Examensarbete
LITH-ITN-MT-EX--05/060--SE

Large planetary data visualization using ROAM 2.0

Anders Persson

2005-12-09

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköpings Universitet
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings Universitet
601 74 Norrköping

LITH-ITN-MT-EX--05/060--SE

Large planetary data visualization using ROAM 2.0

Examensarbete utfört i Medieteknik
vid Linköpings Tekniska Högskola, Campus Norrköping

Anders Persson

Handledare Staffan Klashed
Examinator Anders Ynnerman

Norrköping 2005-12-09

Avdelning, Institution
Division, Department

Institutionen för teknik och naturvetenskap

Department of Science and Technology

Datum
Date

2005-12-09

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English

☐ _____

Rapporttyp
Report category

Examensarbete
☐ B-uppsats
☐ C-uppsats
☒ D-uppsats

☐ _____

ISBN

ISRN **LITH-ITN-MT-EX--05/060--SE**

Serietitel och serienummer
Title of series, numbering

ISSN

URL för elektronisk version

Titel
Title

Large planetary data visualization using ROAM 2.0

Författare
Author

Anders Persson

Sammanfattning
Abstract

The problem of estimating an adequate level of detail for an object for a specific view is one of the important problems in computer 3d-graphics and is especially important in real-time applications. The well-known continuous level-of-detail technique, Real-time Optimally Adapting Meshes (ROAM), has been employed with success for almost 10 years but has at present, due to rapid development of graphics hardware, been found to be inadequate. Compared to many other level-of-detail techniques it cannot benefit from the higher triangle throughput available on graphics cards of today.
This thesis will describe the implementation of the new version of ROAM (informally known as ROAM 2.0) for the purpose of massive planetary data visualization. It will show how the problems of the old technique can be bridged to be able to adapt to newer graphics card while still benefiting from the advantages of ROAM. The resulting implementation that is presented here is specialized on spherical objects and handles both texture and geometry data of arbitrary large sizes in an efficient way.

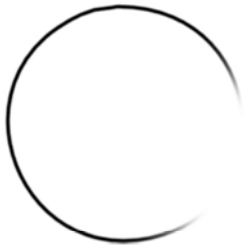
Nyckelord
Keyword

large dataset visualization, continous level-of-detail techniques, ROAM 2.0, real-time applications, terrain rendering, planetary data, adaptive textures

Abstract

The problem of estimating an adequate level of detail for an object for a specific view is one of the important problems in computer 3d-graphics and is especially important in real-time applications. The well-known *continuous level-of-detail* technique, *Real-time Optimally Adapting Meshes* (ROAM), has been employed with success for almost 10 years but has at present, due to rapid development of graphics hardware, been found to be inadequate. Compared to many other level-of-detail techniques it cannot benefit from the higher triangle throughput available on graphics cards of today.

This thesis will describe the implementation of the new version of ROAM (informally known as ROAM 2.0) for the purpose of massive planetary data visualization. It will show how the problems of the old technique can be bridged to be able to adapt to newer graphics card while still benefiting from the advantages of ROAM. The resulting implementation that is presented here is specialized on spherical objects and handles both texture and geometry data of arbitrary large sizes in an efficient way



Large planetary data visualization using ROAM 2.0

Anders Persson*

Master's Thesis in Media Technology
Norrköping, November 2005

Department of Science and Technology
Linköping Institute of Technology
Linköping University

* andpe538@student.liu.se

Acknowledgements

I would like to thank the Sciss-crew; Martin Rasmusson, Per Hemmingsson and Staffan Klashed, for supporting me throughout the whole project. Carl-Johan Rosén and Ulrika Anzén for proof-reading and valuable feedback. Special thanks to Mark A. Duchaineau who patiently answered all my questions about ROAM 2.0.

Table of Contents

1. Introduction 1
1.1. Purpose and Motivation 1
1.2. Problem Description 1
1.3. Objectives 2
1.4. Method 2
1.5. Thesis Layout 3
2. Background and Related Work 4
2.1. History 4
2.2. Reducing Geometry – The Level-of-Detail Problem 4
2.3. Reducing CPU Usage – DLOD vs. CLOD 4
2.4. ROAM 5
2.5. Reducing Occupation of Graphics Pipeline Bandwidth – Modern Graphics Hardware 5
2.6. ROAM 2.0 5
3. The World of ROAM Explained 7
3.1. Binary Triangle Hierarchies 7
3.1.1. Right-angled Triangles 7
3.1.2. Cracks 7
3.1.3. Forced Splitting 7
3.1.4. The Triangle Data Structure 8
3.2. Error Metrics 9
3.2.1. Culling 9
3.3. Frame-to-Frame Coherence and Priority Queues 9
3.4. ROAM 2.0 10
3.4.1. The Diamond Data Structure 11
3.4.2. Tiles and patches 11
3.4.3. Sierpinski Index 12
4. Implementation 13
4.1. Overview 13
4.2. The Base Data Structure 14
4.2.1. Diamonds 14
4.2.2. Priority Queues 15
4.2.3. Tiles and Patches 16
4.2.4. Memory Pools 16
4.2.5. Spherical Base Mesh 16
4.3. Culling and Priority Calculations 17

4.3.1. Culling ... 17
4.3.2. Priority Calculations ... 19
4.4. The Update Loop ... 19
4.5. Asynchronous Tile Loading ... 20
4.5.1. The Tile Loader Thread ... 20
4.5.2. Splitting and Merging Operation Overview ... 20
4.5.3. The Split Function ... 21
4.5.4. The Load Function ... 22
4.5.5. The Merge Function ... 22
4.5.6. The loadedTileApply Function ... 23
4.6. Rendering ... 23
4.6.1. Patch Data Layout ... 23
4.6.2. Drawing a Diamond ... 23
4.7. Pre-processing ... 24
4.7.1. Overview ... 24
4.7.2. Data Structure ... 24
4.7.3. Leaf Tile Creation ... 24
4.7.4. Coarser Low-pass Filtered Tiles ... 25
4.7.5. Disk Layout and IO Functionality ... 26
5. Result ... 28
5.1. Performance ... 28
5.2. Limitations ... 30
5.2.1. Hotspots ... 30
5.2.2. Decoupling of Texture and Geometry ... 31
6. Discussion ... 32
6.1. Other Applications ... 32
6.2. Conclusion ... 32
6.3. Future Work ... 33
6.3.1. Hotspots ... 33
6.3.2. Batched Patches ... 33
6.3.3. Decoupling of Texture and Geometry ... 33
6.3.4. Compressed Tile Loading ... 34
7. References ... 35

Table of Figures

Figure 1 The LOD triangle. 5
Figure 2 T-junction. 7
Figure 3 Forced splitting procedure. 8
Figure 4 Binary triangle tree mesh. 8
Figure 5 The merge and split queue. 9
Figure 6 The diamond - geometrical description. 11
Figure 7 System overview. 13
Figure 8 The diamond – relationships. 15
Figure 9 Spherical base mesh. 16
Figure 10 Diamond parameterization. 25
Figure 11 Texture and geometry low-pass filtering. 26
Figure 12 Performance test cases. 28
Figure 13 Application screenshots. 30
Figure 14 Application screenshot. 34

Equation 1 Vertex sphere projection formula. 16
Equation 2 LOD-level calculation – geometry. 24
Equation 3 LOD-level calculation – texture. 24

Table 1 Summary of discrete and continuous level-of-detail techniques. 4
Table 2 Spherical base mesh. 17
Table 3 Example: Culling flags. 18
Table 4 Example: Diamond Sierpinski index mapping. 27
Table 5 System performance analysis. 29
Table 6 Cache efficiency. 29

1. Introduction

This thesis is the outcome of the development carried out from January to June 2005 at Sciss AB Sweden. It serves as a fulfillment of a Master of Science degree in Media Technology at Linköping Institute of Technology.

1.1. Purpose and Motivation

The main purpose of this thesis has been to develop a visualization system for massive planetary data. The problem case is related to the selection of an appropriate *level-of-detail* (LOD) base technique and implementation for spherical 3d-objects.

Today, a great variation of LOD techniques and strategies exists. One of the bright stars on the *continuous level-of-detail* (CLOD) sky is the *Real-time Optimally Adapting Meshes* (ROAM) technique [2] which was published 1997. The technique benefits from its ability to generate an optimal mesh for a specific view with the least possible effort. ROAM has been shown to be especially efficient for terrain rendering.

However, lately the ROAM star has faded. It has turned out that modern graphics hardware do not suit this method very well [3]. With the high triangle count output available from today's graphics card some people has turned their trust to variants of the old *discrete level-of-detail* (DLOD) method [3]. Yet today, the ROAM technique are still used and developed and with a new variant of the old method being born (ROAM 2.0[1]), the forthcoming of the ROAM star seems bright.

ROAM 2.0 [9][10] was chosen to be most suited for the application. ROAM 2.0 is however not yet a dependable method - there is a need for further evaluation. Therefore this thesis will also work as an assessment of this new technique.

Planetary visualization is today an attractive field and much activity can be noted on the market. Highly detailed planetary data of our own planet is today made public and several public interfaces are also emerging, e.g. Google Earth [4].

1.2. Problem Description

The level-of-detail system was developed for Sciss AB and integrated into UniView[2]. The core of UniView is the ScaleGraph, which makes it possible to travel interactively and seamlessly between any of the extremely different scales of universe - from atoms to galaxies. The application includes a number of classes to render different astronomical objects, e.g. galaxies, stars and planets.

At the start of this project little was done for the purpose of exploring the surfaces of planets. There were some attempts with flat patch replacement when approaching the planet at low altitude. Clipmaps [5] were also employed for the texturing, but the results were not satisfactory.

---

[1] ROAM 2.0 denotes the base technique that was implemented in this thesis and is an extended version of the continuous level-of-detail technique called *Real-time Optimally Adapting Meshes* (ROAM) [2]. However, the technique is not officially named ROAM 2.0, but is only a subset of what will be called ROAM 2.0 in the future. The name is used here for convenience. The original technique from 1997 will be referred simply as ROAM.

[2] UniView is a navigable atlas of universe in which the level-of-detail system was implemented. UniView is a proprietary product developed by Sciss AB in Sweden [1].

The following problems were identified:

- **Geometry resolution.** When approaching close to the surface of a static planet mesh, the resolution of the mesh will appear extremely low, thus few triangles will occupy the screen area. This results in several visual problems. When looking towards the horizon, the expected curved form will appear rough with a few straight lines connecting each others. Specular effects can also suffer from the low polygon count available at the surface when using *per-vertex shading*.
- **Support for high resolution textures.** The old planet class used a single texture object for the whole planet which restricted the maximum texture resolution available.
- **Height map support.** The original application was lacking support for rendering surface height variations which are desirable for close-up surface visualization.

1.3. Objectives

From the problems that were identified in the previous section, the following general objectives were taken upon:

- Evaluate different level-of-detail techniques suitable for planetary visualization.
- Develop an appropriate level-of-detail system for planetary objects.
- Evaluate the final system.

The following requirements were established for the proposed level-of-detail system:

- Height data will be supported.
- High resolution textures will be supported.
- The system will conform to the performance demands of UniView.

These objectives together formed the mission of this project.

1.4. Method

The choice of technique for the implementation was determined relative late during the project.

The project started with a research phase where different LOD and tessellation techniques were compared. One of the more promising was Binary Triangle Trees (BTT) (Chapter 3.1). A simple test application was developed to explore the capabilities of this method. The disadvantage of the BTT method is that the mesh has to be rebuilt every frame even though the mesh changes little between frames. This so called *frame-to-frame coherence* was what opened the way to the world of ROAM. ROAM (Chapter 2.4) is a framework of the BTT structure and take advantages of the frame-to-frame coherence property.

It took a couple of month of continuous work, upgrading the application to ROAM, before realizing what others already had realized [3a]:

> *… a fixed-representation LOD system is a much better solution on modern graphics hardware because each representation can be rendered using precomputed, optimized, indexed triangle strips. A fixed representation system is likely to draw more geometry than strictly necessary, when compared to the approximation mesh a CLOD system might create. However, the rendering performance will be much better because the geometry can be sent to the graphics hardware in an optimized fashion.*

This means that ROAM, which is a CLOD technique (Chapter 2.3), badly adapt to modern graphics hardware. However, when exploring the ROAM field closer it became clear that concurrent development of ROAM was addressing precisely this weakness. Since the ROAM core was already implemented, the decision was made to commit to this new promising approach of the ROAM technique.

