into world space to sample the rectangular height map.*

5.3.4 Discretization

The original Geometry Clipmaps use a rectangular support geometry that is aligned to the grid of the underlying raster data. Since rectangular grids are the native representation of textures on current GPU and also quite common in cartography and artistic terrain generation, we continue to use it although it does not allow a direct correspondence of vertices to height samples. We use spherical coordinates to transform map the height texture (parametrized by (s,t)) to the spherical surface: $(s,t) = (\phi,\theta)$.

The hemisphere $(\tilde{\phi},\tilde{\theta})$ is discretized into quads. $\tilde{\phi}$ is simply divided into n fixed steps. The discretization of $\tilde{\theta}$ depends on the distance to the viewer: Low levels of detail (far away) require less steps per distance than higher levels. The first level of discretization divides the hemisphere into the concentric rings that shrink exponentially: Level i covers $\tilde{\theta} \in (2^{-i}\pi, 2^{-i-1}\pi]$. This sequence is terminated by a fill level that covers $\tilde{\theta} \in (2^{-i-1}\pi, 0]$. Each level is subdivided into m rings by $\tilde{\theta}_{i,j} = \theta_{i,0} * 2^{-j/m}$. Each discrete element of the hemisphere is then partitioned into two triangles. The resulting geometry ensures that the triangles have about the same size in screen space (fig. 5.6).

The disadvantage of this solution compared to the original GPU-based Geometry Clipmaps is the 1:1 correspondence of vertices and height samples had to be dropped. One advantage is that no special case handling is required to circumvent the T-intersections at level boundaries: The constant discretization

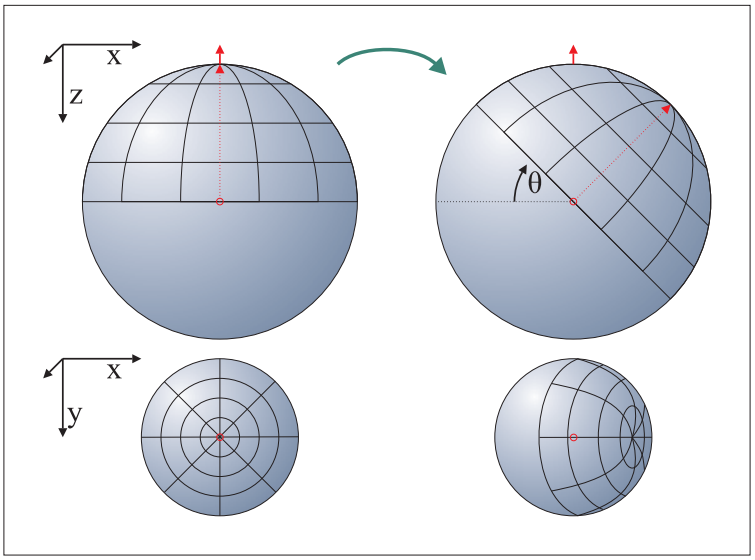


Figure 5.5: *For $\phi_{\mathbf{v}} = 0$, the hemisphere is rotated only around the y-axis.*

of $\tilde{\phi}$ implies the gapless geometry transition.

5.4 Algorithmic details

There are few more algorithmic differences to Asirvatham's Geometry Clipmaps that are caused by the new mapping:

5.4.1 Texture sizes in map space

The size of a support geometry level in map space is now dependent on the position of the viewer: A circular ring with a diameter of $1m$ covers a ϕ range of about $\frac{2\pi}{40{,}000}$ if the viewer is located at the earth equator. The same ring covers the whole 2π if the viewer is standing less than $1m$ away from the north pole. Therefore the map space range of the clipmap texture has to be chosen according to the current $\phi_{\mathbf{v}}$. There is no direct dependence on $\theta_{\mathbf{v}}$ apart from the fact that the texture can be clipped at $\theta < 0$ and $\theta > \pi$ since map sampling does not cross the poles.

This anisotropic range is shown in fig. 5.7. The level covers a map range of $\phi = 2\pi$ if the level diameter $\tilde{\theta}$ is less than the distance to the nearest pole:$(\tilde{\theta} > \theta_{\mathbf{v}}) \vee (\tilde{\theta} > \pi - \theta_{\mathbf{v}})$.

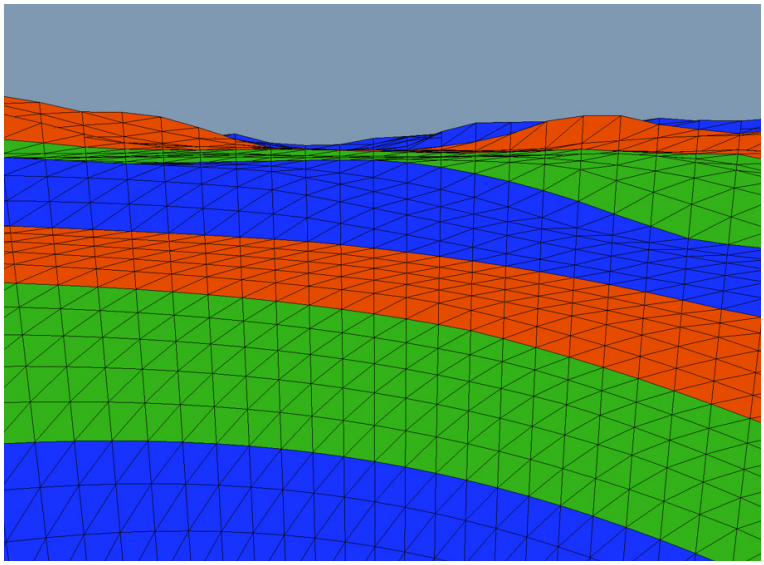


Figure 5.6: *Triangles twice as far away are rendered twice as small, so our expontially growing triangles have about the same size in screen space.*

One of the advantages of the wrap-around clipmap updates is that small movements of the viewer cause small texture updates. If we use the texture completely for any given ϕ-range, a small movement of θ_v would require an update of the whole texture. Therefore we change the texture range in power of two steps and use only a subset of the texture. For instance ϕ-ranges of 0.3π and 0.4π both result in a texture that covers 0.5π. This results in an ammortized complexity that matches the original Geometry Clipmaps.

5.4.2 Aliasing

The missing direct correspondence of height map samples to vertices introduces a possible source of aliasing: The base signal (height map) is resampled at a different rate by the support geometry. The triangles equal a linear interpolation which is a less than perfect reconstruction filter. Resampling at a frequency that is at least as high as the sampling frequency of the source signal ensures that the aliasing is minimized and no detail is lost (fig. 5.8) , but the lower the source signal rate, the lower the visual details of the terrain. A good choice is a resampling rate that roughly equals the source sampling rate: If you discetrize $\tilde{\phi}$ into n steps, then a texture width of $\frac{n}{4}$ is sufficient since this texture has about n boundary texels. Any texture size between this upper bound and $\frac{n}{8}$ should be fine, assuming that $\tilde{\theta}$ is discretized at a similar resolution.

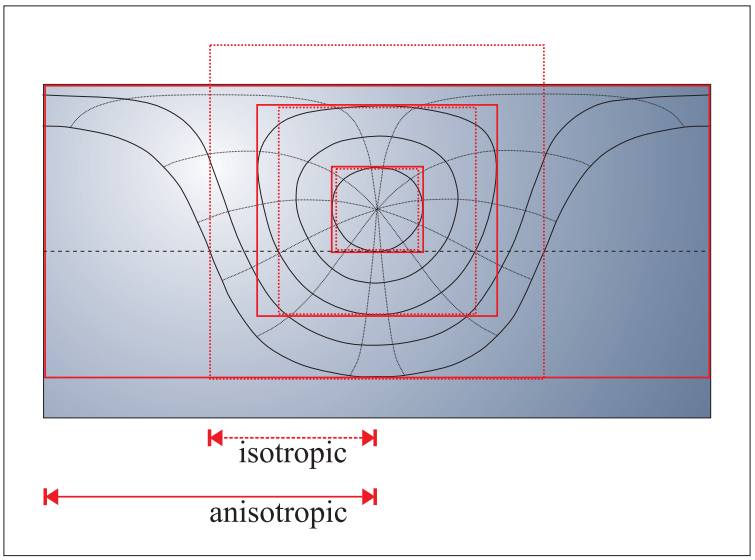


Figure 5.7: *The world space size of the clipmap regions have to be chosen according to θ_v to handle the anisotropy.*

5.4.3 Clipmap filtering

The clipmap pyramid is based on successively downscaled images. This scaling is performed in map space, but resampling by the support geometry vertices is performed in 3D world space. This introduces another possible source of aliasing since the source density in ϕ direction at the poles is far higher than at the equator. Having the same number of ϕ samples is merely an artefact of the chosen parametrisation, so the signal bandwidth has to be limited artificially. This can be done by using a special filter kernel: Common image resampling algorithms use circular kernels. They match the circular shape of the support geometry, so the same strategy can be applied. The filter kernel should be defined in 3D world space and transformed to map space as described in 5.3.3. This way the bandwidth is limited so that the resampling by the support geometry works as expected.

5.4.4 Texture coordinates beyond poles

The arcus tangent in the calculation of p'' in 5.3.3 works based on the assumption that the map wraps around in ϕ direction: The texture coordinates decrease towards $\phi = -\pi$ and increase towards $\phi = +\pi$. They meet at the π-meridian exactly beyond the pole. There's no problem mathematically, but the discretization causes an artefact at that point. The texture coordinates are in-

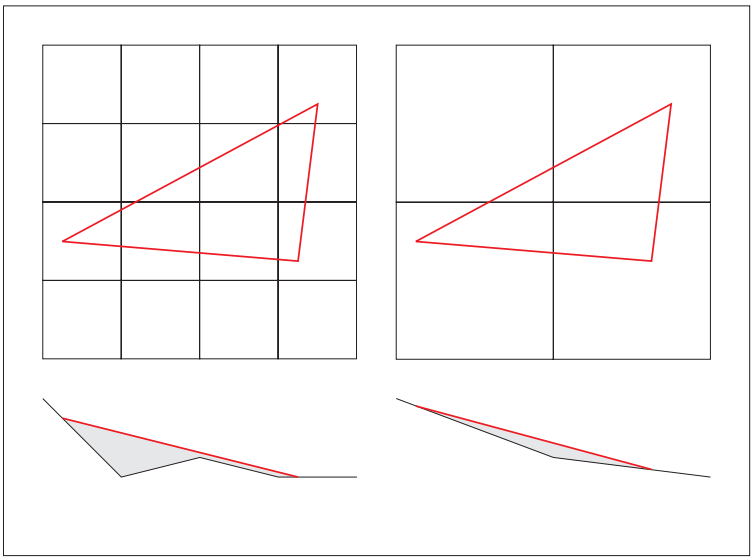


Figure 5.8: *draw: texture=source, geometry=sampling*

terpolated across the triangles, so the last triangle in one of the two directions interpolates from ϵ to 1 instead to 0 (fig. 5.9, left).

