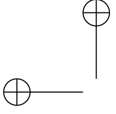
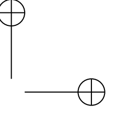
UNIVERSITAT JAUME I
Departament de Llenguatges i Sistemes Informàtics

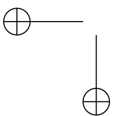
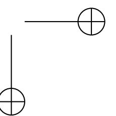
LODStrips: Continuous Level of Detail using Triangle Strips

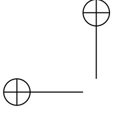
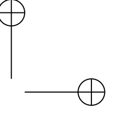
Ph. D. Dissertation
Jose Francisco Ramos Romero

Advisor: Dr. **Miguel Chover Sellés**

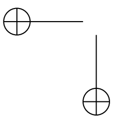
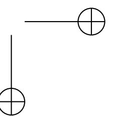
Castellón, April 2008

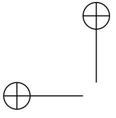
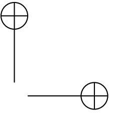



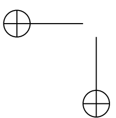
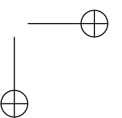



A quien está presente en todos mis sueños.

Preface

This document is the Ph. D. dissertation entitled *LodStrips: Continuous Level of Detail using Triangle Strips*. It is presented by Jose Francisco Ramos Romero, a faculty member at the Department of Computer Languages and Systems of University Jaume I in Castelló, Spain.

Abstract

In recent years, multiresolution models have progressed substantially. At the beginning, discrete models were employed in graphics applications, due mainly to the low degree of complexity involved in implementing them, which is the reason why nowadays they are still used in applications without high graphics requirements. Nevertheless, the increase in realism in graphics applications makes it necessary to use multiresolution models which are more exact in their approximations, which do not call for high storage costs and which are faster in visualization. This has given way to continuous models, where two consecutive levels of detail only differ by a few polygons and where, additionally, the duplication of information is avoided to a considerable extent, thus improving on the spatial cost offered by most discrete models.

Advances have been made in the use of new graphics primitives which minimize the data transfer between the CPU and the GPU, apart from trying to make use of the connectivity information given by a polygonal mesh. For this purpose, graphics primitives with implicit connectivity, such as triangle strips and triangle fans, have been developed. Many continuous models based on this type of primitives have been recently developed. In these last few years, graphics hardware performance has evolved outstandingly, giving rise to new techniques which allow the continuous models to accelerate even more.

In this work, we have improved the interactive render of polygonal meshes. To tackle the problem, we firstly studied fundamental techniques to efficiently render polygonal meshes and we later made use of geometry simplification and level of detail techniques. Thus, we defined a multiresolution model that represents a polygonal mesh at any given resolution. This approach is able

to manage continuous level-of-detail by smoothly adapting mesh resolution to the application requirements. Moreover, the model was modified to take the maximum advantage of the recent GPU features. We also created a modified version of the model for being used in deforming meshes. Finally, we developed an independent library to integrate our model in real-time applications.

Funding

This research has been partially supported by the *GameTools* project from the VIth Framework Program from the European Union (2001/SGR/00296) and from the project *Contiene* by the Comisión Interministerial de Ciencia y Tecnología from the Spanish government (TIN2007-68066-C04-02).

Acknowledgments

First of all, to my advisor Miguel Chover who gave me the chance of looking at the 3D world in a different way and who I have had many professional and personal discussions with. Sincerely, thanks for your patience.

To Minita, you are able to transform normal days into nice days.

To my family, my parents and my sisters, although you do not understand very well what computers graphics are about, you always support me.

To my colleagues and friends, Riki, Joaquin, Oscar, Mike, Jesus and Carlos. Our jokes and freaky conversations made the daily work a little bit more pleasant.

To Ivana and Jindra, for their kindness and support during my stay in Plzen. I miss the hard discussions about morphing in Andel.

And, finally, to all the people involved in one way or another in this work.

Index

List of Figures **VII**

List of Tables **XI**

1. Introduction **1**
1.1. Motivation 1
1.2. Contributions 3
1.3. Document organization 6

2. Previous Work **9**
2.1. Introduction 9
2.2. Simplification 12
2.2.1. Characteristics and classification 12
2.2.2. Operations 13
2.2.3. Metrics 15
2.3. Rendering primitive 16
2.3.1. Triangle strips 16
2.4. Multiresolution modeling 20
2.4.1. Discrete multiresolution models 21
2.4.2. Continuous multiresolution models 22
2.4.3. Characterization 28
2.5. Conclusions 29

3. Level of Detail using Triangle Strips **31**
3.1. Introduction 31
3.2. Construction of the model 32
3.2.1. Simplification 33
3.2.2. Stripification 34
3.2.3. LOD Builder 35
3.3. Representation 38
3.3.1. Theoretical spatial cost 39

3.4. Rendering . . . 41
3.4.1. Uniform resolution algorithms . . . 42
3.4.2. Variable resolution algorithms . . . 44
3.5. Results . . . 46
3.5.1. Spatial cost . . . 47
3.5.2. Temporal cost . . . 47
3.6. Conclusions . . . 48

4. LodStrips: A Uniform Resolution Model 57
4.1. Introduction . . . 57
4.2. Construction of the model . . . 58
4.2.1. Simplifification and stripification overview . . . 58
4.2.2. LOD Builder . . . 59
4.3. Representation . . . 61
4.3.1. LodStrips construction example . . . 62
4.3.2. Theoretical spatial cost . . . 64
4.4. Rendering . . . 65
4.4.1. Level-of-detail extraction . . . 65
4.4.2. Drawing . . . 65
4.5. Results . . . 66
4.5.1. Analysis . . . 67
4.5.2. Spatial cost . . . 69
4.5.3. Temporal cost . . . 70
4.6. Conclusions . . . 71

5. LodStrips on the GPU 79
5.1. Introduction . . . 79
5.2. Stripification techniques . . . 80
5.3. Hardware acceleration techniques . . . 81
5.3.1. High-performance memory . . . 83
5.3.2. Specific library extensions . . . 84
5.4. The model on the GPU . . . 84
5.5. Representation . . . 85
5.6. Rendering . . . 86
5.7. Results . . . 89
5.7.1. Spatial cost . . . 89
5.7.2. Temporal cost . . . 90
5.7.3. Hardware acceleration . . . 91
5.8. Conclusions . . . 93

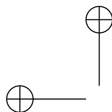
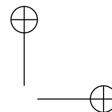
6. LodStrips for Deforming Meshes 95
6.1. Introduction . . . 95
6.2. Background . . . 96
6.2.1. Deforming meshes: morphing . . . 96
6.2.2. Multiresolution . . . 98

6.2.3. GPU Pipeline 98
6.3. Technical background 98
6.3.1. Generating morphing sequences 99
6.3.2. Construction of the multiresolution scheme 99
6.4. Representation in GPU 100
6.5. Rendering 100
6.6. Results 102
6.6.1. Spatial cost 102
6.6.2. Temporal cost 103
6.7. Conclusions 104

7. Applications **109**
7.1. LodStrips Library 109
7.1.1. The VertexData and IndexData interfaces 111
7.1.2. LOD switching 112
7.1.3. Usage 112
7.2. Applications: LodStrips in Ogre 113
7.2.1. Overview 113
7.2.2. LodStrips integration 115
7.2.3. Results 115
7.3. Conclusions 116

8. Conclusions and Future Work **119**
8.1. Conclusions 119
8.2. Publications 122
8.3. Future work 124

Bibliography **127**

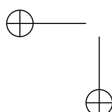
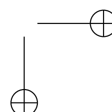



List of Figures

1.1. From left to right, original object with 21,887 vertices and two approximations with fewer vertices: 9,000 and 4,000, respectively. 2
1.2. Continuous multiresolution models rendered at different levels of detail. According to the distance to the viewer, closer objects are more detailed and further less detailed. 3

2.1. Approximations of the *Horse* model. Percentage means the number of vertices composing the mesh with respect to the original one (8,431 vertices). 11
2.2. Example of different simplification operations. 14
2.3. From left to right, examples of: a sequential triangle strip (a), a generalized strip (b) and a triangle fan (c). 17
2.4. A triangle strip of the cow model. 17
2.5. Visualization of a model using different views. From left to right, a) Close view, b) Normal view and c) Distant view 20
2.6. Hierarchical models. 24
2.7. Variable resolution in the horse model. 26

3.1. Half edge collapse and vertex split operations. 33
3.2. Data flow diagram of the model construction process. 34
3.3. Simple geometry represented by means of triangle strips and triangles. 35
3.4. Edge collapse applied to a triangle strip. 35
3.5. Sequence of edge collapses applied to a mesh represented by triangle strips. 37
3.6. Sets of our approach. 38
3.7. Simple representation of the data structures. 40
3.8. Data structures in C++ of our model. 40
3.9. Simple example of a level-of-detail extraction. 43
3.10. Uniform resolution level-of-detail recovery algorithm. 43
3.11. Drawing algorithm. 44

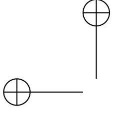
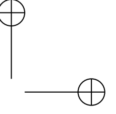
3.12. Cow model at different resolutions. 44
3.13. View frustum criterion applied to a terrain model. 45
3.14. Variable resolution level-of-detail recovery algorithm. 46
3.15. Plane test applied to the bunny model. 49
3.16. Uniform resolution: linear test applied to the cow multiresolution model. 50
3.17. Uniform resolution: linear test applied to the bunny multiresolution model. 51
3.18. Uniform resolution: linear test applied to the phone multiresolution model. 52
3.19. Variable resolution: plane test applied to the cow multiresolution model. 53
3.20. Variable resolution: plane test applied to the bunny multiresolution model. 54
3.21. Variable resolution: plane test applied to the phone multiresolution model. 55

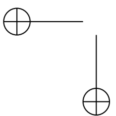
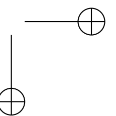
4.1. Triangle strips simplification both filtering and not filtering degenerate triangles. 60
4.2. Simple example of construction of the model. 63
4.3. Fundamental data structures of the multiresolution model. . . . 64
4.4. Level-of-detail extraction algorithm. 66
4.5. Drawing algorithm. 67
4.6. Vertices sent to the GPU per level of detail by applying different filters to the Bunny model. Lowest level of detail means the model simplified at 90 %. 68
4.7. Frames per second per level of detail by applying different filters to the Bunny model. 69
4.8. Percentage of time spent on extraction for different models by applying a linear test with 1000 LOD extractions. 70
4.9. Charts obtained for the Progressive Meshes, MTS and LodStrips multiresolution models based on the Bunny object. 72
4.10. Results obtained for the multiresolution model based on the Cow object. 73
4.11. Results obtained for the multiresolution model based on the Bunny object. 74
4.12. Results obtained for the multiresolution model based on the Dragon object. 75
4.13. Results obtained for the multiresolution model based on the Phone object. 76
4.14. Results obtained for the multiresolution model based on the Isis object. 77
4.15. Results obtained for the multiresolution model based on the Buddha object. 78

5.1. Triangle strips generated by STRIPE and NvTriStrip algorithms for the Cow, Bunny and Dragon meshes, respectively. 82
5.2. Model implementation in GPU. 84
5.3. Simple construction example of the *GPU* model. 87
5.4. Fundamental data structures for the *GPU* multiresolution model. 88
5.5. Level-of-detail extraction algorithm. 89
5.6. Drawing algorithm using the OpenGL *glDrawRangeElements* extension. 90
5.7. Drawing algorithm using the OpenGL *glMultiDrawElemens* extension. 90
5.8. A comparison of extraction and drawing times comparison for the *LodStrips* and *GPU model* using the Bunny object (OpenGL immediate mode was used to better compare the models). . . . 92
5.9. Bunny model performance comparison by using triangle strips generated by Stripe and NVidia algorithms. Rendering mode used was *glMultiDrawElements*. 93

6.1. A simplification sequence from the bug mesh applied to the bunny mesh. 96
6.2. A deforming mesh: Elephant to horse morph sequence. 97
6.3. General construction process data flow diagram. 99
6.4. LOD Builder subprocess. 100
6.5. Multiresolution morphing pipeline using the current technology. 101
6.6. Extraction algorithm. 102
6.7. Temporal cost of the model. 105
6.8. Multiresolution morphing sequence for the FaceToFace model. Rows mean level of detail, 10,522 (original mesh), 3,000 and 720 vertices, respectively, and columns morphing adaptation, aproximations were taken with t=0.0, 0.2, 0.4, 0.6, 0.8 and 1.0, respectively. 106
6.9. Multiresolution morphing sequence for the HorseToMan model. Rows mean level of detail, 17,489 (original mesh), 5,000 and 1,000 vertices, respectively, and columns morphing adaptation, aproximations were taken with t=0.0, 0.2, 0.4, 0.6, 0.8 and 1.0, respectively. 106
6.10. CG implementation of the Vertex shader. 107

7.1. Ogre system overview. 114
7.2. Ogre and LodStrips interaction. 115
7.3. Screenshot of a LodStrips demo. 117
7.4. LodStrips running in OGRE. The colors denote the level of detail used. According to the distance to the viewer, closer objects (in blue) are more detailed and further ones (in red) less detailed. . 117

List of Tables

2.1. A comparison of stripification techniques (ordered according to publication date). 19
2.2. Characterization of different multiresolution models. 30

3.1. Some features and spatial costs (in MB.) for Progressive Meshes, MTS and our approach. 47
3.2. Linear test applied to the multiresolution models PM, MTS and our approach. 48

4.1. List of models with their main features and their original storage cost. 67
4.2. Extraction times of the Bunny model extracting different LODs. The model consists of 31351 levels of detail. 70
4.3. List of models with their features and the *LodStrips* storage cost. 71

5.1. Improvement produced in the spatial cost when implemented with the new data structure. 91
5.2. LodStrips on GPU: Linear test with 1000 extractions applied to the Isis model (187644 vertices and 5141 triangle strips at the highest LOD) by applying different hardware acceleration techniques. 93

6.1. Spatial cost. 103

Chapter 1

Introduction

In a computer graphics context, this dissertation is particularly related to real-time visualization. The aim of this dissertation is to develop a new solution to accelerate 3D visualization by means of geometric level-of-detail techniques. Results can be applied to any real-time application, such as video games or virtual reality.

1.1. Motivation

Nowadays, it is common to represent 3D scenes with a high geometrical complexity. Many of the objects that are included in these scenes come from high-precision scanners, computer-aided design tools, digital terrain models or even from the tessellation of implicit surfaces. In general, objects with millions of polygons. However, graphics resources are limited: processor, memory, graphics hardware capacities, network bandwidth, etc. One possible solution to reduce the cost of representing a 3D scene is to employ level of detail or multiresolution modeling techniques.

The fundamental concept of theses techniques is simple: when rendering, we should use a less detailed representation for small, distant, or unimportant parts of the 3D scene. This less detailed representation usually consists of a selection of different approximations of objects in the scene. Obviously, each approximation is less detailed and faster to render than the one before. Generating and rendering these approximations of objects is known as level of detail or multiresolution modeling. Figure 1.1 shows three levels of detail for an object.

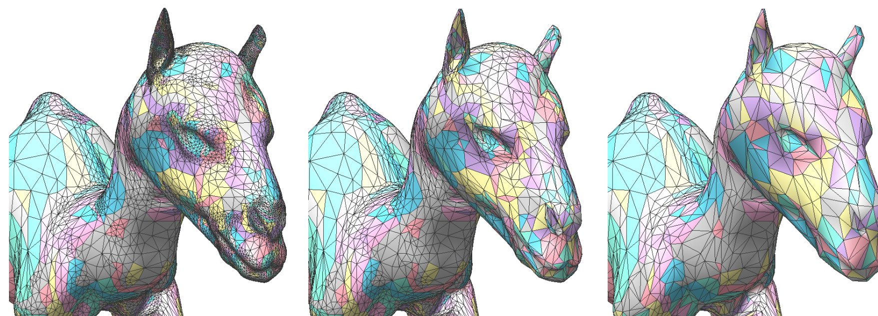


Figure 1.1: From left to right, original object with 21,887 vertices and two approximations with fewer vertices: 9,000 and 4,000, respectively.

In recent years, multiresolution models have progressed substantially. They can be classified into two large groups [RLB+02]: discrete multiresolution models, i.e. those that contain various independent representations of the same object with different levels of detail, and continuous multiresolution models, which are those that manage a vast range of approximations to represent the original object. At the beginning, discrete models were employed in graphics applications, due mainly to their simplicity in implementing them, which is the reason why nowadays they are still used in applications without high graphics requirements. Nevertheless, the main problem of discrete models is the transition between different levels of detail. When changing between the different approximations, an effect, known as *popping*, appears. It consists in a perception of the change in the visual quality. Besides this problem, discrete models present other limitations, such as the number of independent approximations stored and the difficulty to accurately adjust the level of detail to the requirements of the application. Continuous multiresolution models appeared to solve most of the problems presented by discrete models. They are more exact in their approximations (two consecutive levels of detail only differ by a few polygons) and do not call for high storage costs (duplication of information is avoided to a considerable extent, thus improving on the spatial cost offered by many discrete models). In Figure 1.2, we can see a continuous multiresolution model in runtime. Objects are rendered at different levels of detail depending on the distance to the viewer.

Multiresolution meshes are often represented by polygonal models. In particular, most-used polygon primitive is the triangle. However, advances have been made in the use of primitives which make use of the connectivity information given by a polygonal mesh, apart from trying to minimize the data transfer between the CPU and the GPU. For this purpose, graphics primitives with implicit connectivity, such as triangle strips and triangle fans, have been developed. There exist many continuous multiresolution models based on this

type of primitives [Hop97, ESAV99, Ste01, Paj01, DP02, Zac02, SP03].

Triangle strips provide us two ways to improve continuous multiresolution models. On the one hand, the use of triangle strips potentially reduce the number of vertices to be processed by, approximately, a factor of three [ESV96b] by avoiding redundant transformation, clipping, and lighting computations. Besides, such an approach is also an efficient way to encode polygonal meshes.

Therefore, a multiresolution model wholly based on implicit connectivity primitives could improve rendering and spatial cost. Moreover, an easy integration into the GPU would be important as well. Finally, time required to extracting an approximation becomes fundamental so that we can render more objects per scene or make lighter the GPU work. All that, allows us to achieve more realism in real-time visualization.

Figure 1.2: Continuous multiresolution models rendered at different levels of detail. According to the distance to the viewer, closer objects are more detailed and further less detailed.