The final technique that was used for the implementation is almost completely based on [10]. It will be referred to as *ROAM 2.0* throughout the rest of this paper. ROAM and ROAM 2.0 share a set of base algorithms which will be described later. This base structure will be referred to as the *ROAM core.*

1.5. Thesis Layout

The rest of this thesis will be laid out as follows. First the concept of level-of-detail techniques will be explained and discussed (Chapter 2). The ROAM technique will also be explained briefly and put into the context. Chapter 3 will give the technical background of ROAM needed to understand chapter 4**,** which describes the implementation made. Finally the results will be presented in chapter 5 and discussed in chapter 6.

It is assumed that the reader has a basic knowledge of 3d computer graphics and programming.

2. Background and Related Work

This chapter will describe the concepts of LOD technology and briefly describe the ROAM technique which is the foundation of this work.

2.1. History

The level-of-detail problem (see below) is one of the important problems in 3d computer graphics and has been widely employed and explored during the last 30 years. Even though Clark introduced the LOD concept in 1976 [7], it was not until 20 years later that the important continuous and view-dependent variant was announced. The continuous LOD (CLOD) has been found very useful when working with large and complex datasets and is especially central in terrain rendering.

2.2. Reducing Geometry – The Level-of-Detail Problem

A real world scene is characterized by its infinite amount of details. Since a computer generated scene is bound to a finite number of details it is only an approximation of reality.

Today the fundamental bricks of a 3d-graphics scene are triangles. The more triangles used to describe an object the more detailed it gets, but the longer it takes to render. Perspective transformation cause triangles close to the virtual viewer to take up more screen area than triangles located far away from the viewer. Triangles that cover much screen area contribute to less detail in the image. This lead to the problem: how to dynamically limit details (number of triangles) far away from the viewer and maximize details close to the viewer. The problem is known as the LOD problem.

2.3. Reducing CPU Usage – DLOD vs. CLOD

Much work has been done on the LOD problem and there are two important types: Discrete level-of-detail (DLOD) and continuous level-of-detail (CLOD).

DLOD techniques use a number of static meshes with different resolutions which it can alternate between during rendering depending on the specific view. The main advantage of these methods is that the low-resolution meshes are calculated before application execution thus saving calculation power during interaction. DLOD requires a way to blend between the different mesh instances.

The CLOD methods do not use any pre-calculated meshes. Instead a data structure is used to modify the mesh to an appropriate heterogeneous resolution during runtime. This is done by continuously adding or removing triangles. This method better approximate the ideal mesh for a specific view but has the disadvantage of using more of computer power at runtime.

Table 1 Summary of discrete and continuous level-of-detail techniques.

DLOD		**CLOD**	
PRO	**CON**	**PRO**	**CON**
Low CPU usage	Blending/ transitions	Higher granularity	More demanding during rendering
Decouples simplification and rendering		Seamless transitions	

2.4. ROAM

ROAM [2] are a CLOD technique that is based on Binary Triangle Trees (BTT) [8]. BTT incorporates a recursive algorithm that can split or merge any triangle in the mesh without producing any cracks in the mesh. ROAM extends the BTT technique by taking advantage of frame-to-frame coherence. A mesh can therefore be modified optimally for a specific view with the least possible effort.

2.5. Reducing Occupation of Graphics Pipeline Bandwidth – Modern Graphics Hardware

Graphics hardware has become quite advanced even for the home PC market. Modern graphics cards can render more than 200 million triangles per second. For a screen resolution of 1280 x 1024, this gives 2 triangles per pixels at frame rates around 80 Hz. With this kind of triangle throughput it doesn't seem to be important to get a very good approximation of the ideal mesh the way CLOD and ROAM provide since we can output more triangle than can fit on the screen.

However, 200 million triangles per second is not a practical limit, it is only theoretical. In fact, the bottleneck on graphics cards is the busses which transmit triangle information to the graphics hardware. There are several ways to send this information, and it can be done in more or less efficient ways.

The standard way to present a mesh of triangles to the graphics hardware is to send three vertices for every triangle. If the mesh is connected, like a sphere, this way of sending the vertices is clearly redundant, since triangles share vertices. *Triangle strips* reduce redundancy of the geometrical data by only sending shared vertices once. *Triangle strips* require the triangles to be connected in a strip. Only (n+2)/n vertices is sent per triangle, where n is number of triangles in strip.

Other example of reducing graphics card buss traffic is *indexed vertex arrays***,** which will be described later (Chapter 4.6).

2.6. ROAM 2.0

To conclude, it seems to be three important factors to consider when building a LOD-system.

- Reducing geometry.
- Reducing CPU usage.
- Reducing occupation of graphics pipeline bandwidth.

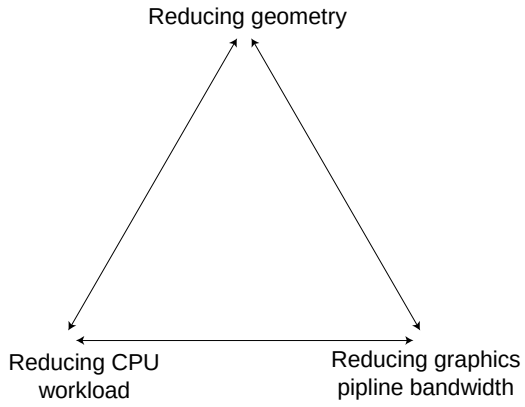


Figure 1 The LOD triangle.
Three important factors when building a LOD system for modern graphics hardware.

ROAM, described above, has long been a promising LOD-technique. But when used with modern graphics cards it seems to suffer from two of these factors: ROAM is really good at reducing geometry but it has difficulties to present the geometry to the graphics card with sufficient high entropy. There have been attempts to create triangle strips of the CLOD mesh but that turned out to overload the CPU so: "… the harder we try to meet the demands of modern hardware with CLOD methods, the worse the situation we find

ourselves in becomes." [3a] Thus, if we try to satisfy the third factor the system is going to suffer from the second factor.

Mark A. Duchaineau, founder and explorer of the ROAM technique, was aware of this problem when he started to work on a new version of ROAM. At this date ROAM 2.0 is not a complete system but several papers have been published that, according to Duchaineau, is a subset of ROAM 2.0 [9][10].

ROAM 2.0 is a hybrid of a CLOD- and a DLOD-system. By extending the original ROAM system to include patches, pre-calculated geometry at different resolutions, it can be possible to eliminate the negative effects of factor two and three above, and still taking advantage of the original technique. ROAM 2.0 also extends the level-of-detail technique to be used with texture data using nearly the same algorithms as for geometry.

3. The World of ROAM Explained

The implementation described in this thesis is based on the ROAM technique. This chapter aim to provide a technical understanding of ROAM [2].

3.1. Binary Triangle Hierarchies

All *continuous* LOD methods define a basic data structure that are used to describe an arbitrary tessellation of a 3d-object.

3.1.1. Right-angled Triangles

The backbone of ROAM is binary triangle trees (a.k.a. bintree triangles, hierarchies of right triangles or adaptive 4-8 meshes), which consists of right-angled triangles. The nice property of right-angled triangles is that they can be divided into two *uniform* triangles only intersecting one of the edges (the hypotenuse). By aligning two (right-angled) triangles that are sharing the hypotenuses, a quadric patch is formed. In this way it is possible to tessellate the patch into more and more triangles by dividing the triangles recursively as described above. Notice that the base mesh can be formed into many other arbitrary forms than quadric patches**.**

3.1.2. Cracks

Crack is one of the important problems when developing a CLOD method. Basically, a vertex (corner) must be shared between all neighbor triangles. In Figure 2 a crack in the BTT structure is displayed, called a T-junction. There are two reasons why you want to avoid T-junctions in a mesh. First, due to limited floating-point precision, gaps in the mesh can be visible. Second, for the purpose of height mapping there can obviously not be T-junctions in the mesh.

Every CLOD technique has its own strategy to deal with cracks – *crack fixing*. BTTs way of handling this is easy - it just do not allow it. In the case of Figure 2, triangle A must be split.

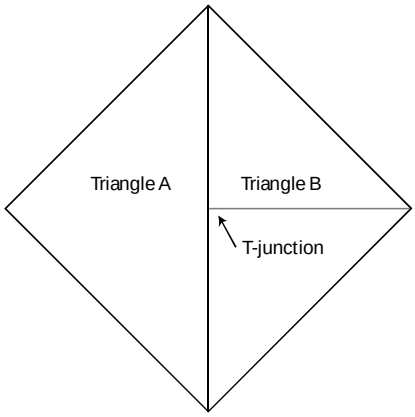


Figure 2 T-junction.
A crack in the BTT structure. The situation can be resolved by splitting triangle A.

3.1.3. Forced Splitting

Since we want to be able to split any arbitrary triangle in a BTT conformant mesh, there must be a way to ensure a crack-free mesh after the split. BTT incorporate a recursive algorithm that takes care of this: If a splitting results in a T-junction, the neighbor triangle must be split first and if that triangle results in a T-junction it has to be split before any of the other triangles, and so on (Figure 3**)**.

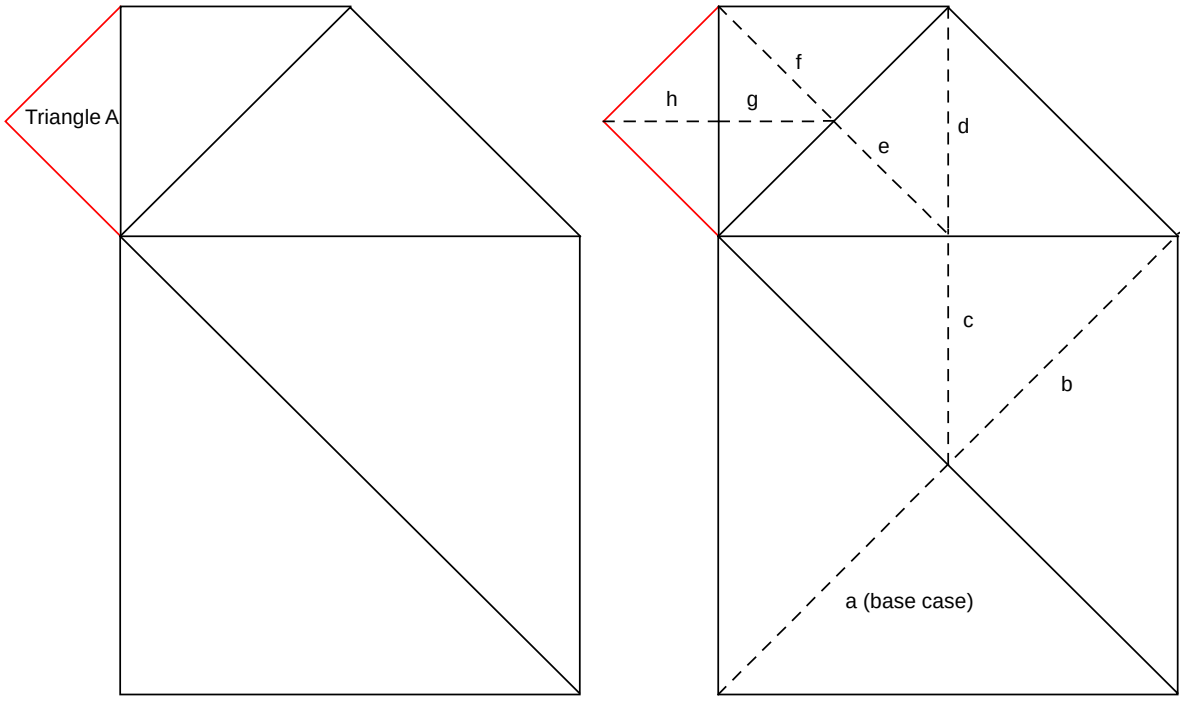


Figure 3 Forced splitting procedure.

This figure shows the recursive nature of the forced splitting procedure. The left figure depicture the initial mesh before triangle A is split. The right figure depictures the resulting split sequence. The recursive split calls will terminate at "a" since this triangle can be split without creating any cracks in the mesh – resulting in the split sequence {a, b, c…, h}.

3.1.4. The Triangle Data Structure

To be able to integrate these algorithms in a computer program the BTT algorithm specify a data structure for a *binary triangle*.

Every triangle has two children (left and right) and three neighbors (left, right and bottom). These relationships are represented by pointers which are used when performing the splitting. All triangles in the mesh then form a hierarchy or a tree of triangles. Additional to the pointers, every triangle has information about its three vertices and other payload data.

To sum up, the BTT technique works as a framework for creating a crack-free mesh of right-angled triangles of arbitrary heterogeneous resolution.