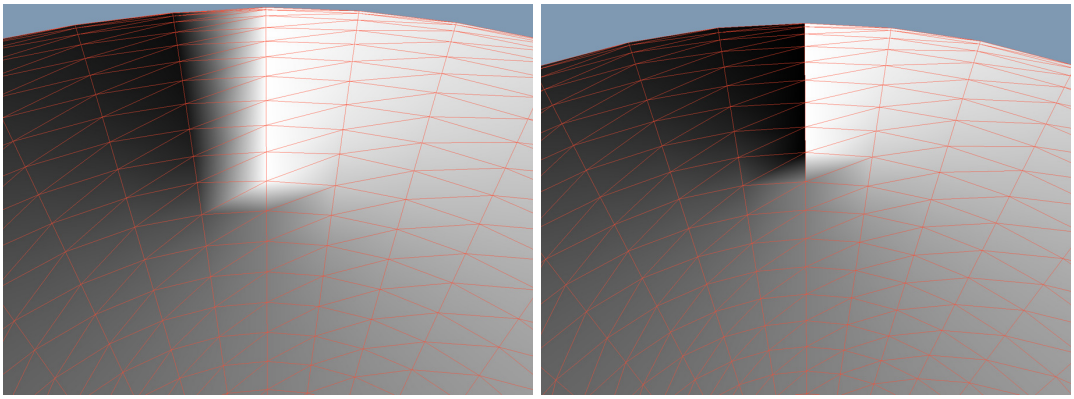


Figure 5.9: *Texture coordinate interpolation results in artefacts beyond the poles, so one line of vertices has to be duplicated.*

This can be fixed by duplicating the vertices on that meridian: Since the support geometry is always oriented with $\tilde{\phi} = 0$ in north direction, only one line of vertices requies special handling. One vertex of each pair gets a special attribute that is used in the vertex shader to correct the texture coordinate. We determine whether the pair lies beyond a pole (compare to $\theta_{\mathbf{v}}$) and subtract 1 from the calculated texture ϕ coordinate, so the interpolation across the triangles works as expected (fig. 5.9, right).

5.4.5 Level visibility

Not all circular rings of the hemisphere are visible from each position. The lower bound (low detail, far away) is determined by the earth curvature, the upper bound (high detail, near) by the height above the local surface. The lower bound can be estimated as shown in fig. 5.10: The height h of the viewer above the spherical planet surface (radius r) determines the tangent cone to the planet. The terrain beyond this $\tilde{\theta}_{max}$ is hidden by the earth curvature (note that the minimum level of detail corresponds to the maximum $\tilde{\theta}$):

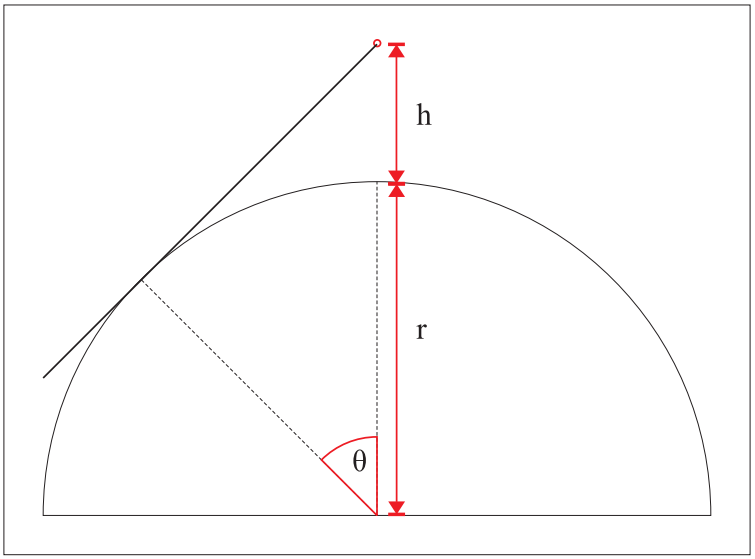


Figure 5.10: *Visibility of the lower levels of detail depends primarily on earth curvature and the distance to the surface.*

$$(r+h)\cdot cos\tilde{\theta}_{max} = r \quad (5.5)$$

$$\Leftrightarrow \tilde{\theta}_{max} = cos^{-1}\frac{r}{r+h} \quad (5.6)$$

This calculation does not take the slope of the terrain into account (e.g. high mountains might be clipped early), so you might want to add a safety factor to this approximation.

The upper bound is calculated based on the requirement that triangles should cover at least one pixel in screen space. The size s of one screen pixel on the surface of the planet depends on the height h of the viewer, the field of view angle fov and the number of pixels $\#p$ per scanline (fig. 5.11):

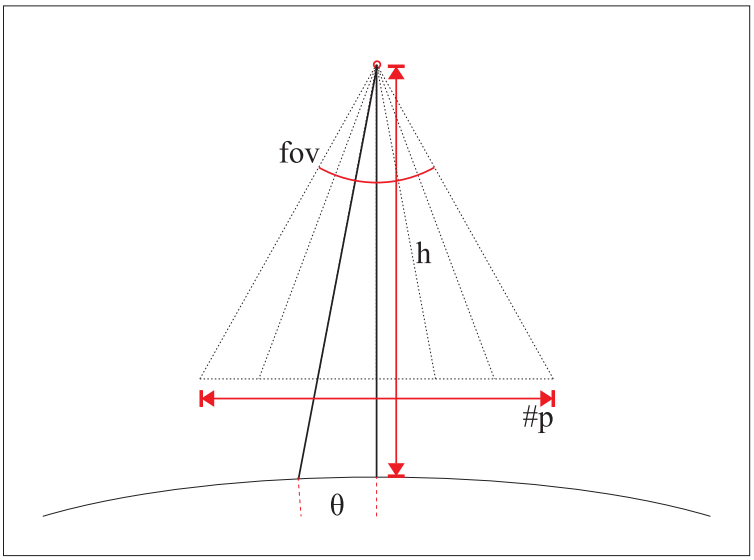


Figure 5.11: *Visibility of the higher levels of detail depends on the screen resolution and the distance to the surface.*

$$s \approx h \cdot tan\frac{fov}{\#p} \tag{5.7}$$

The upper bound $\tilde{\theta}_{min}$ follows directly:

$$\tilde{\theta}_{min} = \frac{s}{2\pi r} \cdot 2\pi = \frac{s}{r} \tag{5.8}$$

5.5 Implementation

The following aspects deal with the implementation on current consumer GPUs. Using vertex texture look-ups currently limits the technique to NVIDIA NV40 and G70 class GPU (Geforce 6600, 6800, 7800) since ATI does not support this feature up to the R520 line (Radeon X1800). A possible work-around is the render to vertex buffer support that allows using the pixel shader to calculate the actual vertex positions in a pre-pass. We focussed on the NV40 and found the following issues:

5.5.1 Trigonometric function replacement

Calling trigonometric functions in the vertex shader is a possible bottleneck. Our map transform algorithm relies on tan^{-1} and cos^{-1} that cannot be precalculated efficiently. But there's another way out: The distortion of the circular ring in map space is quite low for higher levels of detail (small $\tilde{\theta}$). Figure 5.12 illustrates this for $\tilde{\theta} < \frac{\pi}{128}$ and $\theta_\mathbf{v} = \frac{3}{8}\pi$. These inner rings can be transformed using a simple approximation for p''_ϕ:

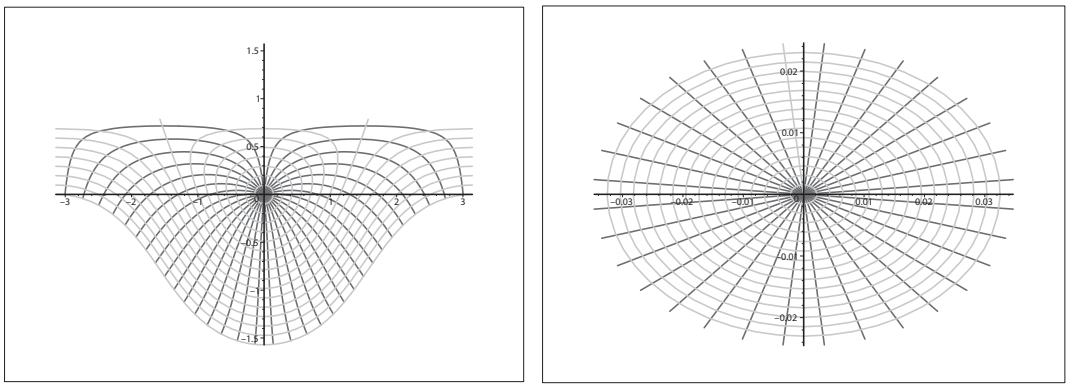


Figure 5.12: *$\tilde{\phi}$- and $\tilde{\theta}$-iso-lines in world space (ϕ, θ): Whereas the overall distortion of the mapped hemisphere is quite large, the area around the viewer is only stretched in ϕ-direction.*

$$\phi = tan^{-1}\frac{p_y}{p_x} \tag{5.9}$$

$$= tan^{-1}\frac{\tilde{p}_y}{cos\theta_\mathbf{v}\cdot\tilde{p}_x + sin\theta_\mathbf{v}\cdot\tilde{p}_z} \tag{5.10}$$

$$\approx \tilde{p}_y\cdot\left(1+\frac{\frac{1}{\pi}}{\frac{1}{4}-(\frac{\theta_\mathbf{v}}{\pi}-\frac{1}{2})^2}-\frac{(\frac{\theta_\mathbf{v}}{\pi}-\frac{1}{2})^2}{6}-\frac{4}{\pi}\right) \tag{5.11}$$

The distortion term depends only on $\theta_\mathbf{v}$ and can thus be precomputed on the CPU. This empirically derived approximation is tuned to the following setting: $\tilde{\theta}$ should be small, for instance $< \frac{\pi}{1024}$, and $\theta_\mathbf{v}$ should be not too near to the poles, e.g. $\pi\frac{1}{48} < \theta_\mathbf{v} < \pi\frac{47}{48}$. These limits result in an relative approximation error $1-\frac{approximated}{exact}$ (fig. 5.13) less than 0.001 for $\tilde{\phi} = \frac{\pi}{2}$. We consider this value acceptable for interactive rendering.

Apart from the slow computation, tan^{-1} has another drawback: The accuracy of `atan(y,x)` on the NV40 is quite limited for small `x`, so higher levels of detail show significant errors in the ϕ texture coordinate (fig. 5.14). The simple solution is to use the approximation formula at least starting at the levels that exhibit the incorrect behaviour.

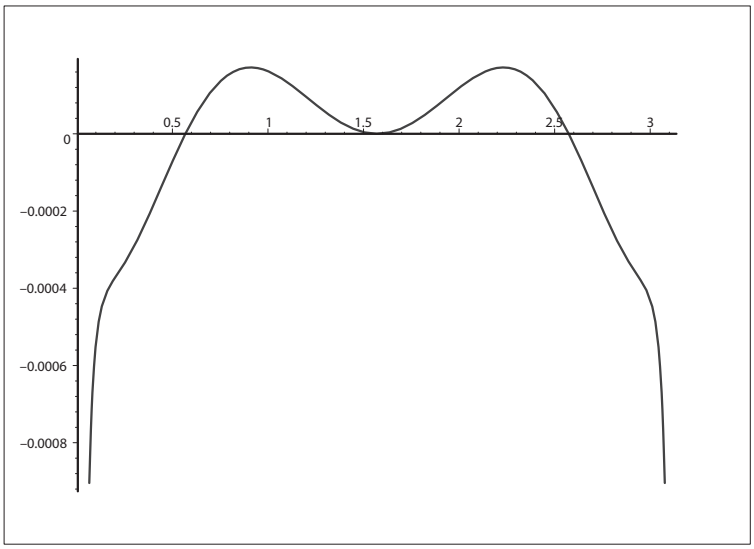


Figure 5.13: *The relative error of the approximation stays below* 0.001 *for* $\pi\frac{1}{48} < \theta_{\mathbf{v}} < \pi\frac{47}{48}$

As motivated above, the inner rings resemble a stretched circle. Therefore the θ-direction requires no further calculations and can be approximated as follows:

$$\begin{aligned} \theta &= cos^{-1}(p_z) - \theta_{\mathbf{v}} && (5.12)\\ &= cos^{-1}(-sin\theta_{\mathbf{v}} \cdot \tilde{p}_x + cos\theta_{\mathbf{v}} \cdot \tilde{p}_z) - \theta_{\mathbf{v}} && (5.13)\\ &\approx \tilde{\theta} && (5.14) \end{aligned}$$

This approximation is also usable under the previously mentioned conditions. If we take all possible view positions into account, $\theta_{\mathbf{v}}$ is relatively large compared $\tilde{\theta}$ in the higher levels of detail in most cases (except close to the north pole). Therefore the numerical error in the calculation of θ introduced by the difference $cos^{-1}(p_z) - \theta_{\mathbf{v}}$ dominates the error of this approximation, so this very simple approximation suffices. Nevertheless the result is visually acceptable.

