Interactive Visualization of a Planetary System

Student Research Project

by

Jochen Winzen

Faculty of Computer Sciene

Institute for Operating- and Dialogue-Systems

University of Karlsruhe, Germany

Karlsruhe, April 2003

Tutors:

Prof. Dr. Alfred Schmitt

Dipl.-Inform. Stefan Preuß

Contents

Chapter 1

Introduction

1.1 Abstract

This paper describes the development of a new 3D engine called "Infinite Universe Engine". This engine renders the geometry of a whole solar system in real-time and demonstrates the capabilities of the new IU-LOD-algorithm (further explanations see 2.1.6). The IU-LOD-algorithm is able to render landscapes of arbitrary size. It uses a top-down approach (2.1.13) and combines the quadtree technique with a scenegraph system.

1.2 Motivation

Previous programs (until march 2003) that are concerned with rendering planetary bodies in **real-time** lack the visual quality of current algorithms in terrain visualization. All astronomical visualization programs (very good program: Celestia [5]) simply render textured spheres as planets, whereas the terrain size of other LOD-algorithms (good overview: VTP [6]) does not scale above some square kilometers. The new IU-LOD-algorithm closes this gap in this field of computer graphics.

The geometry is based on heightmaps as most other algortihms in terrain visualization also use them. The whole geometrical data of a planet is created by functions that combine real data (Digital Elevation Models (2.1.4)) with fractally or procedurally computed values. The elevation data for the whole earth's surface would consume 40 TeraByte of storage space (using resolution of 10m). This amount of

data is (far) too much to be rendered in real-time. So most data is created on the fly with fractal noise functions. The designer of a planet can define special Points of Interest (2.1.9) based on stored data.

Every terrain on earth is rendered by this program with many additional details like plants and buildings to create a really immersive environment. The projection system simulates a first person view. This is sufficient to render close looks at small details (house interior, grass) and to view a whole planet from orbit, too. The view distance is almost infinite (10^{300}m) and rendered objects may be as large as planets. So the IU-LOD-algorithm is really general purpose and can be used in every visualization. Some examples:

- GIS (Geographical Information Systems), 3D-Atlas
- Architecture, city planning
- virtual tourism
- simulations
- 3D games with infinitly large environments

1.3 Overview

Some essential terms in computer graphics and terrain visualization are explained in chapter 2. In chapter 3 a short overview about popular algorithms in terrain visualization is given. The IU-LOD-algorithm is presented in chapter 4. At first it explains why a new algorithm had to be developed to render whole planets. In chapter 4.2 the algorithm is shown in more detail and it will be explained how it actually works. The algorithm will be compared with existing ones and numerical and graphical problems that arise in astronomically large scenes are mentioned. In chapter 5 the results of various measurements of performance are shown. The summary at the end of this paper mentions some possible improvements for the graphics. The appendix contains information about how to control and how to extend the program. Therefore some classes are listed with documented source code.

Chapter 2

Fundamentals

2.1 Technical terms

This chapter explains many terms in computer graphics and terrain visualization. So readers that are not familiar with these topics will have a starting point to understand later chapters. The terms are listed in alphabetical order.

2.1.1 Bintree

A bintree is used as data structure in computer graphics. It is a binary tree so every node has two children. Terrain visualization also uses bintrees: Every node represents a triangle of the terrain. The first layer (two nodes) has two triangles that cover the whole terrain. Each triangle is splitted into two new ones. This subdivision scheme creates different levels of detail for the terrain which causes fewer memory consumption and increasing performance. Since some triangles share edges all neighbour nodes have to be considered when splits are done.

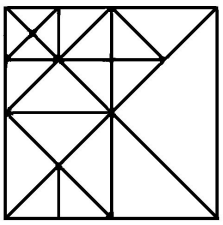

Figure 2.1: Bintree

2.1.2 Bottom-Up-Algorithm

The term Bottom-Up-Algorithm originates of the field of algorithm design. It's a specific way to solve problems. The algorithm starts computation on the smallest units of the problem. Then it aggregates the results and computes again. This forms an hierarchy and leads to a final result for the whole problem. Many tasks can be solved in a bottom-up manner. When using these algorithms in 3D graphics all objects/values have to be iterated. This causes long computation cycles. Top-Down-Algorithms (2.1.13) are often a better choice.

2.1.3 CLOD (Continuous-Level-Of-Detail)

CLOD-Algorithms are special LOD-Algorithms (see 2.1.5). They do not display geometry with some precomputed discrete levels of detail. Thw geometry of an object (e.g. landscape) is recomputed continuously (every frame) with smooth transitions between the different used LODs.

2.1.4 Digital Elevation Model (DEM)

There are a lof of different standards, how to store elevation data. The used formats are Digital Elevation Models that use two-dimensional arrays of height values. These DEMs also contain a latitude/longitude position and data resolution (10m, 1km, ...). Satellite elevation data are most often stored as USGS-DEM files (.dem). In games greyscale images (heightmaps) are a common solution to specify terrain data.

2.1.5 LOD (Level-Of-Detail)

For rendering very large scenes with many objects, performance is critical. LOD (Level-Of-Detail) techniques have proven their advantages in many real-time visualizations. Objects are displayed with different levels of detail, which are determined by the distance from the viewer. This LOD-technique dramatically reduces the number of polygons, which have to be drawn. Performance increases by huge amounts (factor 10, 100 or even more). In chapter 3 some CLOD-algorithms (see 2.1.3) in terrain visualization are presented. Figure 2.2 demonstrates the LOD technique.

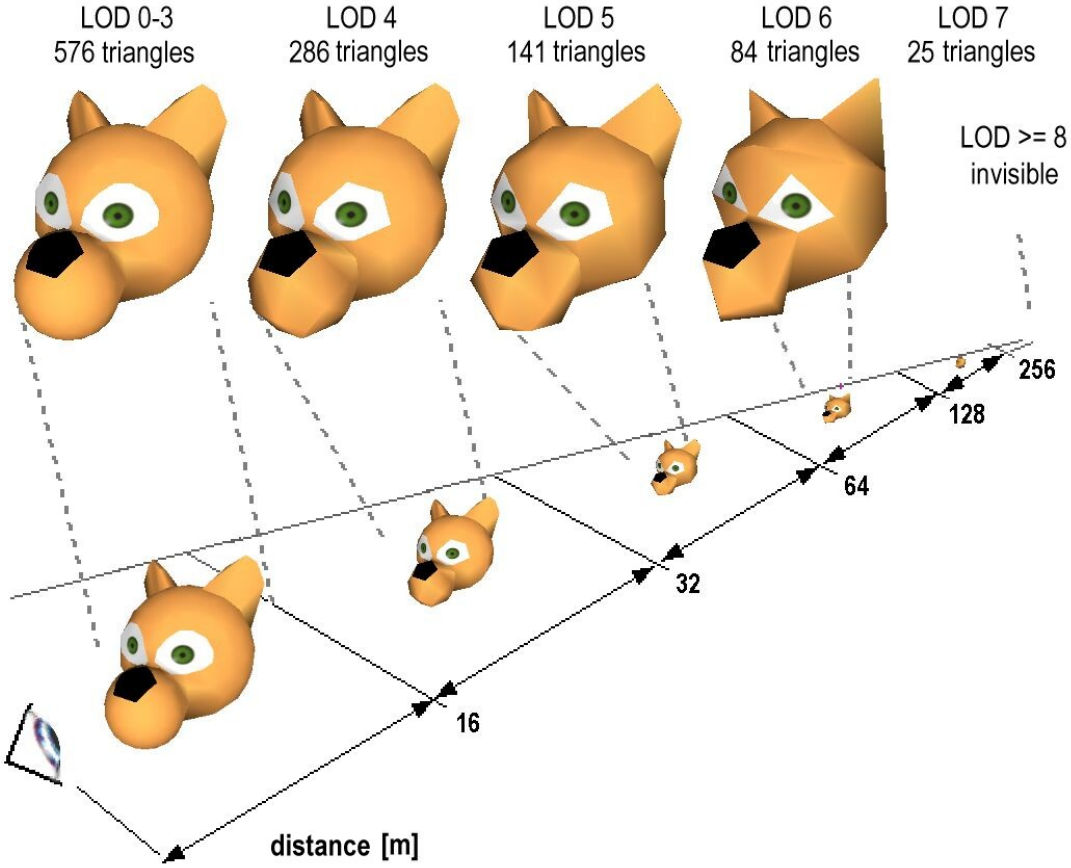

Figure 2.2: LOD for a comic-style dog

2.1.6 IU-LOD-Algorithm

The IU-LOD-algorithm was exclusivly developed for the 3D visualization program "Infinite Universe Engine". IU-LOD is the abbreviation for Infinite-Universe-Level-Of-Detail. The name shows clearly that this algorithm is able to render even astronomically large scenes (whole solar systems).

2.1.7 Patch

In computer graphics a patch is defined as a free form surface that is represented by control points. In this paper a patch is a quadratic surface area of the terrain. Each patch belongs to a node in the quadtree (2.1.10). In the program "Infinite Universe Engine"every patch is stored and rendered with an array of 9×9 vertices (128 triangles). This size was determined heuristically. Larger patches (more vertices) would improve the visual quality but decrease performance because of the increased

triangle count. Smaller patches would not make effectivly use of the fast OpenGL-TriangleFans.

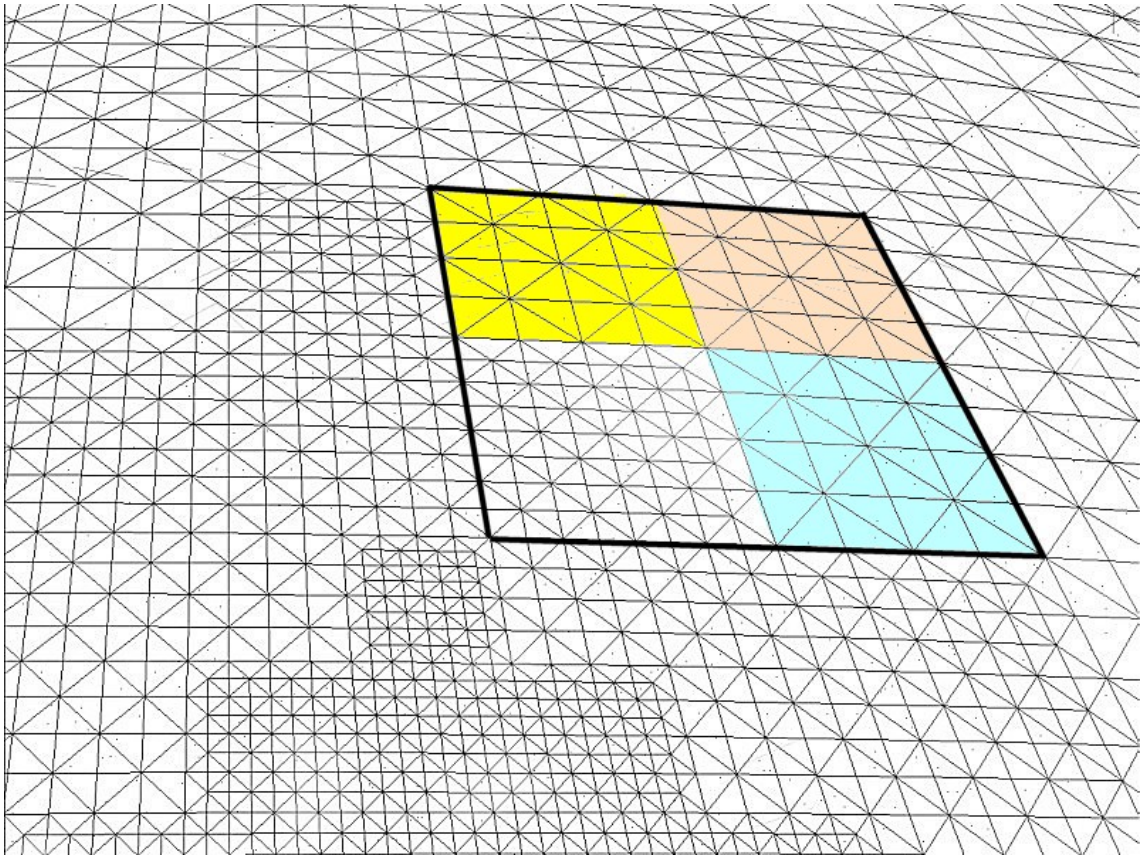

Figure 2.3: Area with black border is a patch of the quadtree. The white part (bottom left) already shows a patch with the next level of detail.

2.1.8 Pixelerror

When using LOD-algorithms (2.1.5) some disadvantages are unavoidable. One disadvantage are visual errors that are caused by the simplification of geometry. The biggest difference between the original image and the projection of the simplified model can be measured in screen pixels. This difference is called pixelerror (projected screen space error). Figure 2.4 shows the pixelerror only for the outline of the model but it can be computed for every projected point of a model (even inner points).

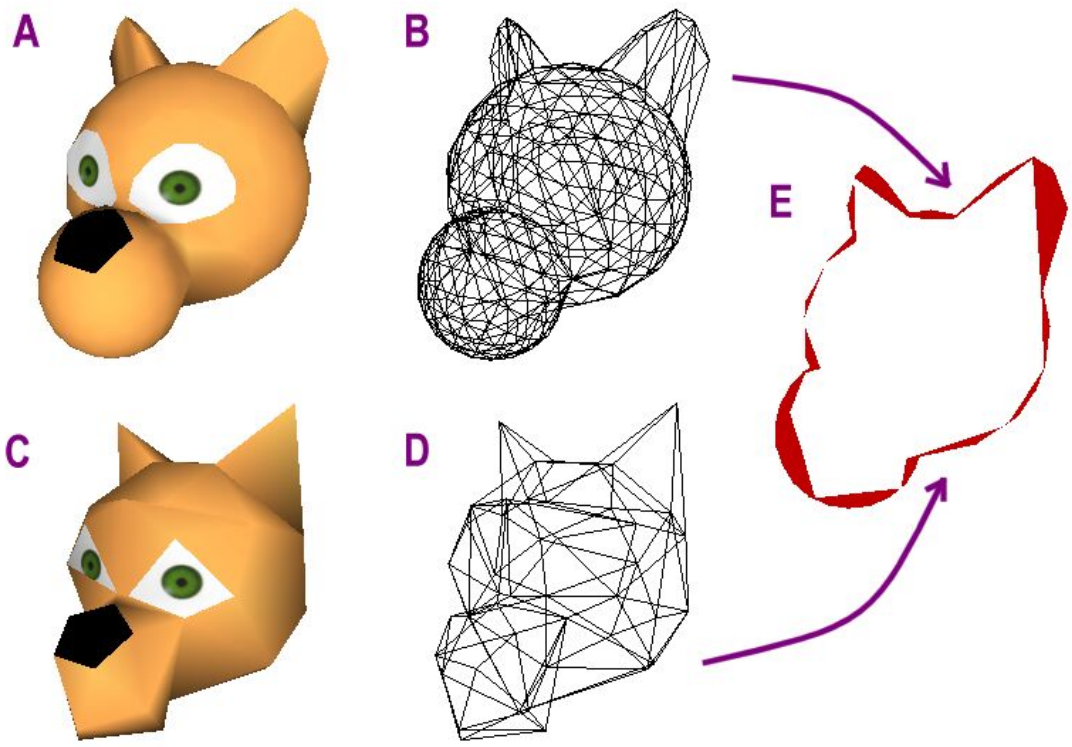

Figure 2.4: Image A and B: Original image (576 triangles). Image C and D: simplified model (84 triangles). Image E shows the difference image that means the pixelerror

2.1.9 Point of Interest

A Point of Interest is a certain area in the landscape where additional information exists. This information can contain exact height data (from DEMs (2.1.4)), vegetation models or even 3D-models of whole cities.