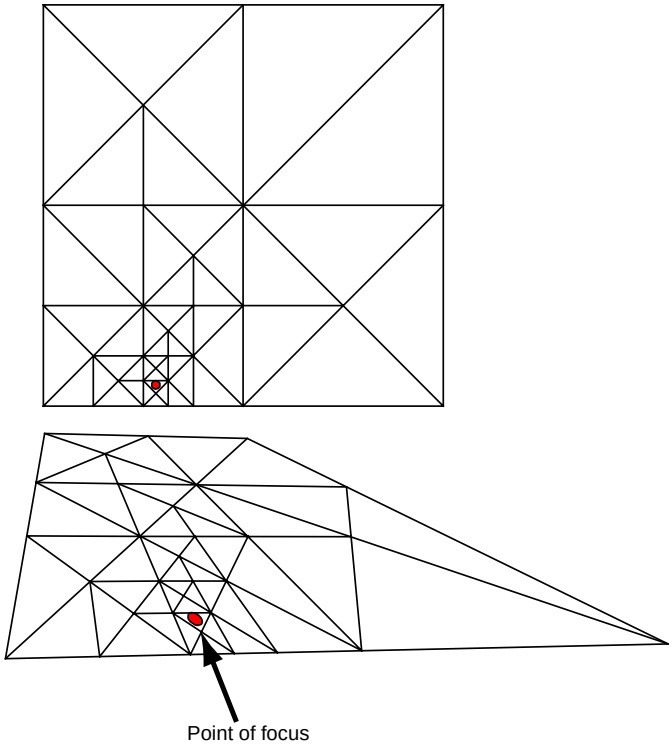


Figure 4 Binary triangle tree mesh.
Example of a BTT mesh of heterogeneous resolution seen from above (top) and with perspective projection (bottom).

3.2. Error Metrics

To be able to build a mesh that adapts to the current view, error metrics are used. Error metrics are used to approximate how much a decimated mesh will differ from the original mesh in the eye of the beholder. In ROAM, every triangle hold a value of how much it is contributing to the summed error of the mesh. This value or error will determine if the triangle should be split. Error metrics can be very easy, like the distance from the viewer to the triangle, or more complicated.

3.2.1. Culling

An important part of ROAM, which can be sorted under error metrics, is culling. Depending on the frustum (viewing volume) some objects or parts of objects will not be visible to the viewer. This fact is used when rendering is performed in 3d computer graphics. This fact is also used in ROAM – triangles that can be identified to be completely out of view or back faced should not be split even if standard error metrics would advice it. This saves memory and processing power when viewing close to the object. Also, since all descendant triangles are located inside the triangle of its parent, not all triangles have to be culled every frame.

3.3. Frame-to-Frame Coherence and Priority Queues

At runtime the BTT framework will build a view dependent mesh, according to some conditions, *every frame*. The conditions include view parameters and an algorithm termination conditions (i.e. maximum triangle count or maximum splitting time). At the start of every frame the algorithms will be provided with only a basic mesh, consisting of a few of triangles, and will then produce the desirable mesh by splitting triangles recursively. Only the leaf-triangles (childless triangles) will be drawn at the rendering phase (except those marked as out by the culling routine). Since this procedure will be executed at least 30 times every second and will involve several thousands of triangles, the splitting process will be quite demanding for the CPU. However, in most applications the viewing parameters will not vary so much between two nearby frames. Comparing the resulting mesh of two adjacent frames, the triangulation will only differ by a couple of percent. This is called *frame-to-frame coherence*, and is used efficiently in the ROAM method.

To be able to reuse the mesh produced in the previous frame, ROAM extends two priority queues and an algorithm for *merging* triangles. One of the queues keeps a sorted list of all the leaf-triangles according to their error metrics value (from now on called *priority*). This queue is called the *split queue* because it consists of triangles that can be subject for a split. The first triangle in the queue has the highest priority and will be the next triangle to split if further refinement of the mesh is in question. Similarly, the other queue (the *merge queue*) consists of a list of all potential mergeable triangles (triangles with childless children).

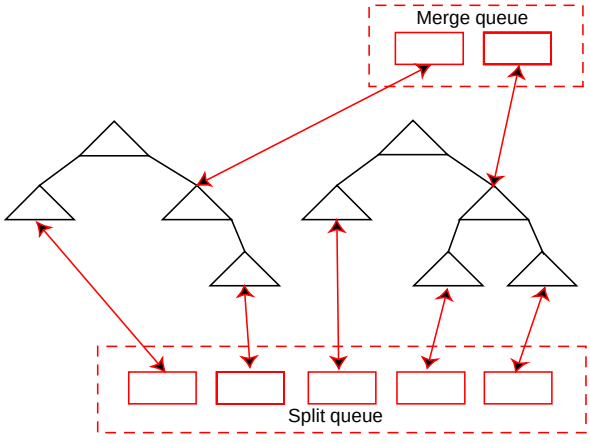


Figure 5 The merge and split queue.
Schematic figure of the linkage between the BTT structure and the two priority queues.

To drive the framewise modification of the mesh, an algorithmic loop is used (*the update loop*). At each iteration the highest split queue triangle will be split if the accuracy of the mesh is too low, else the triangle with the lowest priority in the merge queue will be merged. The priority queues will be updated during each iteration. The loop will terminate if the minimum merge priority is greater or equal than the maximum split priority or if desirable accuracy has been achieved. It is also possible to force termination of the loop if the update time budget has been overdrawn, which can be useful when static frame rates is essential. The way the update loop is designed will ensure an optimal mesh after every iteration lap and thus produce the best possible mesh with the resources available**.**

The merging procedure is quite similar to the splitting, and consists of a way to remove the two children of a triangle and appropriate actions for keeping the mesh crack-free.

The algorithms described in short here represent the ROAM technique (for details see [2]). Here follows a brief algorithmic resume:

The basic mesh is coded into the program, e.g. a patch of two triangles for a planar terrain. Each triangle has pointers to neighbors and children etc. The following takes place during each frame: First of all, the frustum is updated and the current triangulation goes through the culling routine which includes a top-down tree traversing using the child pointers. Triangles completely out of the frustum are marked as *out* in addition to the priority set to zero. The culling procedure does not need to cull the offspring branch of a triangle that has been culled as out, since all the children must clearly also be outside the frustum. Subsequently the priority of all the triangles in the split and merge queue are updated and the updated loop, described above, will drive the tessellation, until terminated. Rendering of the updated mesh is preformed by issue the drawing commands of every triangle in the split queue.

Height mapping can easily be supported with ROAM. All created vertices can be mapped to a height map and elevated accordingly.

3.4. ROAM 2.0

The ROAM technique has successfully been used in real-time applications. However lately, due to technical evolution, ROAM has shown some major weaknesses:

- **Huge datasets.** Data sizes keep up with technical development. Today many datasets are measured in Gigabytes instead of Megabytes. This scenario enforces LOD-engineers to change the architecture of their systems since the data can no longer be kept in main memory. Strategies must be invented to page data dynamically from disk.
- **Graphics hardware.** Since graphics hardware now is able to display more triangles than “can fit on screen”, it is no longer a priority to produce a perfect optimal mesh. Instead priority has fallen upon geometry layout to reduce the load on the graphics pipeline which has been identified as the bottleneck on modern graphics card [3a].

Both of these weaknesses have been addressed in the development of ROAM 2.0. The solution is to integrate the concept of pre-calculated data chunks, à la discreet LOD techniques, into the ROAM framework. In this way the nice features of ROAM remains at the same time as the weaknesses described above can be tackled.

The following definitions are central in ROAM 2.0:

- **Diamonds** are a data structure representing two BTT triangles at the same level sharing the hypotenuse.
- **Patches** are vertex rasters that represents the geometry of a BTT triangle. A diamond is associated with two patches.
- **Tiles** are the pre-calculated data associated with a diamond and are stored on disk before it can be transformed to patch or texture data.

3.4.1. The Diamond Data Structure

Before introducing the new *tile* concept of ROAM 2.0, it is appropriate to explain the diamond data structure, which lay as the foundation of version two.

The diamond data structure (DDS) is an attempt to improve the old triangle data structure. One of the advantages of DDS is that it more closely manage to describe the true nature of the BTT structure and therefore becomes more efficient. It also suites the concept of tiles better. It is important to observe that the difference between these data structures is solely a matter of data description. The functionality of the old binary triangle tree structure is identical.

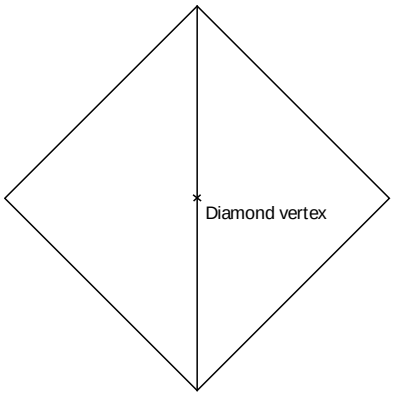


Figure 6 The diamond - geometrical description. A diamond consists of two right-angled triangles at the same level, sharing the hypotenuse. It holds information about the mid-vertex depicted, but its own form is defined from the mid-vertices of its parents and ancestors.

A diamond is defined as two (right-angled) triangles at the same level, sharing the hypotenuse (Figure 6 and Figure 8). Every diamond has a unique one-to-one association with one vertex in the mesh (the mid-vertex), and therefore every vertex is only described once in the DDS (compare with the old triangle structure). The DDS also improves algorithmic efficiency since the two triangles defining a diamond are closely related – one of the triangles can not be independently split without splitting the other ensuring a crack-free mesh.

The price for this data streamlining is a more complex system. Every diamond has now four children, two parents and two ancestors. No pointers are kept to neighbors but they can be located through the other relations. Algorithms are provided to do splitting and merging of a diamond.

The form of each diamond is defined by the mid-vertices of its two parents and two ancestors.

3.4.2. Tiles and patches

With this new data structure it is now possible to replace every diamond in the mesh with a pre-calculated height field and a texture of the right level of detail, called *tiles*. The tiles are stored on disk on can be loaded dynamically during runtime. Geometry tile data is converted to vertex rasters, called *patches*, which represent the BTT triangles of the mesh.

The advantages of replacing the fine-grained data structure with tiles is that large parts of height fields and textures can be read from disk in one go. Furthermore, the tiles can be

presented to the graphics card in an optimal manner. Even if the mesh adaptation will decrease, it will not affect performance since the optimal use of graphics hardware will ensure a much higher triangle count.

Every tile will have the same resolution. This results in a difference of factor two in geometry and texture resolution between two adjacent levels in the mesh. If compared to, for example the quad-tree data structure, usual in DLOD techniques, twice as many levels of details will be achieved.

In order to get the tile replacement working correctly, several mechanisms must be added to the ROAM system:

- **Data creation.** The original geometry and texture data must be low-pass filtered in order to cover all the different levels of details. An out-of-core pre-processor must therefore be built.
- **Efficient disk paging.** All the different tiles created must be stored and labeled in an efficient way. The tiles are labeled in a way to maximize spatial coherence and several caches are used to take advantage of this.
- **Mapping routines.** The orientation of every diamond must be explicit so the system knows how to apply the tile data.
- **Crack-fixing.** Geometry tiles must be given special treatment during the pre-processing to ensure a crack-free mesh around tile borders.

3.4.3. Sierpinski Index

The Sierpinski space-filling curve [10a] has a central role in ROAM 2.0 and has two important applications:

- **Diamond index.** Every diamond needs a unique index that provides a way to access the corresponding tile data. The Sierpinski curve notion offers a way to calculate unique diamond indices with spatial coherence. This spatial property will be shown to be very valuable.
- **Patch data layout.** The space filling curve has the ability to create a continuous curve that completely fills an area. This is used for the patch vertex layout and provides the spatial coherence between vertices essential for the use of indexed vertex arrays (Chapter 4.6.1).

4. Implementation

This chapter will describe the implementation phase of this thesis. The intension of this part is not to provide a recipe for creating a ROAM 2.0 application. For detailed references about the ROAM 2.0, please use [10]. However, all the different steps in the implementation will be described briefly, and on parts were the implementation has diverted from the original paper details will be given. Additionally, more details will be given on parts were the original paper were found too vague.

C++ and O*penGL* were used for the implementation.

4.1. Overview

The level-of-detail system that was implemented was developed within the UniView core. A basic platform where therefore given from start and the basic functionality of a visualization system where already available (i.e. retrieving viewing parameters). In addition to the LOD system a stand-alone pre-processing application, for the tile creation, was also implemented.

Figure 7 shows the basic structure of the final system. The arrows indicate the flow through the system.