Note that you should blend from approximated to exact calculation to avoid gaps in the terrain (fig. 5.15).

5.5.2 Speed

The main bottleneck is the vertex texture look-up: Since we had to drop the 1:1 correspondence, we have to use texture filtering to avoid the strong artefacts of nearest neighbor sampling. The NV40 is not capable of filtering vertex textures,

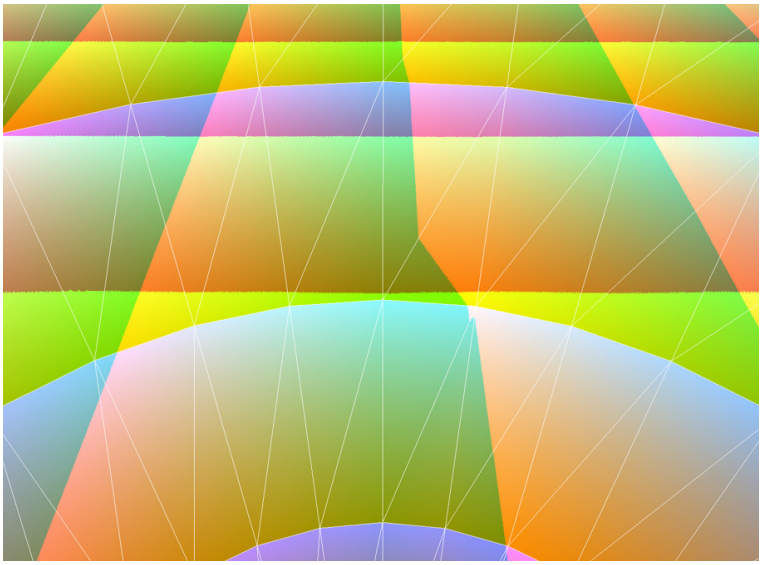


Figure 5.14: *The inaccuracy of the tan^{-1}-implementation causes distortion in the texture coordinate calculation: The central ϕ line should be straight, not jagged.*

but bilinear filtering can be emulated using 4 samples. The blending region between two levels of detail requires trilinear filtering (8 samples). This overhead hides any other possible bottlenecks, even the trigonometric functions do not affect the framerate in this case. It would be a major limitation of the whole technique, but we expect the vertex texture lookups to improve as soon as unified shader architectures become widespread: Common pixel shader units are especially designed to deal with the latency of texture lookups. Implementing this into vertex units in separated architectures would increase the chip complexity disproportionately for a feature that is rarely used in current games. With a unified shader architecture, vertex shaders could use the same technology almost for free, so we believe that this bottleneck will disappear in the next one or two years.

We benchmarked the algorithm on a Pentium 4 (2.4 GHz) system with NVidia NV40 GPU (325 MHz, Geforce 6800). The screen resolution of 1280×960 was no bottleneck since our implementation is vertex shader limited. The test data set consisted of a height map with $43,200 \times 21,600$ pixels (338 MB JPEG 2000 compressed) and a color map with $86,400 \times 43,200$ pixels (203 MB ECW compressed). The clipmap texture sizes for height map and color map are 128^2 and 512^2 respectively. $\tilde{\phi}$ and $\tilde{\theta}$ are discretized into 512 steps (20 $\tilde{\theta}$-steps per level). The approximative transform was used for levels $>= 10$. In this configuration, the following views were used: A overview over Lake Garda (Italy) from the south east with the camera standing on the ground, see fig. 5.16. Levels 6 to 21 were rendered ($\approx 330,000$ triangles per frame) at 25

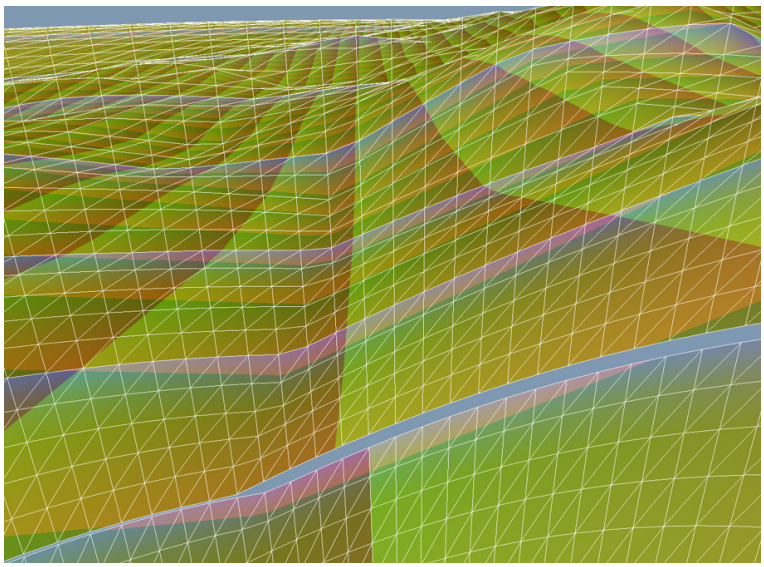


Figure 5.15: *Missing blending between exact calculation and approximation can lead to gaps.*

frames per second.

The second view shows the same area from an aircraft perspective (fig. 5.17). Levels 3 to 9 were rendered ($\approx 140,000$ triangles) at 40fps.

Increasing the altitude again resulted in the third test case, the space view (fig. 5.18). Levels 1 to 4 were rendered ($\approx 100,000$ triangles) at 65fps.

5.6 Conclusions

We presented an extension to the GPU-based Geometry Clipmaps by Asirvatham et.al. that handles spherical terrains. It performs well for a large range of view conditions from space (fig. 5.18) over aircraft heights (fig. 5.17) to a stroller's perspective (fig. 5.16). The implementation is simple and the special cases (texture coordinates beyond poles, arcus tangent accuracy) can be handled in a few lines of code. Additional textures such as color map and normal map can be handled using the same implementation without additional effort.

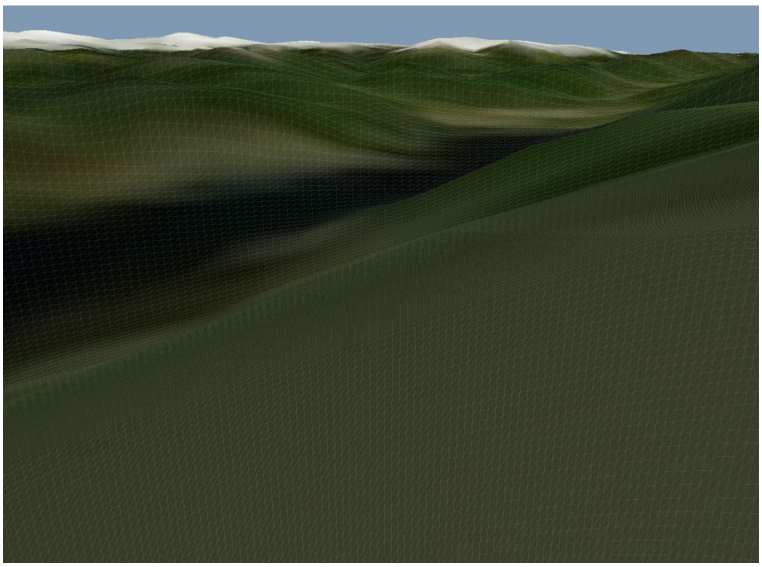


Figure 5.16: *Lake Garda, ground view*

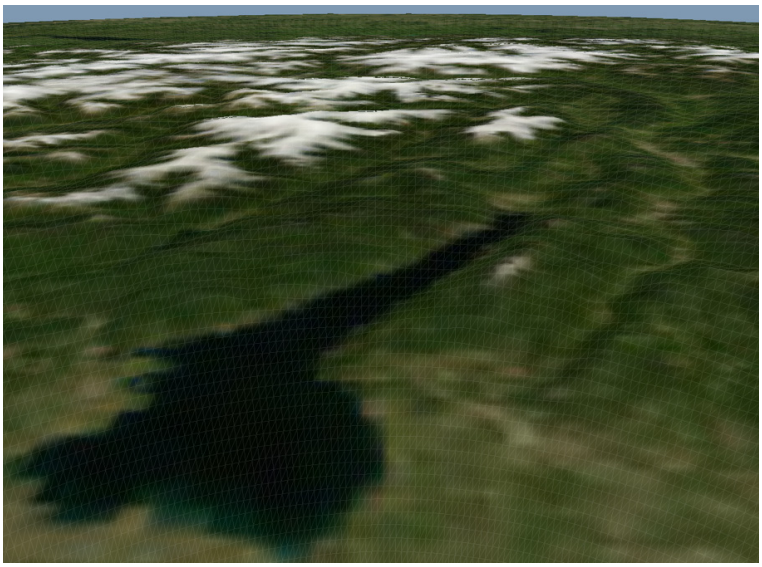


Figure 5.17: *Lake Garda, aircraft view*

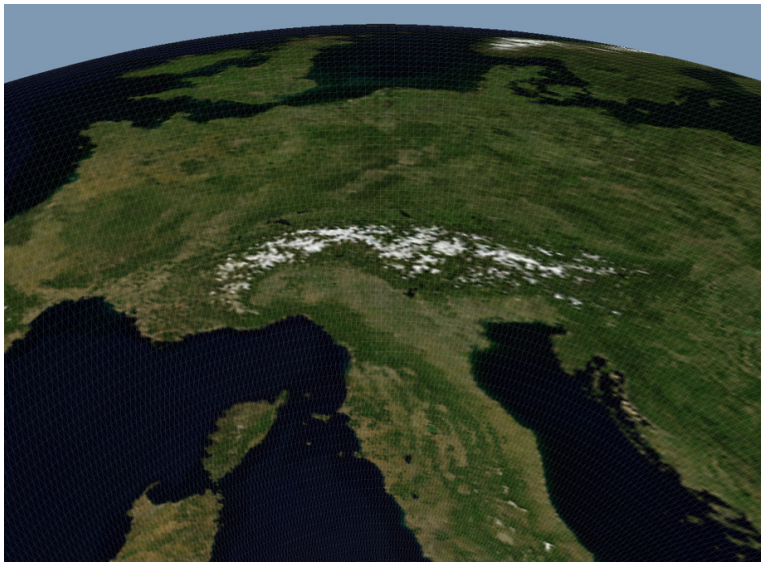


Figure 5.18: *Lake Garda, space view*

Chapter 6

Vegetation Rendering

6.1 Introduction

In chapter 3, we created a LoD hierarchy with the purpose of minimizing time and aliasing in the following renderer. In this chapter, we show how this hierarchy can be rendered efficiently. First, we transform the hierarchy into a sequential tree as proposed by Dachsbacher et al. [2003]. Then we show how to select the appropriate LoD for a specific instance. This instance can then be rendered using GPU raycasting. Depending on the source model, the shading can be improved by precomputing the ambient occlusion term.