2.1.10 Quadtree

A quadtree is often used as data structure in computer graphics. It is a tree in which every node has exactly four children. Quadtrees are also used in terrain visualization: The root node represents the whole terrain. Then the terrain is divided into four equal sized parts (quadrants) that belong to the child nodes. This subdivision is repeated recursivly on the child nodes (Figure 2.5).

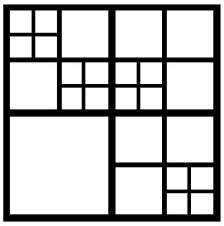

Figure 2.5: Quadtree

2.1.11 Scenegraph

A scenegraph manages all objects in a 3D environment. The functionality of scenegraphs can be quite different. The minimal requirement is that it enables the handling of object hierarchies. That makes it possible to move objects and all belonging parts stay at correct relative positions. Example: If a car drives on a road all tires move with it.

Additional functions are view frustum culling, collision detection, physical simulation of the objects, etc.

2.1.12 Depth Buffer

The depth buffer (z-buffer) is used very often to render 3D graphics. This buffer is as large as the application's window and stores the depth in the scene for every pixel. So it is quite easy to compare the position of every polygone in the scene with each other. The depth value decides which parts of a polygone are hidden or visible. Modern graphics boards store a 16 or 24 bit value for each pixel that means 2^{16} or 2^{24} depth values can be distinguished. This is sufficient for most applications.

2.1.13 Top-Down-Algorithm

The term Top-Down-Algorithm originates of the field of algorithm design. Compared to bottom-up-algorithms (2.1.2) it's the contrary approach to solve a problem. At first the task is viewed as a whole and becomes divided into smaller tasks. This is repeated until the subproblems can be solved easily. With 3D graphics this approach has some advantages: Objects that have to be rendered can be divided into smaller parts. Some of them are maybe outside the view frustum or so small that it is not essential to draw them. These parts can be discarded. So performance improves and memory consumption is reduced.

Chapter 3

Terrain Visualization

3.1 Introduction

Every work in the field of terrain visualization tries to simulate a natural environment. Different applications (architecture, geology, ...) concentrate on different aspects of the simulation. But the common aspect is to try to render the scene as realistic as possible. There exist very good commercial programs (Bryce [20], Vue d'Esprit [21]) and also free ones (TerraGen [22], listing of Virtual Terrain Project (VTP) [23]) that render beautiful landscapes almost as good as a real photo. These images are computed with raytracing which is too expensive to work in real-time (but in some years...). If an application has to be interactive, speed is cruical and rendering must be performed in real-time. In this case CLOD (Continuous Level of Detail)-algorithms (2.1.3) are a good choice. They use 3D-APIs (OpenGL, DirectX) to render the scene. The popular algorithm ROAM [9] (see section 3.2.3) seems to be used most often in programs and games that render large landscapes. Section 3.3 has focus on the height values of terrain. Since a large terrain, especially the whole surface of a planet is far too big to store all height values, details have to be created with fractal or procedural functions.

3.2 Popular LOD-algorithms

Here some popular algorithms in terrain visualization are presented, and their abilities to render large landscapes or even whole planets are examined. A listing of most papers concerned with terrain visualization can be found at the Virtual Terrain Project (www.vterrain.org [6]).

3.2.1 Brute Force, TIN

These two algorithms are only mentioned here, to show the whole spectrum of solutions to visualize terrains. It's possible to render small terrains with these algorithms but never whole planets, because both solutions need all height values as precomputed data. So they are unsuitable for large terrains.

Brute Force is the most simple algorithm to render heightfields. The whole terrain data is rendered with a constant triangluation that is computed once before the visualization. This algorithm is appropriate only for arrays up to 1000×1000 points. Even in this case two million triangles have to be drawn.

TINs (Triangulated Irregular Networks) compute an optimal triangulation for the complete data set. This triangulation is irregular and only takes the pixelerror (2.1.8) into consideration as subdivision criteria. With this algorithm fewer triangles have to be rendered than with regular subdivision schemes (grid structures). Areas with little height variance become approximated through few large triangles (see figure 3.1). But this computation is very expensive.

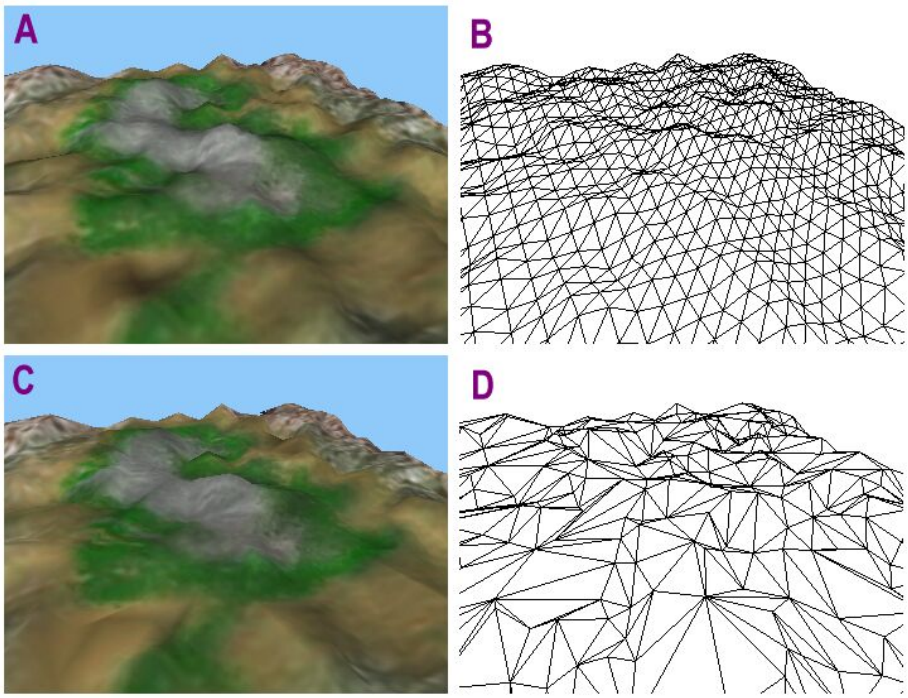

Figure 3.1: Images A and B: terrain rendered brute force (2048 triangles). Images C and D: the same terrain with TIN (503 triangles).

3.2.2 Lindstrom

1996 Lindstrom et al [8] developed a new algorithm for the visualization of height-fields. It uses a quadtree (2.1.10) where the viewer position and distance determine the subdivision. This subdivision scheme is used in many other algorithms (IU-LOD also works this way). The main criteria to split geometry or not is the projected pixelerror (2.1.8). This pixelerror must be computed for every vertex in a bottom-up (2.1.2) manner which is expensive and a clear disadvantage of this approach. Because of the bottom-up computation this algorithm does not scale very well. The whole height data of the terrain must be given in advance and can not be created on the fly. So it needs too much storage space to visualize a whole planet. But this algorithm is still better than the previous ones, as the dynamic subdivision scheme with quadtrees (CLOD-technique (2.1.3)) heavily reduces the number of triangles to draw.

3.2.3 ROAM

The ROAM-algorithm (Real-time Optimally Adapting Meshes) [9] was developed one year after Lindstrom [8] by Mark Duchaineau et al. They used some results of the previous work and avoided the pitfalls. ROAM uses a bintree (2.1.1) with the same subdivision scheme as before. This means the triangles are split if the projected pixelerror is above a predefined border value (e.g. 4 pixels). But all computations are performed top-down (2.1.13), so this algorithm is significantly faster than others. The computations are done top-down to a certain resolution (LOD) and all finer resolutions can be discarded. A bottom-up approach would lead to computations for all resolutions even if they are not necessary for the final result. Every discarded level of the bintree increases performance by a factor of two.

At the subdivision step a priority queue is created, that stores which parts of the geometry have to be splitted or remerged. With this queue it is possible to control how many triangles are created by subdivision. First triangles are merged to reduce the number of triangles. Then the list is sorted by the computed pixelerror for each triangle and the triangles with great values are splitted first. These advantages explain why ROAM is used so often.

First tests to visualize a whole planet with ROAM had the following problem: The geometry is subdivided down to a triangle size of one meter. This means a recursion depth of 20 levels in the bintree. When using several continuous levels of

detail (CLOD, see 2.1.3) it is essential to change the triangluation scheme for transitions between the different LODs to avoid cracks in the visual appearance. ROAM has to check all triangles in the neighbourhood (they share a edge with the current triangle) to find these transitions. The other triangles may have to be splitted and this has to be done recursivly for all children. It is this recursion that causes so many computations that real-time rendering becomes impossible for high numbers of LODs.

3.2.4 Roettger

Stefan Roettger [7] also uses quadtrees in his terrain algorithm. His algorithm divides geometry top-down and performance is comparable to ROAM. The visual quality is improved through geomorphing. All points of geometry are smoothly blended between different LODs so new subdivisions become almost invisible. The error measurement is based on the pixelerror but ensures that neighbour patches (nodes in the quadtree) have no more difference than one level of detail. This error measurement restricts the recursion depth of new subdivisions and is almost identical with the one used in the IU-LOD-algorithm. Since the algorithm is very similar to the one developed in this paper it should be possible to render whole planets in real-time.

3.3 Fractal Landscapes

If one wants to visualize large terrains or whole planets stored data is not enough. Almost every program adds detail information with procedural functions. If there is no real-world data at all for a planetary model everything must be computed with fractals. The book 'Texturing and modeling: a procedural approach' [4] gives a good overview how to use fractals in computer graphics. Here only the most common techniques are presented.

3.3.1 Perlin Noise

This is only a short introduction. Ebert et al. [4] provide a more detailed explanation on perlin noise. Perlin noise is maybe the fractal function that is used most often in computer graphics. It helps to simulate natural structures like clouds, wood, marmor,

fire and other phenomenas in an easy way. Two basic steps have to be done. The first one is a function that "computes"noise depending on the current coordiantes. The second step is to feed in those noise values into different fractal functions. This leads to fractal patterns with a construction process that can be repeated.

Perlin noise does not take values of a simple random noise generator because the pattern has to be the same for every new computation. It uses lattice noise which stores precomputed random values into a grid structure (lattice). Every three-dimensional position in space will be projected onto this lattice to compute a "random value". Fractal patterns are construced by summation of these random values. There is a fixed construction scheme. First a signal with low frequency (few sample points) is interpolated over a certain interval. Then the interval lenght becomes half the size as before (doubled frequency) and new values are added with smaller amplitude. This is repeated until a predefined resolution is achieved. Figure 3.2 shows this computation.

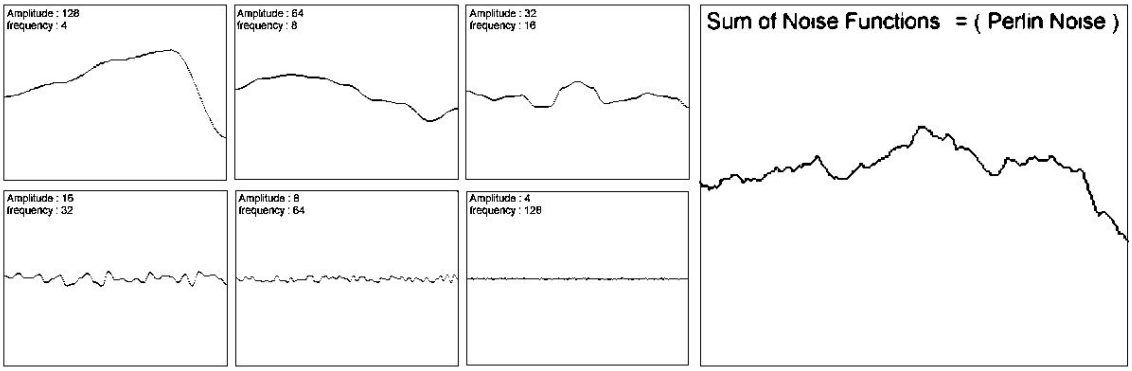

Figure 3.2: Perlin Noise

3.3.2 Fourier-Synthesis

Contrary to perlin noise this technique does not use random values. All parameters have to be defined to compute the result. These are the number of frequencies that should be added and their amplitudes. The final result is a summation of sinus curves with the given parameters. This algorithm works similar to perlin noise but it creates noticable subpatterns. These repeating patterns are caused by the periodicity of the sinus function. In most cases this visual effect is not wanted.

3.3.3 Fractal-Midpoint-Displacement

There exist two versions of the midpoint displacement algorithm, fractal plasma and the diamond square algoritm. Here only fractal plasma will be explained since it is used in the IU-LOD-algorithm (see section 4.2.2: Fractal Heightmaps). The two-dimensional midpoint displacement algorithm is a good choice to create mountanious terrains. It first sets all four corners of a rectangle to random positions with a height value in $[-h, h]$. Then this rectangle is divided into four smaller ones (see figure 3.3). All new points are interpolated from the previous ones. Most important is to add a new random height value in the range $[-h/2, h/2]$.

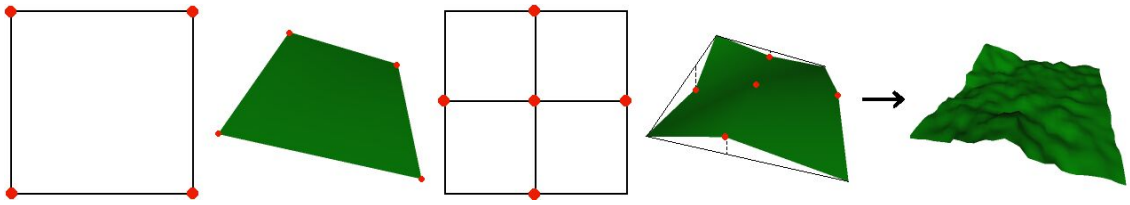

Figure 3.3: Midpoint-Displacement

It is quite common to multiply the height variance by the factor 0.5 for each step. This factor can lie between 0 and 1. If the value goes towards zero the landscape becomes very flat with large hills and valleys. A Factor above 0.5 creates very rough mountains.

Chapter 4

The IU-LOD-Algorithm

4.1 Development History

This section describes techniques that could be used to visualize planets. The main focus is on triangulation schemes of geometry and how to render them. Because of the size of terrain (planet's surface) only LOD-algorithms can be used for this purpose. The different approaches are presented in the same order as they were tested within this work. For every algorithm it is important to mention where this approach failed. At the end this work was implemented with a quadtree based algorithm. The exact algorithm will be explained in later sections. Now the experiences and problems in visualization of whole planets are presented.

Most solutions that visualize whole planets simply map a $m \times n$ grid of height values onto a sphere using polar coordinates. The biggest problem with this solution is a too dense triangulation at the poles as they are singularities (see figure 4.1 (image A)). A level of detail algorithm computes a better triangulation using the pixelerror but only ROAM (3.2.3) was used for this task in previous works (see figure 4.1 (image B)).

Why does the IU-LOD-algorithm not simply implement ROAM? The problem of ROAM is that the geomety is stored in a bintree (2.1.1). The sphere in figure 4.1 has planetary size (diameter is several thousand kilometers). So the bintree of the algorithm consists of more than 20 levels of detail which is also the recursion depth of functions that work with the bintree. The section about bintrees mentioned that all neighboured triangles have to be checked. This causes recursive splits and merges for more and more triangles and is a very expensive computation.