1.2. Contributions

The main objective of this work is to develop a new multiresolution model that accelerates level-of-detail visualization in real-time for polygonal meshes. On the one hand, we developed a basic and general multiresolution model based on triangle strips. This model is able to manage uniform resolution (one level of detail on the whole object) and variable resolution models (different levels of detail on the same object). However, in many applications, such as video games,

most-used models are uniform ones due to their simplicity (data structures and algorithms are simpler than in variable resolution models) and efficiency (they only extract one level of detail, while in variable resolution models we must check the level of detail to be extracted for different parts of the mesh). Thus, we improved this basic scheme in order to generate *LodStrips*, a uniform multiresolution model designed to speed up geometry rendering with level of detail. Moreover, we developed the model to be optionally integrated into the graphics hardware by using new acceleration techniques. Finally, the model was successfully implemented into a game engine and applied to deforming meshes. Below, we will provide a brief description of each contribution.

Previous work in multiresolution modeling

We present some fundamental techniques to efficiently render polygonal models. Key factors are simplification and rendering primitives. Simplification techniques are important to generate the different approximations or levels of detail that we will use in the real-time applications. Other key factor is the rendering primitive, which will enable us to speed up rendering and even to improve the storage cost.

As the simplification method applied is an important factor to produce high quality approximations, we firstly study simplification techniques and the different operators and metrics applicable to polygonal models. Later, we analyze the possibilities of new graphics cards by using the triangle strips graphics primitive. Finally, we survey the different kinds of existing multiresolution models, as uniform as variable ones. A characterization with the main features of lastest models is presented as well.

New ways to improve multiresolution schemes

The technique known as multiresolution or level-of-detail modeling consists in using some kind of simplification or approximation of complex polygonal models to reduce the amount of information to be processed by the graphics system. These models can be improved by including implicit connectivity information, storing the model as triangle strips or triangle fans that reduce the amount of information sent to the graphics pipeline and increase the rendering frame rate. In summary, modeling a mesh as a collection of strips or fans of triangles avoids storing and sending a large amount of redundant information to the graphics system. Therefore, it involves better visualization times and lower storage cost.

We present a general framework to improve existing multiresolution models. It makes use of implicit connectivity primitives such as triangle strips. It represents a mesh as a set of multiresolution triangle strips and maintains the strips both in the data structure and in the rendering stage. This approach offers features such as connectivity exploitation, uniform and variable resolution,

fast extraction and low storage cost.

LodStrips: A new uniform multiresolution method based on triangle strips

New specific models in applications such as computer games could improve their performance. This kind of applications mainly use uniform resolution models, and obviously, they must render and extract levels of detail as fast as possible. Taking into account the framework previously presented, we introduce *LodStrips*, a continuous multiresolution model that manages data structures and algorithms for real-time visualization of multiresolution meshes.

Hardware optimizations in rendering

In these last few years, graphics hardware performance has evolved outstandingly, giving rise to new techniques which allow the continuous models to be accelerated even more. The use of specific stripification algorithms, which try to take the maximum advantage of the GPU cache, and the new extensions of graphics libraries that allow visualization of a whole mesh with only a few instructions are examples of these new techniques.

We introduce some modifications on the *LodStrips* model which allow us to noticeably improve its global performance on GPU. Among them, we underline: modifications on the data structures and application of hardware acceleration techniques. The efficiency of the geometric acceleration techniques have been tested on *LodStrips*. Using hardware acceleration techniques allows us to increase the performance of the models with dynamic geometry. In this sense, the *LodStrips* model greatly increased its performance. This rise is mainly due to the optimized design of the model for the hardware, where level-of-detail extraction times are very low and so graphic acceleration is greatly benefited by avoiding long waits to render approximations.

Multiresolution modeling for deforming meshes

Deforming surfaces are present in many fields such as games, movies and simulation applications. However, as occurs with static surfaces, they are often represented by means of much more geometry than necessary in 3D scenes. Unfortunately, as multiresolution techniques for static meshes are based on specific fixed shapes, the meshes they produce can yield very poor approximations if the mesh highly deforms.

In particular, we focus on mesh morphing, a technique to approximating deforming meshes. A source mesh is transformed into a target mesh by interpolating vertex positions. A possible solution to this problem consists of computing a supermesh representing the union of both meshes [Ale02].

We therefore modified *LodStrips* for being used in deforming meshes. We thereby created a multiresolution model for deforming meshes based on the triangle strip primitive. For our work we used the method proposed by Parus et al. [Par05] in order to get the morphing sequence between two arbitrary meshes, although any other morphing sequence can be also used. Next, we combined that work with a modified version of *LodStrips*. Besides morphing, our model is also able to manage animated models. The model can extract any level of detail for any frame required.

Successful implementation in real-time applications: Ogre3D

We developed an independent library to integrate *LodStrips* in 3D applications. An efficient interface between the application and the geometry modules was designed and implemented. This library provides the ability to construct and render polygonal meshes in 3D applications.

This module contains functions that handle the levels of detail of the input multiresolution polygonal meshes. For any given resolution of an object, this module returns a set of triangle strips representing the object at that resolution, that is, at the requested level of detail. Using *LodStrips*, models take advantage of using triangle strips to reduce storage usage and to speed up realistic rendering.

We extended the graphics rendering engine OGRE3D to efficiently employ a continuous multiresolution model: *LodStrips*. This graphics engine is designed to use discrete multiresolution models and we integrated the *LodStrips* library in such a way that it is not necessary to recompile the engine.

1.3. Document organization

This dissertation is organized as follows:

- Chapter 2: Previous Work
  We introduce a survey of important factors to efficiently render polygonal meshes. We analyze simplification, rendering primitives and multiresolution modeling works. We finally present a characterization of them.

- Chapter 3: Level of Detail using Triangle Strips
  With the aim of improving existing multiresolution models, a general scheme is presented. The proposed model makes use of new implicit connectivity primitives such as triangle strips and it offers the features such as connectivity exploitation, uniform and variable resolution, fast extraction and low storage cost. Data structures, algorithms, as well as results are presented.

- Chapter 4: LodStrips: A Uniform Resolution Model
  We introduce the *LodStrips*, a uniform and continuous multiresolution

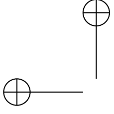
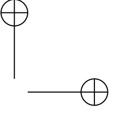
model for real-time visualization. Main data structures and algorithms are also presented. Moreover, we carry out a study about some important questions related to the model such as filtering triangle strips or extraction criteria. Later, we present the results obtained from comparing this model with other well-known multiresolution models such as Progressive Meshes [Hop96] and Multiresolution Triangle Strips [BRR+01].

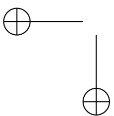
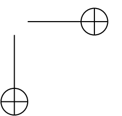
- Chapter 5: LodStrips on the GPU
  We propose some modifications on the original model which allow us to noticeably improve its global performance. Among them, we highlight modifications on the data structures and application of hardware acceleration techniques.

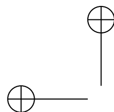
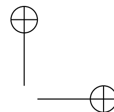
- Chapter 6: LodStrips for deforming meshes
  We present a modification of *LodStrips* for deforming meshes. This approach can extract any level of detail for any frame required of an animation.

- Chapter 7: Applications
  An independent library to integrate *LodStrips* in real-time applications is shown in this chapter. Moreover, the implementation of the library in the Ogre3D game engine is also introduced.

- Chapter 8: Conclusions and Future Work
  Finally, we summarize the contributions of this dissertation and the different publications derived from our work. Future work lines are also presented.

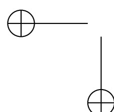
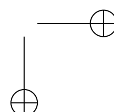
CHAPTER 2

Previous Work

Real-time visualization of 3D scenes is a very important feature of many computer graphics solutions. In applications such as computer-aided design, scientific visualization or even in the growing computer games market, the performance of visualization becomes essential. In addition, the complexity of the scenes is increasing and they now contain objects composed of thousands or even millions of polygons. Therefore, it is necessary to resort to different techniques that allow us to maintain the quality and performance of 3D applications by managing that huge amount of geometry. Among the different solutions, we highlight polygonal simplification, implicit connectivity primitives and level-of-detail approaches. This has led to the appearance of a wealth of research in multiresolution modeling. The main objective of this chapter is to study previous works on this topic, presenting the different solutions that currently exist in the field of real-time visualization of level-of-detail models and to analyze the most notable works published to date.

2.1. Introduction

Nowadays it is common to represent 3D scenes with a high degree of geometrical complexity as the latest technological advances have generated large databases of polygonal models. Many of the objects that are included in these scenes come from high-precision scanners, computer-aided design tools, digital terrain models or even from the tessellation of implicit surfaces. Thus, in general, the output objects are composed of millions of polygons that exceed by far the visualization capacities of present hardware, including the processor, memory, graphics hardware, network bandwidth, etc. Applications such

as computer games, distributed virtual environments or the creation of special effects for films make use of models generated by this type of systems. In all these applications, a balance must be found between the accuracy with which a surface is modeled and the amount of time required to process it.

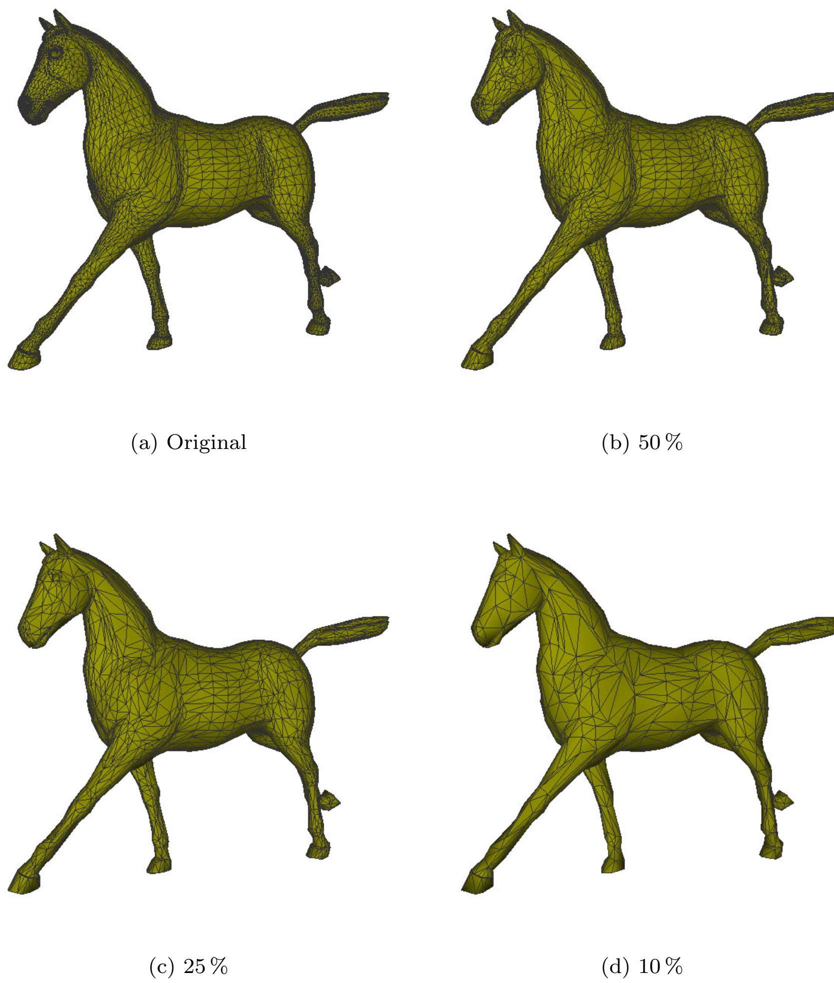
In general, it is assumed that the precision of the approximation is proportional to the number of triangles that form it. The aim is to produce the most simplified mesh that meets the requirements of the application. Nevertheless, simplification in runtime has a high temporal cost. The concept of level of detail or multiresolution modeling emerges with the purpose of supporting real-time simplification operations. This concept entails creating a model in a pre-process that stores several approximations (see Figure 2.1) and that is capable of retrieving any of these approximations in an efficient way.

In 1976, James Clark already described the benefits of representing the objects of a scene at several resolutions [Cla76]. The basic idea of using levels of detail (*LOD*s) is to use simpler versions of an object as it contributes less and less to the rendered image. In general, solutions based on level of detail consist of three parts: generation, selection and switching. Generation is the stage where different representations of a model are generated with different detail (see Figure 2.1). The selection mechanism chooses a level of detail based on some criteria, such as the distance to the viewer, the screen-space area or more complex techniques with feedback or predictive analysis [XESA97]. Finally, we need to change from one level of detail to another, this is named switching.

The mostly used models in interactive graphics applications are 3D polygonal ones, usually composed of triangles due to their simplicity. Before describing the different multiresolution approaches, it is important to comment on two elements that are key for the generation of a level-of-detail model. On the one hand, the simplification process will generate the different approximations of the objects. The application of simplification algorithms to 3D polygonal models is a well-known problem and several algorithms have been successfully developed [MP06]. On the other hand, the visualization primitive becomes a key aspect with respect to the final performance of the multiresolution model.

Traditionally, the triangle has been used as basic. Nevertheless, the evolution of graphics hardware has allowed the appearance of primitives that improve outstandingly the performance of graphics applications. Thus, the use of triangle fans or triangle strips allows us to save bandwidth both in mesh codification and in rendering performance. Therefore, the combination of triangle strips based rendering and simplification methods gives rise to efficient solutions in real-time rendering.

The aim of this chapter is to present the different solutions that currently exist in the field of real-time visualization of models with level of detail. It is possible to find in the literature different reviews [Gar99, RLB+02], but it was necessary to offer a more recent revision which includes the latest advances and tendencies. This chapter is structured as follows. The first two sections



(a) Original (b) 50 % (c) 25 % (d) 10 %

Figure 2.1: Approximations of the *Horse* model. Percentage means the number of vertices composing the mesh with respect to the original one (8,431 vertices).

introduce two very important aspects in multiresolution schemes: simplification algorithms (Section 2.2) and rendering primitive (Section 2.3). Subsequently, Section 2.4 revises the most notable multiresolution models, both of uniform and variable resolution, and a classification of them is also presented. Finally, conclusions and a summary of the main ideas are given in Section 2.5.

2.2. Simplification

Polygonal simplification techniques offer an important alternative to visualize complex models. Many methods have been developed in this field [DZ91, CMS98, Lue01]. These methods simplify the geometry of the model in order to reduce the visualization cost without a significant loss of the visual content of the scene. This idea has been addressed for many years in the field of interactive graphics. These simplified meshes can be generated manually or automatically. Manual simplification is quite costly and many times unattainable due to the complexity of the mesh. As a result, many efforts have been made towards the automatic simplification of polygonal models. In Figure 2.1, the simplification of a polygonal mesh which preserves its appearance can be observed.

The main objective of the automatic simplification of polygonal meshes is to obtain an approximation of a high resolution mesh maintaining its appearance as possible. Formally, given a surface (normally represented as a triangle mesh), the aim is to find an approximation which minimizes both the size and the error of the approximation. Desirable properties for this sort of algorithms are: speed, quality in the approximation, and management of different types of initial meshes. The solutions to this problem are based on the heuristic applied and on the function that measures the quality of the output mesh. Nevertheless, there is no optimum solution since it depends totally on the heuristics or function used, on the order of execution of the simplification algorithm, the objective and so forth.

2.2.1. Characteristics and classification

We can classify the simplification methods into several groups according to:

- The input and output data. Many methods accept triangle meshes as input. However, only a few accept more general meshes, for example, *non-manifold* meshes, where one or more edges are shared by more than two triangles. As for the output, there are many methods that can produce triangle meshes both *manifold*, where every edge must be shared by exactly two triangles, and *non-manifold*. Moreover, it is important to underline that some of these methods also outputs simplification sequences [Hop96, GH97].

- The objective of the simplification. Methods that given a certain error ϵ construct the minimum mesh that approximates the original and sat-

isfies that error (the error is usually measured as a certain number of vertices [Hop96]).

- The heuristics used in the approximation. They can be sub-characterized according to whether the heuristics allow us to measure the error introduced in each simplification step by a defined metric, evaluate the error locally or globally or preserve the geometry or attributes in the mesh.

- The operation applied in the simplification. Operations are local or global [LRC+02], local operations reduces the complexity of a mesh by considering a small portion of the mesh and global operations modify the topology of the whole mesh in a controlled fashion.

- Whether it is incremental or not. A method is incremental if the simplification consists of a sequence of local updates which reduce the size of the mesh in every step while, at the same time, the precision of the approximation obtained decreases.

2.2.2. Operations

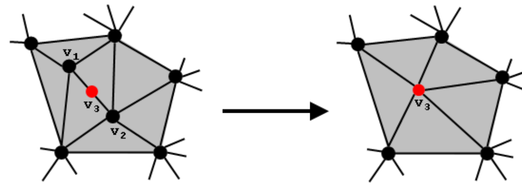
As we have previously commented, the operations applied to produce the approximations can be divided into local and global [LRC+02]. Even though in the work developed by Matias and Pedrini [MP06] each of these operations is detailed and compared, among the existing simplification methods the most populars are incremental methods based on local updates. All these methods have in common that they simplify the original mesh by means of an ordered sequence of local modifications and, besides, every modification reduces the size of the mesh, decreasing in addition the precision of the approximation generated.

Regarding the existing local operations, we emphasize:

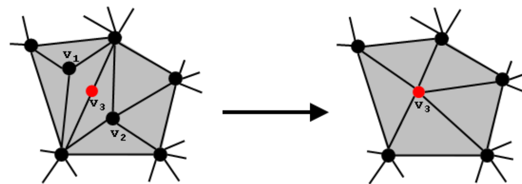
- Edge collapse: an edge (v_1,v_2) collapses in a vertex v_3 producing the elimination of those triangles that contain it (see Figure 2.2a). This operation was firstly introduced by Hoppe [HDD+93] and later used in several works [Hop96, XV96].

- Vertex collapse: two unconnected vertices, v_1 and v_2, are collapsed. Adjacent triangles require to be modified. An important aspect of this operator is that it is able to connect unconnected elements to close holes and gaps (See Figure 2.2b). Important works that use this operation are [GH97, ESV99].

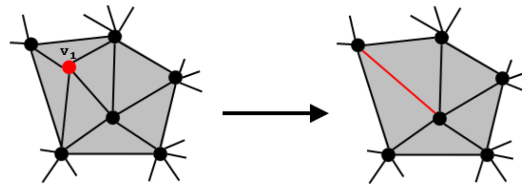
- Vertex decimation: it involves eliminating a vertex, its edges and the triangles it forms. Subsequently, the area is triangularized again (see Figure 2.2c). Notable papers inside this group are [SZW92, Sch97, CCMS97].