Parts of this chapter have been published in [Clasen and Hege, 2005] and [Clasen and Prohaska, 2010].

6.2 Generalized Sequential Primitive Tree

For rendering, the LoD tree is transformed to a sequential generalized primitive tree (from coarse to fine) and stored on the GPU as a whole. For each instance a prefix of the tree is processed using the vertex shader to determine which primitives don't match the error criteria and should be omitted. Transformation and rendering follow directly from Dachsbacher et al. [2003], with the only differences being r_{min} replaced by our node error e, and r replaced by an error threshold e_{max}. Since we don't have an inherent parent-child relationship between the ellipsoids, we compute the error thresholds for whole LoD steps (Sec. 6.3), not for single primitives. For incremental updates during the simplification step, we omit the sorting step and add or replace only the changed nodes (parent and two children) for each merge candidate.

6.3 Level of Detail Selection

The appropriate level of detail for a given model size in screen space is constrained by two soft limits: When the feature size falls below the limit of the Nyquist frequency, aliasing occurs. In this method, the feature size is determined by the size of the lines and ellipsoids, because the noise texture is already frequency capped. The hard edge of the primitives has a theoretically unlimited frequency, so aliasing cannot be avoided. Without resorting to supersampling, we can only reduce it by reducing the number of edges, which means lowering the level of detail to increase the primitive size. This leads to the second soft limit, the artifacts introduced by large primitives. Fig. 3.1 shows that a certain number of primitives is required to faithfully represent a model for a given resolution. We use a default average ellipsoid size of 5 pixels in diameter and a default average line width of 1 pixel. This sounds small, but given the uneven distribution of line widths between trunk and twigs, the most visible lines are a few pixels wide.

Based on these sizes, we set the error value of each primitive to the inverse of the target resolution pixels per meter in screen space. This allows us to merge primitives of different sources (lines and ellipsoids of a single model, or multiple models to a group) without additional error normalization steps.

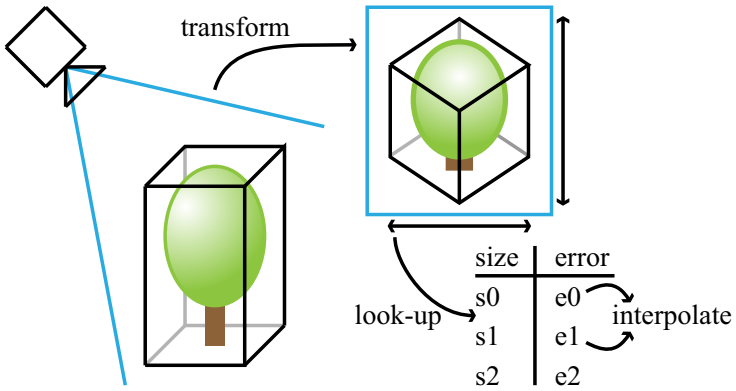


Figure 6.1: *To estimate the maximal allowed error of an instance, we transform its bounding box to screen space and look up the size in the error table.*

At run-time, we compute the screen-space resolution of the bounding box of an instance and use this size to interpolate the necessary LoD from the table (Fig. 6.1). If the target resolution is higher than defined in the look-up table, we switch to the source model for high-quality close-ups.

6.4 Primitive Raycaster

Our simplified plant models consists of lines and ellipsoids. Even though OpenGL supports line primitives directly, we use a custom raycaster for both types of primitives because the OpenGL lines are shaded in a single solid color. Image space screen-door blending as described by Mulder et al. [1998] hides the difference between the highest LoD and the original mesh without depth sorting of the individual primitives.

6.4.1 Ellipsoids

We render the ellipsoids using the raycaster presented by Sigg et al. [2006]. It computes the ray-ellipsoid intersection in the local coordinate system where the ellipsoid is a sphere. This way we can use the coordinates of the intersection as index to the noise cube map, and simply add the normal noise to the normal vector before transforming it to eye space. To avoid aliasing due to the noise texture, we limit the noise frequency based on the screen-space derivatives of the texture coordinates, just like common mip-mapping. The OpenGL function `fwidth()` provides a ready-to-use upper limit on the mip-map level.

The approach by Sigg et al. [2006] includes one premature optimization: Omitting the third component in the equation for the center position,

$$v_c = (r_1^T D r_4, r_2^T D r_4, 0, r_4^T D r_4)^T, \tag{6.1}$$

is correct in symbolic calculations and saves a few computations, but it is numerically unstable for oblate ellipsoids. When the position on the image plane ($z = 0$) is transformed into the local ellipsoid coordinate system, the precision of 32 bit float values can become a bottleneck: If an ellipsoid is about 1 mm flat and 10 m away from the camera, we have a range of 10^4, which leaves little room for calculations and sloppy model definitions like extremely flat leaves. This results in artifacts similar to the common z-fighting of coplanar triangles. When adding the third component,

$$v_c = (r_1^T D r_4, r_2^T D r_4, r_3^T D r_4, r_4^T D r_4)^T, \tag{6.2}$$

all distances are computed relative to the origin of the ellipsoid, which minimizes the artifacts. Apart from this change in the vertex shader, you also have to add the z-coordinate of the center position to the z-coordinate of the intersection point in the fragment shader to yield the correct depth value.

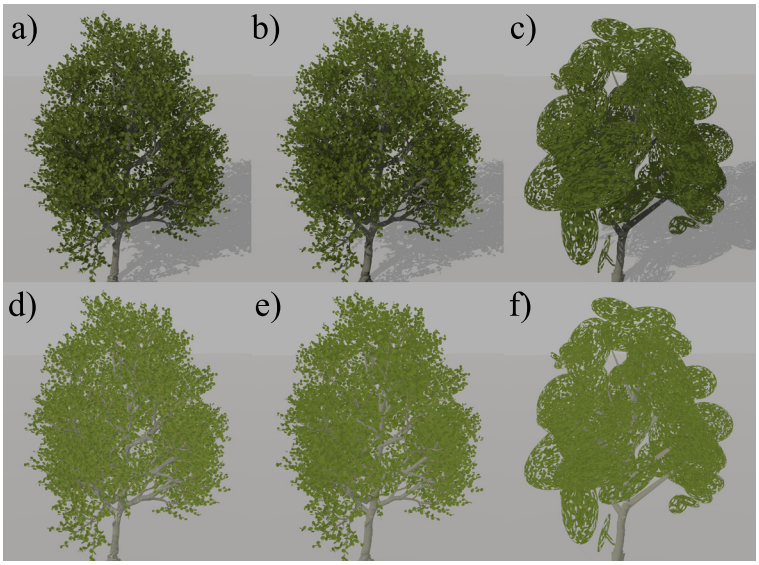


Figure 6.2: *We use shadow mapping and precomputed ambient occlusion (top) to improve the realism over local illumination only (bottom). Left to right: Mesh, highest LoD, coarser LoD.*

6.4.2 Lines

The line renderer is a simplified version of the raycaster proposed by Merhof et al. [2006]. We use only the triangle-based method, not the hybrid version including point sprites, since the resulting artifacts are negligible in our case.

6.5 Shading

We defer the shading of the raycasted primitives using G-Buffers as proposed by Saito and Takahashi [1990] and implemented on the GPU by Hargreaves [2004]. To do this, we have to store all information necessary for shading in the geometry rendering pass. Actual evaluation of the G-Buffer is described in chapter 7. Position (depth) and surface normal are the result of the raycaster or the triangle rasterizer respecively, while material properties such as albedo (the color) and glossiness are given as attributes of the source model.

There is not a single standard for models, but a set of more or less common attributes. Some attributes have to be set manually by modellers. Apart from the shape of the surface, this includes albedo and glossiness. We assume that these are given. Other attributes can be derived. Surface normals simply are the first derivative of the surface, yet they are so common that we rely on

the input models to specify these for triangle meshes. Ambient occlusion as described by Landis [2002] on the other hand is less common. While there are models including this information, we implemented a preprocessing step for those who don't. Combined with shadow mapping, this improved the visual quality significantly (Fig. 6.2). The precomputation works as follows:

6.5.1 Triangles

For each vertex of the triangle mesh, we select a set of random directions in the hemisphere around the surface normal. For each direction, we compute the incoming light using a Monte Carlo Path Tracer (MCPT) in a white (radiance = 1) environment. The average over all direction is the incoming ambient light term, which lies in the target range of $[0, 1]$ by construction. Using a full MCPT instead of evaluating only primary or secondary rays, or capping at a fixed distance from the sample point, improves the accuracy for the inner leaves of a tree top, where most of the light is indirect over several bounces.

6.5.2 LoD Primitives

For our LoD primitives, ellipsoids and lines, we use a slightly modified ambient occlusion computation. Instead of the triangle vertices, we use random points on the primitive surface and the associated surface normals. For each random point, only a single random direction is evaluated. This follows the design principles of MCPT: Using more random points with less directions per point yields more accurate approximation of the overall ambient occlusion term than few points with many directions. Random points which lie inside other primitives are discarded, as they are not visible when rendering the model.

Since the primitives are part of a LoD hierarchy, the path tracer has to take into account whether two primitives are visible at the same time. For the ellipsoids, we could evaluate each LoD step separately, but for the lines we have a quasi-continuous LoD. Therefore we chose a unified approach where the LoD error is used as a fourth coordinate w in addition to the three spatial coordinates x, y, z. The fourth coordinate of the bounding box of each primitive is determined by the minimal and maximal error for which the primitive is drawn. For the path tracer, we set the origin o_w to the error value of the currently evaluated primitive and the direction $d_w = 0$. This way we can use any spatial data structure such as a kd-tree to accelerate the rendering without additional changes. To respect the blending between the levels of detail, we compute the translucency of each intersection and decice by randomization whether a primitive is

#primitives	1 k	5 k	10 k	27 k
time[min]	6	15	22	54
time[%]	85	65	24	14

Table 6.1: *Precomputation times for the ambient occlusion term by number of source primitives for 128 samples per primitive. Center row: Absolute time in minutes. Bottom row: Relative to the LoD generation time.*

hit. This integrates well into the MCPT design.

6.5.3 Calibration

The resulting ambient occlusion terms of the source mesh and the LoD primitives usually do not match. The simplification process takes only the shape into account, not the effect on illumination. Therefore we have to add another calibration step. For each discrete level in the ellipsoid hierarchy, we shoot randomized rays from outside into the model and gather the ambient occlusion term at the intersection point. Then we compare the resulting average ambient occlusion to the average ambient occlusion of the source mesh and scale the primitive terms accordingly. This way we ensure that the overall brightness remains constant, which minimizes the visible artifacts.

6.6 Results

Precomputing the ambient occlusion term is typically done in a batch process after the simplification. We found 128 samples per primitive sufficient, a further increase had no visible effect. For this number, the timings were still lower than the simplification timings (table 6.1), but in the same order of magnitude. Note that the ambient occlusion computation scales better with an increasing number of primitives, probably due to the low algorithmic complexity of raytracing spatial datastructures.

We compared the image quality of our precomputed ambient occlusion term (Fig. 6.3(c)) to a rendering without ambient occlusion (Fig. 6.3(a)) and to the Screen Space Ambient Occlusion (SSAO) method presented by Bavoil and Sainz [2008] (Fig. 6.3(b)). While SSAO works fine for edges in architectural models, it does not capture the complex interreflections inside tree models. All three methods have the same rendering performance characteristic at run-time. While SSAO requires an additional post-processing pass, it only depends on the resolution of the rendered image, not on the scene complexity. For forest scenes,

(a) (b) (c)

Figure 6.3: *(a) Having no ambient occlusion term results in bright images without depth cue. (b) Screen Space Ambient Occlusion generates blurry darkened blobs. (c) Precomputed ambient occlusion yields a realistic impression.*

the image space overhead is negligible. Applying the ambient occlusion term in the fragment shader requires only a single multiplication. This overhead is below the precision of measurement.