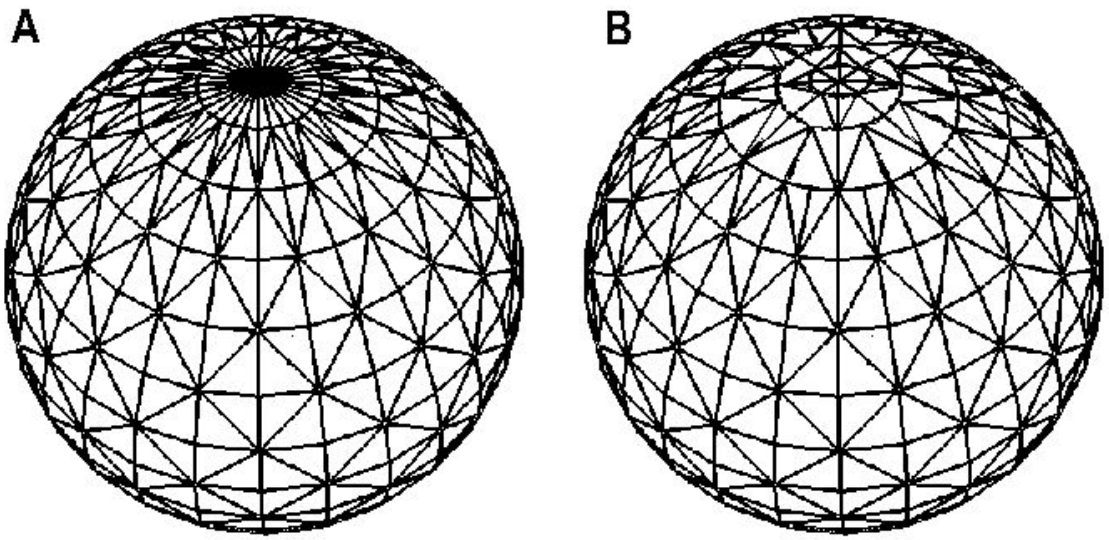

Figure 4.1: image A: singularities with polar coordiantes. image B: sphere with LOD

The height values are computed with fractal functions (see section 4.2.2) in most cases since there is not enough predefined data. It is not possible to compute all subdivisions and store the results to speed up things. The geometry of a whole planet is too much data and has to be created dynamically (on the fly). The triangulation of ROAM can become irregular because only the pixelerror (2.1.8) determines where to split and where not. This gives good visual results. No patterns of the triangulation can be detected. But it has the disadvantage that all computations have to be done per triangle. There are no synergies to use e.g. reuse of values from expensive functions.

The next aim was to create an algorithm that has a regular triangulation scheme. If it was possible, the transitions between different levels of detail should be computed with simple functions without recursive tests.

A first idea was to render grids of different resolutions that lie into each other. Figure 4.2 shows on image A how the triangulation of a surface should be in the optimal case. Image B approximates this triangulation with regular $m \times n$ grids. They are quadratic since the optimal case has perfect circles.

A regular $m \times n$ grid can be directly rendered as vertex array in OpenGL. Such vertex arrays store vertices, normals and texture coordinates for the geometry. Modern graphics boards have hardware acceleration for vertex arrays which makes them very fast. But this grid structure was not so good in practice as it first seems to be. If the camera moves through the world the grids must be changed accordingly.

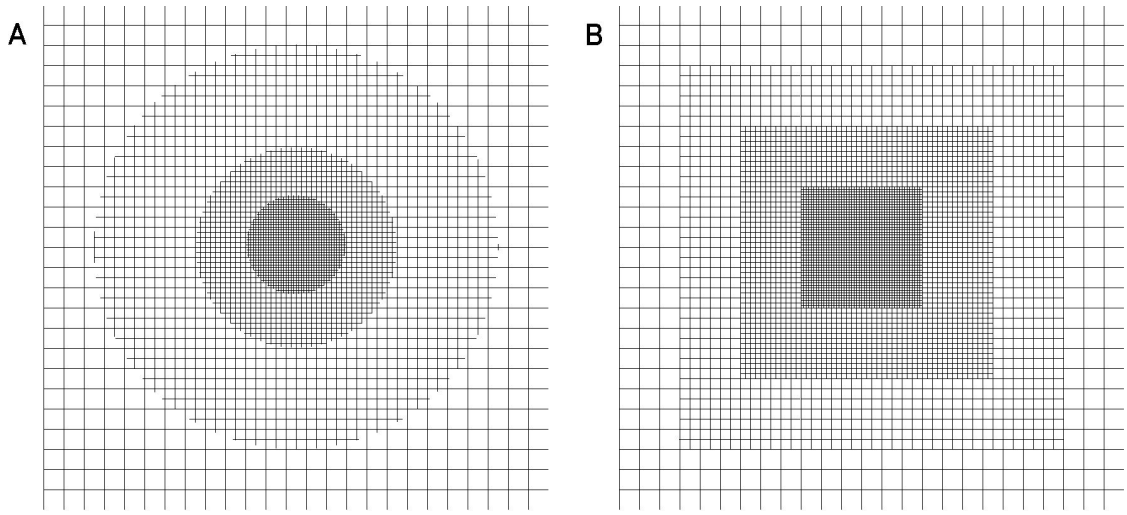

Figure 4.2: Image A: optimal LOD-structure. Image B: LOD with $m \times n$ grids

On one side there are new points on the ohter side some points must be deleted. There are two ways to do this: Copy the whole vertex array or to change the index list. It is not efficient to copy the whole array. In the other case one must use very complicated index functions to adress the geometry. It's not very sensible to store all $m \cdot n$ states of the index lists thus vertex arrays can not be used. Some other papers use this approach (GDC 2001 [13]) but that's not recommendable. It should be only used for small static heightfields.

A logical extension of the previous algorithm is to use tiling techniques on the grids. This reenables the use of vertex arrays. But this idea was discarded since this tiled structure is too similar to quadtrees which should be preferred. The main advantage of a quadtree is its clear and simple structure.

The tests had these results: A planet has very many levels of detail so it is important to avoid expensive computations. The developed algorithm must not use recursive neighbourhood tests (like ROAM) as this causes too many operations. Transistions between different levels of detail only consider the camera distance but no pixelerror. Polar coordinates are no good choice to visualize planetary bodies. Triangulation is very dense at the poles so that a compensation factor must be introduced. This factor is orientied at the angle to the vertical axis as the longitudal distance stays constant but circles by latitude become smaller and smaller. So it is a good decision to choose another projection system. The IU-LOD-algorithm uses a cube model that is projected onto a sphere. In the next section (4.2 How IU-LOD works) this will be explained in detail. The new system also has some problems but they can be solved more easily.

4.2 How IU-LOD works

Here follows an in-depth analysis how the IU-LOD-algorithm actually works. The explanations concentrate on parts of the engine that have essential functionality and already exist for a longer time (until march 2003). Sometimes there were similar solutions (e.g. LOD with bintree (2.1.10) or quadtree (2.1.10)?). In general solutions were preferred that are easier to unterstand and to implement.

In section 4.1 some LOD-algortihms were presented that are not sufficient to visualize whole planets in real-time. It was the right choice to use quadtrees. With them real-time visualization of planets is possible which is demonstrated by the "Infinite Universe Engine".

What's the difference? Most other LOD-algorithms get performance problems with the large number of LODs that are neeeded to render a planet if the minimal triangle size has to be one meter. This requirement comes from the goal to render a planet in full size (scale 1:1). A finer triangle resolution does not seem to be necessary but it is possible to do further subdivisions (down to atoms (10^{-10}m)). Most LOD-algorithms refine geometry by a factor of 2. They divide a triangle's edge into two smaller ones. This gives following approximation for the number of levels of detail.

$$\texttt{\#LOD} = log_2(2 \cdot \pi \cdot radius[m]/4) \tag{4.1}$$

With this equtation earth has $log_2(2 \cdot \pi \cdot 6378000/4) = 23$ levels of detail.

The basic model for this subdivision scheme is a cube. This causes the factor 1/4 in equtation 4.1. Each triangle of the cube has the length: circumference of planet / 4. The many levels of detail enable a smooth approximation of a planet sphere (convergence is $1/2^n$). Figure 4.3 demonstrates the principle.

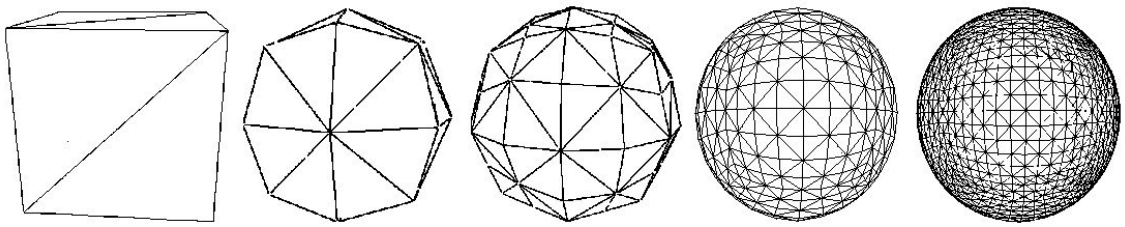

Figure 4.3: Quadtree-subdivision of a planet

All transitions between different levels of detail must use another triangulation (see figure 4.4). For details look at figure 4.9 on page 32. If one does not modify the

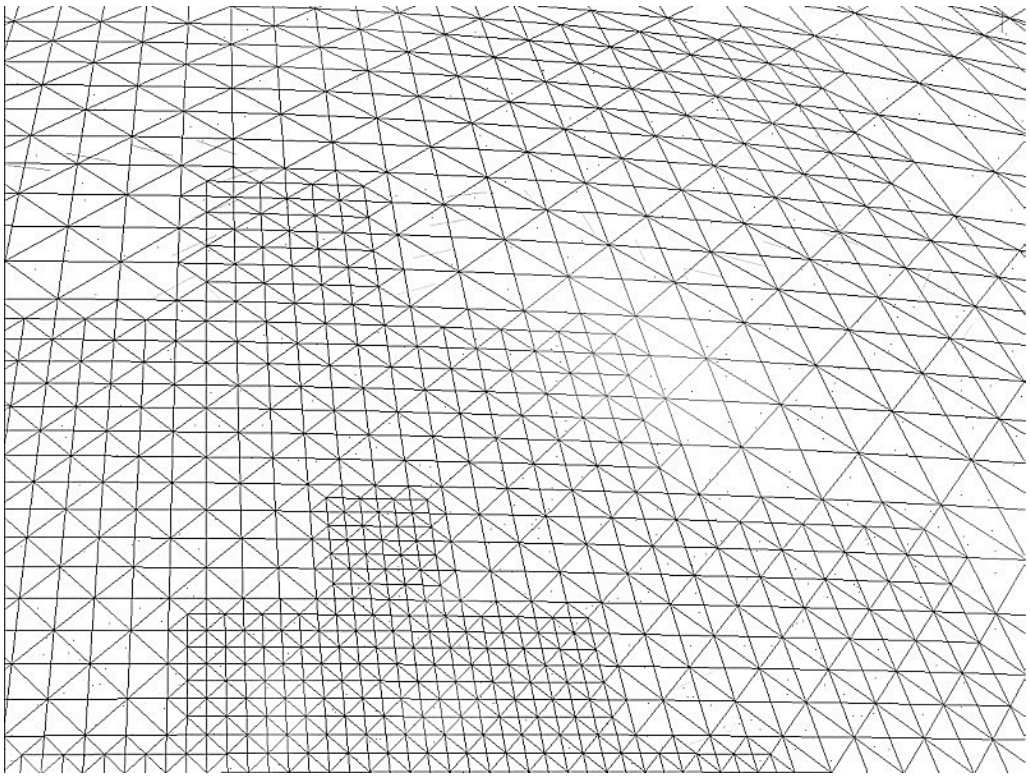

Figure 4.4: Triangulation of a planet

triangulation there can be cracks in the terrain which is not desirable. So every CLOD-algorithm (2.1.3) has to check for these transitions.

Many algorithms simply iterate through all neighbour triangles or patches (2.1.7) to find these transitions. These neighbourhood tests are expensive computations. With the use of quadtrees it is possible to construct an algorithm that does not need these tests but computes all transitions correctly. In section 4.2.4 it will be explained how this works.

Figure 4.3 clearly shows how the cube model is subdivided. This basic model does not use polar coordiantes at all to compute the points on the planet's surface. Height values are usually stored in this coordinate system but it's not necessary to manage geometrie in the same way. Other (and better) geometric models can still use the same height data. They only have to perform some simple (and computationally cheap) transformations. The disadvantages of polar coordinates were described in the previous section 4.1. The main drawback is the distortion of the geometry that almost diminishes with a cube as basic model. In order to visualize the planet the subdivided cube model is mapped onto the planet's sphere. This creates the polyeders in figure 4.3. Geometry still is distorted but by the maximum is $(1 : \sqrt{3})$ which does not require any compensation.

Lemma:
The maximal distortion in geometry is $(1 : \sqrt{3})$.
Proof:
Compare two points with distance d ($d << radius$) on a cube side. If they are near the center of the face they are mapped $1 : 1$ onto the sphere. If they are near a corner of the cube their coordinates have to be divided by $\sqrt{3}$ (lenght of main diagonal in unit cube). Then they lie on the surface of the sphere. The distance between the points now is the smaller distance $d/\sqrt{3}$. □

All urgent visualization problems are solved now:
The IU-LOD-algorithm is able render a planet down to a resolution of 1m. Therefore a cube with planetary size is created. Each side of the cube is a quadtree that is some thousand kilometers in size. Depending on the viewer's distance geometry of the planet is subdivided with new child nodes (child patches). This subdivision is repeated recursivly until geometry resolution is fine enough (small pixelerror (2.1.8)) or the finest resolution of terrain (1m) is reached. This LOD-update will be explained in section 4.2.4. It computes all transitions between levels of detail with a global function that does not need additional tests. This property and the use of quadtrees make it possible to visualize planets in real-time.

Quadtrees have the following advanatages:

- They are well suited for a cube model (cube mapping).
- Geometry subdivision has a simple and regular structure which enables fast computations.
- They have perfect cache behaviour and use spatial and temporal locality of data. Every node of the quadtree stores a patch (2.1.7) of the landscape. Computations for a vertex or triangle are performed on neighboured vertices/trianlges in the next steps. The quadtree nodes can use this spatial locality. Temporal locality is used because the whole quadtree saves a snapshot of the current environment. If a viewer does not move no new computations have to be done in the next frame.

There are some further elements in the IU-LOD-algorithm to solve different visualization problems. These concepts will be explained in the next sections.

A whole planet is a really big tree-dimensional scene. So there are some numerical problems. One problem is that OpenGL can't render scenes of every size. Only

a specific range can be displayed and objects that are too near or too far away are clipped. In the section number spaces (4.2.1) there are several techniques to solve these problems. Another numerical problem is to find stable noise functions. They are essential to create fractal landscapes but even small differences in the position computation of vertices might cause totally different results. The section about fractal heightmaps (4.2.2) tells how this can be avoided. The last problem comes with the challenge to simulate a dynamic planetary system where planets and moons move on their curves around the sun (using simplified physical formulas). For all planets the seasons and self rotation (day/night cycle) are simulated. These transformations can be managed with a scenegraph (2.1.11). It stores all objects in a hierarchy and implicitly uses the IU-LOD-technique. More details are mentioned in section 4.2.3.

Each frame the following steps are done to visualize a planet. The previous sections (4.2.1-4.2.3) provide some background information about the functionality.

- LOD-Update 4.2.4

  View-Update 4.2.5

  Geometry-Update 4.2.6

- Rendering 4.2.7

4.2.1 Number Spaces

Different parts of the IU-LOD-algorithm work on several number spaces that store properties of a planet. What number spaces exist will be explained in the next sections.

4.2.1.1 Double Space

At first there is a continuous three-dimensional space (Double Space = $[double]^3$) that stores the whole geometry of a planetary system. The base unit is 1 meter. Double values have at least 15 digits accurancy so every position in a solar system can be defined down to 1mm. Most objects are not at fixed positions so they are managed in the scenegraph (see section 4.2.3).