(a) Edge collapse.



(b) Vertex collapse.



(c) Vertex decimation.

Figure 2.2: Example of different simplification operations.

2.2.3. Metrics

It is also important to comment on the error function employed to determine the operations to perform and also characterize the simplification methods. Metrics based on the optimization of an energy function [Hop96] and those based on the metrics of quadric errors [GH97, HON04] are the most important and widely used. Depending on the way the error function is calculated, we can classify the metrics into geometry-based and viewpoint-based.

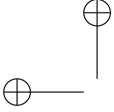
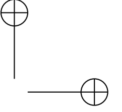
Geometry-based

These metrics use measures based on geometry [MP06], calculating the error by means of simple heuristics (length of the edges, angles, area, etc.), distances to the original surface [Hop96, CCMS97], or other important characterizations such as the one based on quadric errors [GH97, HON04]. We highlight two works:

- Simplification based on quadric errors [GH97]. It is based on iterative edge contractions. A geometric error is maintained for each vertex; this error is calculated as the sum of the squares of the distances to the planes adjacent to the vertex. The overall algorithm is as follows:

    1. Compute the quadric error corresponding to each vertex
    2. Determine the contraction cost for each edge
    3. Place the edges into a min priority queue sorted on contraction cost
    4. Remove the edge *(u,v)* with the lowest contraction cost from the priority queue
    5. Use the quadric to determine the optimal contraction target of *(u,v)*
    6. Contract u and v, and update the cost of all edges adjacent to u and v
    7. Repeat steps 4 through 6 until the desired mesh resolution is reached

- Optimization of an energy function. It allows the reduction of the mesh in an incremental way, eliminating vertices, edges or faces, through the optimization of a function that measures the quality of each of the approximations generated. Given a set of vertices V_0 and an initial triangle mesh M_0, a mesh M_j of the same type and with fewer vertices than M_0 is progressively obtained. To that end, an energy function that contemplates the geometric parameters of the representation is employed. The process is based on a nonlinear optimization, where the number, position and connectivity of the vertices vary in order to minimize such function [Hop96].

Viewpoint-based

Metrics based on viewpoints involve generating different images from several cameras, locating the camera in more than one point around the object, and thus calculating the error when simplifying, comparing the result with the original object [LT00, WLC$^+$03, CSCF07]. These methods generates good realistic results for the viewer, by removing for example parts of the object that are not visible for the user. However, they have a high cost of computation. We underline these works:

- Image-driven simplification [LT00]. Basically, it determines the cost of an edge collapse operation by rendering a model from different points of view uniformly located. Later, it compares the result images to the original ones and adds an error across all the pixels of all the images. Finally, all edges are sorted by the total error induced in the images and the edge collapse that produces the least error is chosen.
- Mesh saliency. Idea of mesh saliency, as a measure of regional importance for graphics meshes, has recently been introduced [CHVJ05]. It consists of generation a saliency map, and then simplifying by using this map.
- Other interesting works are addressed to the application of information theory to the field of metrics based on viewpoints [SFRV07].

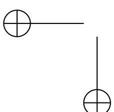
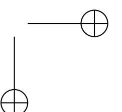
2.3. Rendering primitive

Traditionally, graphical models have been represented with triangles, as the simplicity of their elements and their connectivity allow us to easily manipulate and render them [PS97]. Nevertheless, triangle strips are able to make the most of these characteristics while offering a compact representation of the connectivity of the mesh and a faster rendering, due to the fact that the number of vertices to transform and illuminate is lower.

2.3.1. Triangle strips

In general, the way of encoding a triangle mesh is the specification of the three vertices that compose each of the triangles of the mesh. Owing to the fact that neighbour triangles share an edge and its vertices, all vertices are sent several times to the graphics pipeline. In general, the number of triangles is approximately twice the number of vertices [PS85].

A triangle strip is a more efficient representation which consists of a series of $n+2$ vertices representing n triangles. In Figure 2.3a, the sequence {0,1,2,3,4,5} corresponds with triangles {0,1,2}, {1,2,3}, {2,3,4} and {3,4,5}, that is, a sequential triangle strip. Thus, the transmission cost of n triangles is reduced in a factor of three, from $3n$ to $n + 2$ vertices.

In some situations, the adjacency of the triangles does not allow a sequential encoding. An example of it can be observed in Figure 2.3b. To represent this triangle strip, it is necessary to add an extra triangle to convert the sequence into {0,1,2,3,2,4,5}. This operation is known as repetition or *swap* and triangle strips with *swaps* are called *generalized triangle strips*. Despite this extra information, the transmission cost is reduced in a factor greater than two, from $3n$ to n+2+*swaps* vertices. In some special cases, it is also possible to use a special kind of generalized triangle strip which is termed triangle fan. The fan is defined by a central vertex and its neighbour vertices. In Figure 2.3c, the fan is defined by the sequence {6,0,1,2,3,4,5}. This sort of primitive is not widely used in practice because the length of a fan is normally short: the average number of neighbour vertices is usually six in a manifold mesh [AM04].

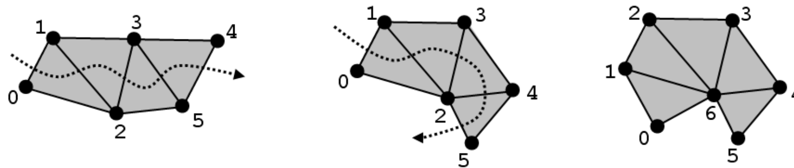


Figure 2.3: From left to right, examples of: a sequential triangle strip (a), a generalized strip (b) and a triangle fan (c).

Owing to the fact that the number of triangles in the meshes grows as fast as the processing capacity of GPUs and the bandwidths of the buses, triangle mesh stripification is very important and many papers have appeared in this direction. Triangle strips provide a compact representation of triangle meshes and they are supported by the most important graphics libraries like OpenGL and DirectX. In general, they allow for a fast transmission and visualization of triangle meshes. In Figure 2.4, we show a triangle strip in the cow model.

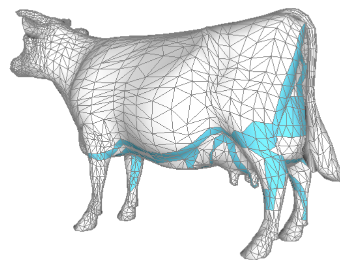


Figure 2.4: A triangle strip of the cow model.

Evans showed that given a polygonal mesh made up by triangles, the prob-

lem of creating an optimum set of triangle strips for this triangulation is NP-hard [ESV96a]. Therefore, to stripify a mesh in a polynomial time it is necessary to use certain heuristics that find an optimum local. A lot of work focused on heuristics has been carried out in an attempt to minimize the number of triangle strips in a polygonal mesh [AHB90, AHMS96, ESV96b, Kor99, Ste01, XHM99, GE04, DGBGP06]. Strip search algorithms, commonly known as *stripification* algorithms, can be classified in different ways [VK07]. Here, they are classified into five different groups according to:

- The kind of input data: isolated vertices, triangles, triangle strips and so forth.
- The kind of mesh: static meshes or meshes with level of detail. It includes the algorithms which receive the triangles of the model and construct the triangle strips with or without changing the mesh topology. Most stripification algorithms have been designed for static meshes. However, due to the increasing complexity of some industrial models, the need of visualization by level of detail becomes even more accentuated. With respect to the use of triangle strips in meshes with level of detail, there are two different approaches. On the one hand, there are methods which work with versions of an original stripification [ESAV99, BRR$^+$01, RC04b]. On the other hand, we can find methods which change the mesh topology for each approximation used [VFG99, Ste01, SP03].
- The type of optimization: minimization of the number of triangle strips or minimization of vertices. The term *optimum stripification* can be understood in several ways. It can be optimized by producing a low number of vertices in the strips to reduce the amount of data sent through the bus and thus accelerate visualization. On the other hand, the initialization of a new triangle strip has a certain cost, though, and that's why it is also interesting to minimize the number of strips generated [Ste01, PS03]. Obviously, it is not possible to minimize both parameters at the same time; reducing the number of strips often involves increasing the number of vertices (due to the high number of *swaps* necessary to preserve the strips) or vice versa. A different approach is offered by those works that create a single triangle strip [GE04, DGBGP06].
- The kind of heuristic: local or global. Other classification considers the type of heuristic function used. Usually, the heuristic function only decides in which direction the triangle strip must continue. To take this decision, it is enough to apply a certain local criterion [ESV96b, Ste01, GE04].
- Hardware: optimization for vertices caches. It refers to the methods capable of generating triangle strips which maximize the use of the cache of vertices. Nowadays, GPUs contain caches of vertices that allow us to reuse already transformed vertices, reducing the bandwidth and therefore

accelerating visualization. Taking into consideration this criterion, many papers have been written [Hop99, BG99, BD02, LY06, NBS06].

Stripification can be applied to multiresolution models in two ways: static or dynamic. Dynamic stripification involves generating the triangle strips in real time, that is, for each level of detail new strips are generated. On the other hand, static stripification entails first creating triangle strips and then working with versions of the original strips. There are several models that use dynamic stripification [Hop97, Ste01, SP03], especially variable resolution models. However, other models such as [ESAV99, RAO+00, BRR+01] use static stripification techniques.

An important problem that poses the use of triangle strips in a multiresolution model is the appearance of degenerate triangles. In those multiresolution models that maintain an original (static) stripification, when the level of detail is decreased the topology of the mesh is altered and it is possible that the initial stripification presents unnecessary triangles that add no information but involve a higher processing time. A solution followed by several authors is the use of filters to eliminate those indices of the strips that are no longer necessaries. Nevertheless, it is worth to refer to those algorithms which were specifically developed for multiresolution models [BRR+01, RCR05]. These algorithms generate the triangle strips following a simplification criterion. Belmonte et al. developed [BRR+01] an algorithm which generated strips starting from the maximum level of detail and applying the subsequent simplifications. A different approach was that presented by Ripolles et al. [RCR05]. In this work they developed a similar algorithm but it constructs the strips starting from the minimum to the maximum level of detail and following the simplification sequence. Nevertheless, both approaches still present a great amount of degenerated triangles in many levels of detail. Table 2.1 summarizes the differences and similarities of some important multiresolution models according to the stripification they apply.

Model	Stripification
VDPM [Hop97]	Dynamic
Skip Strips [ESAV99]	Static
MOM [RAO+00]	Static
MTS [BRR+01]	Static
Tunneling [Ste01]	Dynamic
DStrips [SP03]	Dynamic

Table 2.1: A comparison of stripification techniques (ordered according to publication date).

2.4. Multiresolution modeling

The techniques to control the level of detail of a surface in runtime are very important in real-time visualization systems. In any system, the capacity of the available hardware is essentially limited. Nevertheless, the complexity of a scene can vary substantially. Therefore, in order to keep a constant frame rate, the level of detail of the scene must not exceed the processing capacity of the graphics hardware.

In order to manage the level of detail of an object it is necessary to resort to a surface representation that allows the reconstruction of several approximations adapted to different visualization contexts. An example can be observed in Figure 2.5. Let's suppose we use a distance criterion to select the proper level of detail. Figure 2.5a presents the most detailed approximation, which is suitable when the viewer is close to the object. When the object is further away and it is just covering a few pixels, we do not need such a detailed approximation. Instead, we can use a simplified version that, due to distance, looks approximately the same as the highly detailed version. As a consequence, a significant speedup can be expected. In conclusion, a multiresolution model must be capable of extracting the appropriate approximations in different scenarios. Furthermore, it must change the level of detail in a fast and efficient way, without overloading the system. That is, if the time required to extract a low level of detail and visualize the result exceeds the time required to visualize the higher level of detail, then there is no point in using a multiresolution model [PS99, RLB+02].

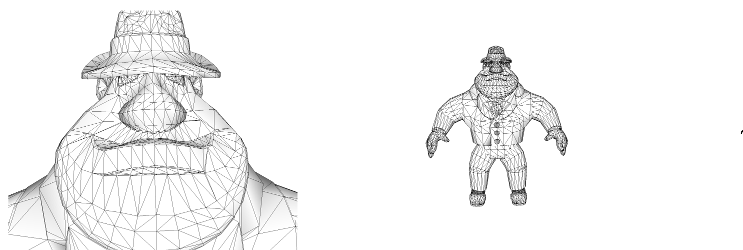


Figure 2.5: Visualization of a model using different views. From left to right, a) Close view, b) Normal view and c) Distant view

Formally, a multiresolution model consists of a representation that stores a range of approximations of an object and that allows to obtain any of them as required [Cla76]. The cost of extracting those approximations must be low as normally many of them will be needed in execution time. It is also important that the size of the multiresolution representation does not exceed very much the size of the object in its most detailed approximation.

Over the last years graphics hardware has undergone a real revolution: not only its computational power has been remarkably increased, but also its

cost has decreased, thus facilitating its availability in any computer in the market. Both aspects, its easy availability and the high processing capacity reached during these years, have driven the developers to make the most of graphics hardware with new objectives, from the production of videogames and computer-generated films to computer-aided design and scientific visualization, or even to solve problems unrelated to computer graphics [GPG07]. Apart from powerful and cheap, graphics hardware has also become flexible, that is, from being a simple memory device to being a configurable unit and, finally, to becoming a parallel processor totally programmable [FHWZ04]. As we will see in the following sections, many authors have resorted to graphics hardware to offer multiresolution approaches that offer better performance and improved visual quality.

2.4.1. Discrete multiresolution models

The simplest method to create a multiresolution model is to generate a fixed set of approximations. In a given instant, the graphics application could select the approximation to visualize. In this case, a series of discrete levels of detail would be being used, as seen in Figure 2.1. This multiresolution model would consist of a set of levels of detail and some control parameters to change between them. The simplicity of this kind of models is its main characteristic. If good approximations of the original mesh can be produced, then a discrete multiresolution model can be generated and successfully applied. Free visualization systems like OSG and commercial ones like Renderman [Ups90], Open Inventor [Wer94] or IRIS performer [JR94] include support for discrete levels of detail. Some 3D games engines such as Ogre3D or Shark3D, also include this feature.

As the number of polygons can differ considerably between two approximations, so does its visual appearance. When changing between levels of detail, it can produce an effect known as *popping*. Specifically, this effect consists in a perception of the change in the visual quality when changing between the different approximations. Despite this limitation, discrete multiresolution models are useful in many applications.

However, the problem of the *popping* artifacts previously mentioned limits them considerably. There are two general solutions that try to lighten this problem: *blending* and *geomorphing*.

Blending entails softening the transitions between the levels of detail by using transparencies, that is, a smooth interpolation of the images [TAF92, GW07]. Nevertheless, the use of this technique causes a significant increase of the visualization cost since the graphics system must visualize two levels of detail at the same time. The other alternative, *geomorphing*, involves interpolating the geometry of two consecutive levels of detail in several *frames* [Zac02]. There are also hardware-based solutions that try to minimize the overload generated by these methods [SG03]. Apart from the overload problem, this type

of models also require a considerable storage space, as each approximation is stored independently, which implies that the number of levels of detail stored should be small. The greatest limitation is that in runtime the levels of detail available are quite limited. That is, the visual display unit must select one of the approximations available, even when it needs an intermediate level of detail. This means that either a model is visualized without enough detail, sacrificing the quality of the image, or a model with excessive detail is selected, wasting processing time.

Another method recently presented [BS05] consists in sending to the GPU a mesh at minimum level of detail and applying later a refining pattern to every face of the model. That pattern is stored in the GPU. The problem is that each pattern corresponds with a different level of detail and as a result we have a discrete model that, according to the authors, suffers popping effects. Another aspect is the load suffered by the GPU when a model does not change the level of detail, as a pass must be made for each face the coarser model has.

2.4.2. Continuous multiresolution models

Instead of creating individual levels of detail, continuous multiresolution models present a series of continuous approximations of an original object. The simplification method employed offers a continuous flow of simplification operations to progressively refine the original mesh.

The main advantage of these models is their better granularity, that is, the level of detail is specified exactly, and the number of visualized polygons is adapted to the requirements of the application. This granularity is usually of a few triangles, as the difference between contiguous levels of detail is usually of a vertex, an edge or a triangle. Moreover, the spatial cost is lower since the information is not duplicated. Obviously, the management of the level of detail in these models is an essential point, that is, the amount of time required to visualize a level of detail should not never exceed the time required to visualize the object at its maximum resolution.

Characteristics and classifications

Several studies [Gar99, PS99, RLB+02] consider important the following characteristics in a continuous multiresolution model:

- Spatial cost similar to the original object: it is important that the storage cost of the multiresolution representation does not surpass excessively that of the original object (usually, two or three times size of the original object).

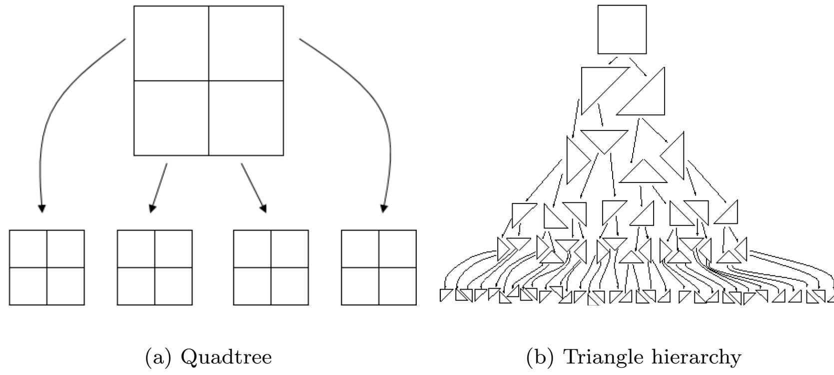
- Continuous surface: any polygonal surface extracted from the model must not have discontinuities.

- Gradual transition between approximations: transitions between levels of detail must be gradual, without abrupt changes.
- Without loss of information: the model must be capable of representing accurately the original object.
- Efficient processing of the information: it is essential that the multiresolution model recovers any level of detail as fast as possible.