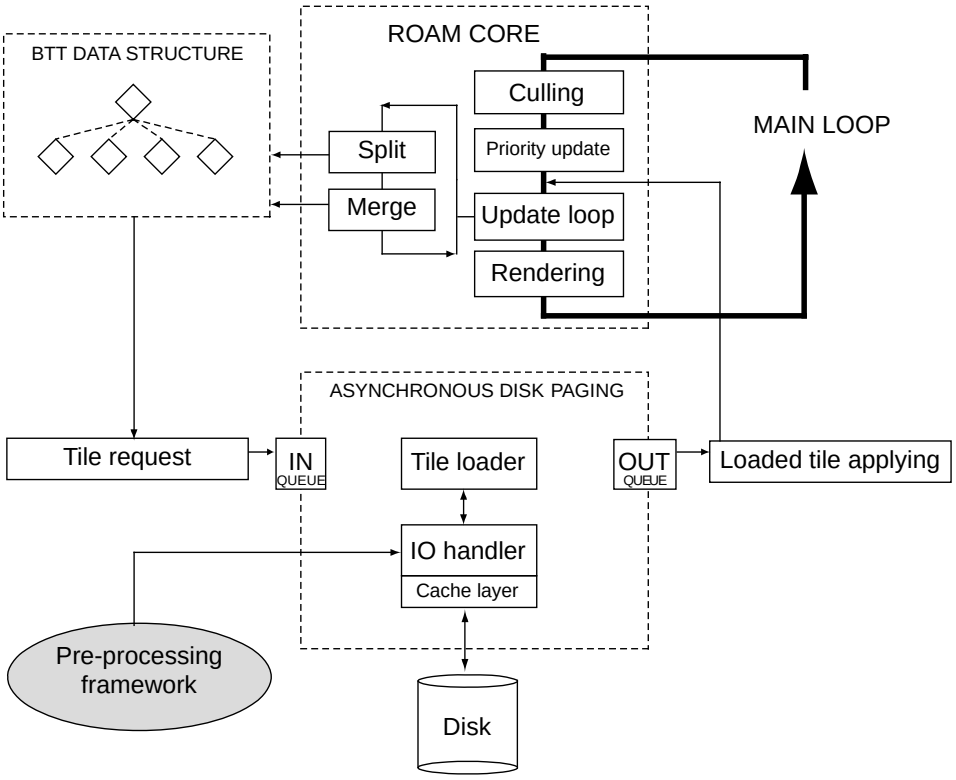


Figure 7 System overview.
The main application consists of two parts. The main thread handles the ROAM core functionality which drives the modification of the BTT structure. Data correlating to newly created diamonds are requested and loaded by the asynchronous tile loader. Communication between these different contexts is handled through two I/O queues. A separate application creates pre-processed data used by the main application.

The main difference from the original paper is the choice of using asynchronous tile loading (Chapter 4.5) instead of utilize the graphics card extension, *vertex array range*. This extension enables storage of vertex information directly onto the graphics card and can reduce both pipeline bandwidth and CPU extensively. However, this is not a standard feature on graphics cards, which is why another approach was taken.

The implementation description will start with explaining the new improved ROAM core, then the asynchronous tile loading mechanism, followed by rendering routines. Finally, the standalone pre-processing system will be described.

4.2. The Base Data Structure

The core algorithms in ROAM 2.0 do not differ much from the original ROAM core. The main change is the new diamond data structure and a new split-and-merge routine for the asynchronous tile loading.

4.2.1. Diamonds

Below is a list of the important structures of a diamond.

- **Center vertex.** Every diamond has a unique association with one vertex, the mid-vertex of the diamond. Vertex information includes normal direction and global texture coordinates.
- **Pointers to relative diamonds.** Every diamond has three arrays of pointers to children (4), parents (2) and ancestors (2). The first ancestor is referred as the *quad-tree ancestor* since the diamond is one of the four quadrants of this ancestor (Figure 8).
- **Pointer to priority queue.** Only one pointer is needed for the priority queues, since a diamond can only be present in one queue (split or merge queue) at the same time.
- **Pointers to patch data.**
- **Texture identification.** An *unsigned int* is used to identify texture tile data in texture memory.
- **Unique diamond identification.** Every diamond has a unique index, primarily used to identify the matching tile data. The index is based on Sierpinski indexing which increases spatial coherence (Chapter 3.4.3).
- **Bounding sphere radius and diamond world coordinate area.** Used for culling and priority calculations.
- **State and culling flags.** Denotes different culling and tile loading states (Chapter 4.3.1 and 4.3.2).

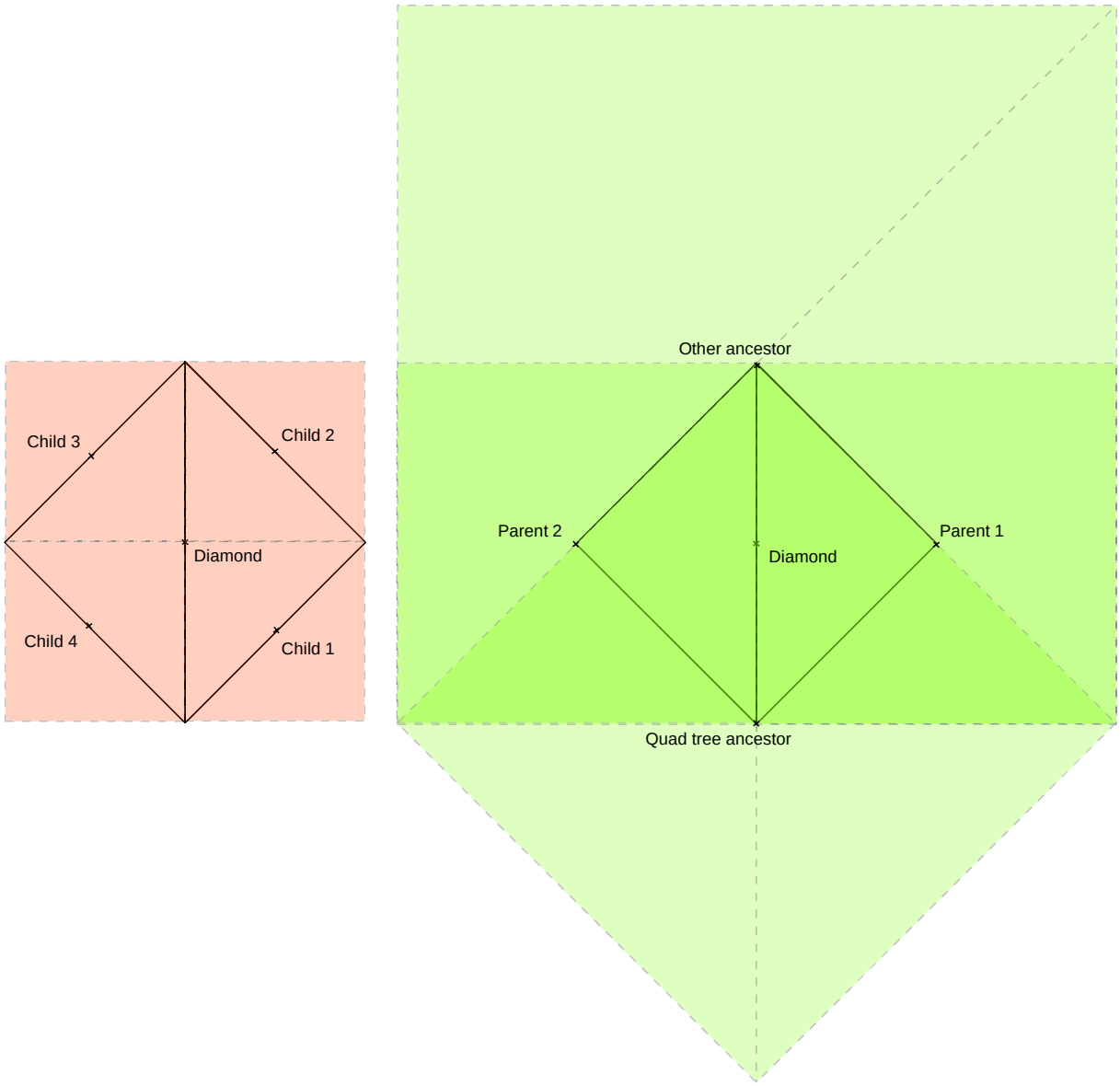


Figure 8 The diamond – relationships.
A diamond has two parents, two ancestors and up to four children.

4.2.2. Priority Queues

There are two priority queues – one split queue and one merge queue – that keep records of which diamonds in the data structure that can be split or merged. The queues can be implemented quite freely, but should include the following features for efficiency:

- Efficient search or sort function (for priority).
- Efficient add and remove item functionality.

The queues were implemented as unsorted double-linked lists, which makes the add-and-remove functions simple and efficient, but enforces the entire list to be searched in order to find the highest or lowest priority item. However, since this is only performed a few times per frame, this arrangement where found adequate. The queue items consist of a pointer to the diamond and a priority value. Notice that the connection between the diamond and the corresponding queue item is double linked.

4.2.3. Tiles and Patches

Every leaf diamond is associated with geometry and texture tile data which are stored on disk. At runtime, an OpenGL texture index identifies the corresponding texture, which was uploaded to texture memory at diamond creation. For geometry, two pointers refer to the correlating two vertex rasters, called *patches*. The patch data is created and initialized at diamond creation. The structure consists of vertex, normal and local texture coordinate information, and is laid out in a way to increase render performance (Chapter 4.6.1**)**. Only leaf diamonds hold patch and texture data.

4.2.4. Memory Pools

Memory pools are widely used in the implementation. Every time when a data structure (i.e. a diamond or a queue item) is created, memory must be dynamically allocated. Since this is a heavy and frequently used operation, memory pools can be used to allocate bigger chunks of data and then ration out small memory chunks when needed. Memory pools are used for diamonds, queue items and geometry patches.

4.2.5. Spherical Base Mesh

As described in chapter 3, the ROAM algorithms need a base structure to work with. For standard terrain visualization this is often a quadric patch consisting of one diamond (triangle-pair). For spherical shapes a cube can be used.

In order to resemble a planet all new vertices that are created during the split procedure must be projected onto the ideal sphere. This is done by projecting new vertices to sphere radius length:

$$(x, y, z) = r * \frac{(x', y', z')}{\|x', y', z'\|_2}$$

Equation 1 Vertex sphere projection formula.
"r" denotes the radius of the ideal sphere.

To form a cube with the diamond data structure 26 diamonds are used. 8 diamonds denote the corners of the cube which vertices lies on the surface of the ideal sphere (*vertex diamonds*). 6 represents the sides of the cube with its vertices at the center each side (*face diamonds*) and the remaining 12 diamonds indicate the edges of the cube (*edge diamonds*). In reality this is not a cube since all the vertices (except the corners) will be projected onto the ideal sphere.

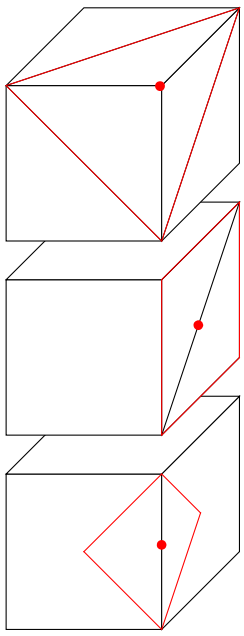


Figure 9 Spherical base mesh.
The spherical base mesh is formed as a cube before projection and consists of eight *vertex diamonds* (top), six *face diamonds* (middle), and twelve *edge diamonds*. Observe that the *edge diamonds* only have three corners.

The vertex diamonds determines the size and orientation of the cube. Edge and face vertex position is computed as the midpoint between the corresponding two vertex diamonds and then projected onto the sphere. Vertex diamonds has no links to its children nor to parents and ancestors (*NULL* pointers are appropriate). Also the face diamonds have no explicit parents or ancestors, but have the edge diamonds as children. The edge diamonds has no

children (yet), but have the corresponding face diamonds as parents and vertex diamonds as ancestors. Table 2 and the corresponding Figure 9 shows how the base mesh was structured.

The i-values in the table are index values that denote its own child index for both parents respectively. These indices are used to efficiently find the different neighbors of a diamond. In this implementation the edge diamonds are defined as the base level-of-detail (level zero), resulting in face diamonds at level minus one and vertex diamonds at level minus two.

Before the update procedure starts, only the edge diamond should be present in the split queue - the merge queue should be empty since the face diamonds should never merge (the base mesh is the coarsest possible mesh).

Table 2 Spherical base mesh.
This table describes the base mesh structure used in current implementation. The first eight columns point out the relationships of every base diamond. *Position* describes diamond vertex position. The two *i* columns denote its own *child index* for both parents respectively and are stored explicitly in the data structure for efficiency. The last column indicates the Sierpinski index of the base diamonds. *v*, *f*, and e should be interpreted as vertex, face and edge diamond in the ID column.