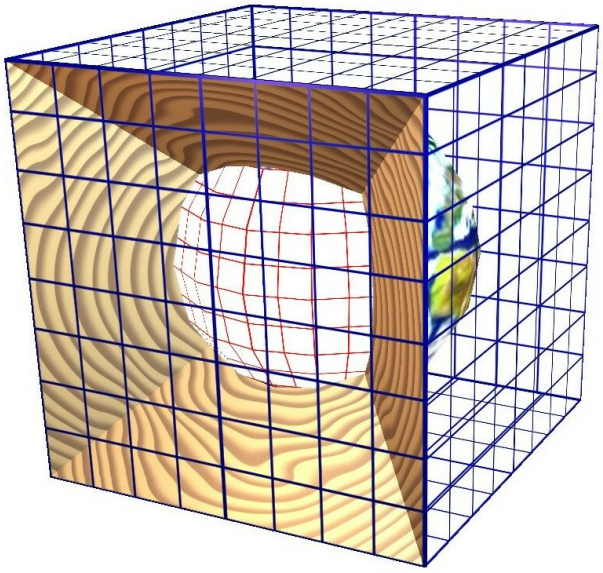

Figure 4.5: Geometry of a planet

4.2.1.2 Integer Space

To simplify management of the levels of detail for a planet there is a discrete three-dimensional space. All discrete lattice points are realized as integer vectors (x,y,z) and are limited for numerical reasons (int with 32 bit) on the range $[-2^{28}, +2^{28}]^3$. The basic model for subdivision in the IU-LOD-algorithm is a cube of planetary size. Equtation 4.1 shows that the earth model has 23 levels of detail. In integer space earth is represented by a cube in the range $[-2^{22}, +2^{22}]^3$. Since the algorithm only visualizes the surface of a planet (for now) only the faces of this cube are of interest (one component always is $\pm 2^{22}$). Each face of this cube is a regular grid whose edges consist of $2^n + 1$ points (for earth $2^{23} + 1$). This guarantees that the quadtree structure can be divided until the minimal size of 1 unit is reached (23 levels of detail). This structure is connected to the rendered geometry by projection of this grid onto the planetary sphere (see figure 4.5). The following equtation (4.2) describes this projection. $\vec{p}$ is a position in $\mathbb{R}^3$ (double space) and $\vec{g}$ in $\mathbb{Z}^3$ (integer space).

$$\vec{p} = \frac{\vec{g}}{\|\vec{g}\|} \cdot (radius + h(\vec{g})) \tag{4.2}$$

The equtation also shows that height values $h(\vec{g})$ are computed with the integer coordiantes. This avoids numerical problems that would cause severe errors in fractal functions. In integer space one can compute random noise values and the result is always the same for a fixed position. With this approach the problem to find a

position dependent but numerically stable noise function is solved. In section 4.2.2 will be explained how this computation works in detail. The integer space has some further advantages.

Simple subdivision of a coarse model for the planetay sphere (see polyeders in figure 4.3) would not lead to a better approximation of the sphere. Newly created vertices would lie on previous edges. It is important to transpose new vertices slightly so that they are positioned on the sphere. The projection from the integer space guarantees this property without any numerical problems (rounding errors).

The distance measurement is very important for the LOD system. The IU-LOD-algorithms computes distances in integer space to choose the level of detail. Therefore the L1 norm ($|x1 - x2| + |y1 - y2| + |z1 - z2|$) is taken which makes it possible to compute all transitions between different levels of detail very easily. This computation will be explained in the LOD-update (section 4.2.4). It is important to know that this computation in integer space avoids rounding errors and recursive tests. Subdivision of a patch can be done independent from neighbour patches so it should be much easier to parallize this algorithm.

The integer space is also used with the static objects that cover a planet. For now there exist plants and houses on earth. The placement of these objects makes use of fractal functions to create a random distribution. In general objects can be placed at every grid coordiante. Integer space is used to store how much area is covered by each object or how much is still free. With integers the needed compare operations and other computations are faster and easier to perform.

4.2.1.3 Camera Space

The third used number space is the limited view range in OpenGL which we here call camera space. The program can only display objects that are between 0.1 and 16000000.0 units away from the camera. The required depth buffer (2.1.12) in OpenGL is on the limit of current graphics boards (24 bit). How the resulting visualization problems were solved will be explained in the rendering section (4.2.7). It is obvious that a whole planetary system does not fit into the camera space. It is only big enough to hold one planet. Geometry that is too far away is scaled down in size and distance to fit into the camera space. So it is possible to render all objects of the scene.

The critical limit is a distance of 8000000.0 units. Everything with a larger size or greater distance is translated by equtation 4.3. The factor H is set to 8000000.0 which is half the size of the camera space.

$$d_{camera} = H + H \cdot (1.0 - e^{\frac{H-d}{d_{invisible}}}) \tag{4.3}$$

The variable $d_{invisible}$ is defined in the scenegraph and is a really far distance ($d_{invisible} \gg H$). The affected object still has to be scaled with d_{camera}/d then it can be rendered. This exponential downscale function makes it possible to render objects of any size. Figure 4.6 shows all number spaces used in the IU-LOD-algorithm.

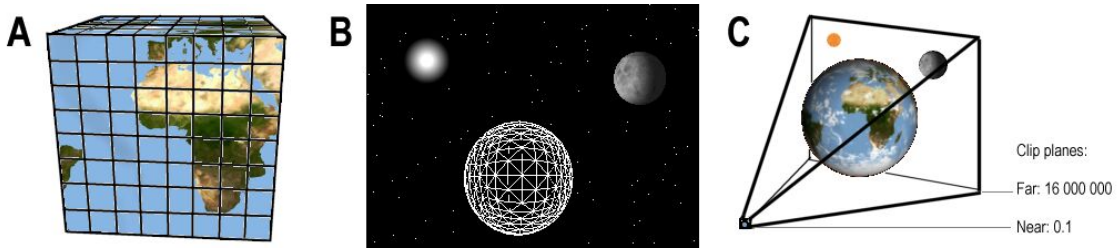

Figure 4.6: Image A: integer space, image B: double space, image C: camera space

4.2.2 Fractal Heightmaps

In section 3.3 some fractal functions were introduced in general. Here it'll be explained why the IU-LOD-algorithm uses Fractal-Midpoint-Displacement to generate terrain.

A fundamental requirement for all fractal landscapes is that a certain area has to be the same each time the terrain is generated. The computation must be deterministic. This requirement is not achieved when using simple random values (pseudo random numbers, PRNs). PRNs could be only used when the whole landscape is created in one step which is not the case for LOD-algorithms. So it is a problem to find position dependent noise functions that are numerically stable and deterministic. In most cases some popular noise functions like Perlin-Noise (3.3.1) or Fourier-Synthsis (3.3.2)) are chosen. Since the IU-LOD-algorithm has a very fine triangle resolution (1m) when the viewer is at low altitude these noise functions have to add many frequencies to generate enough detail information. The previously mentioned functions would require the summation of 23 frequencies for an earthlike planet to compute a single height value. This is too expensive and thus another solution was chosen.

In the section about the functionality the quadtree subdivision of a planetary surface was shown in detail. This structure can be used with the Fractal-Midpoint-Displacement-algorithm, too. The height of every new vertex that is created by subdivision gets a little noise variation. The noise value is computed from the integer coordinate of the vertex. So this computation is numerically stable. The amortized cost of a new vertex is one computation compared to 23 with the other functions.

An interesting problem in a mathematical sense was the creation of the noise function. The position dependent noise function does not really create random values because it is not possible to compute random noise. Stochastic and visual analysis of the noise function show the behaviour of white noise. It was not possible to find repeating patterns in the output. This also holds for different sample rates of the noise function. This means the function simulates random noise for all sample rates. If a value is taken every 1024 units it is the same effect as if a value is taken at all units.

How does this work? The noise function maps all positions in integer space into a single value in $[-1, 1]$. First an array of 256 indices becomes shuffled. The permutation depends on a random seed which is used to initialize the computation. This guarantees that each planet has it's own noise function. Later on the following computation is done for every vertex. Noise4D is a four-dimensional function but the time component is not used in the program.

```
double Noise4D(UINT x, UINT y, UINT z, UINT t)
{
  UINT result = noiseMap[0];
  while (x > 0 || y > 0 || z > 0 || t > 0)
  {
    // Noise1D takes x as input and returns noiseMap[x mod 256].
    result += Noise1D(x + Noise1D(y + Noise1D(z + Noise1D(t))));
    // All variables are right-shifted by eight bits
    // so that every byte is used in the computation.
    x >>= 8;
    y >>= 8;
    z >>= 8;
    t >>= 8;
  }
  // Returned value is in range [-1,1].
  return (((result mod 256)/(double)128) - 1.0);
}
```

As one can see each byte of the integer coordiantes is used to compute the noise value with chained permutations in the index array (noiseMap). Every position is mapped in a unique way and the computed values do not repeat after 256 units as they do with Perlin Noise.

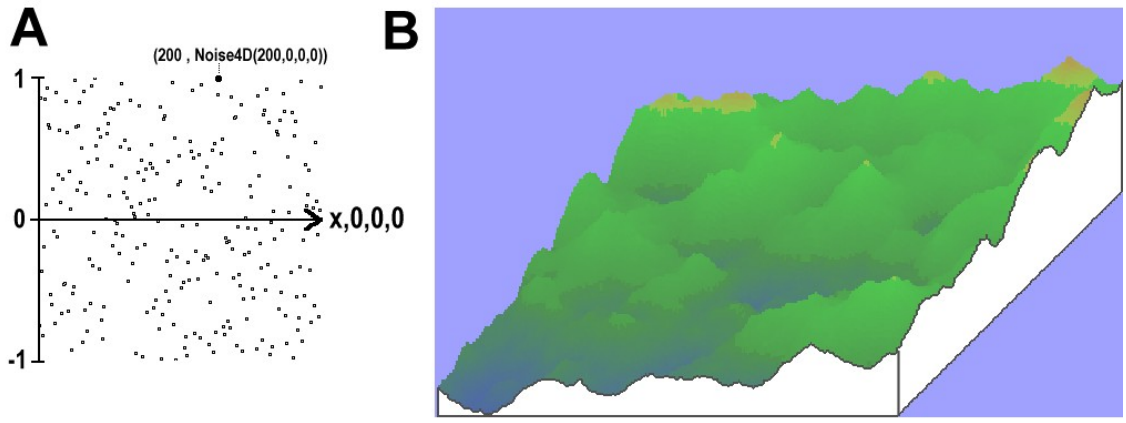

Figure 4.7: Image A: Noise function, image B: Fractal landscape with Noise4D

4.2.3 Scenegraph

The scenegraph stores the whole 3D environment in a hierarchic structure. The visualization's performance is much better when all objects use multiple levels of detail. The first property of the LOD-system in the scenegraph is that objects of all sizes become invisible (not rendered) if they are too far away. Equtation 4.4 is used for each object O in the scenegraph.

$$d_{invisible}(O) = 1000 \cdot radius(O) \tag{4.4}$$

$Radius(O)$ is the radius of the bounding sphere which is at least as large as the convex hull of the object. The factor 1000 was chosen so that objects that appear smaller than 1/1000 of the screen width ($\sim$1 Pixel) are not shown. All parts of the geometry (e.g. patches of the quadtree) are stored with their maximum size (diameter of the bounding sphere) in meters. One additional property is called object-LOD-parameter and stores the base-two-logarithm of the size. An object with object-LOD 10 has a diameter greater or equal 1024m. This parameter has some benefits in the Geometry-Update 4.2.6 of a planet because all static objects (plants, houses) are rendered using the object-LOD. This means big objects (skyscraper) are placed in the landscape at high LODs whereas small objects (grass) are placed and rendered much later.

At the initialization stage a planet is a sphere with a very high object-LOD (earth:23). If the camera gets closer to the planet geometry must be subdivided. In general any technique that has discrete LODs can be used. In VRML it is possible to store a 3D model with different LODs by default. The term LOD was explained in section 2.1.5 in detail. The same technique is used for the geometry of a whole planet. But the geometry can't be stored in the finest LOD (see Introduction, 40 TeraByte). This and the high number of LODs that have to be managed are the main problems that are solved with the IU-LOD-algorithm.

With the scenegraph it is much easier to manage dynamically moving objects. All objects on earth (or in orbit around) are stored relativ to the center of earth and not in absolute coordinates of the double space. So each object stays at the place it should be even if earth rotates and moves around the sun. This movement is really high-speed. Earth moves with a speed of 30km/s through the space. Without the hierarchy of the scenegraph all dependent objects had to be recalculated for every frame. This would cause severe numerical and visual errors.

4.2.4 LOD-Update

This step computes which levels of detail are needed to visualize the planet. The distance from the camera to the object is the only criteria that is used for subdivision. This choice is very important since it enables a subdivision scheme without recursive neighbourhood tests that are required by other LOD-algorithms. Filigree structures are not rendered optimally but this subdivision scheme creates a triangulation where nearly all triangles are equally sized (in screen size) at all distances.

First the so called minimal LOD variable (LOD_{min}) is computed. This minimal LOD is the finest resolution that has to be created by the algorithm. Subdivision is performed down to this LOD. The computation takes the camera's height above terrain and determines how much terrain can be seen. Let the vertical field of view be $fovy$ degrees. The camera looks straight down on the surface. Then $2x \cdot \tan(fovy/2)$ meters of the surface are visible if the camera is x meters above the surface (see Figure 4.8). An object that is viewed from distance x and fills the whole screen is at least $2x \cdot \tan(fovy/2)$ meters in size.

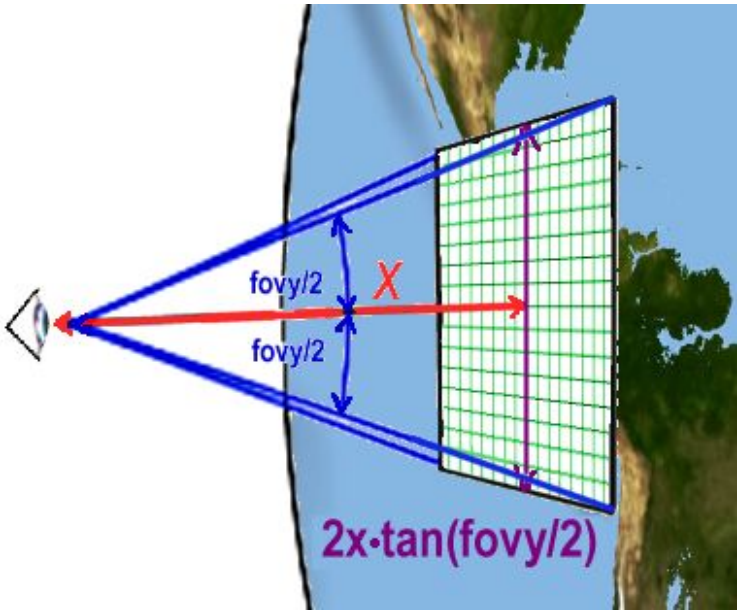

Figure 4.8: Compute minimal LOD with distance to camera

If the logarithm of this object size was taken as minimal LOD all triangles would be as large as the whole screen. So there is a parameter that limits the size of all triangles. Equtation 4.5 shows how to compute the minimal LOD.

$$LOD_{min} = log_2(2x \cdot \tan(fovy/2) \cdot Screensize) \tag{4.5}$$

The result is the finest level of detail that must be rendered. This computation can be changed with the quality parameter *Screensize* that defines the triangle resolution and thus the pixelerror. In the program this pixelerror can be adjusted from 1/16 down to 1/64 of the screen size.