A possible division of continuous multiresolution models is based on their structure. In this classification we can find two main groups:

- Incremental models: they depend on the simplification technique used in their construction, mainly methods based on local modifications of the mesh [Hop96, ESAV99, RAO+00, BRR+01, Ste01, SP03]. The structure of incremental multiresolution models is obtained by storing the evolution of a mesh through a series of local modifications (see Section 2.2.2). Starting from the most detailed mesh, each operation or simplification step generates a new set of triangles that appears in the mesh after the simplification. These models usually encode the approximations with the coarser mesh and the sequence of operations that allows to generate all the approximations; that is why they are called incremental (but also historical [PS97]) models. All historical models are constructed with this kind of techniques. However, there are differences between them, among which the following stand out: the simplification criterion used to generate the model, the amount of information stored, and the operations supported, as well as the efficiency of the algorithms implementing them. The MT or multitriangulation model [DFMP98] is a general scheme for incremental models.
- Hierarchical models: starting from the idea that the resolution of a mesh can be refined recursively by dividing an area into small portions of it, these models are said to be based on consecutive subdivisions of the mesh. Furthermore, the hierarchy of the subdivided areas can be described with a tree [DFMP98, EMB01, BS05, JWLL05, Tur07]. Among hierarchical models we can highlight those based on *quadtrees* and on triangle hierarchies. The first ones are based on the nested subdivisions of regular surfaces. Therefore, they are adequate for regularly arranged surfaces, such as terrains, see Figure 2.6a. On the other hand, triangle hierarchical models are based on the recursive division of the regular space into triangles. In Figure 2.6b we can observe how the binary tree is generated by dividing each triangle. In [ZS00] a wide study of this kind of models is presented.

It is also possible to establish a characterization of continuous multiresolution models depending on the resolution they are capable of showing. Thus, multiresolution models can be divided into two groups:



(a) Quadtree (b) Triangle hierarchy

Figure 2.6: Hierarchical models.

- Uniform resolution models: they always visualize the same level of detail in the whole object.
- Variable resolution models: they allow us to extract and visualize different resolutions throughout the surface of the object. They are also known as view-dependent models when image-space is used to guide the selection of the level of detail for the mesh.

This latter classification is the one that we have adopted in this article to present the works in level-of-detail modeling.

Uniform resolution models

Uniform resolution approaches are characterized by extracting and showing only one level of detail throughout the whole object. These models are very often used in computer games as they are very fast and simpler to implement than variable resolution ones.

One of the first models to offer a neat solution to a continuous representation of polygonal meshes was Progressive Meshes [Hop96]. From version 5.0, it has been included in Microsoft Corporation's DirectX graphics library. Later, its author [Hop97] and other researchers [PH97, PR00, SSGH01] improved the original model.

Progressive Meshes simplifies a mesh $M = M^n$ in consecutive approximations M^i by applying a sequence of n edge collapses:

$$M = M^n \overset{Collapse_{n-1}}{\rightarrow} M^{n-1} \overset{Collapse_{n-2}}{\rightarrow} \dots \overset{Collapse_0}{\rightarrow} M^0 \tag{2.1}$$

The reverse operation can also be performed, recovering more detailed meshes from the simplest mesh M^0 by using a sequence of *vertex splits*:

$$M^0 \stackrel{Split_0}{\rightarrow} M^1 \stackrel{Split_1}{\rightarrow} \dots \stackrel{Split_{n-1}}{\rightarrow} M^n = M \quad (2.2)$$

A further improvement was the development of models based on primitives which implicitly store connectivity information. Thus, storing the meshes as triangle strips or fans considerably reduces the storage cost of a mesh, the amount of information sent to the graphics system and it also accelerates the visualization [ESV96b].

One of the first models to use implicit connectivity primitives was MOM-Fan [RAO+00]. It employed triangle fans both in its data structure and in its visualization process. The main drawback lies in the high number of degenerate triangles produced in the representation, although they can be eliminated before the visualization. Another disadvantage of this model is that the average number of triangles that make up each triangle fan is small [AM04], losing the main advantage of using this kind of primitive.

At a later date the MTS model appeared [BRR+01]. This solution uses the primitive triangle strip both in the data structure and the rendering algorithm. Its core idea consists in using a collection of multiresolution triangle strips, each of them representing a triangle strip for each level of detail. All this is encoded as a graph, which involves a high storage cost. Moreover, the extraction of the level of detail is also a high time-consuming task.

Evolution of graphics hardware has given rise to new techniques that allow us to accelerate multiresolution models. Thus, new papers designed to minimize the traffic between the CPU and the GPU appeared later on. These models try to make maximum use of the cache of vertices of the GPU, minimizing traffic as possible [Cho97, Hop99, BG99], even reducing the pixel redrawing that does not contribute to the final scene [NBS06]. The main idea is to organize the meshes in short and parallel strips, that is, linked along many edges that share as many vertices as possible. The work presented by Chow [Cho97] allows different regions of a geometric model to be compressed with variable precision depending on the level of detail present at each region.

There are also several methods based on points as graphics primitives [ZMK02, DVS03]. The latter proposes a method that converts a hierarchy of points and polygons into a linear list easily rendered by graphics hardware and with a minimum load for the CPU. The problem of this method is the impossibility of applying a hierarchical culling, given the nature of the data structure used.

Ji et al. [JWLL05] suggest a method to select and visualize several levels of detail by using the GPU. In particular, they encode the geometry in a quadtree based on a LOD atlas texture [LPRM02]. In the first pass, the atlas is read as a texture in the GPU, where the level of detail is selected with a pixel shader that dumps the result into a buffer. Subsequently, that buffer is read in the CPU and the vertices selected are checked. Finally, another map with the vertices to be visualized is sent to the GPU, where all the vertices are encoded in a texture, avoiding sending them every time by the bus. The visualization primitive used in this solution is the triangle fan. The problem of this method

is that the CPU must execute in every change of level of detail, testing the selection maps and creating the new map of vertices which is sent to the GPU. Moreover, if the mesh is too complex, the representation with quadtrees can be not very efficient and even the size of the video memory can be an important restriction.

Variable resolution models

As we have previously commented, the concept of variable resolution involves showing different levels of detail of the same object in a given moment. In Figure 2.7 an example is shown where a sphere, interactively placed by the user, selects an area to be represented at maximum level of detail. This area is rendered at highest detail while the rest of the object is rendered with the minimum possible resolution, always assuring the connectivity of the mesh. Consequently, in the same object, two sets of faces can be observed, ones at maximum level of detail and others at minimum.

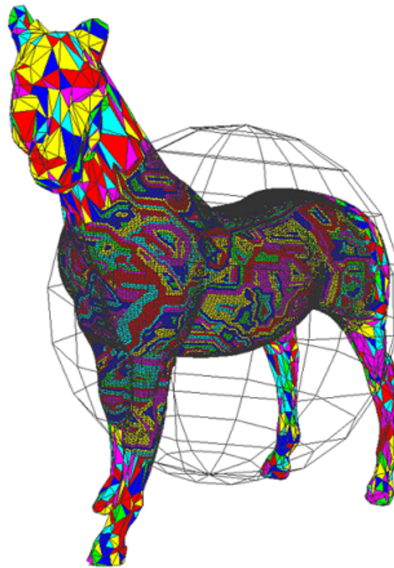


Figure 2.7: Variable resolution in the horse model.

HDS [LE97] and MT [DFMP98] were conceived as triangle-based multiresolution models, although they are both general schemes applicable in the creation of any multiresolution model. The first of them consists of a tree data structure that stores simplification sequences, and the second one of an acyclic graph. There are several models of this type [Hop97, XESA97].

FastMesh [Paj01] introduced a hierarchical framework which offered efficient algorithms and error metrics for both extraction and simplification. Nevertheless, the authors stress their interest on including triangle strips for faster rendering.

One of the first models to use triangle strips was VDPM [Hop97]. In general, this model determines the triangles to be processed and then turns them into triangle strips to visualize them. This task consumes a considerable amount of time, but the final performance in the visualization improves because of the acceleration attained by using this sort of graphics primitive. Later on, Skip Strips [ESAV99] was the first model to store the triangle strips in its data structure. Triangle strips are initially calculated and, afterwards, simpler versions of them are rendered. Skip Strips also uses a series of filters in the visualization to eliminate degenerate triangles. According to [SP03], this model also presents a high storage cost.

DStrips [SP03] is another model based on triangle strips. It calculates the triangle strips to be visualized in execution time, but it only processes the strips affected by the change of level of detail, without recalculating all the strips in the mesh. This mechanism reduces the time of extraction of the level of detail. Nevertheless, according to the results published, it is still a considerable temporal cost. Besides, its data structure is complex and it has a high spatial cost. Another approximation is Tunneling [Ste01], spread by Porcu [PS03, MP05]. Essentially, it involves using an algorithm that allows us to connect triangle strips that become more and more simplified and thus obtain strips with many triangles and reduce their total number. Its main problem is the time required to carry out these operations, which implies a high extraction cost of the levels of detail.

Present representations and extraction algorithms are not scalable for models made up of tens or hundreds of millions of polygons. The extraction cost is proportional to the size of the model, and it can be prohibitive for massive models.

Massive or *out-of-core* models are multiresolution approaches which have been specifically developed for meshes which exceed by far the capacities of present hardware. Therefore, only parts of the original mesh can be used for visualization. Given that these meshes surpass the capacity of the main memory, specific multiresolution techniques must be adopted for rendering in real-time this type of meshes. In general, these models create a simplification hierarchy in external memory that will be employed later for the online visualization of the required approximation of the original mesh.

The first paper directed at the visualization of massive models suggested the use of levels of detail in external memory [ESC00]. This model generated in a pre-process a simplification hierarchy based on edge collapses that will be used later in the visualization. Afterwards works were presented, improving substantially the spatial cost by creating new formats of disk storage and the performance in visualization by using efficient data structures [DP02].

The model introduced by Erikson et al. [EMB01] was developed to manage very large environments, and was later improved by Lakhia [Lak04] to offer better results in dynamic scenarios. Hierarchical models are essentially discrete models and they suffer from *popping* artifacts when the scene changes in a fast way. Lakhia tries to reduce these effects by means of certain heuristics that advance the load of levels of detail before they are visualized. Another problem of hierarchical models is that the quality of the simplifications diminishes as the simplification hierarchy becomes deeper.

More recent models are directed towards exploiting the graphics hardware with massive models [SM05]. In general, the previous models does not have a gradual transition between the different levels of detail. They are mainly based on updating the level of detail before the change is perceived. This model performs *geomorphing* in the GPU gradually, avoiding the typical *popping* effect. This model doubles the number of buffers used in the GPU when it performs geomorphing at coarser levels of detail, besides performing a great amount of rendering calls.

A model that incorporates simplification based on vertex hierarchy, *out-of-core* functions and occlusion culling is [YSG05]. This model shows in its results a lower spatial cost than other models such as [DP02] or [Hop97]. It is based on a cluster hierarchy of Progressive Meshes, where each cluster is made up of several nodes, and, in turn, each of these nodes contains a progressive representation of a portion of the original mesh [Hop96]. This allows us to improve the performance through a hierarchical *culling*.

2.4.3. Characterization

In Table 2.2 we can observe a description of the multiresolution models for arbitrary polygonal meshes considered in this review. The description takes into account several aspects:

- Type: it indicates whether the model is discrete or continuous.
- Primitive: the sort of primitive the model uses in the visualization.
- Resolution: whether the model allows us to visualize various resolutions in the same mesh (variable) or it is only possible to see a level of detail throughout the approximation (uniform).
- Simplification: it indicates if the simplification technique used in the construction of the model is incremental or hierarchical.
- GPU: it indicates if the model analyzed uses any of the latest capacities of present GPUs.
- Massive: whether the model is prepared for extremely big meshes.

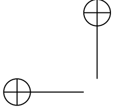
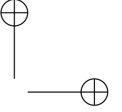
2.5. Conclusions

The use of a discrete or continuous multiresolution model in an application depends on its requirements. Discrete multiresolution models are a good alternative in applications which require few levels of detail in their objects. These models also offer a simple implementation and these levels of detail can be polished and improved considerably. Nevertheless, continuous models offer a high granularity, allowing us to adapt exactly the level of detail desired in each case. Furthermore, their evolution has allowed us to obtain very low level-of-detail extraction times and significant accelerations due to the irruption of the new and powerful GPUs.

As previously commented, multiresolution models can be divided into uniform and variable resolution. Although variable resolution schemes are powerful and very flexible, they usually have a high storage cost and they are usually slower than uniform ones. Obviously, requirements will determine its applicability. The design of schemes for specific applications, such as computer games or interactive visualization of vegetation, is an open line.

Model	Type	Primitive	Resolution	Simplification	GPU	Massive
Prog. Meshes [Hop96]	Continuous	Triangles	Uniform	Incremental	No	No
VDPM [Hop97]	Continuous	Strips	Variable	Incremental	No	No
HDS [LE97]	Continuous	Triangles	Variable	Incremental	No	No
MT [DFMP98]	Continuous	Triangles	Variable	Hierarchical	No	No
Skip Strips [ESAV99]	Continuous	Strips	Variable	Incremental	No	No
MOMFan [RAO+00]	Continuous	Fans	Uniform	Incremental	No	No
ExtVD [ESC00]	Continuous	Triangles	Variable	Incremental	No	Yes
MTS [BRR+01]	Continuous	Strips	Uniform	Incremental	No	No
Tunneling [Ste01]	Continuous	Strips	Variable	Incremental	No	No
Fastmesh [Paj01]	Continuous	Triangles	Variable	Hierarchical	No	No
HLods [EMB01]	Discrete	Triangles	Variable	Hierarchical	No	Yes
XFastMesh [DP02]	Continuous	Strips	Variable	Hierarchical	No	Yes
Geomorph [Zac02]	Discrete	Strips	Uniform	Hierarchical	No	No
DCLod [ZMK02]	Dis. & Con.	Point	Variable	Incremental	No	No
CLodGPU [SG03]	Discrete	Triangles	Uniform	Incremental	Yes	No
DStrips [SP03]	Continuous	Strips	Variable	Incremental	No	No
SequentialPoint [DVS03]	Continuous	Point	Variable	Incremental	No	No
EfficientHLod [Lak04]	Discrete	Triangles	Variable	Hierarchical	No	Yes
Prog. Buffers [SM05]	Discrete	Triangles	Variable	Incremental	Yes	Yes
Prog. Mult. Meshes [KG05]	Continuous	Triangles	Dis. & Con.	Incremental	No	No
QuickVDR [YSG05]	Continuous	Triangles	Variable	Incremental	No	Yes
MeshGPU [BS05]	Discrete	Triangles	Variable	Hierarchical	Yes	No
DynamicLOD [JWLL05]	Continuous	Fans	Variable	Hierarchical	Yes	No
SWPM [Tur07]	Continuous	Triangles	Variable	Hierarchical	Yes	No

Table 2.2: Characterization of different multiresolution models.

CHAPTER 3

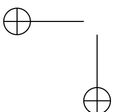
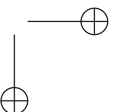
Level of Detail using Triangle Strips

In this chapter, we present a general framework to improve existing multiresolution models. It makes use of new implicit connectivity primitives such as triangle strips. It represents a mesh as a set of multiresolution triangle strips and maintains them both in the data structure and in the rendering stage. This approach offers features such as connectivity exploitation, uniform and variable resolution, fast extraction and low storage cost.

3.1. Introduction

As presented in the previous chapter, one possible solution to reduce the amount of information to be processed by the graphics system consists in using some kind of simplification or approximation of complex polygonal models. Management of the level of detail of an object implies to represent it by means of a surface description that allows the reconstruction of several approximations. Multiresolution models consists of a representation that stores a range of approximations of an object and that allows us to obtain any on them as required [Cla76].

Multiresolution models are often represented by triangle meshes. However, large triangle meshes are difficult to render at interactive frame rates due to the large number of vertices to be processed by the graphics hardware. Although each triangle can be specified by three vertices, it is desirable to order the triangles so that consecutive triangles share an edge. Such ordered sequences

of triangles are referred to as triangle strips. Using such an ordering, only the incremental change of one vertex per triangle need be specified, potentially reducing the rendering time by a factor of three by avoiding redundant transformation, clipping, and lighting computations. Therefore, multiresolution schemes can be improved by using triangle strips. Storing the model as triangle strips reduces the amount of information sent to the graphics pipeline and increases the rendering frame rate. In summary, modeling a mesh as a collection of strips or fans of triangles avoids storing and sending a large amount of redundant information to the graphics system. Hence, it involves better visualization times and lower storage cost.

With the aim of improving existing multiresolution models, here we present a general framework [RCBR04]. The proposed model makes use of new implicit connectivity primitives such as triangle strips. It represents a mesh as a set of multiresolution triangle strips. This approach offers the following features:

- Continuity. Transitions between levels of detail are smooth. They mean eliminating or adding one vertex.
- Connectivity exploitation. The model is based on the use of triangle strips both in the data structure and in the rendering stage. This leads us to a reduction in the storage and rendering costs.
- Uniform and variable resolution. This scheme permits both uniform and variable resolution.
- Fast extraction. It avoids the intensive use of the CPU that usually takes place with continuous multiresolution models.

In this chapter, section 3.2 shows how our approach is constructed. After that, data structures and algorithms are explained in section 3.3 and 3.4, respectively. Later, in section 3.5 we present the results obtained from this approach. Finally, in section 3.6, conclusions are presented.

3.2. Construction of the model

Nowadays, the most available sources of geometric data are polygonal meshes composed of triangles. Construction of the model is performed by a preprocess consisting of different stages that allows us to obtain a multiresolution representation from a polygonal mesh. Thus, the model is able to retrieve, at anytime, the level of detail required by the application. The data flow diagram of the construction process is shown in Figure 3.2.

In general, the construction of a continuous multiresolution model based on triangle strips requires two fundamental tasks. On the one hand, it is necessary to use a simplification algorithm which allows us to obtain a sequence of operations to generate the different approximations from an object. On the other

hand, we also need an algorithm capable of converting a polygonal mesh that is usually composed of triangles into triangle strips.

3.2.1. Simplification

There exist different methods for polygonal simplification, for an overview see section 2.2. The simplification method applied here is an incremental method, that is, it is based on a sequence of local updates. For every simplification step, the mesh is reduced in size and precision. In this context, we can underline the work on [HDD+93], where the edge collapse and vertex split local operations were presented. A sequence of n edge collapse operations is applied to simplify an arbitrary mesh M^n to a much simplifier mesh M^0 of the same topology, reducing the number of vertices by n. Reciprocally, M^n can be reconstructed by applying n vertex split operations to the base mesh M^0.