The run-time performance of the renderer is presented in section 3.6.

Chapter 7

Shading

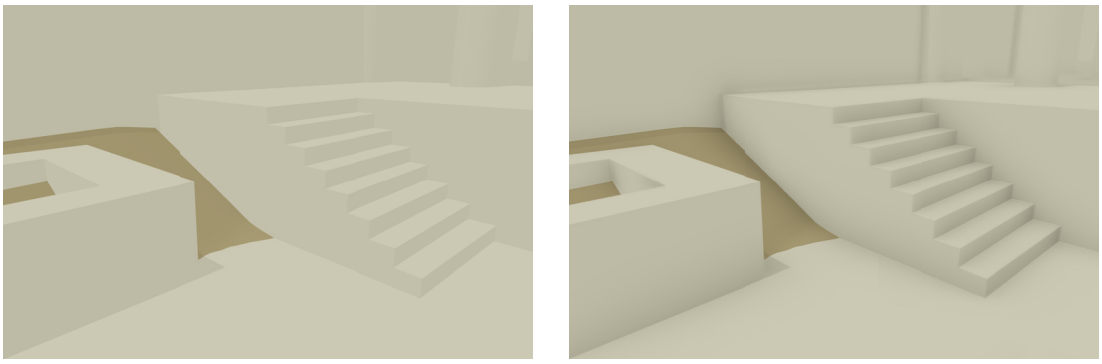
In the rendering passes for terrain and vegetation, we gather all information necessary for shading in the G-Buffer first presented by Saito and Takahashi [1990] and implemented on the GPU by Hargreaves [2004]. This allows us to apply relatively expensive shading operations which would be difficult using forward shading, given the inherently high overdraw of vegetation scenes.

7.1 Illumination

We target photorealistic images. Kajiya [1986] introduced the rendering equation, which models the physical behavior of surface reflections. We use the BRDF by Schlick [1993] for all surfaces, since it is physically plausible and can be parametrized to model a wide range of materials. As usual in GPU-based rendering, we model the incoming radiance by two terms: Direct lighting by a directional sunlight, and indirect ambient light.

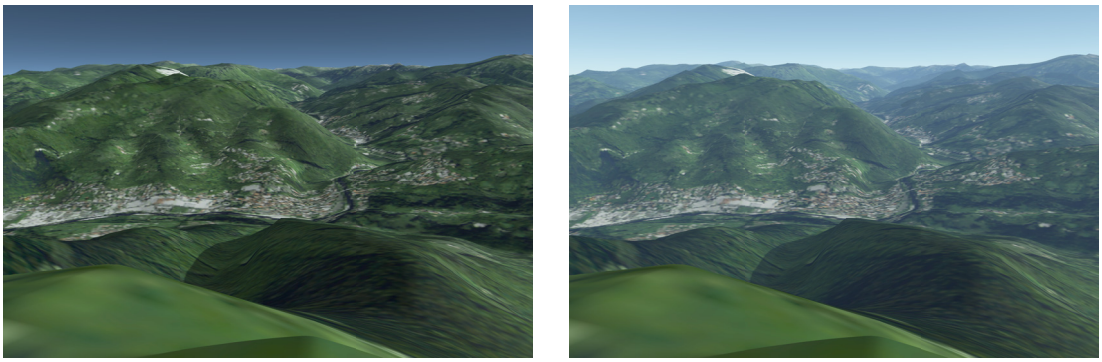
We compute the shadows for the direct lighting term using the Cascaded Shadow Maps (CSM) method first introduced by Zhang et al. [2006]. CSM is robust enough to be useful for outdoor scenes at any light and view direction, so we ported the reference implementation by Dimitrov [2007] from Direct3D to OpenGL without additional changes.

The ambient light term affects all surfaces, regardless of orientation or position. Landis [2002] suggest adding an Ambient Occlusion (AO) factor to attenuate the ambient light. AO is a property of the rendered surface, similar to albedo. It is usually specified per vertex or in a light map texture. For models supporting AO, for example plant models with the preprocessing from section 6.5, we store this value in the rendering pass in the G-Buffer. For surfaces with-



(a) (b)

Figure 7.1: *Screen Space Ambient Occlusion effectively emphasizes edges in architectural models.*



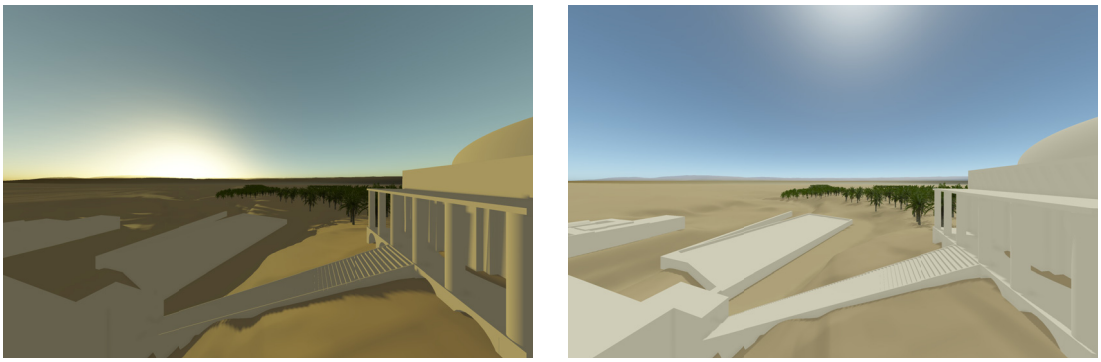
(a) (b)

Figure 7.2: *Atmospheric scattering is an important depth cue for distant mountains.*

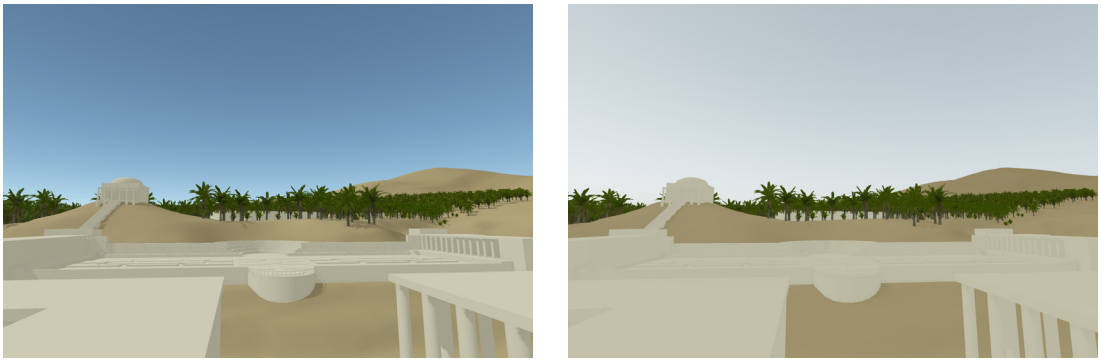
out associated AO value, we compute it on-the-fly based on the G-Buffer using the horizon-based Screen Space Ambient Occlusion (SSAO) by Bavoil and Sainz [2008]. While SSAO does not capture complex interreflections (Fig. 6.3(b)), it is quite effective at emphasizing edges in architecture models (Fig. 7.1).

7.2 Atmospheric Scattering

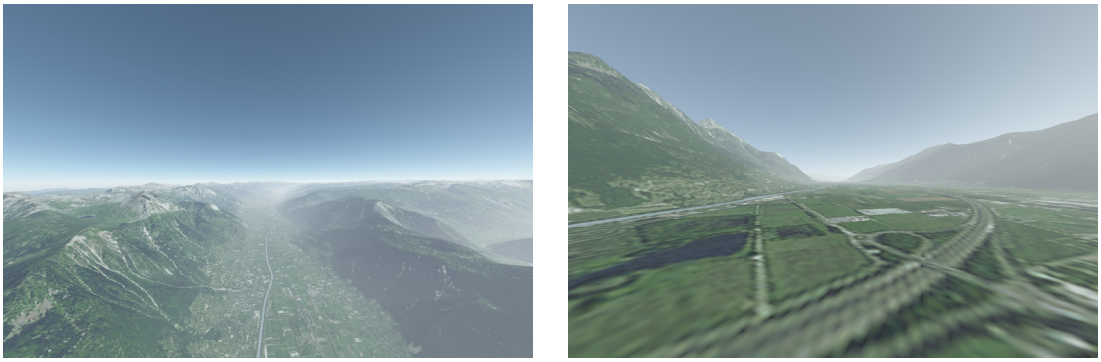
On the scale of typical landscape visualizations, the atmosphere cannot be treated as transparent medium. Pale, foggy and slightly blue mountains are an important depth cue (Fig. 7.2). The color of the sky not only indicates time and weather, but also the current altitude (Fig. 7.3). While these effects can be handled separately, for example using skybox textures and depth-based blend-



(a) (b)

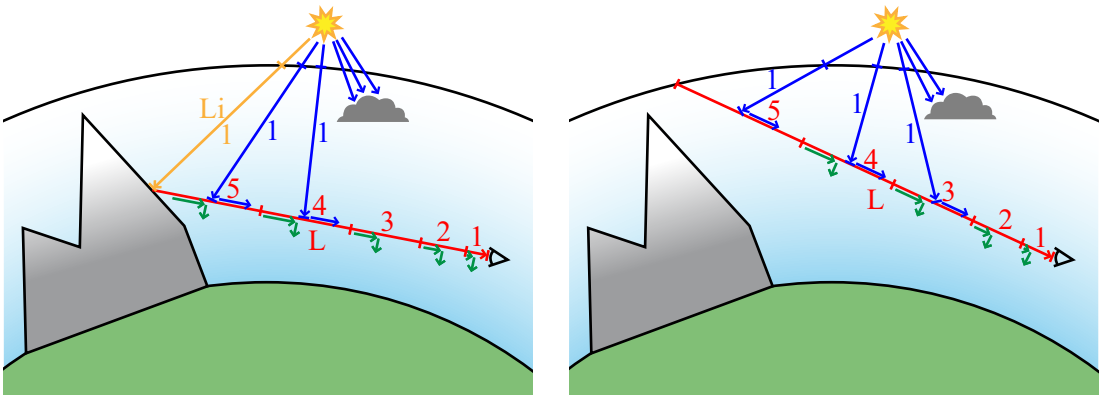


(c) (d)



(e) (f)

Figure 7.3: *The color of the sky caused by atmospheric scattering indicates time of day (a, b), weather conditions (c, d) and altitude (e, f).*