ID	a[0]	a[1]	p[0]	p[1]	c[0]	c[1]	c[2]	c[3]	position	i[0]	i[1]	Sierpinski index
v0	-	-	-	-	-	-	-	-	(-ct,ct,ct)	-	-	-
v1	-	-	-	-	-	-	-	-	(-ct,ct,-ct)	-	-	-
v2	-	-	-	-	-	-	-	-	(-ct,-ct,-ct)	-	-	-
v3	-	-	-	-	-	-	-	-	(-ct,-ct,ct)	-	-	-
v4	-	-	-	-	-	-	-	-	(ct,-ct,ct)	-	-	-
v5	-	-	-	-	-	-	-	-	(ct,-ct,-ct)	-	-	-
v6	-	-	-	-	-	-	-	-	(ct,ct,-ct)	-	-	-
v7	-	-	-	-	-	-	-	-	(ct,ct,ct)	-	-	-
f0	-	-	-	-	e0	e1	e4	e3	0.5*v0+v2	-	-	-
f1	-	-	-	-	e5	e4	e6	e7	0.5*v2+v4	-	-	-
f2	-	-	-	-	e3	e5	e11	e9	0.5*v0+v4	-	-	-
f3	-	-	-	-	e10	e11	e7	e8	0.5*v4+v6	-	-	-
f4	-	-	-	-	e8	e6	e1	e2	0.5*v2+v6	-	-	-
f5	-	-	-	-	e9	e10	e2	e0	0.5*v0+v6	-	-	-
e0	v1	v0	f0	f5	-	-	-	-	0.5*f0+f1	0	3	16
e1	v1	v2	f4	f0	-	-	-	-	0.5*f1+f2	2	1	17
e2	v1	v6	f5	f4	-	-	-	-	0.5*f1+f6	2	3	18
e3	v3	v0	f2	f0	-	-	-	-	0.5*f0+f3	0	3	19
e4	v3	v2	f0	f1	-	-	-	-	0.5*f2+f3	2	1	20
e5	v3	v4	f1	f2	-	-	-	-	0.5*f3+f4	0	1	21
e6	v5	v2	f1	f4	-	-	-	-	0.5*f2+f5	2	1	22
e7	v5	v4	f3	f1	-	-	-	-	0.5*f4+f5	2	3	23
e8	v5	v6	f4	f3	-	-	-	-	0.5*f5+f6	0	3	24
e9	v7	v0	f5	f2	-	-	-	-	0.5*f0+f7	0	3	25
e10	v7	v6	f3	f5	-	-	-	-	0.5*f6+f7	0	1	26
e11	v7	v4	f2	f3	-	-	-	-	0.5*f4+f7	2	1	27

4.3. Culling and Priority Calculations

As depicted in Figure 7, the main loop starts with culling routines and priority update. UniView provides updated frustum information (frustum normals, camera position and direction).

4.3.1. Culling

The *child diamond* does not inherit its parents culling property as in the case with the BTT structure. However, all descendant diamonds *every second level*, inherit the culling property in view of the fact that they all reside inside the bounding volume of the *ancestor diamond*. This is used in the culling routine and can be called *branch exclusion*. The branch exclusion feature enables the application to terminate a recursive culling thread if a diamond is clearly

in or out of the view frustum, why the culling routine only have to cull to a level where it can decide if a diamond is *clearly* in or out.

This results in a very greedy culling routine. Only when the culling state of a diamond is changed, the application must continue the recursion, marking all the successor diamonds with the correct culling label (see pseudo code below).

A sphere is employed as the bounding volume for the diamonds instead of wedgies that was proposed in the original ROAM version. This reduces culling to one test per frustum plane and diamond. The culling routine also tests for back-faced diamonds. If the standard culling routine cannot determine if a diamond is in or out, all the normals of the four diamond corners will be compared with (dot product calculation) the *camera-to-corner* direction to determine if all diamond vertices are turned away from the camera. If that is the case, the diamond can be marked as out, together with all its descendants. The same reasoning is valid here: all children diamonds must have normals that are equal or lies between the values of the ancestor diamond.

An unsigned short is used as a bit pattern to keep track of the current culling status of the diamonds (Table 3). Every bit represents a frustum plane test or back-face corner culling test which is used internally in the culling routine. Observe that in this implementation near and far clipping planes are not used in the culling routine. Besides the culling routine, the culling labels are also used in priority calculation and rendering.

As mention earlier, the culling inheriting goes between the quad-tree ancestors and the diamond which practically means that only every *even* LOD-level will be culled since the culling recursion starts with the *edge diamonds*. This enforces additional routines in priority calculations and rendering to determine the culling status of a diamond.

Table 3 Example: Culling flags.
Three examples of the cull flag structure.

Interpretation	**Bit pattern (the least significant bits of an unsigned short)**								
	x	x	x	x	x	x	x	x	LSB
Diamond is culled as completely out of view frustum or totally back-faced (**OUT**).	1	0	0	0	0	0	0	0	0
Diamond is culled as completely in view frustum and totally front faced (**IN**).	0	1	1	1	1	1	1	1	1
Diamond partly in view frustum; children should be tested against frustum plane one and four.	0	0	0	0	0	0	1	1	0
		Front cull flags for diamond vertices				**IN flags for frustum planes**			

Below follows pseudo code for the culling routine:

```
For every edge diamond do cullingRecursion().

cullingRecursion()
{
      culldiamond()

      if the old cull flag was marked as IN or OUT and the result is the same after
      cullDiamond() terminate recursion,

      else cull all possible quad-tree children (the grandchildren that have the current
      diamond as quad-tree ancestor) by repeating cullingRecursion() for those diamonds.
}

cullDiamond()
{
      if the quad-tree ancestor is OUT or IN set diamond label respectively and exit.
      else
      {
            ancestor cull flags are inherited.

            test all not IN marked frustum planes against diamond bounding sphere and, add
            IN marked flags if IN or set whole culling label to OUT if one test fail.

            if culling label is still not IN or OUT do back-face culling and set back-face
            flags accordingly. If the diamond can be identified as totally back-faced set
            culling label to OUT.
      }
}
```

4.3.2. Priority Calculations

The *priority update* is performed after the culling routine and calculates new priorities for all diamonds in the split and the merge queue.

First culling status are checked – if culling label is set to OUT or both parents are culled as OUT, priority is set to standard zero priority. Otherwise, the screen space area of the diamond is employed as error metrics for the priority calculations. This is standard calculations where field-of-view and screen resolution must be regarded.

The priority values are scaled so that 1.0 represents the desired diamond projection size, which are a user defined parameter. Higher values indicate a too large projection size and that splitting is appropriate. Diamonds with lower priority values than one can be merged.

4.4. The Update Loop

There are variants of the update loop, which drives the mesh refinement. Two problems can be enlightened that are important during the design work:

- **Deadlocks.** Some designs can lead to deadlocks. This means that the loop will never terminate and the application will crash.
- **Oscillation effects.** If the update loop is implemented badly, sometimes the mesh refinement work will never stabilize even when the frustum is unchanging. This results in visual flickering.

After trying several approaches for the update loop designed, a quite different one was found best suited for the needs. This implementation does not provide an optimized mesh during each iteration step; neither does it include any triangle count limit. Nevertheless it works well with the rest of the implementation.

Below follows pseudo code for the update loop:

```
Repeat until the maximum priority of the split queue is less or equal than 1.0 or
minimum merge priority is greater or equal to the maximum split priority (maximum number
of iteration loops is also specified):
{
      if the maximum priority of the split queue is greater than 1.0
      {
            set priority of the split diamond to a minimum value (use for the special
            asynchronous tile loading mechanism explained below)
            split(diamond with maximum priority in split queue)
      }
      else
      {
            set priority of the merge diamond to a maximum value (use for the special
            asynchronous tile loading mechanism explained below)
            merge(diamond with minimum priority in merge queue)

      }
      find new maximum and minimum priority diamonds in split and merge queue
}
```

The specially designed split and merge functions for the asynchronous tile loading functionality will be described below.

4.5. Asynchronous Tile Loading

The asynchronous tile loading approach was chosen to be able to keep smooth frame rates concurrent with extensive disk loading routines. The idea was taken from [3b]. The tile loading mechanism includes a threaded tile loader with in and out job queues and special split and merge functionality.

The main assumption is that all needed tile data (due to splits and merges in the update loop) cannot be page from disk during the same frame. This implies that split and merge routines must be extended beyond the boundaries of a frame. The following sections will deal with the components that have to be built or extended to accomplish this.

4.5.1. The Tile Loader Thread

The tile loader is the main component in the asynchronous tile loading framework and is running in an own context. The thread utilizes an IO-handler (Chapter 4.7.5) for tile loading and an input and an output queue for communication with the main thread.

The functionality is straight forward. The tile loader continuously checks and deals with tile requests from the input queue (a double linked FIFO queue). A request consists of a pointer and a Sierpinski index for the current diamond plus diamond corner information. The correct texture and height field data is loaded from disk. Height field data is transformed to vertex data and laid out in an efficient way (Chapter 4.6.1). Pointers to the loaded texture and the created geometry patches are added to the output queue together with diamond identification.

Mutex objects for the queues are used to ensure critical session.

4.5.2. Splitting and Merging Operation Overview

For the new splitting and merging mechanism a special state flag is used. Every diamond can be in one or, in some cases, two of the following states (the diamond life cycle):

- **LOADING** denotes a diamond that has requested tile data due to the split procedure of one of its parents. Diamonds in this states have been created and linked into the tree structure but does not formally exists, because it has not been added to the split queue yet.
- **LOADED** marks diamonds that have retrieved the requested tile data and are ready to be formally added to the BTT tree.

- **CHILDREN LOADING** indicates a diamond which splitting process has started. Tile data for its children have been requested and children are marked either as LOADING or LOADED.
- **SPLIT**. Diamonds with this label are formally split and will have children.
- **SPLIT + (LOADING or LOADED)** exist during the merge procedure of a diamond. The diamond is split and is either waiting for the requested data or have it.
- **DELETED** marks removed diamonds and are used primary for clarity.

The **split** routine will run the **load** function to create children diamond and request tile data through the **tileRequest** function. The **merge** function which uses the **tileRequest** function to request data can also trigger the **abortLoading** function to cancel earlier issued load request of diamonds being removed due to a merge. The **loadedTileApply** function is executed before the update loop to add possible loaded tile data to corresponding diamonds.

When enforcing the split and merge operations to be performed during several frames a new concept has to be introduced: t*rue* and *false splits/merges*. A *false* split/merge means that the operation has started but not finished - formally nothing has happen with the BTT structure. A t*rue* split/merge denotes that the operation was completed and diamonds have been added or been removed from the BTT tree.

4.5.3. The Split Function

This function is triggered in the update loop and returns a boolean that indicates if a *true split* was performed. In this chapter *the diamond* will refer to the diamond that is up for the split. The function will return *true* immediately if the diamond already has its status set to SPLIT.

Both parents of the diamond must first be split through two recursive call to the function itself, to enforce a crack-free mesh. If both function calls return true, e.g. the parents was already split or *true splits* where performed, splitting procedure can continue, otherwise it will have to wait another frame for the parents to split resulting in a false split.

The next step is to allocate all the children. This is performed by calling the **load** function for all four children. Then the status of all the children is checked. All children must present the LOADED flags for the function to continue. Otherwise the function will set the status of the diamond to CHILDREN LOADING and return false.

When all children have been loaded and the above described test succeeds the function will perform the *true split* of the diamond. The *true split* involves three operations - possible removal of parents from the merge queue, moving the diamond from the split queue to the merge queue and adding the children to the split queue. Culling status and priority calculations will also be performed for the new children. Finally, the tile data for the diamond will be deleted, since it will no longer be used in the rendering step and the function will return true to indicate a *true split*.

Since *false splits* do not remove the diamond’s top priority position from the split queue, the update loop will choose the same diamond for splitting over and over, during the entire update cycle. To overcome this, the priority of the diamond is forced to a minimum (or maximum for the merge queue) before splitting (or merging) as described in the update loop chapter (Chapter 4.4). The nice outcome is that these forced priorities resets every frame (due to the priority update) resulting in new attempts to perform true splitting or merging of these diamonds.

4.5.4. The Load Function

This function is called from the split function and allocates and sets up the needed data structures for a child diamond. It will also issue the tile data request.

Since we only want to add a child once a simple test will terminate this function immediately if the child already exists. Notice that children already can be present from the other split parent.

The child allocation includes:

- Setting up all new links to and from parents and ancestors.
- Setting default values for flags etc.
- Calculating new projected diamond vertex, bounding sphere radius and Sierpinski index.
- Requesting tile data through the **tileRequest** function.

The status of the child will be LOADING when leaving the function.

The **tileRequest** function simply creates a suitable tile request item and put it at the back of the input queue of the tile loader.

4.5.5. The Merge Function

For a diamond to merge it must first have the correct tile data (which was deleted during an earlier split procedure), why a similar approach as for the split operation must be taken. In this section *the diamond* will refer to the diamond that is up for the merge.