In particular, we use the half edge collapse variant of this operation of simplification. This variant collapses one vertex into another one, that is, no new vertices are created because we always refer to an existing vertex to compute the operation. As shown in Figure 3.1, edge pc collapses into vertex p, producing the simplified mesh. In the complementary operation, vertex p can be split into edge pc, thus refining the mesh.

In order to obtain the simplification sequence, any metric can be used in our approach. We use a metric based on quadric errors [GH97]. In Figure 2.1 we can see the result of applying a simplification algorithm based on quadric errors to a mesh. This kind of algorithms receives a polygonal model composed of triangles as a data input and it outputs an ordered edge collapsing sequence. In order to obtain this sequence, the algorithm maintains a geometric error for each vertex. This error is calculated as the sum of the quadric of the distances to the planes lying adjacent to the vertex. As commented before, we deal with half edge collapses.

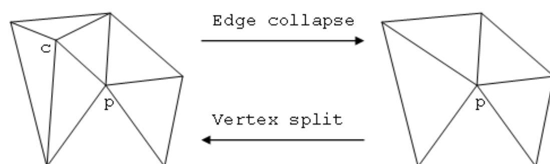


Figure 3.1: Half edge collapse and vertex split operations.

Thus, this type of simplification produces an ordered sequence of edges to be removed in the mesh to progressively obtain coarser meshes. This sequence can be seen as a series of edge collapse operations and their complementary operation, vertex split.

Formally, we define a polygonal mesh composed of triangles M, as:

$$M = \{V, T\} \tag{3.1}$$

where V is a set that contains all the vertices and T is another set that contains every triangle used to represent the mesh at the highest level of detail.

As shown in Figure 3.2, this process receives a polygonal mesh M={V,T}. Internally, it refines this mesh by applying a simplification algorithm. From this process, we obtain E, that is, the simplification sequence. It is important to stress that set E is an ordered set of edge collapses that allows us to simplify the polygonal mesh M. Taking into account that n is the number of levels of detail available, we can state that:

$$M^0 \xrightarrow{e_0} M^1 \xrightarrow{e_1} \ldots \xrightarrow{e_{n-2}} M^{n-1} \tag{3.2}$$

Therefore, every atomic operation of simplification applied to the multiresolution mesh will mean a different level of detail. Obviously, n will be less than the number of vertices in the mesh.

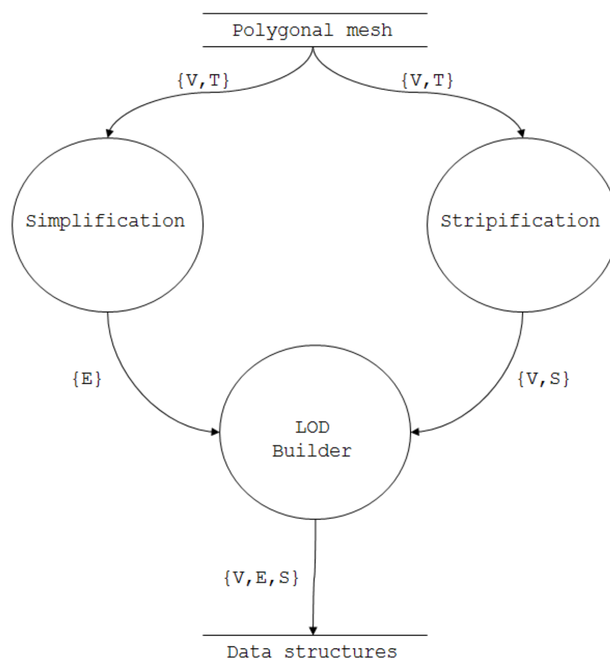


Figure 3.2: Data flow diagram of the model construction process.

3.2.2. Stripification

As mentioned in the previous chapter, 3D polygonal models, usually composed of triangles, are the most used in interactive graphics. However, triangle strips represent, in a compact way, the connectivity of a triangular mesh. An

ideal triangle strip codifies a sequence of n triangles using $n+2$ vertices, see Figure 3.3. This implies important savings over the codification based on triangles, which requires $3n$ vertices per triangle. Therefore, by using triangle strips we obtain improvements in storage costs.

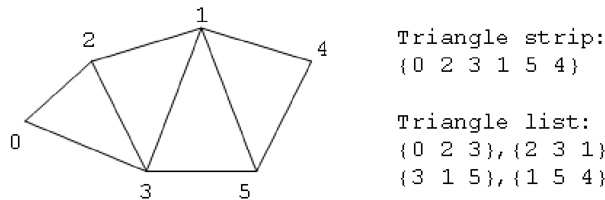


Figure 3.3: Simple geometry represented by means of triangle strips and triangles.

The process of converting a polygonal model into triangle strips is called *stripification*. As commented in the previous chapter, many algorithms implement this process. We notice that any stripification method can be applied to our approach.

In the approach that we propose, we use static stripification (see section 2.3.1). We prefer this kind of technique since we thereby avoid strip creation and destruction, which would imply an additional cost that would make the model less competitive. This is due to the fact that we should calculate the new triangle strips at each level of detail, which would penalize the level-of-detail extraction. Thus, the use of static strips will enable us to have a better implementation in the GPU, as we will see in chapter 5.

Thus, we process a mesh M={V,T}, by converting it into another {V,S}, where V contains all the vertices in the model and S is the set of triangle strips that compounds the model at the highest level of detail.

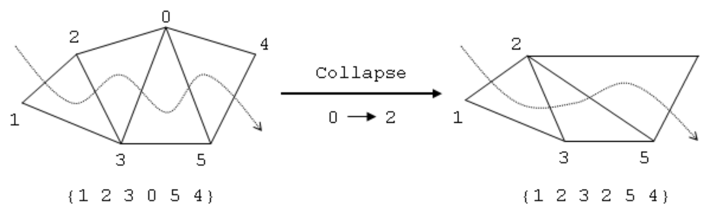


Figure 3.4: Edge collapse applied to a triangle strip.

3.2.3. LOD Builder

One objective of this model is to diminish the cost produced by the operations of edge collapse and vertex split into multiresolution triangle strips.

Moving among levels of detail implies a high temporal cost in extracting the geometry. Here, we pre-calculate the information that will change, when level-of-detail transitions take place, and thus store it in a suitable way. Later, we can query this information in runtime to quickly locate the vertices to be changed.

In Figure 3.4, an edge collapse is applied to a triangle strip. As we can observe, the result of this collapse can be represented by replacing every occurrence of vertex 0 by the vertex 2, that is, vertex 0 collapses to 2. As shown in Figure 3.5, we can generate different approximations or levels of detail by applying a sequence of edge collapses to a mesh represented by triangle strips.

As shown in Figure 3.2, this process receives two fundamental types of information with which to construct the multiresolution model. On the one hand, and from the simplification process, we obtain the simplification sequence E, that is, the information needed to transit among the levels of detail. On the other hand, the stripification process provides us with the mesh in triangle strips at the highest level of detail $\{V,S\}$.

With this information, it is already possible to build a multiresolution model. However, every time a change in level of detail is required, we have to search for the edges to be collapsed in each triangle strip. Obviously, this operation has a high cost in real time, and it will noticeably affect the performance of the application. With the aim of creating a more efficient multiresolution model, the *LOD Builder* process performs a fundamental task: it calculates and stores the information required to transit among the levels of detail.

The calculation of transitions consists in extracting every level of detail of the model, saving the number of strips, that changes in that level of detail and the place where the vertices to be processed are located. This precalculated information will allow a fast transit among LODs because it prevents time from being wasted on looking for the vertices to modify in the real-time application.

Specifically, we name the set of changes that modify the different triangle strips C. This set is an extension of set E and it is defined as:

$$C = \bigcup_{i=0}^{n-1} C^i \ , n > 0 \tag{3.3}$$

Where n is the number of LODs available, and C^i is the set of changes to be applied in the triangle strips at LOD i. Every item in C^i consists of the modifications to be applied in a particular triangle strip, that is, it stores the strip that changes and where collapses take place. Formally:

$$\forall\ C^i_j \ \in C^i, C^i \ = \{\ E^i, \{t^i_0, p^i_0\}\ , \{t^i_1, p^i_1\}\ , ...\}\ ,\ 0 \leq j < s \tag{3.4}$$

s being the number of strips to be modified at LOD i. Next, we detail the meaning of every tuple that compounds C^i_j:

- E^i: is the edge collapse to be applied at LOD i.

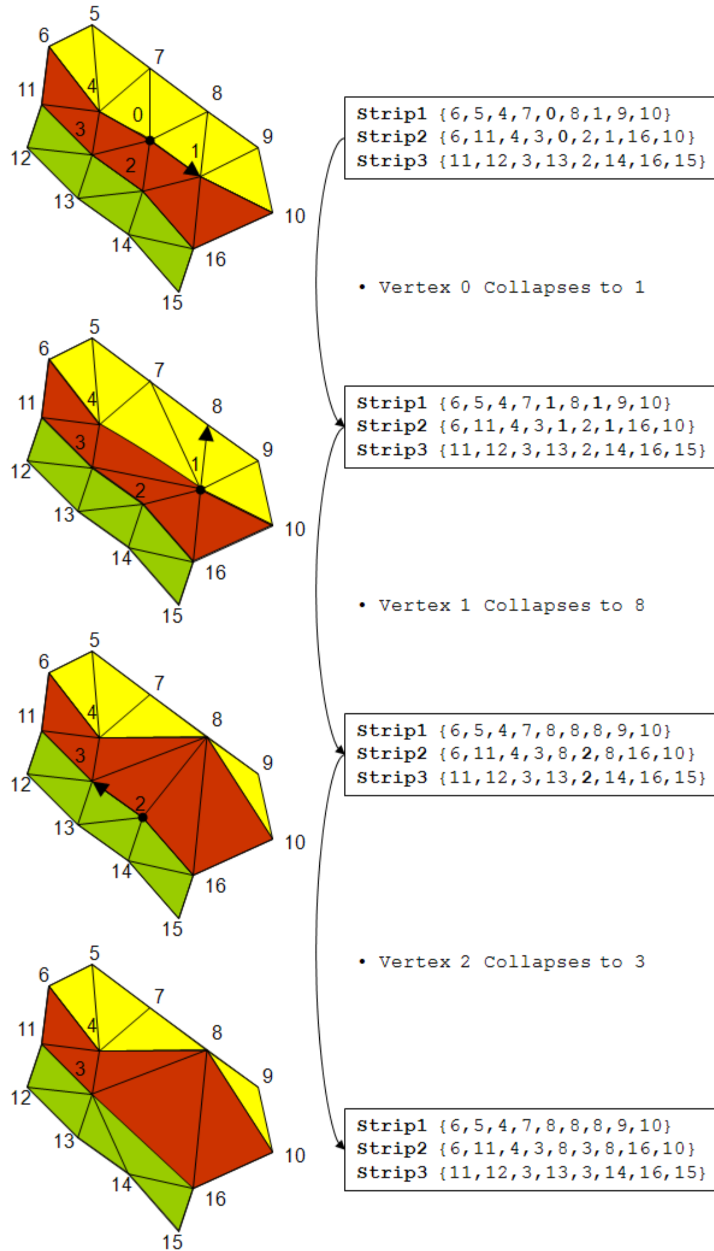


Figure 3.5: Sequence of edge collapses applied to a mesh represented by triangle strips.

- t_j^i: is a scalar that stores the index to one of the triangle strips that is modified at LOD i.
- p_j^i: is a scalar that informs us about the position, in the strip t_j^i, where vertices to be modified are located.

In Figure 3.6, we can observe a simple example where an edge collapse takes place from edge {0,1} to vertex 1 (notice that an edge always collapses to the second vertex of the same edge) which implies a modification in strips 0 and 1. Vertex 0 is located in position 4 within both strips. Information about set C comes wholly from the construction pre-process. In this pre-process, every modification produced in the triangle strips is calculated and stored in C. This process goes on its flow until the lowest LOD is reached, storing each and all of necessary changes to transit among the different LODs.

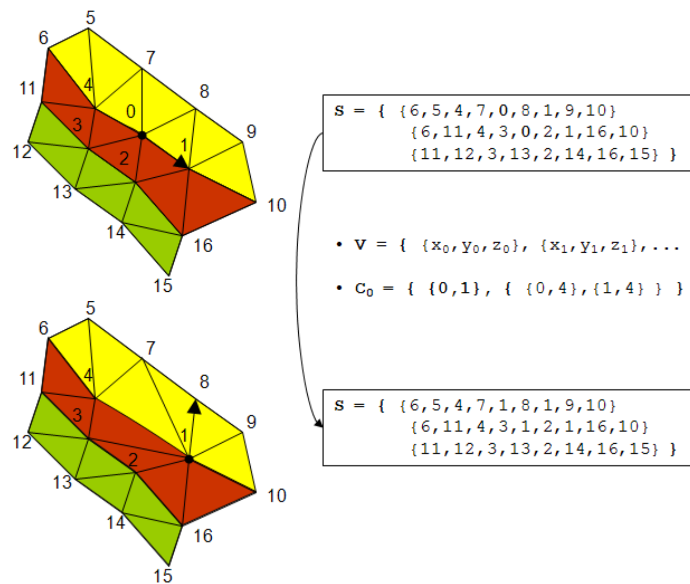


Figure 3.6: Sets of our approach.

Construction of the model finishes by storing the information in the data structures used to manage the LOD. We will introduce them in next section.

3.3. Representation

This model represents a mesh as a set of multiresolution triangle strips. Let M be the original polygonal surface and M^t its multiresolution representation. M^t can be defined as:

$$M^t = \{V, S^t\},\ 0 \leq t < n \tag{3.5}$$

where t and n are: any level of detail and the number of levels of detail available, respectively. V is the set of all the vertices and S^t is the triangle strips used to represent any resolution. Thus, we can express S^t as a tuple $\{S^0,C\}$, where S^0 consists of the set of triangle strips at the lowest level of detail and C is the set of changes required to simplify or refine them.

$$S^0 \stackrel{C_0}{\rightarrow} S^1 \stackrel{C_1}{\rightarrow} \dots \stackrel{C_{n-2}}{\rightarrow} S^{n-1},\ n : levels\ of\ detail \tag{3.6}$$

In order to efficiently represent the set C, we perform some operations in the sets of our model:

1. We order the vertices in the set V according to their simplification sequence.

2. We later update the set S so that it reflects a new vertex order.

3. Taking into account that we use the half edge collapse operation to simplify and that this operation collapses one edge into an existing vertex of that edge, we create an ordered set of vertices W. This set contains the vertices into which every vertex in V collapses to.

4. We finally remove the edge collapse information E from the set C as it is now implicitly stored in our model.

Once construction is completed, the data structures are fed and prepared for their use in runtime. A simple example of this can be observed in Figure 3.7. On the one hand, the *Vertices* data structure stores the 3D coordinates and one index to the vertex where it collapses to. On the other hand, *Triangle strips* store indices to the vertex that they are composed of. And, finally, the structure *Changes* saves the information required to quickly locate the vertices which are modified by moving between different levels of detail. A simple implementation in C++ is shown in Figure 3.8.

3.3.1. Theoretical spatial cost

Before calculating the spatial cost of the model, we assume that cost of storing a real, integer or pointer value as being a word. Thus, the cost can be calculated as a sum of the different components of the multiresolution model: V, W, S and C.

We suppose $|V|$ to be the number of vertices in the model. For each vertex we must store their 3D coordinates, and so the spatial cost for V is $3|V|$ words. On the other hand, W contains indices to the set V, considering that the number of indices will be at the most $|V|$, we can state that the spatial cost for W is $|V|$ words. In brief, the spatial cost, including V and W, is $4|V|$.

Vertices

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3D	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ	XYZ
p	1	8	3	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Triangle Strips

0	6	5	4	7	0	8	1	9	10
1	6	11	4	3	0	2	1	16	10
2	11	12	3	13	2	14	16	15	

Changes

0	0	4	1	4				
1	0	4	0	6	1	4	1	6
2	1	5	2	4				

Figure 3.7: Simple representation of the data structures.

```
struct Vertices {
  float *listOf3DCoordinates[3];
};
struct Collapses {
  struct Vertices *CollapseVertex;
};
struct Strip {
  integer *listOfIndices;
};
struct Strips {
  struct Strip *listOfStrips;
};
struct Change {
  struct Strip *stripToChange;
  integer Position;
};
struct ChangesPerLOD {
  struct Change *listOfChangesPerLOD;
};
struct Changes {
  struct ChangesPerLOD *listofChanges;
};
```

Figure 3.8: Data structures in C++ of our model.

In a polygonal mesh, if $|T|$ is the number of triangles and $|V|$ the number of vertices, according to Euler's formula, we can suppose that $|T|$ is approximately $2|V|$. On the other hand, the theoretical spatial cost for a mesh composed of triangle strips is $|T| + 2|S|$ words, where $|S|$ is the number of triangle strips. Obviously, $|S|$ depends on the algorithm applied and varies noticeably. Therefore, the spatial cost for S is approximately $2|V| + 2|S|$ words.

Finally, we must analyze the changes produced in the triangle strips that allow us to transit among the levels of detail. Empirically, we have proved that, on average, an edge collapse produces a modification in two triangle strips. However, as we do not remove vertices from the triangle strips, we accumulate these modifications to the next edge collapses where the collapsed vertex be involved in. Thus, taking into account that we have a maximum of $|V|$ levels of detail, we will store $2^{|V|}|V|$ records of C. Moreover, each element of C contains two words. Therefore, the spatial cost of C is $2|V|2^{|V|}$.

Hence, the spatial cost of the *LodStrips* multiresolution model is $4|V| + 2|V| + 2|S| + 2|V|2^{|V|}$ words, that is, $(6+2^{|V|})|V| + 2|S|$ words.

3.4. Rendering

In general, visualization algorithms try to efficiently manage complex scenes. Multiresolution modeling improves visualization by rendering the appropriate level of detail of an object or group of objects within the scene. The most-used criterion to select the level of detail is the distance from the observer to the object [Hop97]. Thus, objects closer to the observer are drawn at a high detail and distant ones at a low detail. Regarding resolution, on the one hand, if the entire object is drawn at the same detail, we are using *uniform resolution*. On the other hand, if the object is represented by different levels of detail coexisting along the same mesh, then we are using *variable resolution*. Often, the parameters of the view are used to select the part of the surface to be represented at a high detail and, in this case, it is called *view-dependent visualization*.

Our model is able to manage both types of resolution. Obviously, we need different algorithms to support them. In this section, we will examine both algorithms.