Now it is clear how far the terrain has to be subdivided at the camera's position to meet the required visual quality (pixelerror). The position difference from one frame to the next one is also stored to achieve fast framerates even for fast movements through the terrain. The base-two-logarithm of this distance is compared to the minimal LOD and the maximum of both is used. The idea behind this concept is that it is not necessary to show every flower on a field when moving with some km/s across the landscape. High speed movement has the same effect as a larger viewer distance, the changed minimal LOD does not force the finest terrain resolution anymore.

A very large object like a whole planet can not be rendered using only one level of detail. The minimal LOD value is a lower border for the subdivision of terrain. All existing levels of detail have to be connected with a continuous triangulation scheme as described before. The more complicated step is now to find all transitions between different levels of detail.

At first the camera's position is transformed into integer coordinates of the planet (integer vector: *cam*). So it is possible to compute exactly and numerically stable where the LOD-structure has to be changed for the current frame. Therefore the L1-distance (*dist*) from the current patch to the camera is calculated.

$$dist = |patch.x - cam.x| + |patch.y - cam.y| + |patch.z - cam.z| \tag{4.6}$$

The next step is to define a border that limits the area which uses a certain level of detail. This area is centered at the camera's position. This border distance is directly dependent from the current level of detail. In the scenegraph section was described that an object with LOD i has the size 2^i. This is also true for patches. In the following computation there is an additional quality parameter (QualityScale) that influences the triangle resolution. It also ensures that two neighboured patches have no more difference than one LOD.

$$border = (2 + QualityScale) \cdot 2^{patch.LOD} \tag{4.7}$$

At the end the following test is performed for all patches in the quadtree to decide if further subdivisions have to be done or not.

$$divide = (LOD_{min} < patch.LOD)\ AND\ (dist \leq border)\ AND\ (isVisible) \tag{4.8}$$

The variable $isVisible$ is computed in the view-update 4.2.5. If the current patch is not visible there is no need for further subdivisions. The child nodes are deleted in this case. If the patch is visible and subdivision is aquired new child nodes are created. The vertex subdivision routines are explained in the geometry-update (section 4.2.6).

After subdivision a non-recursive neighbourhood test is done. It uses the global distance metric (L1-norm) and computes for each patch in the quadtree what LOD the neighbour patches have. This computation is independent from other patches and could be done in parallel. This test uses the same equtation as the subdivision test but the distance comes from the neighbour patches and the value of the variable border is doubled. Figure 4.4 on page 19 shows an example triangulation of the LOD-system.

4.2.5 View-Update

Early versions without visibility check could not visualize terrain in real-time. If one uses the hierarchy of a quadtree a very efficient visibility check can be done. One way is to use the bounding spheres of all objects. If the bounding sphere of a patch is not in the view frustum of the camera none of it's child nodes is visible. Image C in figure 4.6 shows the view frustum of a camera in OpenGL. It has the shape of a pyramid. Each side defines a clip plane. Then it is calculated if the distance of the object to each plane is greater or equal the negative value of the bounding sphere's radius. This can be done be building scalar products. This simple test is done for all patches to avoid rendering more triangles than needed.

4.2.6 Geometry-Update

The LOD-update computes for all patches if they must be subdivided or not. Since all data is stored in the quadtree this means to create or delete child nodes. If a new child node is created there are some steps to perform. The first one is to copy the geometry from the parent node to the child. The new patch contains new vertices as it has a finer resolution (LOD-1). These new vertices have to be created with a special height function for all objects. On planets this function uses bicubic interpolation for the stored heightmap of a planet. The resolution of this map is not fine enough if the triangles become smaller than the interval between the sample data points of the

map. Then the data has to come from additional Digital Elevation Models (2.1.4) or fractal details are generated. If fractals are generated a slightly modificated version of the Fractal-Midpoint-Displacement-algorithm is used as explained in section (4.2.2).

The geometry-update generates all information needed to render the newly created patch. One important task is to transform integer space coordiantes into valid points of the double space. Further computations create color values, texture coordiantes and normal vectors for all new points.

After these computations an additional function is called which places the static objects in the current patch. In a patch of LOD k only objects with object-LOD $i \leq k - 3$ (1/8 of the patch size) can be placed. On one side this guarantees that all objects lie in the bounding sphere of the patch and on the other side that all objects have the same subdivision criteria (pixelerror) as the triangles of the terrain. At the beginning all objects are sorted by their object-LOD value. This avoids a long search at runtime. For all objects some properties like preferred height above sea level (climatic zone) or distribution density are specified. These factors influence the placement of objects. For houses there is the restriction that ground may not have more slope than 15%. All properties are used to compute a value in $[0, 1]$. Then a position dependent random value must be greater than this border to place an object. This statistic element avoids full placement at all positions and rather creates a fractal distribution pattern.

4.2.7 Rendering

For rendering all quadtrees are visited recursivly. If a patch (current node of the quadtree with 9×9 vertices, 128 triangles) is visible it must be rendered. Each forth part of the patch is checked for child nodes. If a child patch exists this patch is rendered. Otherwise the corresponding part of the current patch is shown. All patches have the same triangulation but transitions between different levels of detail have to be considered to avoid cracks. These transitions were found in the LOD-update by a global function. As two neighboured patches have no more difference than one level of detail there are two possible triangulations (see figure 4.9). Image A shows two patches with the same LOD, image B shows how the transition to the next LOD has to be done.

All possible triangulation patterns are stored in index lists that are created when the program starts. Performance is optimal when using TriangleFans for rendering

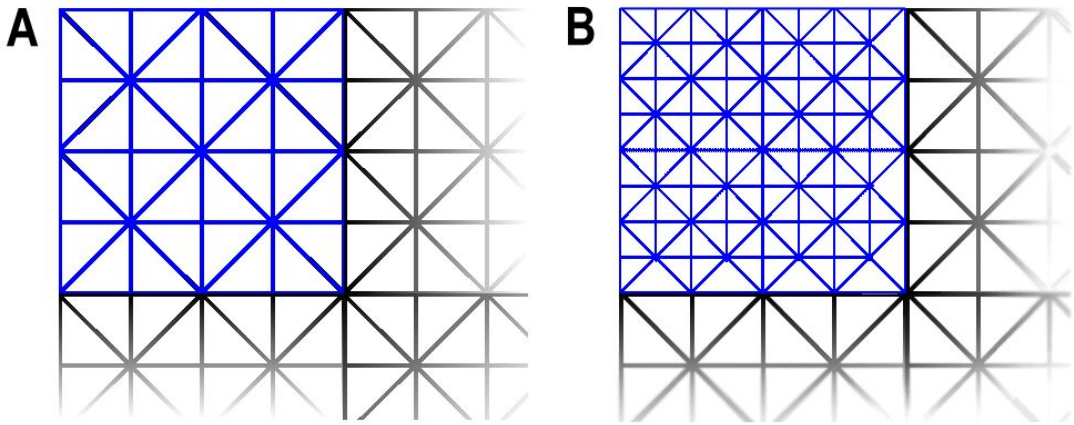

Figure 4.9: Triangulation of LOD-transitions

with OpenGL. TriangleStrips did not perform faster since the patches are relativly small and thus there are no long strips. The whole geometry of a patch is stored in fields that can be directly used as vertex arrays.

For now there was a brief description how the terrain is rendered. A planet has much more things to show that define a landscape. On earth these are water, clouds, plants and houses (the program does not contain more, yet). All these details are visualized with multipass rendering. Blending of transparent objects requires dynamic sorting of objects. If the camera is underwater the water surface is drawn last. If the camera is above the cloud layer they are rendered over the whole image. In every quadtree the whole visual information is stored that means not only terrain but objects like trees and houses, too. They are rendered if the patch is visible.

In practice an additional step is necessary. Even modern graphics boards have 24 bit depth buffer but not more. This is not enough to render astronomically large scenes. The engine shall be able to render objects from a distance of 10 cm (near clip plane) to infinity. Graphics boards can only display a limited depth range (i.e. 1-10000). So geometry is splitted using two-pass rendering. First all distant geometry ($>$ 10000m) is drawn, then the near range. Each pass uses a modified depth buffer. If graphics boards had at least 32 bit depth buffer in the future this would simplify rendering.

Chapter 5

Results

5.1 Measurements

All measurements were done on a Pentium 4 machine with 1.8 GHz and 256 MB RAM. The graphics board was a GeForce3 Ti 200 equipped with 64 MB VRAM. The first test (table 5.1) simulates a flight on the moon with a speed of 200m/s (720km/h) and 50m above terrain. This performance measurement represents all planets without static objects and determines how fast terrain can be rendered. The second test (table 5.2) is a stress test. Here the camera is located at Hawaii and all visual effects are activated this means plants, houses, water and clouds are rendered. Speed and height are set as before. The height data of Hawaii is taken from a 16 MB DEM-file and not created with fractal functions.

The tests used a viewport size of 1280×1024 pixels and 32 bit color depth. The tables are to understand in this way: Column Quality classifies the pixelerror. The range is from 'low' for a maximal pixelerror of 33 pixels to 'perfect' for a maximal pixelerror of one pixel. The quality parameter is displayed as status information when running the program. This pixelerror not only holds for terrain but all visible objects. This means if the quality is set to 'perfect' it is almost not noticable when an object appears on the screen.

The next columns show the framerate (FPS) and the number of triangles in this frame. The number of triangles in counted in 'normal' mode and without clipping to show its improvement.

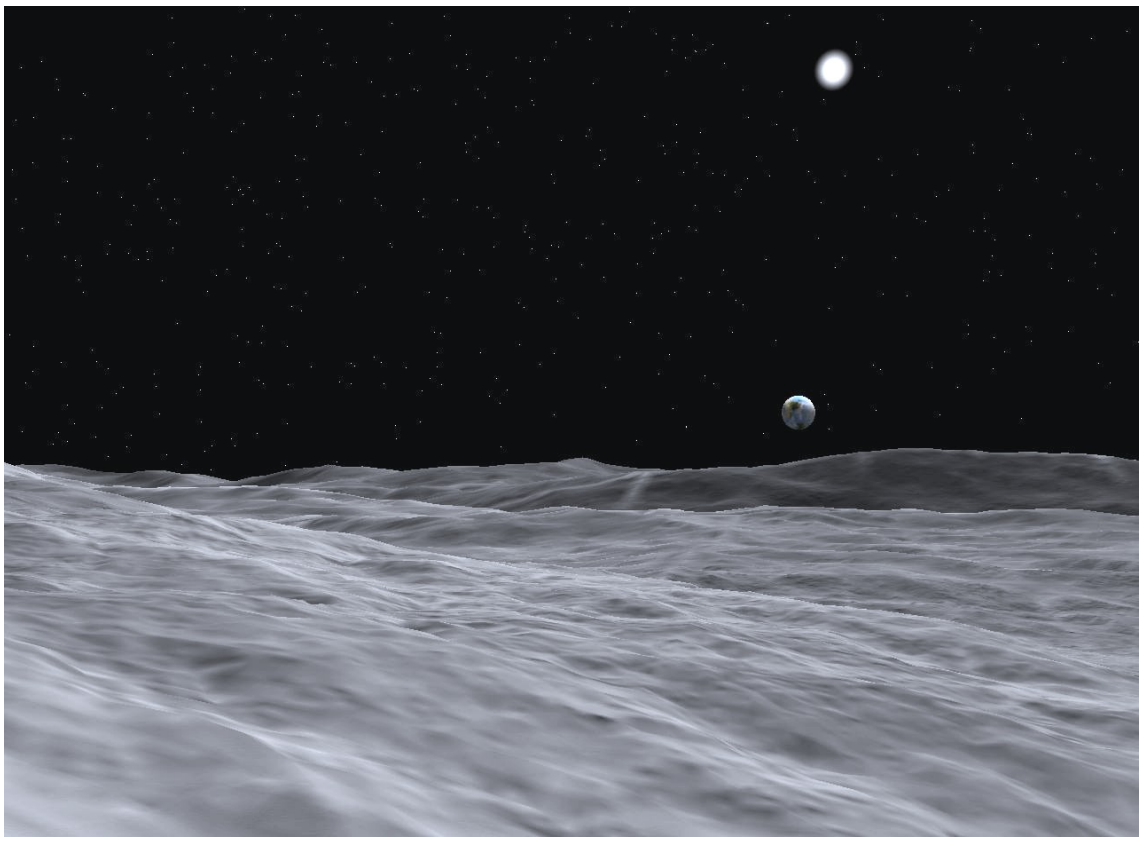

Figure 5.1: Flight on the moon

Quality	Pixelerror	FPS (without clipping)	#triangles (without clipping)
low	32	42 (24)	24k (63k)
medium	16	32 (14)	40k (125k)
high	8	22 (6)	63k (214k)
very high	4	16 (4.5)	90k (300k)
super high	2	12 (3)	120k (430k)
perfect	1	8 (1.2)	150k (550k)

Table 5.1: Measurement: Flight on the moon

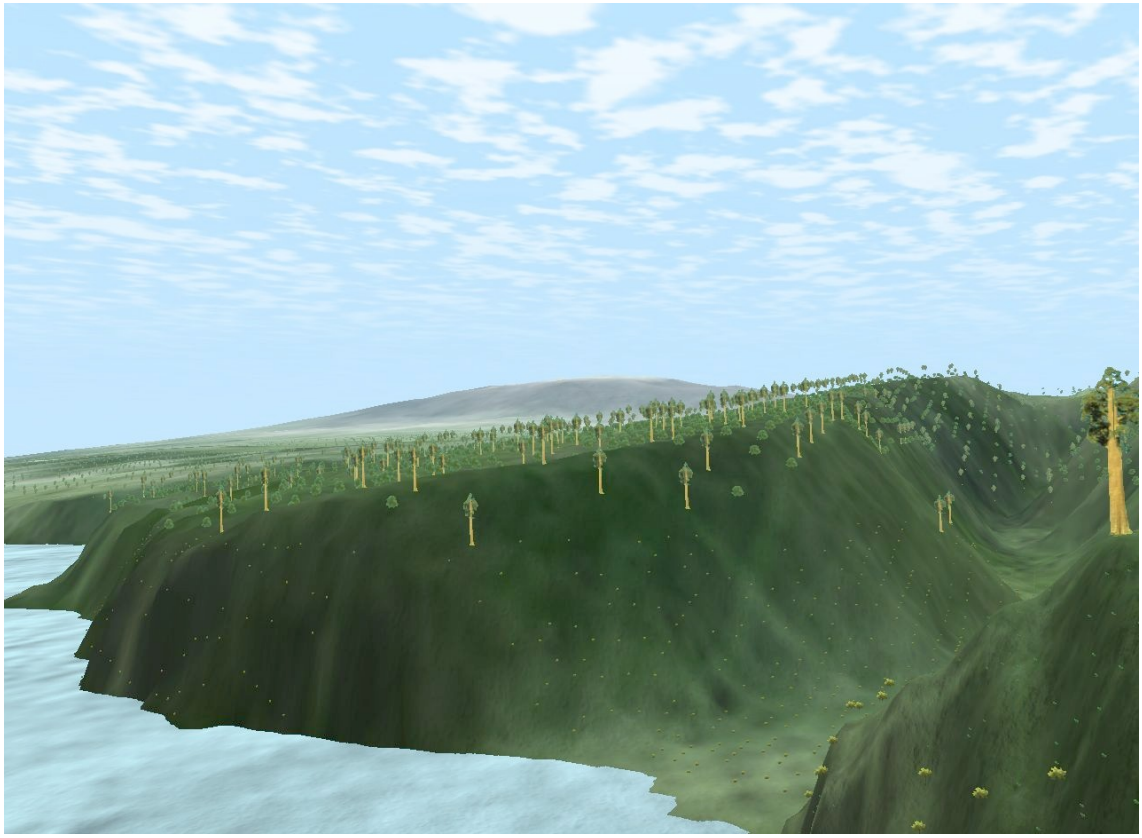

Figure 5.2: Flight over Hawaii

Quality	Pixelerror	FPS (without clipping)	#triangles (without clipping)
low	32	21 (9)	40k (130k)
medium	16	13 (4.4)	72k (274k)
high	8	9 (3)	110k (440k)
very high	4	6 (1.8)	160k (662k)
super high	2	5 (1.5)	220k (850k)
perfect	1	3 (0.5)	280k (1.0M)