The function will terminate immediately if the diamond is already merged (negative SPLIT flag test). In the merge routine the status can show both the SPLIT and the LOADED (or LOADING) flag simultaneously, denoting that the diamond is not merged yet but tile data is loading or has been loaded. If the status of the diamond does **not** indicate that the tile data has been loaded (SPLIT + LOADED) or already requested (SPLIT + LOADING), it will make a request for the data. The LOADING status will also be added to the status so no further data request will be made. The function will then terminate.

If, on the other hand, the not merged diamond indicate that the tile data has been loaded the diamond can be truly merged. Here a problem arises: What if the children of the diamond are in the CHILDREN LOADING state (e.g. grandchildren are on the way)? In order to enforce that no tile requests are left behind in the tile request queue, the children must be forced to stop loading data before merging the diamond. This is done by calling the **abortLoading** function for every child of the diamond (see below).

The diamond merge includes removing children from the split queue and update linking. Children diamonds and the corresponding tile data are removed and status is set to DELETED. Furthermore, the merged diamond is moved from the merge queue to the split queue, status is set to LOADED (indicating that the diamond is merged) and priority is updated.

The **abortLoading** function will terminate the loading of grandchildren, which also can include the possible child-loading sons-in-law (other parents to the grandchildren) which also have to stop loading (status reset). All already partially loaded grandchildren must be removed, with one exception: Grandchildren with other parents of status SPLIT. The function also updates all linking and cleans the tile loader queues of canceled jobs.

4.5.6. The loadedTileApply Function

To finish the description of the asynchronous tile loading mechanism, the **loadedTileApply** function will be described briefly. This function will be executed every frame before the *update loop*. It will deal with all newly loaded tile data in the output queue of the tile loader. If the loaded data correspond to a diamond with status DELETED the tile data will be deleted. This should not happen but can prevent memory leaks during debugging. Otherwise the patch data pointers of the current diamond will be redirected to the location of the newly loaded tile data. Textures will be uploaded to texture memory and given texture name will be stored in the diamond data structure. The function will change the status of the diamond to LOADED.

4.6. Rendering

Rendering is the least complex part in the implementation. Basically, all diamonds in the split queue will be rendered unless they are culled out.

4.6.1. Patch Data Layout

As been point out, geometry layout is crucial for rendering performance. For this implementation *indexed vertex arrays* are used. *Vertex arrays* enable a chunk of vertices to be sent to the graphics hardware using a single draw command. *Indexed* vertex arrays rationalize the layout even further by only sending the vertex data once and then sending indices to indicate which vertices should be drawn the rest of the times. Since the layout of the patch mesh was based on Sierpinski space filling curves (Chapter 3.4.3), *indexed vertex arrays* are efficiently employed.

The vertex transformation is a recursive routine that divides the four corners of the diamond into two homogeneous sphere projected triangular meshes of desired resolution (256 triangles per patch have been used which results in a 17 x 17 vertex raster per diamond). This data structure is called a patch and every diamond has two patches associated with it. The transformation algorithm uses the Sierpinski space filling curve as base to maximize spatial coherence between the vertices. Height field data is added in the projection step.

The patch vertex indices of all diamonds are identically, why two arrays, one for each type of patch, can be hard-coded into the application.

4.6.2. Drawing a Diamond

First culling status of the diamond is checked – if culling label is set to OUT the function will terminate because the diamond will not be visible. Otherwise the culling label of the two parents to the diamond is evaluated. If an OUT label is found, the patch on the side of that parent will not be drawn. This procedure is necessary because the culling routine only culls every second level.

The drawing of a diamond patch involves providing indices and pointers for the patch data to the graphics hardware. OpenGL commands used are `glVertexPointer, glTexCoordPointer, glNormalPointer and glDrawElements.` The corresponding texture data for the diamond is *binded* prior to the drawing.

4.7. Pre-processing

4.7.1. Overview

A separate application was implemented to produce pre-filtered data of the original texture and geometry data. The pre-processing system has two main functions:

- To create transformed and low-pass filtered tiles for the diamonds.
- Storing the data in an efficient disk layout.

The system can be extended to include other image processing abilities, as described in the original report, but those has not been implemented.

4.7.2. Data Structure

The data structure described in chapter 4.2 and some of the basic functionality of the ROAM core (base mesh creating, (synchronous) splitting) was reused in the pre-processing system. This enables the application to build a replica of the BTT-tree that will be used in the main application. This reproduced diamond tree is be used to provide information on how the original data should be transformed to suit different diamonds.

The diamond structure is created at application start. This is done by iteratively splitting diamonds in the split queue, starting with the base mesh, until desired level of detail is reached. The outcome is a flat tree with all leafs at the highest level.

The number of levels depends on source data size and desired tiles resolution. The source data for a sphere is normally a so called longitude-latitude map with a 2:1-width-height-ratio. Equation 2 and 3 were used to calculate the highest possible LOD-level for geometry and texture data (notice the level definition in chapter 4.2.5, where the *edge diamonds* are defined at level 0). The padding parameter, which applies for texture data only, will be described in chapter 4.7.3.

$$toplevel_{geometry} = 2 * \log_2\left(\frac{width_{source}}{8 * (width_{tile} - 1)}\right) + 1$$

Equation 2 LOD-level calculation – geometry.

$$toplevel_{texture} = 2 * \log_2\left(\frac{width_{source}}{8 * (width_{tile} - 2 * padding_{tile})}\right) + 1$$

Equation 3 LOD-level calculation – texture.

4.7.3. Leaf Tile Creation

The next step is to remap the source data to the top level diamonds. ROAM geometry consists of a collection of flat patches while a longitude-latitude map is matched to a perfect sphere. The original data must therefore be remapped to conform to the diamond parameterization.

The four ancestor and parent vertices defines the diamond area. The *quad-tree ancestor* vertex is interpreted as the origin of the diamond and separates the two triangular patches of the diamond together with the other ancestor. The two parents define the parameter space vectors (u and v) in relation to the origin (Figure 10).

Since the two triangular patches of the diamond do not have to be uniform or reside in the same plane, they must be processed separately. As depicted in Figure 10 the area of the

diamond can be traversed by using one of the parameter space vectors and the origin-to-other-ancestor vector for each triangular patch individually. The desired tile resolution determines the step length in each direction. Also the data type and the use of *texture padding* affect the layout of the patch point raster.

For texture data the point raster will be located a half step length in from each tile side while for geometry, the raster will start directly from the sides of the tile. This is because height data tiles will share points at borders with neighbors tiles (to avoid cracks) while textures do not. Since texture data often must be interpolated at rendering when the *texel-to-pixel* ratio is more than one visible seams can occur at the tile borders because no data is available beyond the borders. Therefore *texture padding* can be used to include redundant border data that can be used for interpolations at the tile borders. Every unit of *texture padding* will increase the coverage of the point raster one step length at each tile side. For example using one unit of *texture padding* will locate the raster a half step length *out* from each tile side.

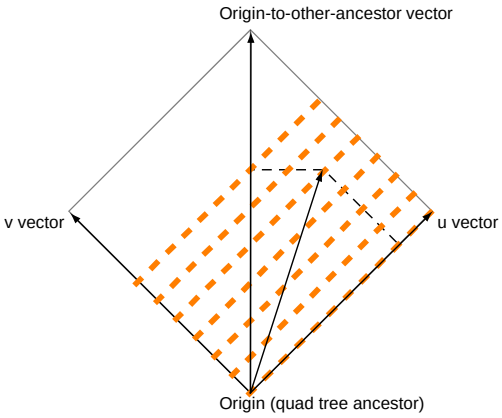


Figure 10 Diamond parameterization. Since the two patches of a diamond does not have to be uniform, the origin-to-other-ancestor vector and the u or v vector are used to describe an arbitrary point in a diamond tile area. This is used to create a point raster which can be mapped to the source data.

Every calculated patch point is projected on to the sphere and converted to spherical coordinates where after it can be mapped to the original data. The projection is identical to the one described in chapter 4.2.5.

The mapping involves two problems. First of all the sampled value must be interpolated. Simple bilinear interpolation is used successfully. The other problem is that the source data may not fit in main memory at the same time. This was solved by using a least-recently-used data cache. The cache can read modest chunks of the source data into main memory without reading the whole file (the source data was converted to raw data – no compression). The spatial coherence of the remapping process will ensure an efficient use of the cache with few chunk loadings. The chunk cache system also supports reading from chunks of source data. This can be very useful when dealing with extremely large datasets that has been divided into different sub-images.

4.7.4. Coarser Low-pass Filtered Tiles

When all the data for the top level tiles has been acquired, the coarser tiles can be computed. This is a recursive process starting with the twelve *edge diamonds*. A coarser tile will cover twice the area of its child but will contain the same amount of data (number of color or height values). This means that a coarser tile should be a low-pass filtered version of its children tiles.

The low-pass filtering operation starts with temporary resemble the data to filter. Depending on the size of the filter kernel used, different number of higher level tiles must be resembled. In this implementation twelve different tiles where used during a filtering pass. This is enough for the use of filter kernels half the size of the coarser tile width. The source tiles are located and correctly placed and orientated in a temporary raster (see [10] for details).

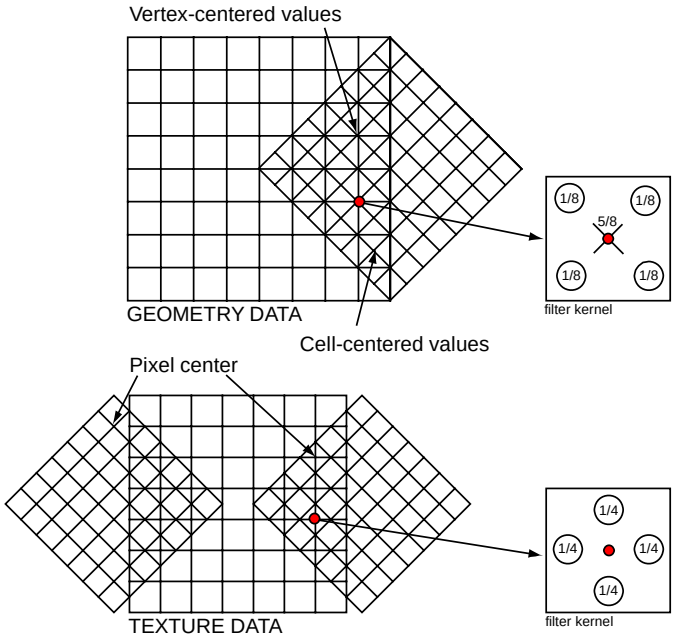


Figure 11 Texture and geometry low-pass filtering.
To create data for a coarser tile, data for at least four higher level tiles have to be resembled (twelve are used for larger filter kernel). The tile data points are laid out as depicted for geometry (top) and texture data (bottom) respectively, forming *cell-centered* and *vertex-centered* points. The values for a new coarser tile are calculated using the corresponding filter kernels.

In view of the fact that the area of tiles at two adjacent levels differs only with a factor of two and that they are rotate 45 degrees from each other, the use of a special filter kernel is required. When a higher level tile is laid out correctly on the coarser tile (as depicted in Figure 11) two types of points can be classified: *cell centered*, which lay between coarser tile points, and *vertex centered*, which lay at the same location as the coarser tile points. Figure 11 shows the filter kernels that were used in this implementation and perform quality low-pass filtering by using both vertex and cell centered points from the source tile. The temporary raster is therefore divided in two arrays – one for vertex centered points and one for cell centered points. Data types (geometry or texture) and *texture padding* must also be taken into account on how to resemble the points into the temporary raster.

The filtering procedure is straight forward when having all source points laid out using the filter kernel described above. Geometry data points at borders should not be low-pass filtered, but be copied form the corresponding vertex centered value. This reassures that no cracks can occur to neighbor tiles at adjacent levels.

The filtering process described above is the base case of the recursive function for creating all data for the coarser tiles in the tree. If the function cannot find all depending tiles for the filtering operation recursive calls will be made for them first.

4.7.5. Disk Layout and IO Functionality

The IO-class is providing functionality for reading and writing to the tile database and is used both by the main application and the pre-processing system. Data for a tile must be efficiently identified and read or written to.

The database is a structured file system with a number of directory levels which contain files. Every file consists of a number of blocks which contain the data for a number of tiles. The exact layout is user specified.

The paging system is based on the sierpinski index - the unique identifier for a diamond. When a tile is requested this index is first converted to another format. The new format consists of the old index bit pattern left shifted one step beyond the most significant bit. A one bit marker is put after the least significant bit. The new format expresses the spatial properties of the Sierpinski index better - all diamonds with the same bit combination of the most

significant bits in the new index format will be spatially related. Directory and file names in the file system names will literally consist of chunks of bit combinations from the index. This allows the file system to reflect this spatial property – tiles closely located will reside in the same block.