As previously commented, once the model has been constructed, we need algorithms to be able to support multiresolution capabilities. Our approach and indeed most multiresolution models have two main algorithms to carry out these tasks, i.e. a level-of-detail recovery algorithm and a drawing algorithm. We assume the rendering stage to be a stage that contains these two algorithms, which are applied in a sequential order: first extraction and then drawing.

3.4.1. Uniform resolution algorithms

In this section, we will explain the two main algorithms needed to render the multiresolution mesh at the appropriate level of detail. That level of detail will be the same in the whole mesh, that is, these algorithms are for uniform resolution rendering.

Level-of-detail recovery algorithm

This algorithm is responsible for extracting a level of detail from the triangle strips by applying the pre-calculated operations available in the data structure *Changes*. The model uses three data structures. From a visualization point of view, two of them are static: *Vertices* and *Changes*, that is, they never change in runtime, and the other one, the *Triangle Strips*, is dynamic. These are adapted to the level of detail required during visualization. Obviously, every time an operation is produced, it is applied to the triangle strips. In this way, triangle strips always have the geometry corresponding to the level of detail used in that time.

To illustrate this algorithm, we will see how the model transits between two consecutive levels of detail by generating the appropriate triangle strips for visualization. In Figure 3.9, we proceed to extract a coarser level of detail in the model. In order to clarify, to apply an edge collapse or vertex split operation means to change the level of detail. Of course, we could easily consider a level of detail as a group of operations. Thus, from the information calculated in the pre-process, we know that vertex 0 collapses to 1 (information about collapsing is stored in field p, vertex where it collapses to). Moreover, we also know the triangle strips that contain that vertex and the position within them. In the example, we need to change triangle strips 0 and 1, both in position 4, where vertex 0 is located. In Figure 3.10 we can see the algorithm that was implemented. With this algorithm, we are able to quickly transit between two non-consecutive levels of detail.

Drawing algorithm

The aim of the drawing algorithm is to send all the triangle strips to the graphics system. However, as the model moves to coarser levels of detail, the triangle strips begin to accumulate repeated vertices. Some multiresolution models use filtering [ESV99] to remove these vertices. Our work follows the same line. Most repeated vertices follow the pattern $a(aa)^+$ or $ab(ab)^+$, where a, b are indices to vertices of the model. Sending this kind of vertices is equivalent to sending degenerate triangles that do not contribute at all to the final scene but just add overhead to the graphics system. Thus, filtering or removing these vertices noticeably improves the model because less vertices to be processed are needed. Therefore, this algorithm also filters degenerate triangles so

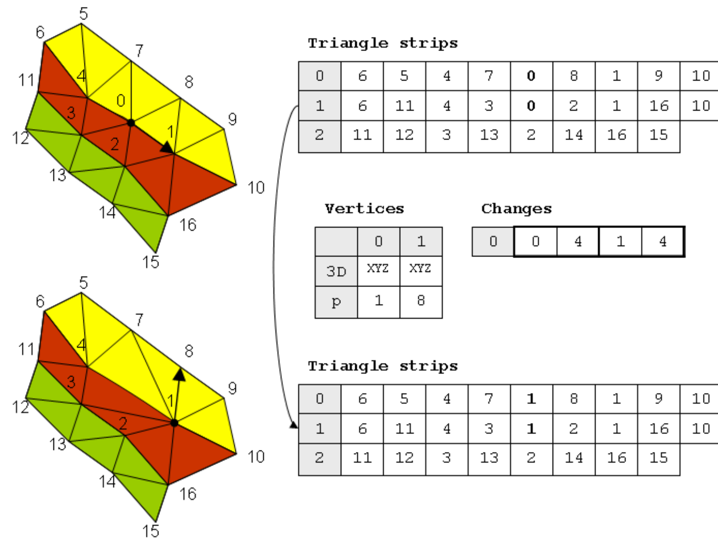


Figure 3.9: Simple example of a level-of-detail extraction.

```
Function  ExtractLevelOfDetail (newLOD)
  // To a coarser mesh
  if newLOD>currentLOD then
    for lod=currentLOD to newLOD
      for change=Changes[lod].Begin to Changes[lod].End
       change.ApplyCollapses(Strips);
      end for
    end for
  // To a more detailed mesh
  else
    for lod=newLOD to currentLOD
      for change=Changes[lod].Begin to Changes[lod].End
       change.ApplySplits(Strips);
      end for
    end for
  end if
  currentLOD=newLOD;
end Function
```

Figure 3.10: Uniform resolution level-of-detail recovery algorithm.

```
Function Draw()
  for strip=Strips.Begin to Strips.End
    strip.RemoveDegenerate();
    strip.Render();
  end for
end Function
```

Figure 3.11: Drawing algorithm.

as to prevent them from being sent to the graphics system. In Figure 3.11 we can see the algorithm that was implemented.

3.4.2. Variable resolution algorithms

The meaning of variable resolution consist in displaying different levels of detail on the same object. In Figure 3.12, a sphere interactively located by a user selects the area to be represented in high detail. This area is drawn at the highest resolution LOD, while the rest of the object is drawn with the coarsest resolution possible while maintaining the connectivity of the mesh. Thus, in the same object, two sets of faces can be observed: a set of faces with a low level of detail and another one with a high one.

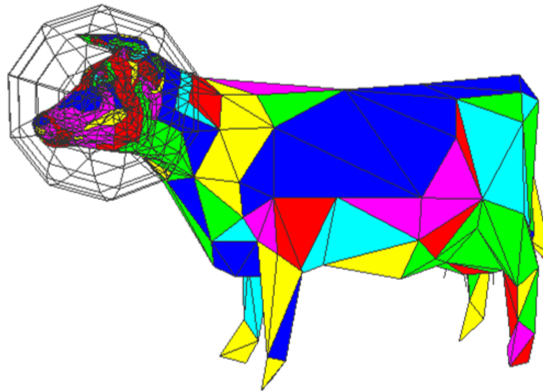


Figure 3.12: Cow model at different resolutions.

To retrieve a variable resolution LOD a criterion or set of criteria is needed. The criterion is used to decide which part of the object is simplified and which is refined. This decision could be taken by the user interactively, indicating which regions should be displayed with the higher or lower resolution, or by the application. The most-used criteria to represent areas at a high detail,

are [Hop97]:

- View frustum, increasing detail in the regions inside the view frustum. See Figure 3.13.
- Silhouette boundaries, increasing detail in the regions where there are edges for which one of the adjacent faces is visible and the other is invisible.
- Orientation surface, increasing detail at the regions oriented near the viewer.
- Screen-space projections, increasing or decreasing detail in the region depending on the length of its screen-space projection.
- Local illumination, increasing detail in a direction perpendicular to, and proportional to, the illumination gradient across the surface.

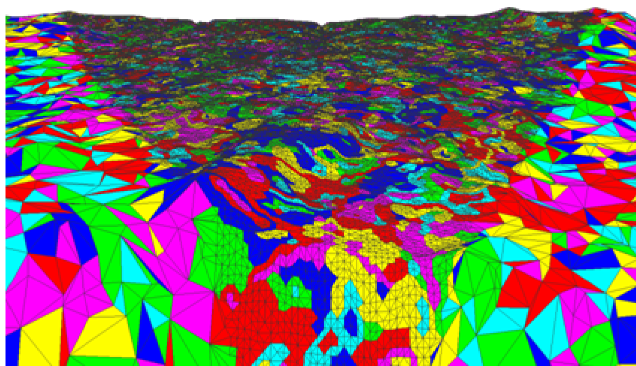


Figure 3.13: View frustum criterion applied to a terrain model.

These criteria can be easily added to our model because the algorithm is independent of the criteria used.

Level-of-detail recovery algorithm

In general, this algorithm traverses every vertex, and a test function is evaluated to decide whether the criterion is achieved. Depending on the result, the vertex included inside the test is processed or not, obtaining a mesh with different levels of detail. This algorithm is shown in Figure 3.14.

After the level-of-detail recovery algorithm has processed the multiresolution strips and the criteria have been applied to them, the drawing algorithm takes over, traversing each strip to obtain their vertices in order to send them to the graphics system. We can use the same algorithm mentioned in Figure 3.11.

```
Function ExtractLevelOfDetail (newLOD)
  for vertex=0 to NumberOfVertices-1
    if (test(vertex,newLOD)==TRUE) then
      for change=Changes[newLOD].Begin to Changes[newLOD].End
        change.ApplyCollapses(Strips);
      end for
    else
      for change=Changes[newLOD].Begin to Changes[newLOD].End
        change.ApplySplits(Strips);
      end for
    end if
  end for
end Function
```

Figure 3.14: Variable resolution level-of-detail recovery algorithm.

3.5. Results

In this section, we present the obtained results. Some tests were conducted to compare our approach to Progressive Meshes [Hop96] and Multiresolution Triangle Strips [BRR+01], from now on PM and MTS, respectively. The first of them has been, and still is, a reference model in the multiresolution modeling field. The second one was one of the first models to use triangle strips. These multiresolution models were coded in C++ and OpenGL. It is also important to underline that OpenGL immediate mode was used to render the models as it facilitates comparisons with other multiresolution models

Tests were applied to three standard meshes from the Stanford University Computer Graphics repository. This makes it easier to compare them to other existing models. The hardware used consists of a PC with an Intel Xeon 3.6 Ghz processor and 512 Mb RAM, and with an NVidia GeForce 7800 GTX (512Mb) graphics card.

We first analyze the spatial cost of our model, comparing it to other existing models. After that, we also analyze the temporal cost of the model both in uniform and variable resolution. As regards uniform resolution, we apply the tests proposed by Ribelles et al. [RCLH99]. In particular, we use the linear test, which extracts the levels of detail of the model in a linear and proportionately increasing or decreasing way. Finally, in order to test the variable resolution capabilities, we apply a plane test to the models. Plane test consist of a plane that goes through the object from one extreme to the other in a linear way, thus in front of the plain and behind it the level of detail is high and low, respectively (see Figure 3.15).

3.5.1. Spatial cost

Table 3.1 shows a comparison between the multiresolution models commented above. We can observe that our model offers a similar spatial cost to the other models. However, as the complexity of the model increases, the cost also grows considerably. This is due to the presence of the degenerate triangles in the data structures, which are in the triangle strips even though they are removed when rendering. It is important to underline that our model allows us to represent a model both in uniform and variable resolution, which affects spatial cost.

	#Vertices	#Faces	Progressive Meshes	MTS	Previous Approach
Cow	2,904	5,804	0.27	0.25	0.39
Bunny	34,834	69,451	3.28	2.96	3.56
Phone	83,044	165,963	7.86	6.76	8.10

Table 3.1: Some features and spatial costs (in MB.) for Progressive Meshes, MTS and our approach.

3.5.2. Temporal cost

Uniform resolution

Table 3.2 shows the results obtained after applying a linear test from the highest level of detail to the coarser. We extracted the one per cent of the available levels of detail. As commented before, the models in the comparison are: Progressive Meshes, MTS and the approach here presented. The total visualization time is shown first, while the lower part shows the percentage of this time used to extract the level of detail and the percentage used to draw the resulting mesh.

As shown in the table, the PM model is very fast in extracting the level of detail. However, its basic rendering primitive is the triangle, and for this reason the total visualization time is approximately three times higher than our approach. The MTS model also shows times that are around twice as high. It is because MTS produces more strips and it increases the cost to extract the level of detail.

In Figures 3.16, 3.17 and 3.18 a series of charts allows us to graphically analyze the performance of the three multiresolution models used in the comparison. The x axis shows the level of detail in the interval [0,1], zero being the highest level of detail and one the lowest one. Essentially, we observe that as our approach moves to lower levels of detail it sends more vertices than PM due to the accumulation of degenerate triangles in the triangle strips, . On the other hand, MTS sends fewer vertices because it breaks the triangle

Models	Progressive Meshes		MTS		Our Approach	
	Render (ms)		Render (ms)		Render (ms)	
	% Rec	% Drw	% Rec	% Drw	% Rec	% Drw
Cow	0.711		0.370		0.245	
	26	74	62	37	40	60
Bunny	6.625		4.070		1.956	
	3	97	51	49	37	63
Phone	16.265		8.839		4.705	
	2	98	38	62	34	66

Table 3.2: Linear test applied to the multiresolution models PM, MTS and our approach.

strips dynamically, but this task considerably affects the performance of the model. Our model offers a noticeable trade-off between vertices that are sent and performance in rendering. As shown in the charts, our approach obtains visualization curves that are always below the other models it is compared to.

Variable resolution

In order to test the variable resolution capabilities on our approach, we performed a plane test based on position along the x axis to the cow, bunny and phone models. In Figure 3.15, we can see a model simplified with the mentioned test.

In Figures 3.19, 3.20 and 3.21, we show the results of these tests. The higher parts show the vertices sent to the graphics pipeline while the plane is traversing the object. Plane position is in the interval [0,1], zero being the plane on the left of the object by showing the whole model at a high level of detail, and one being the plane on the right of the object by showing the whole model at a low level of detail. A 0.5 value means that the plane is located in the middle of the object by showing the left part at a high level of detail and the rest at a low one. In these figures, the lower parts show the frame per second rates reached while running the plane test.

As expected, the number of vertices sent decreases as plane traverses the x axis. In this way, frame per second rates increase as more surface at a low level of detail is processed.

3.6. Conclusions

We introduced an approach that improves the time required to extract different levels of detail in multiresolution schemes based on triangle strips [RCBR04]. Moreover, by means of a filter in the visualization, it removes most degenerate

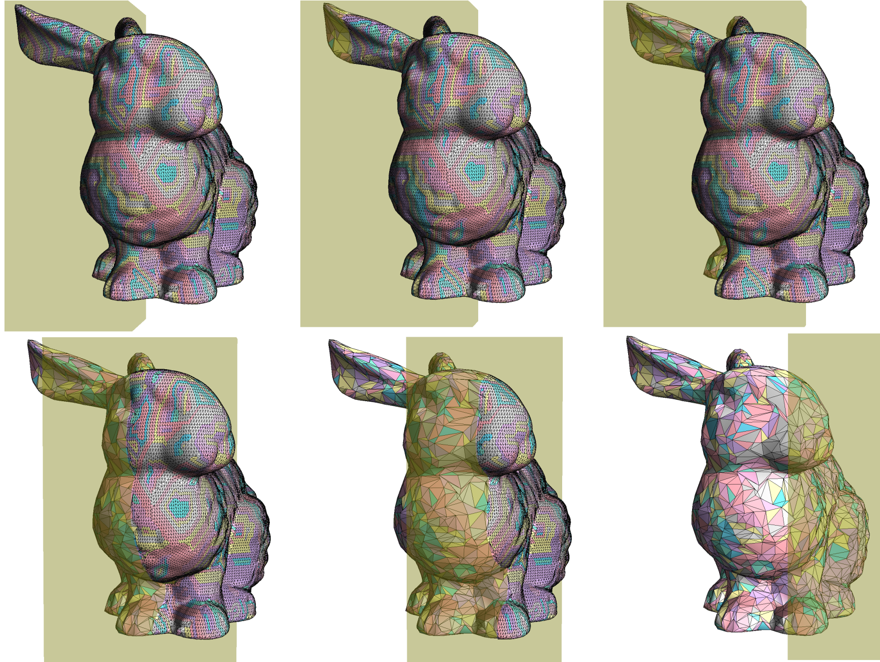
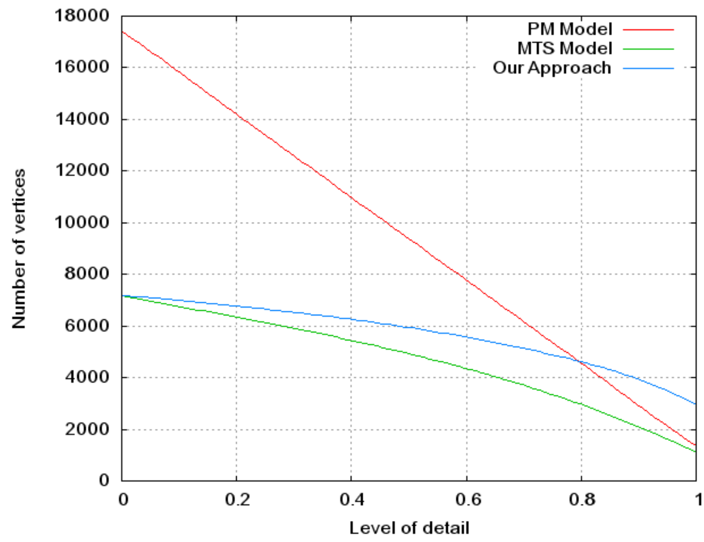


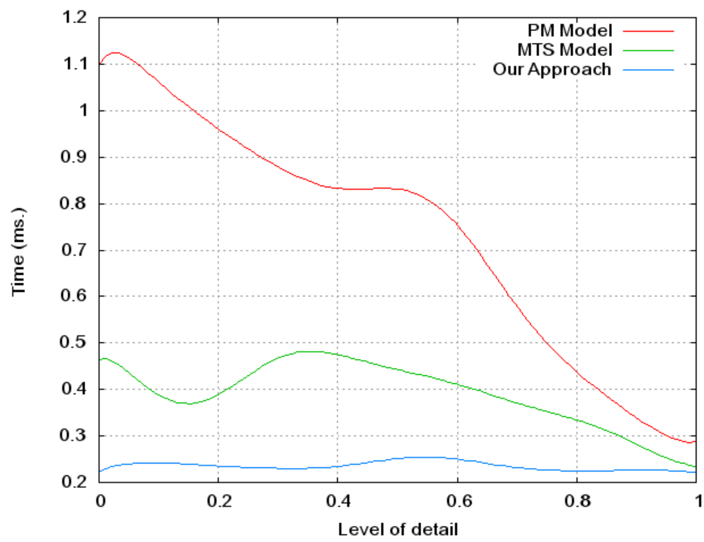
Figure 3.15: Plane test applied to the bunny model.

triangles produced in the lower levels of detail of the model. However, filtering these repeated vertices is a time-consuming task and it is applied every frame when rendering.

With respect to the temporal cost, the model presented here has a better cost than the models it has been compared with [RC04a]. Indeed, the time employed to recover the level of detail is quite low compared to other models based on triangle strips, like MTS, and similar to models based on triangles, like PM. Nevertheless, the model can be further improved as far as its extraction cost is concerned by removing more degenerate triangles and moving detection and elimination to the pre-process. Thus, we can previously detect and save that information and create a uniform resolution model faster than the approach here introduced, although we would lose variable resolution capabilities [RC04c, RCG04].