Table 5.2: Measurement: Flight over Hawaii

5.2 Sample images

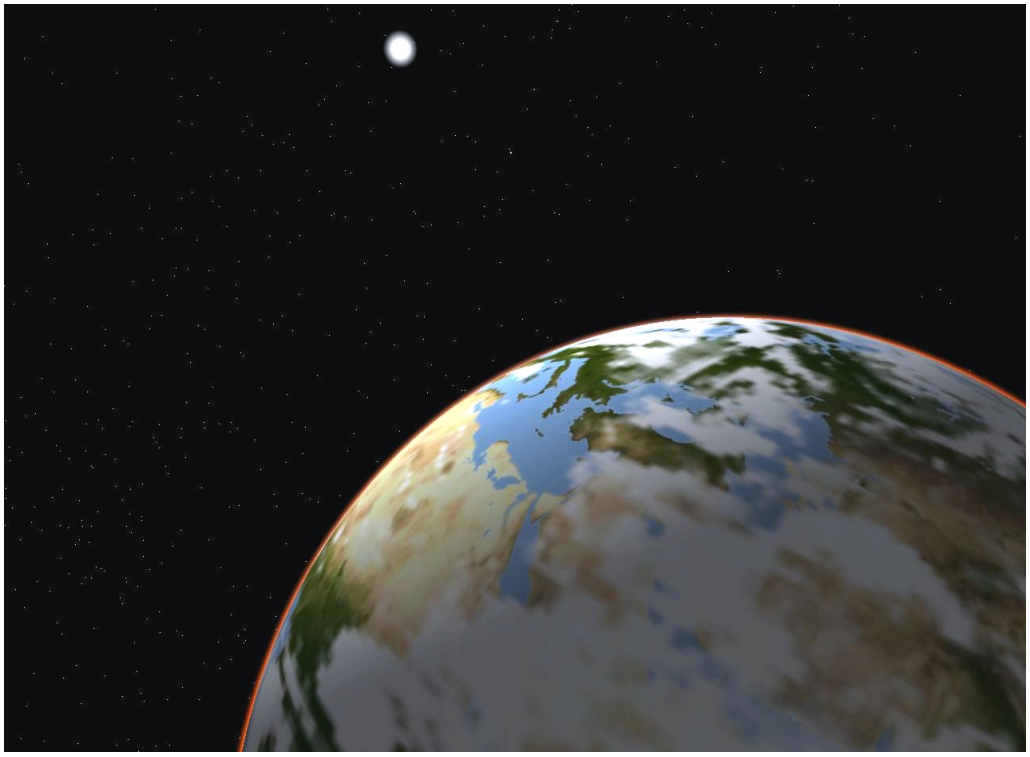

Figure 5.3: Flight to earth, Bild 1/4

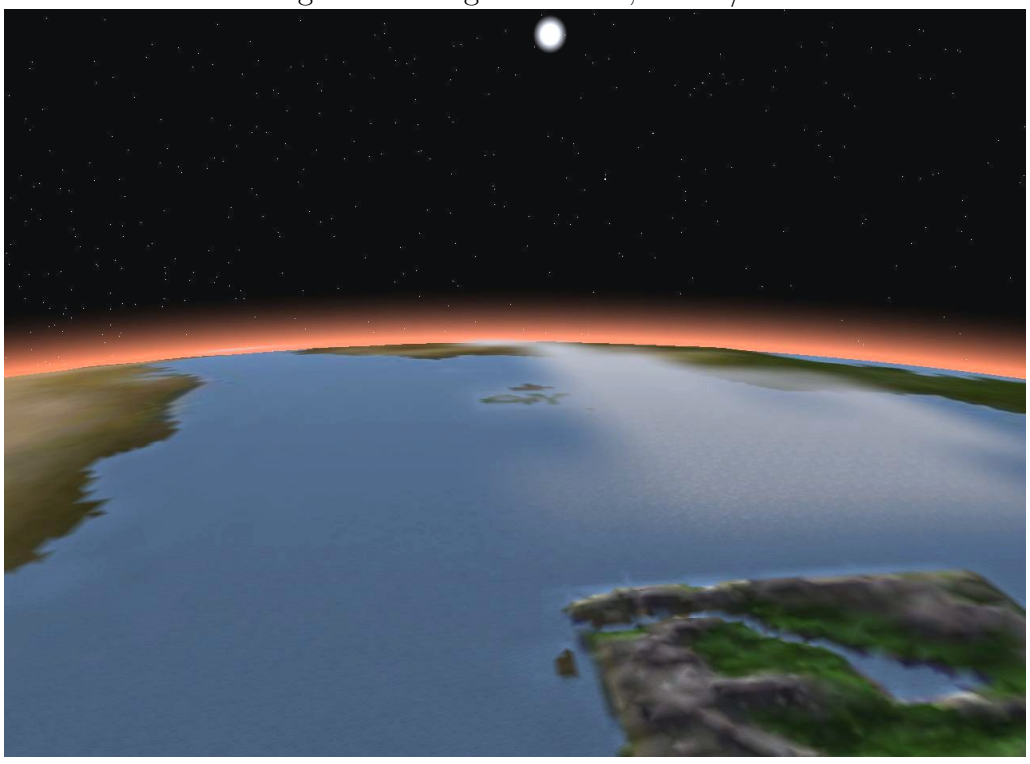

Figure 5.4: Flight to earth, Bild 2/4

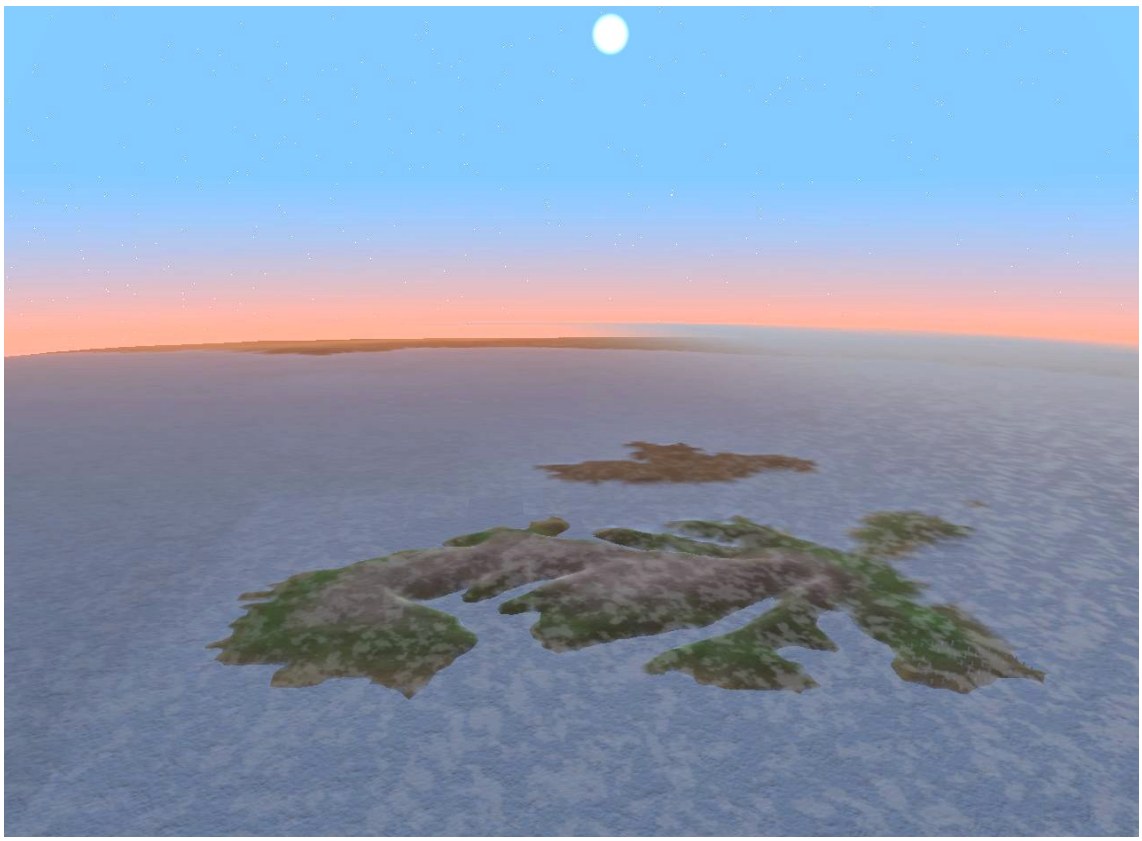

Figure 5.5: Flight to earth, Bild 3/4

Figure 5.6: Flight to earth, Bild 4/4

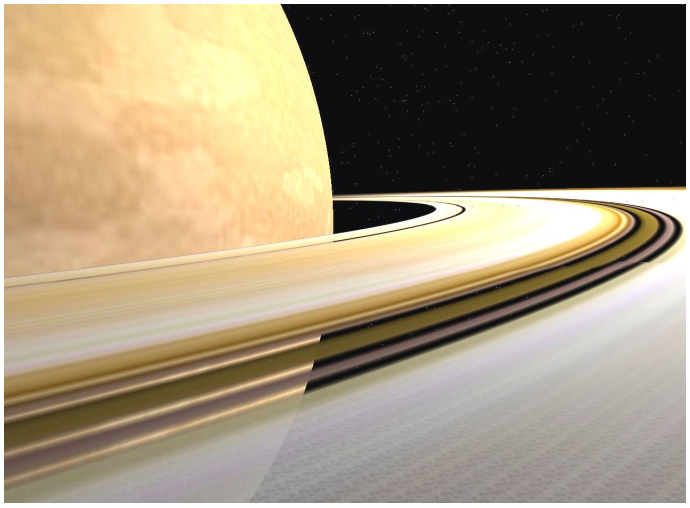

Figure 5.7: Flight to saturn, Bild 1/3

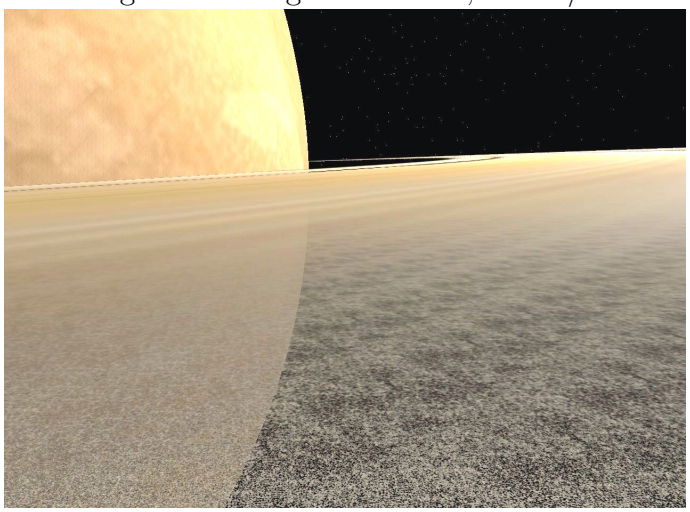

Figure 5.8: Flight to saturn, Bild 2/3

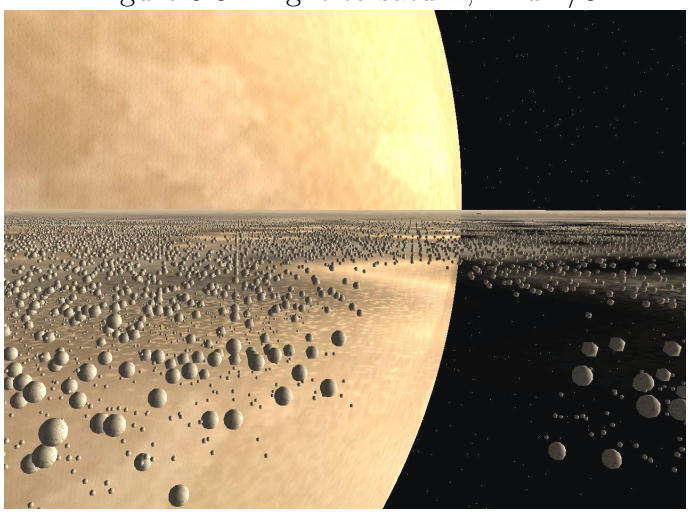

Figure 5.9: Flight to saturn, Bild 3/3

Chapter 6

Summary and Outlook

This work demonstrates that current PCs (not only workstations) are fast enough to visualize three-dimensional environments of arbitrary size. Therefore it is necessary to partition 3D objects with LOD-techniques and use clipping to compute and render only the visible geometry. Even high demands on the visual quality of the visualization (1 pixel error) can be satisfied with a scene complexity of half a million triangles per frame. This amount of triangles can be rendered in real-time in the near future (or with appropriate optimizations even now).

The developed program "Infinite Universe Engine" visualizes a whole planetary system from any point in space with at least 15 frames per second and a visual quality that comes close to that of current 3D games. The IU-LOD-algorithm is similar to other algortihms in terrain visualization but with strong focus on scaleability. The computational overhead grows logarithmically with the size of the 3D scene not linearly. But it is clear that the rich variety of natural environments is not simulated good enough. There is still much to improve.

The following extensions are imaginable (and partly under construction)

- 3D-Stereo-Visualization.
- Adaption to other scenegraph systems (OpenSG, Java3D).
- Parallelism of the IU-LOD-Algorithm for Clustered Computing.
- Support of common 3D formats (3ds, obj, vrml, Q3-bsp) for objects on planets.
- LOD-Universe from atoms to galaxies.

- Volumetric clouds and fog (best paper on this field (cloud rendering) by Mark Harris [15]).
- Physical simulation of objects.
- Simulation of cities with people, traffic etc. A very interesting paper to urban modeling comes from Parish and Mueller [14].

One aspect of this work is that most data is generated artificially with fractal functions. This is done at runtime without precomputation to keep this approach more genenral. If a specific application needs huge amounts of user defined data (i.e. 10 GB satellite data) it would be essential to create improved functions that dynamically load and unload detail information. At the moment the program loads all needed data in one bunch into memory.

In conjunction with these (un)load functions the use of a 3D format that stores hierarchical clustered data in all resolutions would be optimal. Such a Hierarchical Clustered Multiresolution Mesh (HCMM) was the best choice to store data for all scenegraph systems that support LOD.

Bibliography

Alle given URLs were last checked on April/21/2003.

[1] Dante Treglia: *Game Programming Gems 3.* Charles River Media, Hingham, Massachusetts, 2002.

[2] Sean O'Neil: *A Real-Time Procedural Universe, Part Two, Rendering Planetary Bodies.* Tutorial bei Gamasutra, 10. August 2001, http://www.gamasutra.com/features/20010810/oneil_01.htm

[3] Tomas Moeller, Eric Haines: *Real-time rendering.* A K Peters, Ltd., Natick, Massachusetts, 1999.

[4] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley: *Texturing and modeling: a procedural approach.* Academic Press, Chestnut Hill, Massachusetts, second edition, 1998.