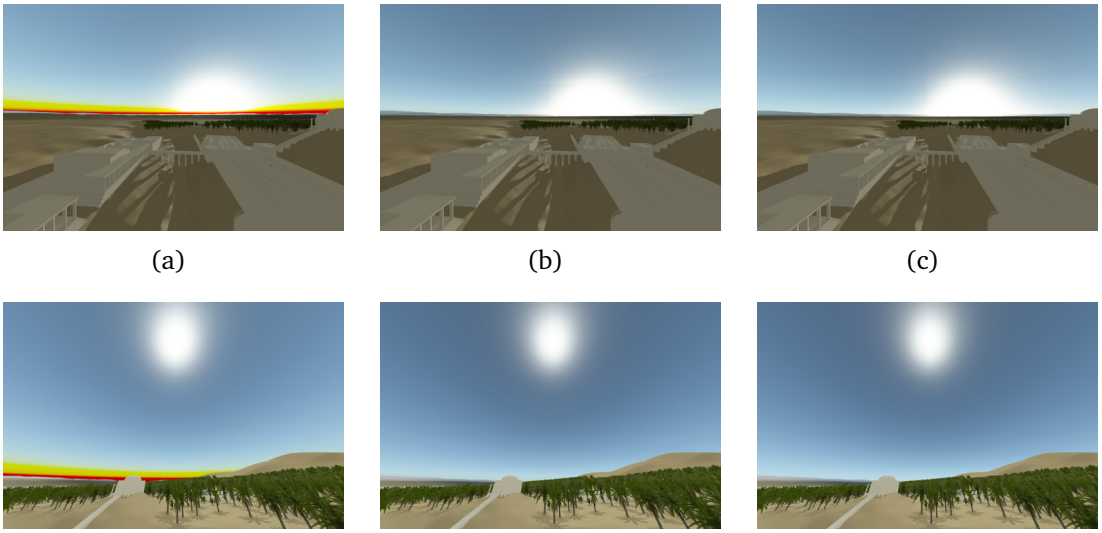
(a) (b)

Figure 7.4: *We sample the ray L from the viewer to (a) the nearest surface (illuminated by L_i), or (b) the boundary of the atmosphere, in discrete steps. For each step, we compute in- and out-scattering into and from L. For in-scattering and L_i, we attenuate the sunlight in a single out-scattering step over the interval from the boundary of the atmosphere to the in-scattering position. We use shadow mapping to avoid in-scattering into shadowed positions.*

ing to a constant color (depth fog), they are both visual effects of the same physical principle, the scattering of light at the molecules in the air.

We model the atmosphere by using the Rayleigh and Mie scattering for a single scattering step, similar to [O'Neil, 2005]. For each pixel in screen space, we perform ray marching to the nearest surface and compute in- and out-scattering (Fig. 7.4). Depending on the light and view directions, 12 to 36 steps are sufficient for a realistic approximation (Fig. 7.5). Note that we do not distinguish between terrain and sky shading: When shading the G-Buffer, we process all pixels in a uniform way. When no surface is hit, we stop the ray marching at a user-defined height, where the atmosphere is thin enough to be considered transparent. On earth, 50 km is a practical value.

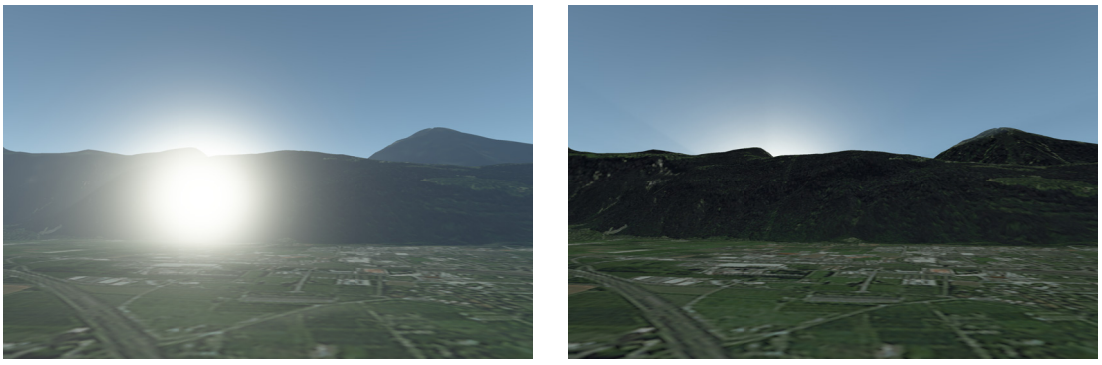
When computing the in-scattering, we have to sample the shadow map to determine whether there is actually light that can be scattered into the view. Ignoring this leads to distracting glowing regions at the position where the sun would be if it was visible (Fig. 7.6). This requires a quite high number of texture accesses to the shadow map, but it did not become a bottleneck in our experiments.



(a) (b) (c)

(d) (e) (f)

Figure 7.5: *The required number of steps depends on the light and view directions: 4, 12 and 36 steps for dawn (top row) and noon (bottom row): While using 12 steps still yields small artifacts at dawn (note the curved line at the top of the glow), there is no visual difference between 12 and 36 steps at noon.*



(a) (b)

Figure 7.6: *It is important to use shadowing when computing the inscattering to avoid glowing regions around the hidden sun.*

Chapter 8

Applications

The following chapter is based on the content already published in [Paar and Clasen, 2007], [Clasen and Paar, 2008] and [Paar et al., 2008].

8.1 Interactive Visualization with Biosphere3D

The methods described in the previous chapters are implemented in the open-source software Biosphere3D. Visualization projects with Biosphere3D start with a coarse representation of the entire earth (Fig. 8.1). This ensures that there are no end-of-the-world artifacts. The user then adds further data to refine and adjust the visualization. No precalculations are required, all settings can be changed interactively. This includes for example layer visibility, scenarios (content presets which can be blended), camera settings, atmosphere properties and all rendering preferences such as shadow map resolution. This enables quick development cycles and semi-interactive participation processes, where scenes are modified according to stakeholder requests.

Biosphere3D supports various file formats for the different kinds of landscape data (Fig. 8.2):

Relief (digital elevation models)

- JPEG2000 (`.jp2`)

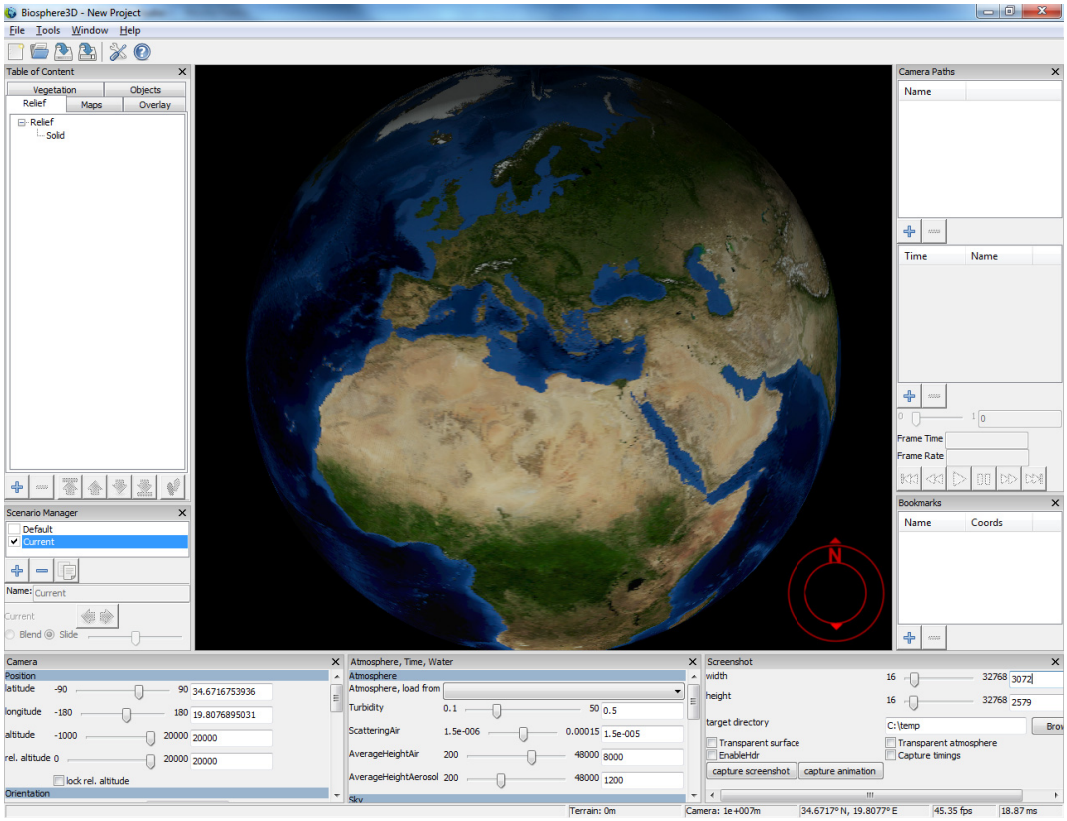


Figure 8.1: *Biosphere3D starts with a coarse globe.*

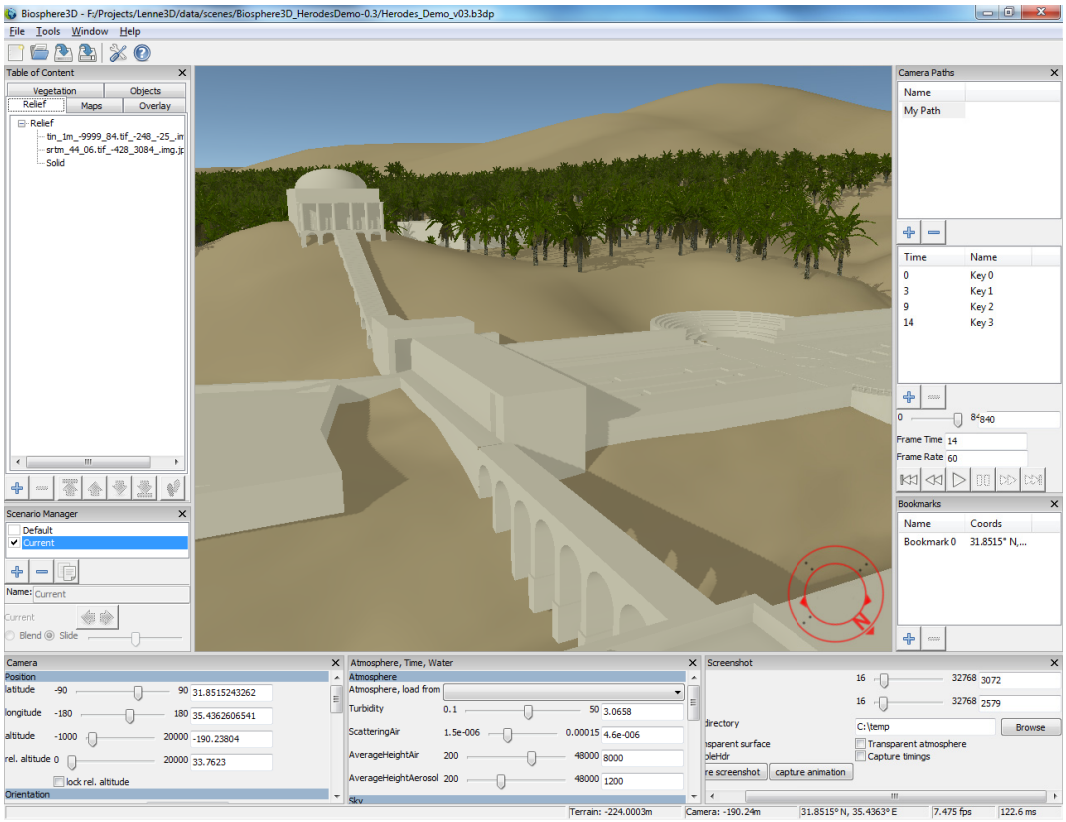


Figure 8.2: *A Biosphere3D project consists of Relief data (digital elevation models), Maps (color surface textures), Vegetation (plant instances), Objects (buildings), and Overlays (screen-space images).*

Maps (color surface textures)

- JPEG2000 (`.jp2`)
- ERMapper Compressed Wavelet Raster (`.ecw`)

Vegetation (plant instances)

- ESRI Shapefile (`.shp`)
- KML (`.kml`, `.kmz`)
- Lenné3D ASCII Ecofile (`.eco`)
- Lenné3D Plant Meshes (`.txf`)
- Lenné3D Plant Billboard Clouds (`.txfc`)
- Lenné3D Interchange Plant Format (`.lipf`)

Objects (buildings)

- ESRI Shapefile (`.shp`)
- KML (`.kml`, `.kmz`)
- Collada (`.dae`)

Overlays (screen-space images)

- PNG (`.png`)
- JPEG (`.jpg`)

The Lenné3D formats `.eco`, `.txf` and `.txfc` have their origin in the predecessor system Lenné3D Player, presented in Werner et al. [2005]. We support these to allow a smooth transition to Biosphere3D. Most new content is created using the standard formats.