Some low-level tiles cannot describe a complete file path because of insufficient bits. Therefore a special root file is used to store those tiles. The root files will contain as many blocks that is necessary to keep all these tiles. Table 4 shows three examples of this mapping.

Table 4 Example: Diamond Sierpinski index mapping.
This table illustrates the mapping of three diamond indices to a tile database. The diamond indices used has 64 bits precision and the file system used has two levels of 16 directories each (4+4 bits) and with 16 files (4 bits) in each. Each file holds 16 tile blocks (4 bits) with 16 tiles (4 bits) in each. This layout will handle a maximum of 15 levels and the root file will consist of 4096 tiles stored in 256 blocks. Notice that the tile data for the first two example diamonds will reside in the same block due to spatial coherence of the data structure.

Diamond index	Index bit representation	Converted (shifted) index	File path	Block index	Tile index
325689	…00100111100000111001	001111100000111001100…	"0011/1110/0000.file"	14 (1110)	6 (0110)
325691	…00100111100000111011	001111100000111011100…	"0011/1110/0000.file"	14 (1110)	14 (1110)
38	…00100110	00110100…	"root.file"	52 (00110100)	0 (0000)

The spatial coherence of the file system allows a very efficient use of caches. In this implementation three different layers of least-recently-used caches where used. When data for a tile is requested for reading, the system first looks in the three caches – if the data is there it will be copied to the right location without any heavy disk operations necessary.

- **Block cache.** The block cache contains blocks of tiles. When a specific tile is needed the whole block for the tile will be read and put into the cache. If another tile located in the same block is requested, it can be copied directly from the block in the cache if it is still available. This cache will be very efficient when close to the surface of the mesh since the spatial coherence in the blocks are high for high level tiles.
- **Tile cache**. This cache is quite simple and contains only single tiles. It is useful when tiles are re-requested several times.
- **Low-level cache.** This cache is static and contains tiles from the coarsest levels. At application start the cache is filled with as many levels down that can fit. Low-level tiles cannot benefit from the block cache since the tiles will be distributed in many files and blocks. On the other hand low-level tiles will generally be used more often than high-level tiles. Therefore it will be profitable to always store these tiles in main memory. Because of the pyramid structure of the data (each new level of detail is twice the size of the previous) relative many levels can be fit into this cache.

For writing operations only the tile cache is used.

5. Result

The intension of this chapter is to evaluate the resulting application that was created. Performance statistics will be presented and some limitations of the system will be highlighted.

Generally speaking, the implementation process was found to be successful. The integration of the system into UniView was quite painless and the system worked as expected.

5.1. Performance

The system has, at the time of this writing, been used with source data of resolution 43200 x 21600 (24 bits per texel) [11] for texture data and height map data of resolution 16384 x 8192 (8 bits per height value) [12].

To measure performance of a system like this is not an easy task. There are several factors that have to be taken into account.

- A real-time interactive application is non-deterministic. Thus, performance analysis will depend on user inputs.
- Asynchronous execution is hard to analyze since context changes will occur and cannot be predicted.
- The variation of execution conditions is great. Apart from hardware choice there are lots of application parameters that can be altered for different behavior.

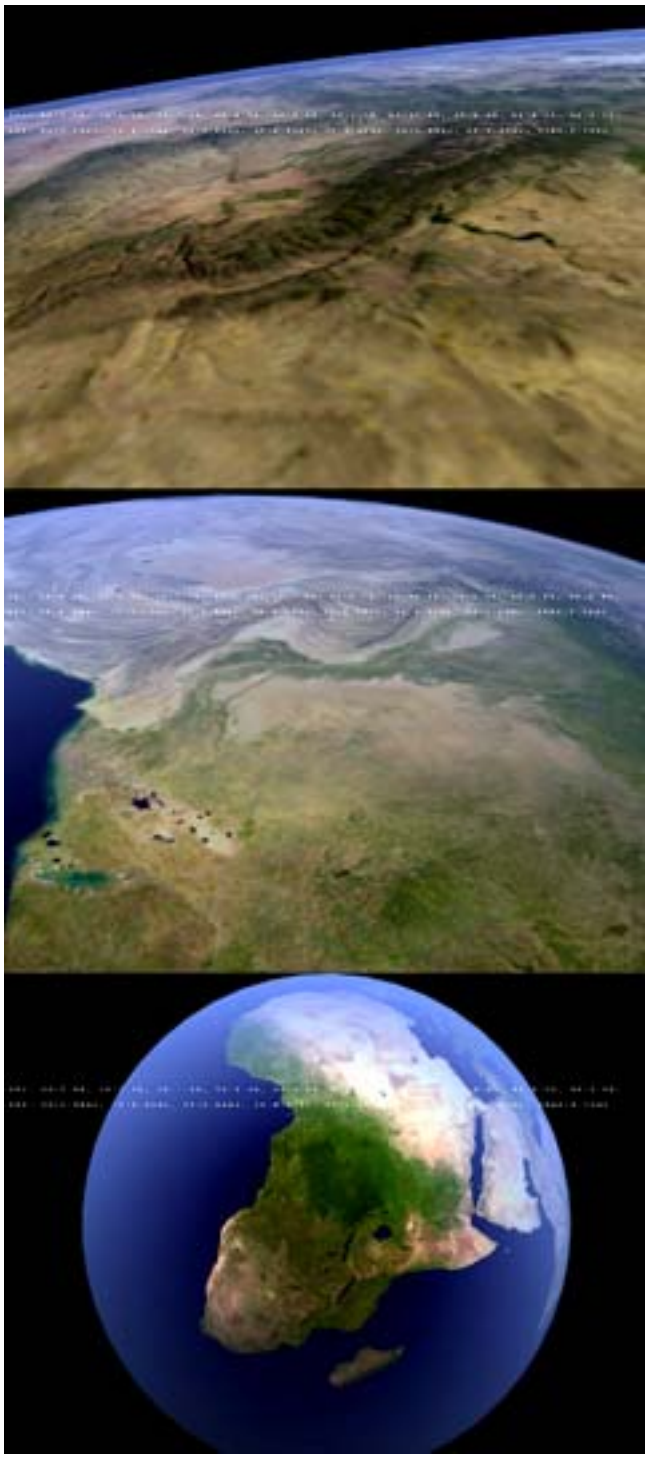


Figure 12 Performance test cases.

To negate the effects of these factors as much as possible, a simple test environment was set up. A timer class was used to measure execution time for different parts of the system. The time was taken as the mean value of the accumulated time for a large number of frames to cancel out the effect of context changes. Due to lack of supporting deterministic flight path at the time of the testing, led to the use of a couple of specific test cases which could describe the system performance in specific situations. A set of execution conditions that was found to work well generally, was chosen for the testing and will be described in detail below.

Three test cases were used, using the dataset described above. The *texel-to-pixel* ratio was set to one. Texture tiles of resolution 64 x 64, with 1 texel of padding at each side, was used in relation with geometry tiles of resolution 17 x 17. This results in a *texel-to-vertex* ratio of 15.5 which also becomes the *pixel-to-vertex* ratio in this case. The IO cache sizes for texture loading were set to 40, 40 and 5 Mb for the tile, block and low-level cache respectively. Cache sizes for geometry loading were set to 0.5, 0.5 and 1.5 Mb for the tile, block and low-level cache respectively.

The *update loop* was constrained to do a maximum of 6 split or merge operations per frame where a merge only counts as a quarter of a split. The patch loader thread was set to standard priority. A screen resolution of 1280 x 1024 was used. The application ran on a 3GHz Intel Pentium 4 processor with 1 Gb RAM with a Radeon X800 XT of 256 MB on Windows OS. Table 5 presents the result.

Table 5 System performance analysis.
Absolute and relative mean time consumed by different parts of the system.

	Culling		**Priority update**		**Tile data applying**		**Update loop**		**Drawing**		**Sum**
Low altitude fly-by	0.15	5.0	0.05	1.7	0.02	0.6	0.02	0.6	2.82	92.1	3.06
High altitude fly-by	0.21	2.9	0.11	1.5	0.15	2.0	0.14	1.8	6.8	91.8	7.41
Planet rotation overview	0.09	1.2	0.1	1.3	0.18	2.3	0.19	2.4	7.25	92.8	7.81
	ms	%	ms	%	ms	%	ms	%	ms	%	ms

The test cases used were two fly-bys on low respectively high altitude above the surface at decent speeds and a planet overview with a full planet revolution every 24:th second. Figure 12 illustrates the different test cases.

The statistics presented above do not take the tile loader thread into account. But with standard performance tools one could conclude that not more than 20 % of the total processor time was used for this task at maximum workload. Notice that the performance of tile loader could generally be improved with larger cache sizes.

Table 6 shows the efficiency of the caches during the test cases for the current cache sizes. It clearly shows the diversity in efficiency for different detail levels.

Table 6 Cache efficiency.
Proportion of cache (re)use and disk access for tile loading.

	Low-level cache	**Tile cache**	**Block cache**	**Disk access**
Low altitude fly-by	9%	5%	72%	14%
Medium altitude fly-by	14%	12%	40%	34%
Planet rotation overview	56%	43%	0%	1%

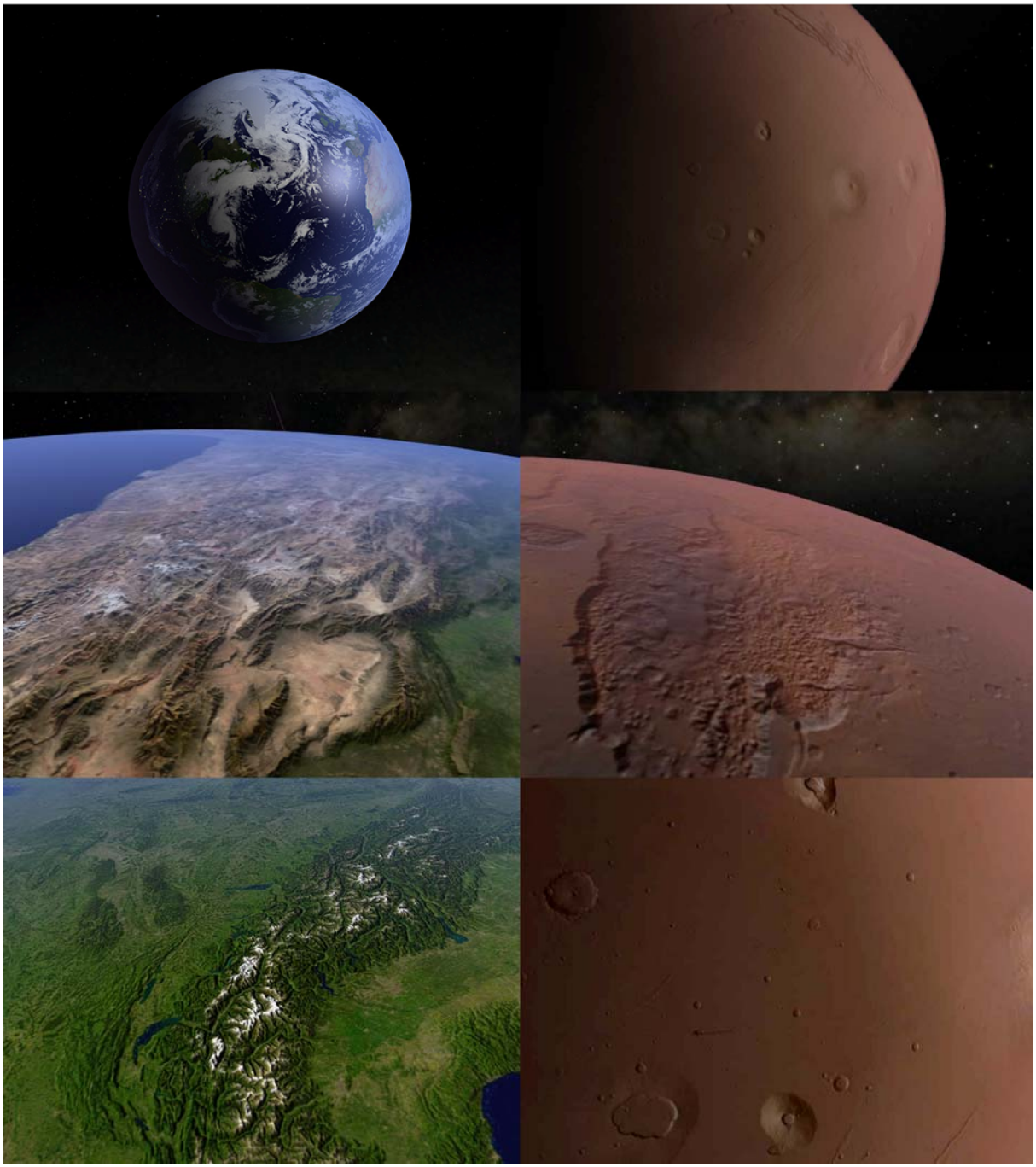


Figure 13 Application screenshots.
Overview and close-up images from Earth and Mars.