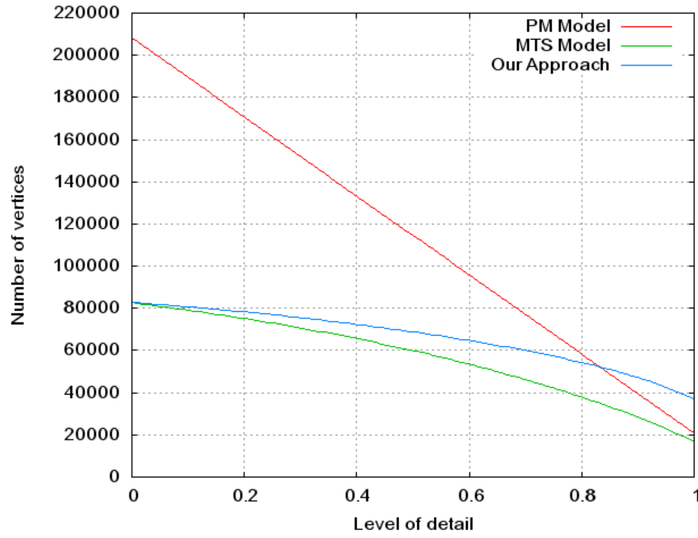


(a) Vertices sent per level of detail.

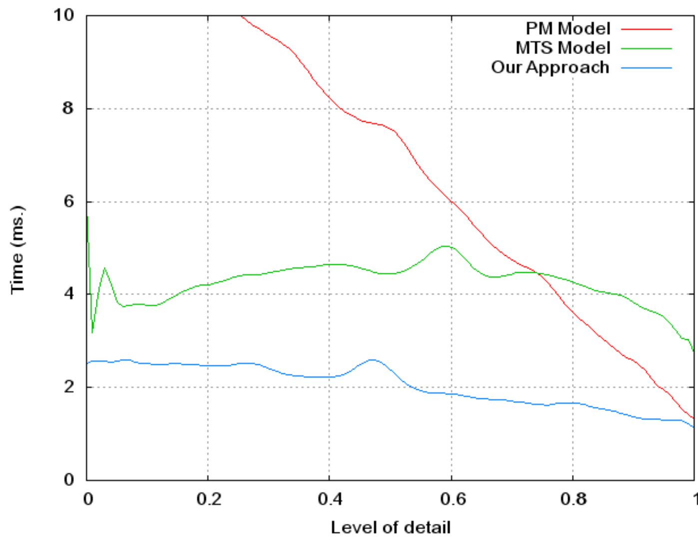


(b) Rendering times per level of detail.

Figure 3.16: Uniform resolution: linear test applied to the cow multiresolution model.

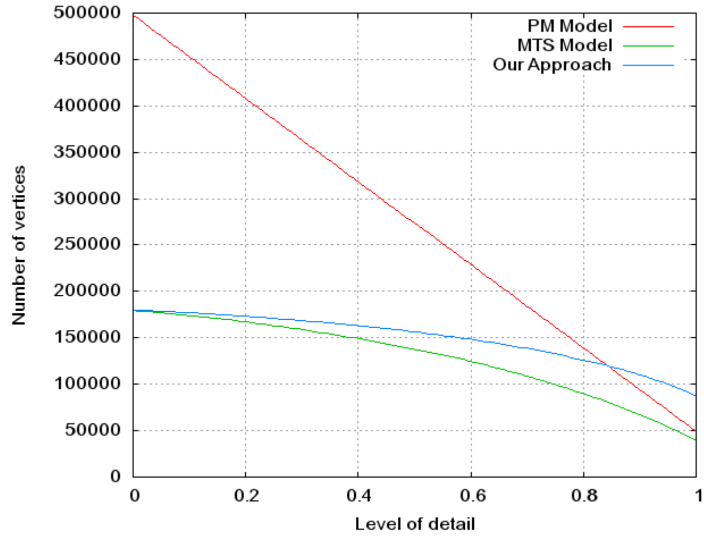


(a) Vertices sent per level of detail.

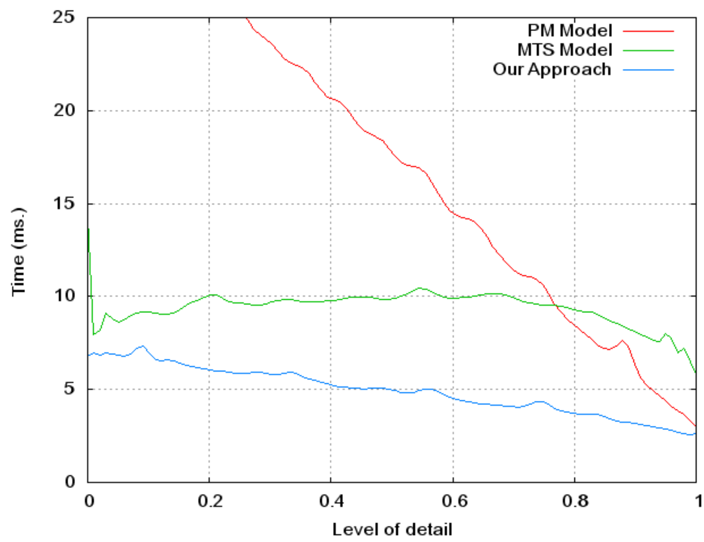


(b) Rendering times per level of detail.

Figure 3.17: Uniform resolution: linear test applied to the bunny multiresolution model.

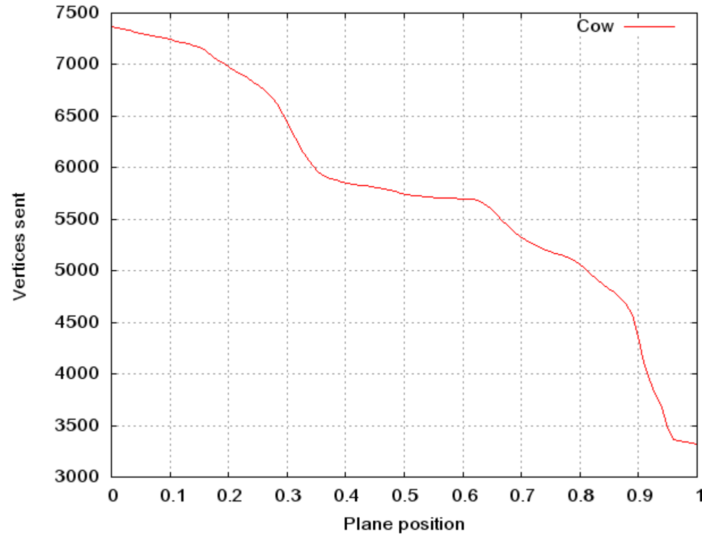


(a) Vertices sent per level of detail.

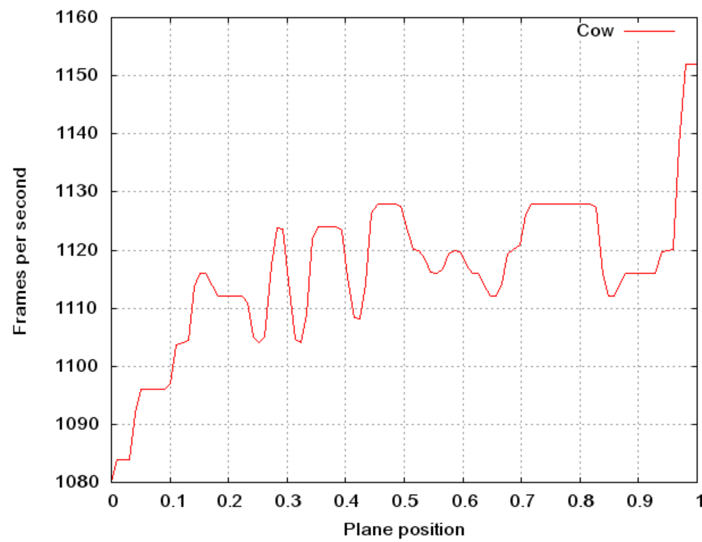


(b) Rendering times per level of detail.

Figure 3.18: Uniform resolution: linear test applied to the phone multiresolution model.

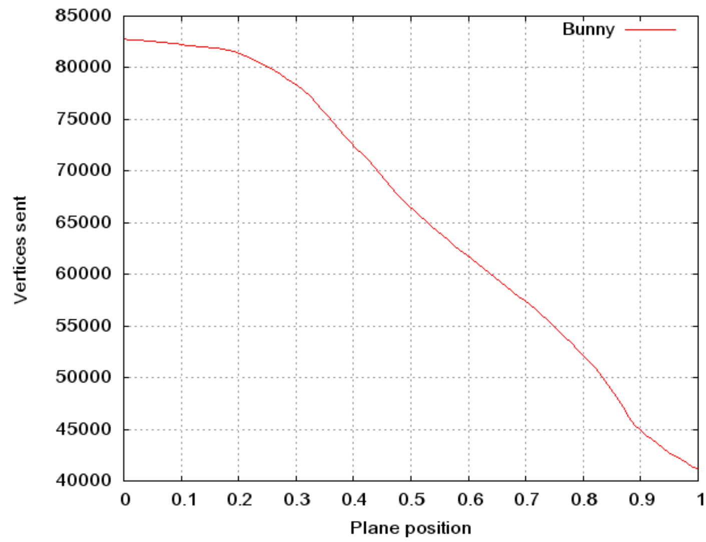


(a) Vertices sent during the plane test.

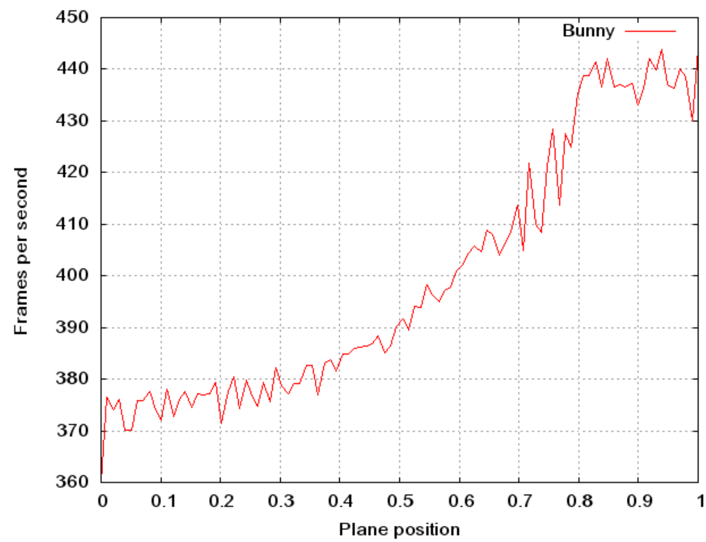


(b) Frames per second during the plane test.

Figure 3.19: Variable resolution: plane test applied to the cow multiresolution model.

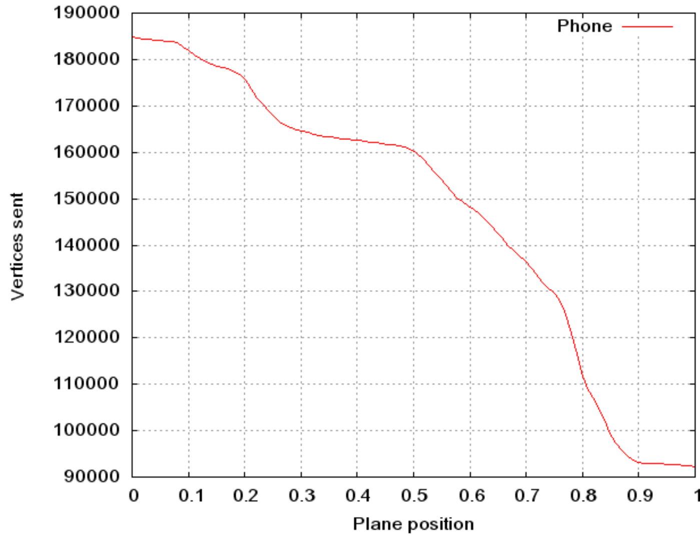


(a) Vertices sent during the plane test.

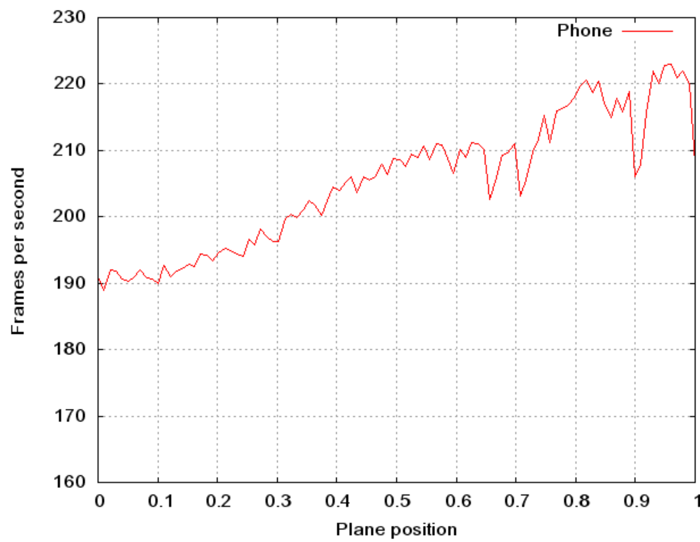


(b) Frames per second during the plane test.

Figure 3.20: Variable resolution: plane test applied to the bunny multiresolution model.

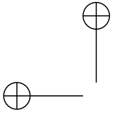
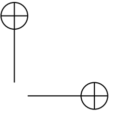


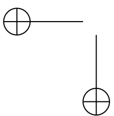
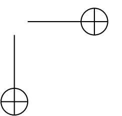
(a) Vertices sent during the plane test.



(b) Frames per second during the plane test.

Figure 3.21: Variable resolution: plane test applied to the phone multiresolution model.

Chapter 4

LodStrips: A Uniform Resolution Model

In the previous chapter, we conducted a detailed study of some fundamental aspects involved in creating a continuous multiresolution model based on implicit connectivity primitives. We introduced a model that enables multiresolution meshes to gain the rendering speed-up from using optimized rendering primitives, such as triangle strips. Taking that preliminary study into account, we specifically design *LodStrips*, a uniform resolution model based on triangle strips that features a fast level-of-detail extraction and a low spatial cost.

4.1. Introduction

Applications such a videogames usually makes use of uniform multiresolution models due to their simplicity (data structures and algorithms are simpler than in variable resolution models) and efficiency (they only extracts one level of detail, while in variable resolution models we must check the level of detail to be extracted for different parts of the mesh). With that aim, we specifically develop a uniform resolution model. As explained in section 3.4, the basic approach uses filtering in visualization to remove degenerate triangles produced when simplifying. However, filtering these repeated vertices every frame is a time-consuming task that could be improved if we could precalculate that information. Using uniform resolution enables us to previously detect and remove degenerate triangles. Moreover, it greatly improves visualization and storage cost. This model features:

- Uniform resolution. We specifically design and implement a uniform resolution model.
- Degenerate triangles elimination. We remove degenerate triangles from a pre-process, allowing us to speeding the extraction up. On the one hand, vertices in the triangle strips are not continuously filtered at the rendering stage. On the other hand, degenerate vertices are removed from the triangle strips before rendering, taking advantage of frame to frame coherence in the triangle strips.
- Low storage cost. We improve spatial cost from the previous approach. As we detect and remove degenerate triangles before rendering, we do not need to save the information of these repeated vertices into the data structures that allows us to change the level of detail.
- Rendering speed up. Due to removing degenerate triangles from the triangle strips before the rendering stage, this stage is noticeably improved.

This chapter is organized as follows: construction of the *LodStrips* model, with internal details, is presented in section 4.2. Section 4.3 and 4.4 introduce its data structure and algorithms. Later, in section 4.5 we present the results obtained from comparing this model with other models. Finally, in section 4.6, conclusions are presented.

4.2. Construction of the model

This model is built mainly upon two algorithms: construction of a simplification sequence [GH97] and generation of triangle strips [ESV96b]. We combine the information obtained from those algorithms in order to apply the simplification sequence to a model represented by triangle strips. Let us overview again the general process followed, which is shown in Figure 3.2.

4.2.1. Simplifification and stripification overview

In general, we define a polygonal mesh composed by triangles M, as:

$$M = \{V, T\} \quad (4.1)$$

where V is a set that contains all the vertices and T is another set that contains every triangle used to represent the mesh at the highest level of detail.

The most important information obtained from the simplification process is the edge collapse sequence necessary to simplify the mesh M. This information allows us to obtain some versions, at different level of detail, from a polygonal mesh, see Figure 2.1.

The simplification receives a polygonal mesh $M=\{V,T\}$, and it refines this mesh by applying the simplification E, where E is the ordered set of edge

collapses that allows us to simplify the polygonal mesh M. We underline again that every atomic operation of simplification applied to the multiresolution mesh will mean a different level of detail. Moreover, the highest level of detail will be zero and the lowest one *n-1*.

Regards to stripification, this process consists of creating, from a polygonal mesh geometrically composed of triangles, another mesh composed of triangle strips. This process does not change from the basic scheme either. Therefore, it processes a mesh $M=\{V,T\}$, by converting it into another $\{V,S\}$, where V contains all the vertices in the model and S the set of triangle strips that compounds the model at the highest level of detail.

4.2.2. LOD Builder

As commented in the introduction, one feature of the *LodStrips* model is that it removes degenerate triangles from the triangle strips by avoiding the scanner needed in the rendering stage of the basic scheme. It removes every degenerate triangle from the data structure used to render. Thus, we can pre-calculate those eliminations in a pre-process.

As expected, this process receives the information needed to transit among the levels of detail, simplifying the geometry E and the mesh in triangle strips at the highest level of detail $\{V,S\}$. This information enable us to build a simple multiresolution model that extracts the different levels of detail. In order to accelerate extraction and efficiency, the initial construction process performs two fundamental tasks:

- it calculates and stores every transition among the levels of detail.
- it removes degenerate triangles in every transition, storing that information.

On the one hand, calculation of transitions consists in transiting every LOD of the model, storing the strip or strips which changes in that LOD and the place where vertices to be processed are. On the other hand, we detect and remove degenerated triangles produced by the simplification. This pre-calculated information will allow a fast transit among LODs because it avoids to waste time looking for the vertices to modify in the real-time application. Moreover, it enable us to quickly remove degenerate triangles and therefore, filtering in rendering is not needed anymore. Thus, we will use the following patterns, which represent the 90 per cent of the degenerate triangles to be appeared in the triangle strips:

- Type 1: we replace pattern vertices such as $a(aa)^+$ by aa, where $a \in\mid V \mid$.
- Type 2: we replace pattern vertices such as $ab(ab)^+$ by ab, where $a,b \in\mid V \mid$.