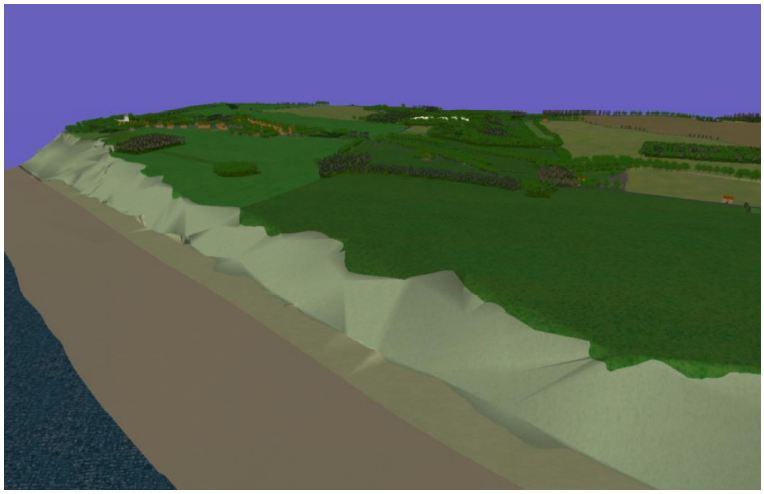


Figure 8.3: *Screenshot created in Presagis Terra Vista of an interactive visualisation developed by Brown et al. [2006] showing the coastal environment in 2001.*

Biosphere3D is a pure visualization system. Users can only adjust the rendering settings as mentioned above, not the data itself. This is a design decision: Different applications require different editing capabilities, and full GIS editors such as ESRI ArcGIS are highly complex tools. Therefore we minimized the loading times, so external editors can be used to modify the data with low turn-around times to the updated visualization.

8.2 Interactive Visual Simulation of Coastal Landscape Change

In [Paar et al., 2008], we described and evaluated the application of Biosphere3D in a case study of cliff erosion in Norfolk, on the eastern coast of England. It is a typical case where communication between decision-makers and members of the public is important: The cliffs between Sheringham and Happisburgh, currently up to 40 m high, are subject to rapid erosion of 0.5 m to 2 m per year. It is probably not possible to protect all parts of the coast, so priorities have to be set. This involves thorough evaluation of scientific facts and careful communication with all stakeholders, especially with the residents.

The Tyndall Centre for Climate Change Research previously worked on the visualization of cliff erosion scenarios and policy options using Presagis Terra Vista (Fig. 8.3, [Brown et al., 2006]). While this visualization showed the basic effects of cliff erosion, realism was limited. The lack of a true horizon and

Figure 8.4: *Biosphere3D enables the visualization of foreground vegetation.*

proper illumination create the impression of an abstract model. Only coarse models of selected buildings and trees are added, which has a negative impact on views close to the ground.

For [Paar et al., 2008], we imported the data from [Brown et al., 2006] to Biosphere3D. To enhance realism, we added a DEM based on the hole-filled SRTM V3 data provided by the King's College, London, and the Blue Marble satellite image by NASA. This ensured that the scene does not end at the end of the study area, but extends to the natural horizon. We further added more plant models and developed a reflective water shader to create more realistic ground-level views on land (Fig. 8.4) and sea (Fig. 1.3). This increased the usability for decision making processes as described by Sheppard [2005], who analyzed the potential role of realistic visualizations in climate change communication. It also showed that true realism is yet to be achieved: Our terrain model does not support true 3D surfaces with overhangs, a common property of cliffs. Furthermore, a visually appealing presentation could profit from animations. While we currently only support image-space blending between scenarios, model morphing could make the change itself more comprehensible. This would require linking the simulation software closer to the visualization system, because a transition based solely on the geometric properties of the final states could be misleading. The tile generation framework presented in section 4.4.1 already supports this kind of on-the-fly generation.

Chapter 9

Conclusions

9.1 Contributions

In this thesis, we presented the core components of a landscape visualization system. Based on the observation of the requirements of landscape planners and other landscape visualization users, we identified the aspects where existing solutions were insufficient: Terrain rendering, plant rendering, plant preprocessing and data management. We presented the published solutions for these tasks and additional shading improvements. Finally we presented the application of Biosphere3D to a climate change visualiztion scenario.

Level of Detail for Vegetation describes a novel Level of Detail method for plants based on ellipsoid and line primitives. Individual plant models are simplified using a combination of fuzzy partitional clustering for the leaves and agglomerative clustering for branches. The solution outperforms previous methods up to 6 times at run-time.

Tiling describes a unified framework to generate and manage scene tiles for both terrain and vegetation. We show how a dependency graph can be used to compose scenes on-the-fly. Only visible portions are generated and cached, which enables composition of terabyte-scale source data.

Terrain Rendering shows how to render the generated digital elevation model and surface texture. We map the clipmap data structure to the spherical domain. This combines the advantages of minimal memory transfer overhead

with the realism of true planetary surfaces. A unified texture handling is used to process elevation, color and normal data.

Vegetation Rendering brings the previously simplified plant models to the screen. We use a sequential tree data structure to handle the models on the GPU, where they are rendered using raycasting. We achive a realistic look by adding noise textures for alpha-test and surface normals, and a proper ambient occlusion term.

Shading processes the G-Buffer filled by terrain and plant renderer to the final image. We use deferred shading to enable otherwise expensive atmospheric scattering effects. We use shadow mapping when computing the atmospheric in-scattering to avoid leaking light artifacts.

9.2 Future Directions

In retrospect, we conclude that the work at hand is a major step towards interactive landscape visualization, but only a step. The resulting open-source software package is already widely used for both academic research and commercial visualizations, yet there is room for improvements:

3D Terrain is necessary to model overhangs. In our cliff visualization, we observed that this is a real-world demand which currently has to be worked around by using 3D models in addition to the 2.5D terrain. This involves not only an update to the visualization system, but also to the data source applications, where 2.5D terrains are still state-of-the-art.

Plant Groups could lower the burden of large numbers of plants. While we can already simplify plants to single primitives, the overhead of the draw calls becomes the next bottleneck. Instancing support on the GPU can alleviate this, but only to the moment when a single primitive plant is smaller than necessary. At this point, plant groups could be the base of a futher LoD hierarchy, eventually converging to surface textures for very large distances.

Interactivity is currently enabled by employing fast rendering methods, but there are no guaranteed frame-rates. Moving the LoD selection for both terrain

and vegetation to a time-budget based solution could improve the usability. Currently users can already scale the desired LoD, but this is only an indirect tool for the aim of steadily high frame-rates.

List of Tables

List of Figures

Index

Bibliography

K. Appleton and A. Lovett. Gis-based visualisation of rural landscapes: defining "'sufficient"' realism for environmental decision-making. *Landscape and Urban Planning*, 65:117–131, 2003. 3

K. Appleton, A. Lovett, G. Sünnenberg, and T. Dockerty. Rural landscape visualisation from gis databases: a comparison of approaches, options and problems. *Computers, Environment and Urban Systems*, 26:141–162, 2002. 3

Arul Asirvatham and Hugues Hoppe. *GPU Gems 2*, chapter Terrain Rendering Using GPU-Based Geometry Clipmaps, pages 27–46. Addison-Wesley, 2005. 36, 37, 44, 46, 49, 52

Xiaohong Bao, Renato Pajarola, and Michael Shafae. SMART: An efficient technique for massive terrain visualization from out-of-core. In *Proceedings Vision, Modeling and Visualization (VMV)*, pages 413–420, 2004. 36

Louis Bavoil and Miguel Sainz. Image-space horizon-based ambient occlusion. ACM SIGGRAPH 2008 talks, 2008. URL `http://developer.nvidia.com/object/siggraph-2008-HBAO.html`. 76, 80

Stephan Behrendt, Carsten Colditz, Oliver Franzke, Johannes Kopf, and Oliver Deussen. Realistic real-time rendering of landscapes using billboard clouds. *Comp. Graph. Forum*, 24(3):507–516, 2005. 17

Daniel R. Berger. Spectral texturing for real-time applications. Siggraph 2003, Sketches and Applications, July 2003. 41

Stefan Bischoff and Leif Kobbelt. Ellipsoid decomposition of 3d-models. In *3D Data Processing Visualization and Transmission*, pages 480– 488, 2002. 24

Eric Bodden, Malte Clasen, and Joachim Kneis. Arithmetic coding revealed - a guided tour from theory to praxis. Technical Report SABLE-TR-2007-5, Sable Research Group, School of Computer Science, McGill University, 2007. 112, 114

Frederic Boudon, Alexandre Meyer, and Christophe Godin. Survey on Computer Representations of Trees for Realistic and Efficient Rendering. Technical Report RR-LIRIS-2006-003, LIRIS Lab Lyon, 2006. 3, 17

I. Brown, S. Jude, S. Koukoulas, R. Nocholls, M. Dickson, and M. Walkden. Dynamic simulation and visualisation of coastal erosion. *Computers, Environment and Urban Systems*, 30:840–860, 2006. 89, 90

Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet-sized batched dynamic adaptive meshes (p-bdam). In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 20, Washington, DC, USA, 2003a. IEEE Computer Society. ISBN 0-7695-2030-8. doi: http://dx.doi.org/10.1109/VISUAL.2003.1250366. 36, 52

Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Bdam – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum*, 22 (3), 2003b. 52

Malte Clasen and Hans-Christian Hege. Realistic illumination of vegetation in realtime environments. In *Trends in Real-time Visualization and Participation, New Technologies for Landscape Architecture and Environmental Planning*. Wichmann Verlag, 2005. vii, 71, 112, 114

Malte Clasen and Hans-Christian Hege. Terrain rendering using spherical clipmaps. In *EuroVis Proceedings*, 2006. vii, 36, 51, 112, 114

Malte Clasen and Hans-Christian Hege. Clipmap-based terrain data synthesis. In Thomas Schulze, Bernhard Preim, and Heidrun Schumann, editors, *Proc. SimVis 2007*, pages 385–398. SCS Publishing House e.V., 2007. vii, 36, 112, 114

Malte Clasen and Philip Paar. Globalisierung der landschaftsvisualisierung. In *Proceedings of AGIT 2008 Symposium und Fachmesse für Angewandte Geoinformatik*. Zentrum für Geoinformatik der Universität Salzburg, 2008. vii, 85, 112, 114

Malte Clasen and Steffen Prohaska. Image-error-based level of detail for landscape visualization. In *Proceedings of VMV - Vision, Modeling & Visualization*, 2010. vii, 16, 17, 20, 27, 28, 29, 31, 33, 71, 112, 114

Liviu Coconu. *Enhanced Visualization of Landscapes and Environmental Data with Three-Dimensional Sketches*. PhD thesis, University of Konstanz, July 2008. 28