[5] *Celestia*, http://www.shatters.net/celestia

[6] *Virtual Terrain Project*, http://www.vterrain.org

[7] Stefan Roettger, Wolfgang Heidrich, Philipp Slusallek, Hans-Peter Seidel: *Real-Time Generation of Continuous Levels of Detail for Height Fields.* Proceedings of the WSCG '98, 1998, http://wwwvis.informatik.uni-stuttgart.de/∼roettger/data/Papers/TERRAIN.PDF

[8] Peter Lindstrom, D. Koller, et al: *Real-time continuous level of detail rendering of height fields.* Computer Graphics (SIGGRAPH '96 Proceedings), p.109-118, 1996, http://www.cc.gatech.edu/gvu/people/peter.lindstrom/papers/siggraph96

[9] Mark Duchaineau, Murray Wolinsky, et al: *ROAMing Terrain: Real-time Optimally Adapting Meshes.* IEEE Visualization '97 Proceedings, 1997, http://www.llnl.gov/graphics/ROAM/

[10] ZHAO Youbing, ZHOU Ji, SHI Jiaoying, PAN Zhigeng: *A Fast Algorithm For Large Scale Terrain Walkthrough.* Paper of the State Key Lab of CAD&CG Zhejiang University, 2001, http://www.cad.zju.edu.cn/home/zhaoyb/paper/2001terranwalk_release.pdf

[11] Thatcher Ulrich: *Quadtree tiling / unique full-surface texturing.* 17. Februar 2000, http://tulrich.com

[12] Jonathan Blow: *Terrain Rendering at High Detail Levels.* Proceedings of the GDC 2000, 11. M"arz 2000, http://bolt-action.com (not active at the moment)

[13] Glenn Corpes: *Procedural Landscapes.* Proceedings of the GDC 2001, http://www.cix.co.uk/∼glennc/gdcetalk_files/frame.htm

[14] Yoav I H Parish, Pascal Mueller: *Procedural Modeling of Cities.* Paper der ETH Z"urich, 2001, http://www.cg.inf.ethz.ch/Downloads/Publications/Papers/2001/p_Par01.pdf

[15] Mark J. Harris, Anselmo Lastra: *Real-Time Cloud Rendering.* Department of Computer Science, University of North Carolina, Chapel Hill, North Carolina, USA. Proceedings of the EUROGRAPHICS 2001, http://www.cs.unc.edu/∼harrism/clouds/

[16] OpenGL, http://www.opengl.org

[17] Jackie Neider, Tom Davis: *OpenGL Programming Guide.* Addison-Wesley, Reading, Massachusetts, second edition, 1997

[18] The Independent JPEG Group, www.ijg.org, ftp://ftp.uu.net/graphics/jpeg/

[19] The Qt library, http://www.trolltech.com

[20] Bryce, http://newgraphics.corel.com/products/bryce4.html

[21] Vue d'Esprit, http://www.e-onsoftware.com/

[22] TerraGen, http://www.planetside.co.uk/terragen/

[23] Overview of programs to generate photorealistic landscapes, http://www.vterrain.org/Packages/Artificial/index.html

Appendix A

The Program

A.1 Requirements

Here the software and hardware requirements of the "Infinite Universe Engine"are described.

A.1.1 Software

The "Infinite Universe Engine"was developed to run with OpenGL [16] that needs to be supported by your operating system. OpenGL 1.1 drivers should be sufficient. The program was successfully tested with Windows 98, Windows XP and SuSE Linux 8.0. Under Windows two external dlls that are needed to run are delivered with the program. These are the runtime libraries QT-2.3.0-Noncommercial from Trolltech (siehe [19]) and LibJPEG of the Independent JPEG Group (siehe [18]). Linux distributions (i.e. SuSE) most often have native support of these libraries. Otherwise they have to be installed manually.

A.1.2 Hardware

The engine was developed on a Pentium 4 machine with 1.8 GHz and 256 MB RAM. The graphics board was a GeForce3 Ti 200 equipped with 64 MB VRAM.

Tests with other computers showed these minimal requirements:

Computer with CPU $\geq$ 800 MHz and 128 MB RAM, graphics board $\geq$ 32 MB.

A.2 Operation

Here is explained how to control the "Infinite Universe Engine". The most essential commands can be found on the help page of the program (Key 'H').

A.2.1 Controls

The program has to be controlled by a combination of mouse and keyboard commands. To look around the mouse is used by moving the mouse with a pressed mouse button. This avoids erroneous movements.

Until now there only exists one aircraft model that is controlled this way:

- Mouse move with left button pressed to change view direction.
- Mouse move with right button pressed to roll around view axis.
- Press 'SPACE' for immediate stop.
- Keys '1' to '0' set the acceleration (1 = 0.1m/(ss), 2 = 1m/(ss), ... 0 = light speed/s).
- Press 'UP' to accelerate.
- Press 'DOWN' to slow down.
- Key '+' zooms into the planet, '-' zooms out.
- 'C' allows to move the camera independently from the aircraft model (look around).
- 'R' resets the camera if it was freely moved with 'C'.

A.2.2 Options

The engine has some further graphic options that can be (de)activated with keyboard commands.

- 'B' (de)activates backface culling in OpenGL.
- 'F' (de)activates fog for atmosphere effects.
- 'G' marks all patches of a specific LOD.
- 'L' (de)activates the spotlight of the camera.
- 'M' (de)activates the rendering of the aircraft model.
- 'Q'(+'SHIFT') switches through 6 Qualities 'low'-'perfect'.
- 'W' (de)activates wireframe mode.
- 'Z'(+'SHIFT') sets the timeSpeedup-factor (0.01 - 10000) which defines a day's duration. (1 = real-time, one day is 24h; 10000 = one day lasts 8,6 sec)

A.3 User Data

In the future the "Infinite Universe Engine"shall be able to create every kind of data with procedural techniques. At the moment the whole terrain of a planet can be created with fractal functions. Objects that are placed in the landscape have to be defined by the user. Therefore a new XML-format was created to support user defined data within the engine. So it is possible to define plants, houses or specific geographic information with DEMs and greyscale images. All this data is taken to improve the visualization.

The following steps are performed: when the program starts it first reads the file config.xml and determines in which file the planetary system is defined (i.e. allplanets.xml). This file is loaded and every defined planet is generated with all its objects to be rendered.

The following sections show parts of the valid specification of the XML-format (until April 2003) that enable the user to modify the program data base. Additional information can be gained with the XML-files that come with the program. The following definitions contain comments in XML-style `<!-- comment -->`, so it should be possible to copy and paste them as errorfree containers.

A planetary system has the following stucture:

```
<SOLARSYSTEM NAME="MySystem">
  <!-- List of planet definitions     -->
  <!-- This list contains objects of  -->
  <!-- type STAR, PLANET, RING        -->
</SOLARSYSTEM>
```

A in depth explanation how to define planets is in section A.3.1. All objects have an attribute `NAME` which has to be unique since objects are adressed by name in the program! All sizes are defined in meters. It is important not to mix up lower and upper case characters because it could cause file loading errors. Images can be defined only in 'bmp', 'jpg' and 'tga' format.

A.3.1 Planets

The visualization of whole planets in rich variety and with full details is the main purpose of this work. So the definition of a planet is quite extensive.

A.3.1.1 Planet Definitions

A planet is defined with the following structure:

```
<PLANET NAME="Testplanet"
        <!-- optional list of parametes -->
>
  <!-- optional list of object definitions                -->
  <!-- objects are the tags DEM, PLANT, HOUSE, ASTEROID   -->

  <!-- optional elements: WATER, SKY, CLOUDS              -->
</PLANET>
```

The list of parameters is explained in the next section. It defines the base type of the planet (earthlike, moonlike, gas giant, ...) and further geometric properties (size, orbit, roughness of terrain, ...).

A planet can have three additional elements: `WATER, SKY, CLOUDS`. With `WATER` it is defined if a planet has oceans full of water (or other liquids). `SKY` defines the atmospheric hull of a planet if it has one. Earth is covered with an atmosphere that glows blue and becomes red on mornings/evenings. The moon has no atmosphere so the star background is always visible. Clouds are defined with the `CLOUDS` tag. For an earthlike planet it is sensible to declare all these tags.

A.3.1.2 Parameters

Now follows an example to show the extensive list of parameters in the start tag of a planet. Nearly all parameters are optional and missing attributes get default values:

```
<PLANET NAME="Earth"
        CENTER="" <!-- moons must mention their planet    -->
        <!-- TYPE defines a basic visual appearance if    -->
        <!-- there is no further specification           -->
        TYPE="EARTH" <!-- or SUN, MOON, MARS, GAS         -->
        RADIUS="6378000.0"
        HEIGHT="10000.0" <!-- Height variation in terrain -->
        <!-- Mountains and sea valeys reach to +/- 10000m -->
        <!-- FINETEX1-3 define detail textures (opt.)     -->
        FINETEX3="Data/detailjo.bmp"
        <!-- ORBIT defines average distance to sun        -->
        ORBIT="149600000.0e3"
        <!-- axis angle in degrees against global up-axis -->
        AXIS_ANGLE="20.0"
        YEAR_DURATION="365" <!-- lenght of year in days   -->
        DAY_DURATION="24"   <!-- lenght of day in hours   -->
        <!-- TEXTURE is used as planetary texture instead -->
        <!-- computation of fractal texture with TYPE.    -->
        TEXTURE="Data/planetmaps/earth2k.jpg"
        <!-- Height values of planet come from greyscale  -->
        <!-- image, more details are created fractally.   -->
        HEIGHTFIELD="Data/planetmaps/earthbump2k.jpg"
>
  <!-- same as before ... -->
</PLANET>
```

A.3.1.3 Element: Water

If there is a WATER-tag, all terrain below zero altitude is covered with oceans.

```
  <!-- If Terrain under 0m should be covered with water   -->
  <!-- simply define the WATER tag.                       -->
  <WATER>
    <COLOR RED="0.6" GREEN="0.8" BLUE="1.0"/>
    <!-- r,g,b-definition for the color of water          -->
  </WATER>
```

A.3.1.4 Element: Atmosphere

An atmosphere is defined this way:

```
<!-- If the planet has an atmosphere define a SKY tag.  -->
<SKY ATM_RANGE="100000.0"> <!-- Depth of atmosphere hull-->
  <!-- Day-/night-color of the sky (blue)               -->
  <COLOR_DN RED="0.5" GREEN="0.8" BLUE="1.0"/>
  <!-- Red glow of sky on morning/evening               -->
  <COLOR_ME RED="1.0" GREEN="0.3" BLUE="0.0"/>
</SKY>
```

A.3.1.5 Element: Clouds

The planet can have an additional cloud layer to improve the visual appearance of the sky.

```
<!-- If the planet has an atmosphere with clouds        -->
<!-- define a cloud sphere around the planet.           -->
<CLOUDS NAME="Earthclouds"
        TYPE="CLOUDS"      <!-- fixed type for clouds!  -->
        RADIUS="6388000.0" <!-- Clouds are 10km above   -->
        <!-- terrain, therefore define 6378km + 10km.   -->
        HEIGHT="0.0"       <!-- fixed value             -->
        <!-- FINETEX1 defines cloud detail structure.   -->
        FINETEX1="Data/clouds3.tga"
        ORBIT="0"          <!-- fixed value             -->
        YEAR_DURATION="0"  <!-- fixed value             -->
        DAY_DURATION="100" <!-- Clouds move slowly      -->
        <!-- around the globe, one time in 100 hours.   -->
        <!-- Global distribution of clouds (cloudmap).  -->
        TEXTURE="Data/planetmaps/earthclouds.tga"/>
```

A.3.2 Planetary Rings

To visualize saturn with its beautiful rings it was necessary to create a separate object class. A planetary ring (`RING`) has parameters similar to a planet but slightly modified to fit its geometry. All rings of a planet should be aggregated in one object and the separation is done with a ring texture that has transparent zones within. A ring has this definition:

```
<RING NAME="Rings of Saturn"
      CENTER="Saturn"     <!-- rings surround saturn       -->
      RMIN="70000.0e3"    <!-- inner radius of rings       -->
      RMAX="140000.0e3"   <!-- outer radius of rings       -->
      HEIGHT="400.0"      <!-- vertical extension          -->
      AXIS_ANGLE="20.0"   <!-- angle against planet's axis -->
      DAY_DURATION="20.0" <!-- duration of self rotation   -->
      <!-- This texture defines the global appearance.     -->
      RINGTEX="Data/planetmaps/rings.tga"
      <!-- This texture visualizes the random distribution -->
      <!-- pattern of the asteroids.                       -->
      FINETEX="Data/asteroids2.bmp">

   <!-- list of objects, only asteroids are sensible       -->
   <ASTEROID NAME="Big Stone_nocolor"
             WIDTH="40"
             HEIGHT="40"
             TEXTURE="Data/planetmaps/asteroid.jpg"
             DENSITY="1.0"
             MIN_H="-1000.0"
             MAX_H="1000.0"/>
</RING>
```

A.3.3 Stars

Stars have almost the same definition as planets. Though it is sensible to define them as separate objects. A star is a global source of light in the scene, a planet does not emit light. Stars also have a bright corona. The definition of our sun is given as example (only new attributes are explained):

```
<STAR NAME="Sun"
      TYPE="SUN"
      RADIUS="695000000.0"
      HEIGHT="100000.0"
      ORBIT="0"
      YEAR_DURATION="0"
      DAY_DURATION="0">

  <!-- 1st source of light in the system   -->
  <LIGHT NUMBER="0">
     <!-- Color of light                   -->
     <COLOR RED="1.0" GREEN="1.0" BLUE="1.0"/>
  </LIGHT>

  <!-- Texture for corona of the star      -->
  <CORONA TEXTURE="Data/corona.bmp">
     <!-- Colorisation of hot and cold -->
     <!-- zones in plasma (only visible    -->
     <!-- at low distance to sun)          -->
     <COLOR_HOT RED="1.0" GREEN="1.0" BLUE="0.3"/>
     <COLOR_COLD RED="1.0" GREEN="0.5" BLUE="0.3"/>
  </CORONA>

  <!-- Somewhat redundant description of the -->
  <!-- corona as atmosphere of the star.     -->
  <SKY ATM_RANGE="100000000.0">
     <COLOR_DN RED="1.0" GREEN="1.0" BLUE="0.3"/>
     <COLOR_ME RED="1.0" GREEN="0.5" BLUE="0.3"/>
  </SKY>

 </STAR>
```

A.3.4 Heightmaps (DEM)

The global height data for a planet has for memory reasons a very coarse resolution ($\sim$ 10km). The whole detail information is created with fractal functions but sometimes it is wanted to use exact data for a specific area of the planet. Therefore such Points of Interest can be defined with satellite data or other maps that contain height information for the terrain. The required files that contain this data are called DEMs (Digital Elevation Models). The program can read DEM-data from Binary Terrain Files 'bt', satellite data in USGS 'dem'-format and greyscale images. Pure greyscale images need some further geometric information. Some examples for DEM-definitions:

```
<DEM NAME="USA Lake Oregon"
     <!-- Height data -->
     FILENAME="Data/dem/crater_0513.bt"
     <!-- Texture for this area -->
     TEXTURE="Data/dem/lakeoregon.jpg"/>

<DEM NAME="Sardinien"
     FILENAME="Data/dem/terrainheightmap.jpg"
     TEXTURE="Data/dem/terraintexture.jpg"
     <!-- latitude/longitude information         -->
     LEFT="8.0"     <!-- 8 degrees east   -->
     RIGHT="10.0"   <!-- 10 degrees east  -->
     BOTTOM="39.0"  <!-- 39 degrees north -->
     TOP="41.0"     <!-- 41 degrees north -->
     MIN_H="-1500.0" <!-- lowest point in terrain -->
     MAX_H="4000.0" <!-- highest point in terrain -->
/>
```

A.3.5 Objects (Plants, Houses)

Here the definition of static objects in the landscape will be explained. For now there exist the tags (`PLANT, HOUSE, ASTEROID`) which visualize plants and houses on earth and asteroids in the rings of saturn. Plants, houses and asteroids are all derived from one base class and consume the same list of parameters, only the tags are different. These objects also use LOD-techniques but they are still pretty simple. Here a plant is given as example, the other object definitions are analogue.

```
<!-- HOUSE, ASTEROID analogue  -->
<PLANT
       NAME="Palme"
       WIDTH="10"   <!-- width of the object -->
       HEIGHT="20"  <!-- height of the object-->
       <!-- texture for the 3D model -->
       TEXTURE="Data/plants/palm.tga"
       <!-- distribution density in landscape   -->
       DENSITY="0.5"
       <!-- Plants only grow in specific height -->
       <!-- ranges that simulate climatic zones.-->
       MIN_H="0.0"
       MAX_H="300.0"
/>
```

Appendix B

The Sourcecode

In this appendix only some fundamental classes are explained in detail. This should help to understand the functionality of the different classes and shorten the time needed to work with the whole engine's source code.