5.2. Limitations

In the ROAM variant that was developed during this project two important limitations exists which will be described here.

5.2.1. Hotspots

The concept of attaching data of higher resolution for specific areas can be called *hotspots*. At present the system can only handle source data of homogeneous resolution and therefore *hotspots* cannot be utilized. The importance of this feature in planetary visualization is quite

significant since this kind of data is not acquired uniformly. There exists, for example, extremely detailed data of New York while the steps of Siberia are badly covered.

The leading applications for planetary visualizations support this feature but it is important to mention that they are built around quite primitive level-of detail techniques which create visual seams.

5.2.2. Decoupling of Texture and Geometry

In the system developed every diamond own a static association with *one* specific data tile of each kind (geometry or texture). This leads to a static *texel-to-vertex* ratio that can never change during execution. The only way to alter this ratio is to rebuild the tile database changing the tile resolution of either the geometry or texture data.

In the original report a way to decouple geometry from texture during runtime was described, but the importance of this feature was identified too late during the implementation process and was therefore never implemented.

There are several reasons for keeping dynamic relations between geometry and texture resolution. First of all, relative texel and vertex performance will vary on different graphics hardware [10]. Secondly the *texel-to-pixel* and the *vertex-to-pixe*l ratio are not naturally statically bound. In an orthogonal viewing angle against the planet surface many vertices will be redundant while all texels still will be needed.

6. Discussion

The task of implementing a ROAM 2.0 application has not been easy. Compared to the original technique the degree of difficulty has increased. This mostly depends on the new diamond data structure which is hard to comprehend at first.

The results from the performance analysis clearly show that this new ROAM variant can bridge the problems of the old technique. The new tile layout enable a much higher triangle throughput than before without excessive workload on the CPU. Only around 8 % of the time is used for data structure manipulation which had been impossible to achieve with the original ROAM technique for the same triangle count. The summed time indicate that decent frame rates can be reached (frame rates of 60 Hz equals 16.6 ms per frame).

From another point of view, the achieved drawing time might be considered as quite long. In a real application, like UniView, other things need to be rendered as well and the draw time budget will be quite tight. Therefore the focus for further optimizations should be on limiting the triangle count and even more efficient rendering.

It was difficult to measure how the tile loader affects the system. What could be concluded is that the IO operations of loading tile data from disk are the second heaviest procedure in the system. The tile loading mechanism also suffers from a certain amount of inertia. When the view changes quickly the loading of new tiles cannot keep up and the *texel-to-pixel* ratio will become too high or low. This leads to texture seams at tile borders (when too low) or anti-aliasing effects (when too high).

6.1. Other Applications

This system was developed for planetary data visualization. It is important to emphasize that this technique have many other applications. The ROAM data structure is not bound to flat patches or spheres but can be used with any arbitrary form once a procedure is found to set up the base diamond structure.

6.2. Conclusion

This thesis has shown how the basis of the old ROAM techniques could be extended to conform better to modern graphics hardware. The new method has proved to provide a much better geometry layout by creating adaptive meshes of higher granularity than before. The outcome is a great increase in triangle throughput and a reduction of CPU workload.

The level-of-detail system that was developed is dedicated for planetary data and was successfully integrated into the UniView system. The system handles both texture data and height data of high resolution. A special asynchronous tile loader mechanism was invented to avoid the *Vertex Buffer Range* approach taken in [10].

The general conclusion of this thesis is that the ROAM technique is a very good alternative to a level-of detail system. The main features/advantages are:

- **Frame-to-frame coherence.** Due to this feature very little CPU time has to be spent to decide how the mesh should be refined for a specific view. Additionally, with the higher granularity introduced in ROAM 2.0 even less time have to be spent on this.
- **Twice the levels.** Opposite LOD system built around quad-trees, the ROAM framework, built around the binary triangle tree structure, offers twice as many levels of detail. The seams at borders between adjacent tiles therefore become much more subtle or not noticeable at all.

- **No crack-fixing.** The binary triangle tree structure inherently does not contain any cracks between the triangles. No special crack-fixing method is therefore needed.
- **Flexibility.** The ROAM technique consists of a very flexible and open architecture which easily can be extended without changing the core structure. Also, as said before, this technique can be used with any type of 3d objects.

6.3. Future Work

6.3.1. Hotspots

No method yet exists to use data of heterogeneous resolution with the ROAM system. Since this feature is very important for some level-of-detail applications, i.e. planetary visualization, a solution on this problem would be extremely treasured.

6.3.2. Batched Patches

A way to improve the efficiency of rendering is to further reduce the amount of calls to the graphics hardware. This could be done if several (batched) patches could be rendered at the same time. This approach would include the following extensions:

- **Texture banks.** Instead of using one texture object per tile many tiles can share one texture object. This makes it possible to render many patches with one call. Moreover, this functionality will prevent frequent reallocation of texture memory and decrease the number of heavy *bind* operations.
- **Batched patch management.** A data structure for storing and manipulating these batched patches must be invented. The functionality of this class will include a way of repacking the batches every frame so that they stay continuous. This procedure will not affect performance since only a small number of patches will change draw status each frame (due to culling, splitting and merging). The vertex array data of a patch must be interleaved to be able to create batches of patches.

This functionality could also easy be combined with the use of vertex buffers if available. Only the changed patches of the batches need to be updated to the graphics card memory each frame.

6.3.3. Decoupling of Texture and Geometry

ROAM 2.0 describes a way of decoupling texture and geometry. The importance of this is described in chapter 5.2.2**.** The original approach consists of remapping geometry tiles to different texture tiles at lower levels using resulting composite mapping to alter the texture coordinates.

Another, simpler way to accomplish this, is to instead reduce geometry concentration for all tiles at certain situations. Since *indexed vertex arrays* have been used in this implementation, an altered index array could be used to specify the reduced vertex patch while still using the same texture. Any uniform refinement of triangle patches that is a power of four will create a crack-free mesh [10c]. For example, a standard patch size of 256 triangles could be reduced in four discreet steps: 64, 16, 4 and 1 triangles per patch. Since all patches uses the same index array only five static variations are needed. This approach is really simple but has the disadvantage of not controlling the mapping of individual patches. Instead the triangle concentration of the patches must be determined in a global scope.

6.3.4. Compressed Tile Loading

As mentioned in chapter 6, the tile loader has problems to keep up with the *update loop* when the view changes quickly. The most important cause to this is that disk operations are heavy. Besides using larger or better designed caches, compressed tiles could be used to improve the efficiency of the tile loader. Also this is described in the original report [2], but few details are given.

To accomplish this, a new tile data structure must be invented (from the one given in this text). The new structure must incorporate a way to store tile data of different sizes which will be the case for compressed tiles.

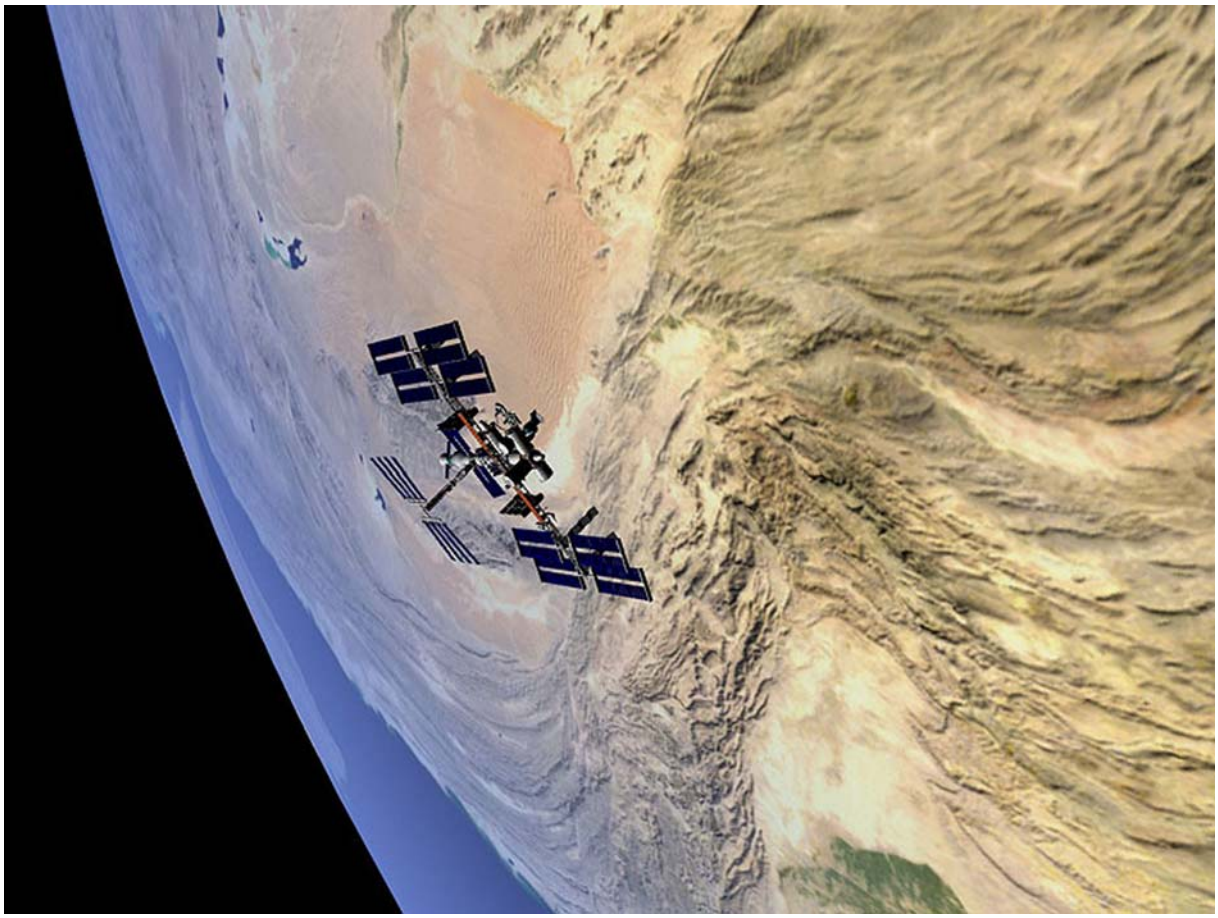


Figure 14 Application screenshot.
ISS (International Space Station) over Hindu Kush (Afghanistan).

7. References

[1] *Smart Content for Interactive Systems AB.* http://www.sciss.se/.

[2] Mark A. Duchaineau, Murray Wolinshy, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein (1997). *ROAMing terrain: Real-time optimally adapting meshes.* In Proc. IEEE Visualization, pages 81-88, October 19-24 1997.

[3] David Hill (2002), *An efficient, hardware-accelerated, level-of-detail rendering technique for large terrains.* University of Toronto: Department of Computer Science. http://www.dgp.toronto.edu/~dh/downloads/thesis.pdf (Acc. 2005-10-30). [3a] page 46, [3b] pages 57-61

[4] *Google Earth.* http://earth.google.com/

[6] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones (1998). *The clipmap: A virtual mipmap.* SIGGRAPH 98 Conference Proceedings, pages 151-158, July 1998.

[7] Clark, J. H. (1976). *Hierarchical geometric models for visible surface algorithms.* Communications of the ACM, 19:547-554.

[8] McNally, S (1999). *Binary Triangle Trees and Terrain Tessellation.* Longbow Digital Arts web page, http://www.longbowdigitalarts.com/seumas/progbintri.html (Acc. 2005-10-30)

[9] Lok M. Hwa, Mark A. Duchaineau, and Kenneth I. Joy (2004). *Adaptive 4-8 Texture Hierarchies*. 15th IEEE Visualization 2004 (VIS'04) pages 219-226, October 2004.

[10] Lok M. Hwa, Mark A. Duchaineau,, and Kenneth I. Joy (2005). *Realtime Optimal Adaptation for Planetary Geometry and Texture: 4-8 Tile Hierarchies.* IEEE Transactions on Visualization and Computer Graphics, March/April 2005 (Vol. 11, No. 2). [10a] page 9, [10b] page 13, [10c] page 12

[11] *Blue Marble: Land Surface, Shallow Water, and Shaded Topography* http://visibleearth.nasa.gov/view_detail.php?id=2433 (Acc. 2005-10-30)

[12] *e43 elevation.* http://www.space-graphics.com/e43_elevation2.htm (Acc. 2005-10-30)