Robert L. Cook, John Halstead, Maxwell Planck, and David Ryu. Stochastic simplification of aggregate detail. *ACM Trans. Graph.*, 26(3):79, 2007. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1276377.1276476. 33

Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Trans. Graph.*, 22(3):657–662, 2003. 20, 24, 71

X. Décoret, F. Durand, F. X. Sillion, and J. Dorsey. Billboard clouds for extreme model simplification. In *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 2003. 17

A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Metholodigical)*, 39(1):1–38, 1977. 17, 20

Oliver Deussen and Bernd Lintermann. A modelling method and user interface for creating plants. In *In Proceedings of Graphics Interface 97*, pages 189–197. Morgan Kaufmann Publishers, 1997. 18

Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis. Interactive visualization of complex plant ecosystems. In *IEEE Visualization*, 2002. 17, 33

Rouslan Dimitrov. Cascaded shadow maps. NVIDIA white paper, 2007. URL `http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf`. 79

Jürgen Döllner, Konstantin Baumman, and Klaus Hinrichs. Texturing techniques for terrain visualization. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 227–234, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press. ISBN 1-58113-309-X. 40

Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. Roaming terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press. ISBN 1-58113-011-2. 52

Anton Ephanov. Understanding virtual texture. MultiGen-Paradigm Support, March 2006. 36

S. M. Ervin. Digital landscape modeling and visualization: a research agenda. *Landscape and Urban Planning*, 54:49–62, 2001. 3, 10

Bernd Freisleben and Thilo Kielmann. Coordination languages and models, second international conference coordination 97 berlin, germany, september

1–3, 1997 proceedings. *Lecture Notes in Computer Science*, 1282:414 – 417, 1997. 44

Anton L. Fuhrmann, Eike Umlauf, and Stephan Mantler. Extreme model simplification for forest rendering. In *EG Workshop on Natural Phenomena*, 2005. 17

Guillaume Gilet, Alexandre Meyer, and Fabrice Neyret. Point-based rendering of trees. In *EG Workshop on Natural Phenomena*, 2005. 17

Enrico Gobbetti, Fabio Marton, Paolo Cignoni, Marco Di Benedetto, and Fabio Ganovelli. C-bdam - compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum*, 25(3), sep 2006. To appear in Eurographics 2006 conference proceedings. 36

C. V. Haaren and B. Warren-Kretzschmar. The interactive landscape plan - use and benefits of new technologies in landscape planning, including initial results of the interactive landscape plan koenigslutter am elm, germany. *Landscape Research*, 31(1):83–105, 2006. 1

Shawn Hargreaves. Deferred shading. GameDevelopers Conference (GDC) talks, 2004. URL http://www.talula.demon.co.uk/DeferredShading.pdf. 74, 79

David Hill. An efficient, hardware-accelerated, level-of-detail rendering technique for large terrains. Master's thesis, Graduate Department of Computer Science, University of Toronto, 20002. 52

James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM. ISBN 0-89791-196-2. doi: http://doi.acm.org/10.1145/15922.15902. URL http://doi.acm.org/10.1145/15922.15902. 79

Oliver Kersting and Jürgen Döllner. Interactive 3d visualization of vector data in gis. In *Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Systems (ACMGIS 2002)*, pages 107–112, Washington D.C., November 2002. 40

J . Kim and J.W. Woods. Spatio-temporal adaptive 3-d kalman filter for video. *IEEE Transactions on Image Processing*, Volume 6 Issue 3:414–424, 1997. 17

Hayden Landis. Production-ready global illumination. Siggraph 2002 Course 16: RenderMan in Production, 2002. URL http://www.siggraph.org/s2002/conference/courses/crs16.html. 75, 79

E. Lange. The limits of realism: perceptions of virtual landscapes. *Landscape and Urban Planning*, 54:163–182, 2001. 15

Peter Lindstrom. *Model simplification using image and geometry-based metrics*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2000. Adviser-Turk,, Greg. 25

Peter Lindstrom and Valerio Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/TVCG.2002.1021577. 36

Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *Siggraph 2004*, volume 23 (3), pages 769–776, New York, NY, USA, 2004. ACM Press. doi: http://doi.acm.org/10.1145/1015706.1015799. 36, 39, 46, 52

Rafał Mantiuk, Scott Daly, Karol Myszkowski, and Hans-Peter Seidel. Predicting visible differences in high dynamic range images - model and its calibration. In *IS&T/SPIE's 17th Annual Symp. Electronic Imaging*, volume 5666, pages 204–214, 2005. ISBN 0277-786X. 17, 27

Dorit Merhof, Markus Sonntag, Frank Enders, Christopher Nimsky, Peter Hastreiter, and Guenther Greiner. Hybrid visualization for white matter tracts using triangle strips and point sprites. *IEEE TVCG*, 12(5):1181–1188, 2006. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/TVCG.2006.151. 74

Jurriaan D. Mulder, Frans C. A. Groen, and Jarke J. van Wijk. Pixel masks for screen-door transparency. In *Proc. of VIS '98*, pages 351–358. IEEE CS Press, 1998. ISBN 1-58113-106-2. 73

Sean O'Neil. Rendering planetary bodies. *Gamasutra*, August 10, 2001, 2001. 52

Sean O'Neil. *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Accurate Atmospheric Scattering. Addison-Wesley, 2005. 82

Philip Paar. Landscape visualizations: Applications and requirements of 3d visualization software for environmental planning. *Computers, Environment and Urban Systems*, 30:815–839, 2006. 3, 15

Philip Paar and Malte Clasen. Earth, landscape, biotope, plant. interactive visualisation with biosphere3d. *Proceedings of CORP - 12th International Conference on Urban Planning and Spatial Development in the Information Society, May 20th - 23rd*, pages 207 – 214, 2007. vii, 85, 112, 114

Philip Paar, Katy Appleton, Malte Clasen, Maria Gensel, Simon Jude, and Andrew Lovett. Interactive visual simulation of coastal landscape change. In *Proceedings of the Digital Earth Summit on Geoinformatics 2008, International Society for Digital Earth*, 2008. vii, 2, 85, 89, 90, 112, 114

H. Repton. Observations on the theory and practice of landscape gardening. Taylor, London; Phaidon, Oxfort (facs.), 1803. 1

Wieland Röhricht and Malte Clasen. Multum, non multi. hierarchische bittrees bei der pflanzenverteilung mit oik. In *Simulation in Umwelt- und Geowissenschaften, Workshop Dresden 2005*, 2005. 112, 114

Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, pages 197–206, New York, NY, USA, 1990. ACM. ISBN 0-89791-344-2. doi: http://doi.acm.org/10.1145/97879.97901. URL http://doi.acm.org/10.1145/97879.97901. 74, 79

Christophe Schlick. A customizable reflectance model for everyday rendering. In *In Fourth Eurographics Workshop on Rendering*, pages 73–83, 1993. 79

S. R. J. Sheppard. Landscape visualisation and climate change: the potential for influencing perceptions and behaviour. *Environmental Science & Policy*, 8: 637–654, 2005. 90

Christian Sigg, Tim Weyrich, Mario Botsch, and Markus Gross. Gpu-based ray casting of quadratic surfaces. In *Proceedings of Eurographics Symposium on Point-Based Graphics*, 2006. 73

Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1998. ACM Press. ISBN 0-89791-999-8. doi: http://doi.acm.org/10.1145/280814.280855. 35, 36, 53

Christoph Ueffing. Wavelet based ecw image compression. *Photogrammetric Week 01, Wichmann Verlag, Heidelberg*, pages 299–306, 2001. 39

Roland Wahl, Manuel Massing, Patrick Degener, Michael Guthe, and Reinhard Klein. Scalable compression and rendering of textured terrain data. In *Journal of WSCG*, volume 12, 2004. 36, 37

Lujin Wang and Klaus Mueller. Generating sub-resolution detail in images and volumes using constrained texture synthesis. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 75–82, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8788-0. 41

Z. Wang, E. P. Simoncelli, and A. C. Bovik. Multi-scale structural similarity for image quality assessment. In *IEEE Asilomar Conference on Signals, Systems and Computers*, 2003. 27

A. Werner, O. Deussen, J. D"ollner, H.-C. Hege, P. Paar, and J. Rekittke. Lenn'e3d - walking through landscape plans. In *Trends in Real-time Visualization and Participation, Proc. at AnhaltUniversity of Applied Sciences*, pages 48–59, 2005. 3, 88

Fan Zhang, Hanqiu Sun, Leilei Xu, and Lee Kit Lun. Parallel-split shadow maps for large-scale virtual environments. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, VRCIA '06, pages 311–318, New York, NY, USA, 2006. ACM. ISBN 1-59593-324-7. doi: http://doi.acm.org/10.1145/1128923.1128975. URL http://doi.acm.org/10.1145/1128923.1128975. 79

Lebenslauf

Persönliche Daten

Malte Clasen
Schillerstr. 83
10627 Berlin
Deutschland

Tel.: +49 (0)30 22 43 88 71
Email: info@malteclasen.de
WWW: http://www.malteclasen.de/

geb. am 05.06.1980 in Köln

Bildung

08/1990–05/1999	Albert-Schweitzer-Gymnasium, Hürth
10/2000–09/2003	Informatik an der RWTH Aachen
10/2003–04/2005	Informatik an der TU Berlin
seit 10/2005	Informatik (Promotion) an der TU Berlin

Berufserfahrung

11/1999–07/2001	Softwareentwickler bei Globalpark GmbH (Hürth)
07/2004–05/2005	Studentischer Mitarbeiter am Zuse-Institut Berlin
06/2005–05/2011	Wissenschaftlicher Mitarbeiter am Zuse-Institut Berlin
seit 11/2006	Gesellschafter der Lenné3D GmbH
seit 05/2007	Gründer und Softwareentwickler von Rezeptefuchs.de
seit 06/2011	Softwareentwickler, adesso AG (Dortmund)

Publikationen

Forschung	[Röhricht and Clasen, 2005], [Clasen and Hege, 2005], [Clasen and Hege, 2006], [Clasen and Hege, 2007], [Clasen and Prohaska, 2010]
Anwendung	[Bodden et al., 2007], [Paar and Clasen, 2007], [Paar et al., 2008], [Clasen and Paar, 2008]

July 2, 2011

Curriculum Vitae

Personal Data

Malte Clasen
Schillerstr. 83
10627 Berlin
Germany

Phone: +49 (0)30 22 43 88 71
Email: info@malteclasen.de
WWW: http://www.malteclasen.de/

born on 05.06.1980 in Cologne, Germany

Education

08/1990–05/1999	Albert-Schweitzer-Gymnasium, Hürth
10/2000–09/2003	Computer Science, RWTH Aachen
10/2003–04/2005	Computer Science, TU Berlin
since 10/2005	Computer Science (PhD), TU Berlin

Professional Experience

11/1999–07/2001	software engineer, Globalpark GmbH (Hürth)
07/2004–05/2005	student assistant, Zuse-Institut Berlin
06/2005–05/2011	research assistant, Zuse-Institut Berlin
since 11/2006	partner, Lenné3D GmbH
since 05/2007	founder and software engineer, Rezeptefuchs.de
since 06/2011	software engineer, adesso AG (Dortmund)

Publications

Research	[Röhricht and Clasen, 2005], [Clasen and Hege, 2005], [Clasen and Hege, 2006], [Clasen and Hege, 2007], [Clasen and Prohaska, 2010]
Application	[Bodden et al., 2007], [Paar and Clasen, 2007], [Paar et al., 2008], [Clasen and Paar, 2008]

July 2, 2011