B.1 JWPlanet

This class represents a planet in the program. There are many visual attributes (planetary textures, water, clouds etc.) and the geometry itself (quadtree). All comments are in english as the whole source code is treated this way.

```
class JWPlanet : public LODObject {
private:
  // reference counter for the number of planets
  static int planetcount;
  // default detail textures for all planets (used LOD for earth)
  static TEXTURE *staticTexDetail;          // LOD > 15
  static TEXTURE *staticTexFineDetail;      // LOD > 7
  static TEXTURE *staticTexSuperFineDetail; // LOD <= 7
  static GLuint waterId[32];                // 32 animated water textures

public:
  typedef enum PLANET_TYPE { SUN=1, EARTH=2, MOON=3,
                             MARS=4, GAS=5, WATER=6, CLOUDS=7 };
  // the style is important for the color scheme of the planet
  PLANET_TYPE style;

  // 6 quadtrees store the geometry of the planet (one per cube side)
  JWQuadtree *jwq[6];
  // vectors needed for orientation on each surface
  CIntVector dx_s[6];
  CIntVector dy_s[6];
```

```
float *HF;                  // heightfield for planet
TEXTURE *Tex[6];            // texture for each cube side
TEXTURE *TexDetail;         // detail texture for terrain
TEXTURE *TexFineDetail;     // fine detail texture for terrain
TEXTURE *TexSuperFineDetail;// super fine detail texture
TEXTURE *colorMap;          // use a planetary texture as color map

std::vector<ElevationGrid *> dems; // here we store digital elevation maps

int waterNum;               // current used water texture
bool hasWater;              // planet has oceans
float rgbWater[3];          // color of water
JWPlanet *waterSphere;      // help object for water

bool hasSky;                   // planet has atmosphere
float rgbSkyDayNight[3];       // 1st color of atmosphere
float rgbSkyMorningEvening[3]; // 2nd color of atmosphere
bool hasClouds;             // planet has clouds
JWPlanet *cloudSphere;      // help object for clouds
float ATM_RANGE;            // height of atmospheric hull
float fog_depth;            // depth of view in atmosphere

// factor between 1 and 100 (0 to disable),
// defines how dense cities are placed on planet
int cityDistance;

double radius;              // radius of this planet

// ******************** function declarations ***************************
// ********************** class interface *******************************

JWPlanet();           // constructor
virtual ~JWPlanet();  // destructor

/**
* This method initializes the whole planet and fills in all data.
* @param node    the <PLANET>...</PLANET> node in a XML document
*/
virtual void init(QDomElement node);

/**
* The "heart" of my algorithm. We recompute the LOD structure every frame.
* @param pCamera  camera that defines our viewport in the virtual world
*/
virtual void updateGeometry(C3DCamera *pCamera);

/**
* We draw the whole planet by calling special subroutines.
* @param pCamera  camera that defines our viewport in the virtual world
*/
virtual void draw(C3DCamera *camera);
```

```
// ******************** function declarations ****************************
// ********************** internal routines ******************************

// *********** conversions between integer and double space **************

/**
* This method converts a (x,y,z)-vector into a cube position (CIntVector).
* We need the int^3-vector for LOD management and random noise functions.
* @param x    x position
* @param y    y position
* @param z    z position
* @param c_s  index of the cube side (0-5) for the position (x,y,z)
* @param dist distance from geometry surface (in meters above sea level)
* @return     cube position in range [-CUBE_HALF, CUBE_HALF]^3, c_s and dist
*/
virtual CIntVector CoordinatesToSurfaceVector(const double x,
                                              const double y,
                                              const double z,
                                              int *c_s, double *dist);
/**
* This method computes the vertex in double space for a given cube position.
* @param pos   position in integer space
* @param c_s   cube side of integer position
* @param h     height of terrain at pos (in meters above sea level)
* @return      vertex position
*/
virtual CDoubleVector SurfaceVectorToCoordinates(CIntVector pos,
                                                 int c_s,
                                                 double h);

/**
* This method computes the vertex normal for a given cube position.
* @param pos   position in integer space
* @param c_s   cube side of integer position
* @return      vertex normal
*/
virtual CDoubleVector SurfaceVectorNormal(CIntVector pos, int c_s);

/**
* This is one of the core routines. Here we get a height value for every
* position on our planet. We first check the loaded DEMs (1), then use the
* global HF for high LODs (2) or 3D noise for fine LODs (3).
* @param point    cube position in range [-CUBE_HALF, CUBE_HALF]^3
* @param LOD      the level of detail we need for this vertex
* @param h_old    interpolated height value from higher LODs
                  (good for multifractal functions and sharp coast lines)
* @return         the current height value at this position and this LOD
*/
double get_HeightField_And_RandomOffset(CIntVector point,
                                        int LOD,
                                        double h_old);
```

```
/**
* This method creates a color value for each terrain vertex according to the
* planet type. Only height and position on terrain are used to create a color.
* @param hf   height value of vertex where the terrain color is computed
* @param pos  integer position of vertex
* @return     quadrupel (r,g,b,a) as color
*/
CFloatVector4 computeColor(const double hf,
                           CIntVector pos = CIntVector(0,0,0));

/**
* Computes the height of the given object over terrain of this planet.
* We search the affected quadtree and delegate the query to it.
* @param obj    object e.g. player model
* @return       height above terrain in meters
*/
virtual double getHeight_above_Surface(C3DBase *obj);

// ********************* internal init subroutines ***********************

/**
* Geometry construction of our planet. We initialize basic geometry and
* generate index lists for triangulation.
* @param pStyle   Still the main influence factor for appearance.
                  It's a constant like MOON, EARTH,... and is used very often
*                 to switch between the different cases.
* @param rad      radius of the planet in meters
* @param H_R      height range of terrain in meters.
*                 That means how high are mountains (< H_R),
*                 how deep are oceans/valleys (> -H_R)
* @param axis_angle   angle agaist up-axis
*/
void initPlanet(PLANET_TYPE pStyle, double rad, double H_R,
                double paxis_angle = 0);

/**
* This method inits our planetary textures which can also be fractal generated.
* We use 3 multiplicative layers of multitexturing for the planet's surface.
* @param tex            1st texture layer (high LODs > 15)
* @param finetex        2nd texture layer (medium LODs > 7)
* @param superfinetex   3rd texture layer ("detail" texture for LODs <= 7)
*/
void initTextures(char *tex=0, char *finetex=0, char *superfinetex=0);

/**
* Here we compute the 6 cube side textures for a planet.
* @param id     cube side number
* @param base   center position of the quadtree for this cube side
* @param dx,dy  vectors for 2-dimensional surface orientation
*/
void createCubeTexture(const int id, const CIntVector base,
                       const CIntVector dx, const CIntVector dy);
```

```
/**
* Here we init a global height field (HF) for the whole planet.
* We can either use a bitmap file source (filename in tex)
* or create a random HF of size pHF_size^2 with create_fractal.
* @param tex      filename of bitmap that is used as height field
                  (if 0 we create a fractal map)
* @param HFisComplete If false, underwater terrain uses fractal map.
* @param pHF_size size of the HF, a bigger map will look nicer but
                  current systems are limited. (TNT2: 256, GeForce3: 4096)
*/
void initHeightField(char *tex, bool HFisComplete, int pHF_size=256);

/**
* Uses fractal midpoint displacement to calculate a new landscape.
* @param MAP  size of HF we want to create
* @return     the created HF
*/
float* create_fractal(int MAP);

/**
* This function loads digital elevation maps.
* @param node    A <DEM ... /> node in the XML document
*/
void loadDEM(QDomElement node);

/**
* This function inits the water surface for a planet.
* @param node    A <WATER ... /> node in the XML document
*/
void initWater(QDomElement node);

/**
* Initialize water for the planet with color (r,g,b).
* We load 32 animated water textures and set hasWater = true
* @param r    color attribute
* @param g    color attribute
* @param b    color attribute
*/
void initWater(float r, float g, float b);

/**
* Creates a subobject for water surface rendering.
*/
void initWaterSphere();

/**
* This function inits the atmosphere for a planet.
* @param node    A <SKY ... /> node in the XML document
*/
void initSky(QDomElement node);
```

```
/**
* Init the sky of a planet with the colors (rdn,gdn,bdn) and (rme,gme,bme)
* for day/night and morning/evening atmosphere colors.
* @param rdn  color attribute
* @param gdn  color attribute
* @param bdn  color attribute
* @param rme  color attribute
* @param gme  color attribute
* @param bme  color attribute
* @param atmh sets the ATM_RANGE for the planet that means
              the thickness of the atmosphere in meters
*/
void initSky(float rdn=0, float gdn=0, float bdn=0,
             float rme=0, float gme=0, float bme=0, float atmh=10000.0f);

// ********************** special draw routines ************************

/**
* Here we draw the terrain of the planet
* @param pCamera  camera that defines our viewport in the virtual world
*/
void drawTerrain(C3DCamera *pCamera);

/**
* Here we draw the static objects that are placed on the terrain.
* @param pCamera  camera that defines our viewport in the virtual world
*/
void drawStaticObjects(C3DCamera *pCamera);

/**
* Here we draw a water surface for planets with hasWater == true
* @param pCamera  camera that defines our viewport in the virtual world
*/
void drawWater(C3DCamera *pCamera);

/**
* Here we draw the sky effect of a planet. We draw a circle of quads around
* the camera position that clips with the horizon of the planet. This models
* an orthogonal cut through the atmosphere layer and gives the nice effect
* of air around the planet in space and nice color variations on ground.
* @param pCamera  camera that defines our viewport in the virtual world
*/
void drawSky(C3DCamera* pCamera);

/**
* We compute if we are inside the atmosphere (if there is one) and how dense
* fogging should be. The visible distance can vary from from short view (1000m)
* up to no fog at all. We have to check the day/night cycle to adjust fog color.
* @param camera   camera that defines our viewport in the virtual world
*/
void updateFog(C3DCamera* camera);
```

```
  /**
  * We draw a fog sphere around the camera. In sky fogged geometry
  * only gets fog color but underwater we can clip at far distance.
  * @param camera   camera that defines our viewport in the virtual world
  */
  void drawFog(C3DCamera* pCamera);

}; // end of JWPlanet
```

B.2 JWQuadtree

The class `JWQuadtree` is a quadtree that stores all data needed to render the planet. So every patch is directly stored in the quadtree. The whole geometry is managed with arrays that are compatible with vertex arrays in OpenGL.

```
class JWQuadtree : public C3DBase {
private:
  bool isVisible;       // save the visibiliy computation here
public:
  // 4 child nodes: top left[2], bottom left[0], top right[3], bottom right[1]
  JWQuadtree *child[4];

  LODObject *jwp;       // pointer to parent object (planet or ring)
  JWPlanet *planet;     // dynamic_cast of jwp to planet class
  JWRing *ring;         // dynamic_cast of jwp to ring class

  // ************************* patch data ********************************

  // a pointer field to the static objects that should be rendered
  JWStaticObject **object;
  // Here we store how much space each object consumes at its position.
  // For new objects this is sqrt(2)*width/2, otherwise the remaining
  // overlap of an object at a neighbour vertex position.
  int *objLOD;

  int id;                 // surface id (cube side index [0,5])
  int LOD;                // LOD of this patch, size is (2^LOD)
  int xmid, ymid;         // 2D center position of this patch (deprecated)
  double hmin, hmax;      // minimal and maximal height in this patch

  CDoubleVector offset;   // special vertex offset for geometry with LOD <= 13
  bool isActive;          // Is patch initialization completed?
  // These flags store the LOD state of the 4 neighbour patches.
  // This is important to prevent cracks in the triangluation.
  // true = neighbour patch has the same LOD; false = neighbour's LOD is bigger
  bool no,nl,nu,nr;
```

```
// All following fields contain data for 9*9 vertices.
double *vertexData;   // vertex positions (x,y,z) in double space ([m]^3)
double *crackData;    // modified vertex positions to prevent cracks
GLfloat *normals;     // one normal vector for every vertex
GLfloat *textureCoords;   // 2D-coordinates in [0,1]x[0,1]
GLfloat *textureCoords2;  // 2D-coordinates in [0,1]x[0,1]
GLfloat *textureCoords3;  // 2D-coordinates in [0,1]x[0,1]
GLfloat *colorData;       // RGBA color value
double *heightField;      // height value for every vertex
CIntVector *surfacePos;   // cube position in range [-CUBE_HALF, CUBE_HALF]^3

// ******************** function declarations ***************************

/**
* Construction of a quadtree node with detail level pLOD.
* @param pId            id of this quadtree node
* @param pXmid, pYmid   2D position on this cube side (deprecated)
* @param pLOD           level of detail for this patch, defines size
* @param pJwp           pointer to parent object
*/
JWQuadtree(const int pId, int pXmid, int pYmid,
           const int pLOD, LODObject *pJwp);
virtual ~JWQuadtree(); // destructor

/**
* This method returns the height value for the given position.
* @param point   integer position of vertex
* @return        height at this position
*/
double getHeight_of_Terrain(const CIntVector point);

/**
* Initialize geometry for a top-level quadtree node. This function is
* only called for the quadtree nodes that cover a whole cube side.
* All other patches are created by subdivision of these nodes.
* @param base   center position of the quadtree for this cube side
* @param dx,dy  vectors for 2-dimensional surface orientation
*/
void initAsCubeSide(const CIntVector base,
                    const CIntVector dx, const CIntVector dy);
/**
* Core routine of the IU-LOD-algorithm. Here we subdivide the
* quadtree structure downto LODmin. The subdivision criteria is
* the L1-distance of each patch center to the camera position.
* This method is recursive and calls createChild and
* generateStaticObjects to create and initialize new patches.
* @param pCamera      Camera position in double space. This is needed
                      because we also compute if a patch is visible or not.
* @param pos          Camera position in integer space
* @param LODmin       lower bound for subdivision of geometry
*/
void update(C3DCamera *pCamera, const CIntVector pos, int LODmin);
```

```
  /**
  * Create all geometry data for the child[k]-patch.
  * @param k     create data for child[k]
  */
  void createChild(const int k);

  /**
  * Here we place static objects in the child[k]-patch.
  * @param k     create data for child[k]
  */
  void generateStaticObjects(const int k);

  /**
  * This method draws the whole terrain. Calls all child nodes recursivly.
  * @param pCamera  camera that defines our viewport in the virtual world
  */
  void draw(C3DCamera *pCamera);

  /**
  * This method draws the water surface. Calls all child nodes recursivly.
  * @param pCamera  camera that defines our viewport in the virtual world
  */
  void drawWater(C3DCamera *pCamera);

  /**
  * This methos draws the static objects. Calls all child nodes recursivly.
  * @param pCamera  camera that defines our viewport in the virtual world
  */
  void drawStaticObjects(C3DCamera *pCamera);

}; // end of JWQuadtree
```