Figure 4.1 shows the effect of applying degenerate-triangles filter to a triangle strip. The first column shows a triangle strip geometric representation along successive simplifications. In the second and third columns, we can see the triangle strip indices obtained from the simplification. In the first edge collapse, when applying the simplification we generate a *4 7 4 7* sequence, which is an $ab(ab)^+$ pattern, and which could be removed from the strip while keeping its geometry intact. After that, when applying the second edge collapse a *3 3 3* sequence appears, a $a(aa)^+$ pattern, and it is also removed from the triangle strip. It is obvious that removing degenerate triangles results in sending fewer vertices to the graphics system and a lower system load. In this example, after four edge collapses, filtering the number of vertices sent to the pipeline to render the same geometry halves the number produced without filtering.

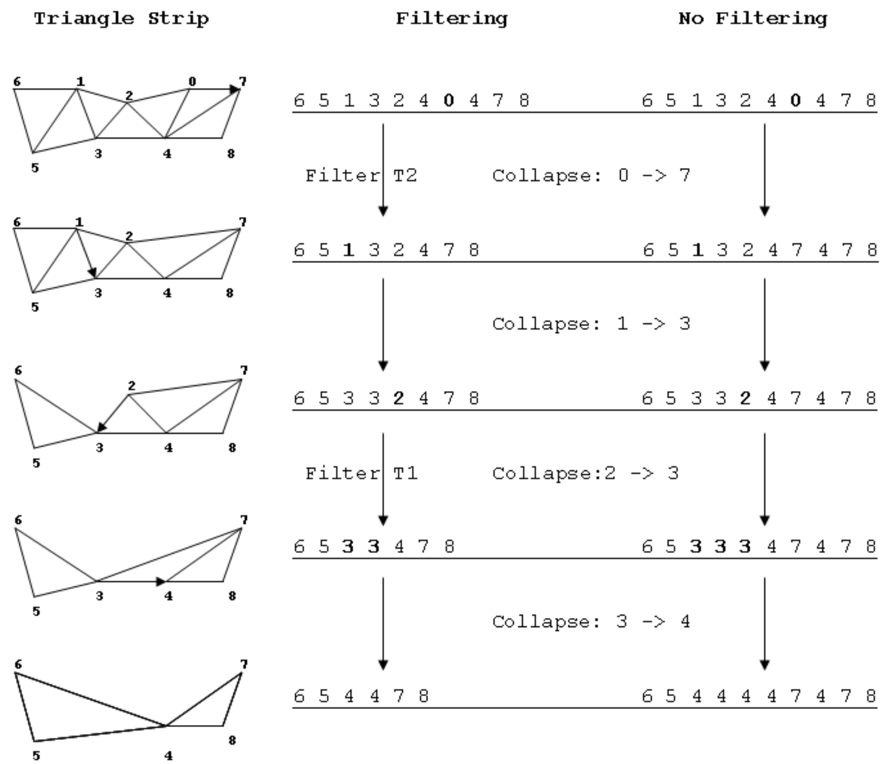


Figure 4.1: Triangle strips simplification both filtering and not filtering degenerate triangles.

We name C the set of changes that modifies the different triangle strips. In order to add filtering to the model, we need to reformulate this set. We define C as:

$$C = \bigcup_{i=0}^{n-1} C^i \ , n > 0 \tag{4.2}$$

where n is the number of LODs available, and C^i is the set of changes to apply in the triangle strips at the LOD i. Every item in C^i consists of the modifications to apply in a particular triangle strip, that is, it stores the strip that changes, where collapses take place and where are the degenerate triangles to be removed after applying those collapses. Formally:

$$\forall \ C^i_j \ \in C^i, C^i \ = \{ \ E^i\{\}, \{t^i_0, p^i_0, r^i_0\} \ , \{t^i_1, p^i_1, r^i_1\} \ , ...\} \ , \ 0 \leq j < s^i \tag{4.3}$$

being s^i, the number of strips to modify at the LOD i. Next, we detail the meaning of every tuple that compounds C^i_j:

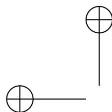
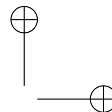
- E^i: this is the edge collapse to be applied at LOD i.
- t^i_j: this is a scalar that stores the index to one of the triangle strips that is modified at the LOD i.
- p^i_j: this is a scalar that informs us about the position, in the strip t^i_j, where vertices to be modified or removed are located.
- r^i_j: this scalar can adopt positive values (indicating how many degenerate triangles of type one there are to be removed at position p^i_j), negative values (indicating how many degenerate triangles of type two there are to be removed at position p^i_j) and zero (indicating that a vertex collapse will take place at position p^i_j).

Finally, the model is composed of the sets: V, S and C.

4.3. Representation

We perform some operations in the sets of the model to efficiently represent it. These operations have already been commented in section 3.3. We thereby obtain the sets:

- V: Vertices ordered according to their simplification sequence.
- W: Indexes of the vertices where each vertex in V collapses to.
- S: Triangle strips at the highest level of detail.
- C: Changes to be applied to the triangle strips that enable us to obtain the different approximations or levels of detail.

4.3.1. LodStrips construction example

In the following example, we show the construction sequence of the model obtained from the cited information. Thus, if we suppose that the sets W and S are:

$$W=\{1,8,3,...\}$$

$$S=\{\{6,5,4,7,0,8,1,9,10\},\ \{6,11,4,3,0,2,1,16,10\},\ \{11,12,3,13,2,14,16,15\}\}$$

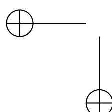
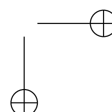
In Figure 4.2, we can observe the construction data flow diagram. In previous sections, we have already explained that, in this model, the simplification vertex sequence follows an increasing natural order, that is, the first vertex to be simplified is zero, the second vertex is one, etc. Furthermore, the set W is implicitly ordered in such a way that vertex zero collapses to one, one to eight and so on.

Initially, we start from the highest level of detail, S, and we apply successive edge collapses on the triangle strips. During their simplification some degenerate triangles could be produced and these would be removed from them. All this information is stored in C. Thus, when the process finishes, we have the necessary information to reconstruct every LOD in the model.

Therefore, the process begins by applying the first collapse to set S. This implies collapsing vertex zero to one. The process search for the vertex zero in every triangle strip in order to replace it with vertex one. Once we have found the places where this vertex is located, we then proceed to update S and we store that information in C. In the example case, vertex zero is in position four at strip zero, $\{0, 4, 0\}$, and position four at strip one, $\{1, 4, 0\}$. The last zero informs us about the type of register, in this case an edge collapse. Thus, the fist subset in C, that is C^0, is equal to $\{\{0, 4, 0\}, \{1, 4, 0\}\}$. After that, we proceed with the next collapse: vertex one collapses to eight. As well as storing positions of vertex one in order to collapse it (strip zero, positions four and six and strip one, positions four and six: $\{\{0, 4, 0\},\{1, 4, 0\},\{0, 6, 0\},\{1, 6, 0\}\}$), we detect a degenerate triangle in strip zero position four. Finally, the subset C^1 contains $\{\{0, 4, 0\},\{1, 4, 0\},\{0, 6, 0\},\{1, 6, 0\},\{0, 4, 1\}\}$. This process continues its flow until the lowest LOD is reached, storing each and every change needed to transit among the different LODs.

The data structures proposed for saving the model are shown in Figure 4.3.

In particular, V contains every vertex of the model, which are stored in the data structure *Vertices*. The sequence of collapses to be applied to the different vertices, that is W, is saved in the structure *Collapses*. Triangle strips at the highest level of detail are stored in *Strips*. Finally, the changes needed to modify the resolution of the triangle strips are saved in *Changes*.

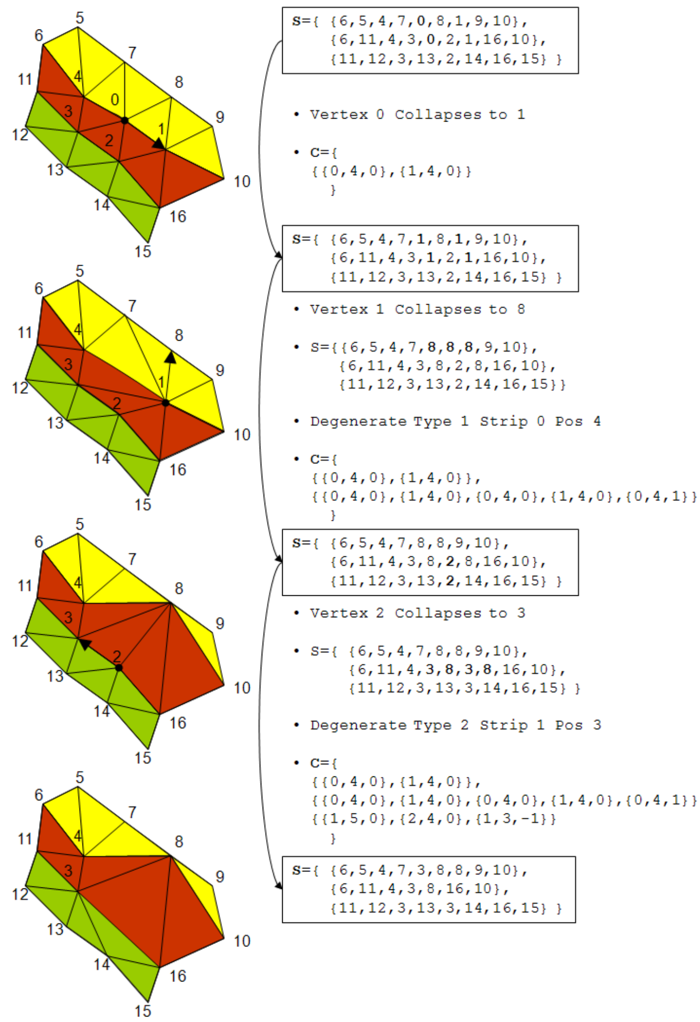


Figure 4.2: Simple example of construction of the model.

```
struct Vertices {
  real *listOf3DCoordinates[3];
};
struct Collapses {
  struct Vertice *CollapseVertex;
};
struct Strip {
  integer *listOfIndices;
};
struct Strips {
  struct Strip *listOfStrips;
};
struct Change {
  struct Strip *stripToChange;
  integer Position;
  integer TypeOfRecord;
};
struct ChangesPerLOD {
  struct Change *listOfChangesPerLOD;
};
struct Changes {
  struct ChangesPerLOD *listofChanges;
};
```

Figure 4.3: Fundamental data structures of the multiresolution model.

4.3.2. Theoretical spatial cost

Before calculating the spatial cost of the model, we assume that the cost of storing a real, integer or pointer value as being a word. Thus, the cost can be calculated as a sum of the different components of the multiresolution model: V, W, S and C. As commented in section 3.3.1, the spatial cost of V, W and S is $6|V| + 2|S|$ words.

Finally, we proceed to analyze the changes produced in the triangle strips that allow us to transit among the levels of detail. Empirically, we have proved that, on average, an edge collapse produces a modification in two triangle strips and one degenerate triangle. Taking into account that we have a maximum of $|V|$ levels of detail, we will store $3|V|$ records of C. Moreover, each element of C contains three words. Therefore, the spatial cost of C is $9|V|$.

Hence, the spatial cost of the *LodStrips* multiresolution model is $6|V| + 2|S| + 9|V|$ words, that is, $15|V| + 2|S|$ words. A mesh composed of triangle strips has a spatial cost of $5|V| + 2|S|$ words. Therefore, our model has a spatial

cost approximately three times higher than the original model.

4.4. Rendering

Obviously, besides data structures, multiresolution models need algorithms to manage the level of detail. The model presented here, and indeed most models, have two fundamental algorithms to perform this task. Thus, we assume the rendering stage to be a stage consisting of two algorithms applied sequentially. The first algorithm extracts the level of detail and the second visualizes the resulting mesh.

4.4.1. Level-of-detail extraction

The level-of-detail extraction algorithm processes requests from graphics applications that imply a modification in the level of detail. This algorithm allows us to efficiently obtain a mesh in uniform resolution from a geometric representation.

The graphics dataset is represented as a set of triangle strips. We have two representations for each triangle strip. On the one hand, we use a representation that supports constant or linear time in insertion and removal of elements to perform modifications in the triangle strip. On the other hand, we draw the geometry by means of a representation with a constant access time. Thus, when the level of detail does not change we access the triangle strips in constant time. Both representations are maintained and suitable for the current level of detail. We will refer to them as *recovery strips* and *draw strips*, respectively.

Each time a change in level of detail is required, edge collapses or vertex splits take place in the *recovery strips*. While changing the strips, they are also marked as *modified*. After that, the drawing algorithm goes into action and the *recovery strips* that remain unmodified are rendered by means of their copy in *draw strips*. However, if a *recovery strip* is modified, its content is copied to its corresponding *draw strip* while rendering.

Therefore, this algorithm comes into operation when the graphics application requires a change in the level of detail of the represented object. Set C is processed from the current level of detail to the new one ($C^{CurrentLOD}$ to C^{NewLOD}) by applying geometric transformations to the set of triangle strips. It is important to underline that edge collapses or vertex splits are applied to a mesh when the LOD demanded is higher or lower than the current one.

In Figure 4.4 we can observe the pseudo-code associated to this algorithm.

4.4.2. Drawing

After the level-of-detail extraction algorithm has modified the object geometry, the drawing algorithm begins to run. As commented before, this algorithm updates the *draw strips* when required and finally renders them. In general, we

```
Function ExtractLevelOfDetail (newLOD)
  // Determine whether newLOD is higher or lower than the current one
  if newLOD>currentLOD then
    for lod=currentLOD to newLOD
      for change=Changes[lod].Begin to Changes[lod].End
        if change.isCollapse() then
          change.ApplyEdgeCollapses(RecoveryStrips);
        else
          change.RemoveDegenerateTris(RecoveryStrips);
        end if
      end for
    end for
  else // To a more detailed mesh
    for lod=newLOD to currentLOD
      for change=Changes[lod].Begin to Changes[lod].End
        if change.isSplit() then
          change.ApplyVertexSplit(RecoveryStrips);
        else
          change.RestoreDegenerateTris(RecoveryStrips);
        end if
      fin para
    end for
  end if
end Function
```

Figure 4.4: Level-of-detail extraction algorithm.

will render the geometry by means of a triangle strip representation with constant time access, which implies optimum performance when sending them to the graphics system. In conclusion, by using both representations we take advantage of constant time to access the triangle strips and linear time to insert and remove elements from them.

4.5. Results

Tests and experiments were carried out with a Dell Precision PWS760 Intel Xeon 3.6 Ghz with 512 Megabytes of RAM, the graphics card was a NVidia GeForce 7800 GTX 512. Implementation was performed in C++ and OpenGL as support graphics library. We used the immediate mode to render the geometry because it facilitates comparisons with other multiresolution models. Among other aspects, the next chapter will deal with acceleration in rendering.

```
Function Draw ()
  for indexStrip=0 to NumberOfStrips-1
    // Refresh draw strip content if recovery strip modified
    if RecoveryStrips[indexStrip].Modified then
      RecoveryStrips[indexStrip].CopyTo(DrawStrips[indexStrip]);
      RecoveryStrips[indexStrip].SetUnModified();
    end if
    DrawStrips[indexStrip].Render();
  end for
end Function
```

Figure 4.5: Drawing algorithm.

The different objects used to generate multiresolution models are meshes in OBJ format, which allows us to manage its geometry easily. In Table 4.1, we show some objects used as a reference. We specify the number of vertices and triangles which compose them, together with the cost of storing all that information.

	Original		
	Vertices	Triangles	MB.
Cow	2,904	5,804	0.10
Capone	3,618	7,124	0.12
Boat	6,082	12,268	0.21
Car	21,310	42,964	0.74
Bunny	34,834	69,451	1.19
Dragon	54,294	108,588	1.86
Phone	83,044	165,963	2.85
Isis	187,644	375,284	6.44
Buddha	543,699	1,085,634	18.65

Table 4.1: List of models with their main features and their original storage cost.

4.5.1. Analysis

Before presenting the results obtained, we will carry out a preliminary study of some important questions related to the model. On the one hand, filtering or removing degenerate triangles is a key question in the model. On the other hand, the number of level of detail extractions is important in order to determine the model's performance.

Filtering

With the aim of analyzing filtering of degenerate triangles, we studied the impact of applying different filters to a multiresolution model: the Bunny model. Thus, in Figure 4.6, we compare a multiresolution model based on *LodStrips* applying different filters and a model based on triangles, in this case Progressive Meshes [Hop96]. Moreover, we can observe the difference in times obtained by applying filtering in Figure 4.7. Obviously, as the model moves toward coarser LODs, the number of vertices sent is noticeably decreased due to filtering. It is important to notice that from certain LODs, PM sends fewer vertices to the *pipeline*. However, temporal cost is better in *LodStrips* due to the graphics primitive which is based on.

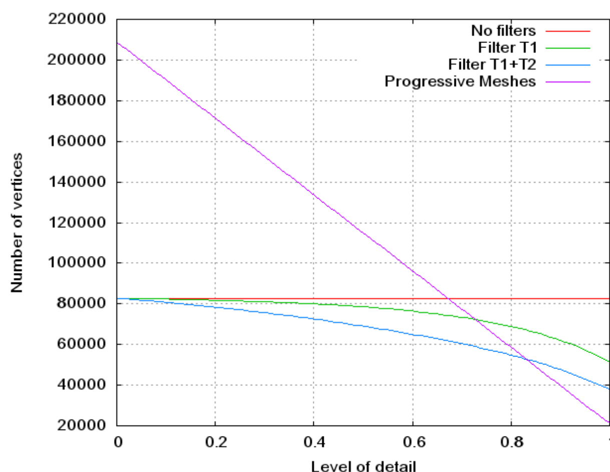


Figure 4.6: Vertices sent to the GPU per level of detail by applying different filters to the Bunny model. Lowest level of detail means the model simplified at 90 %.

Extraction

Before evaluating the temporal cost of the model, it is necessary to analyze the context where it will take place. We have implemented the linear test proposed by Ribelles [RCLH99]. This test extracts and renders, in a linear way, a number of levels of detail given by the user or application. Moreover, we must decide on the parameters to be used in running the linear test on the models. On the one hand, we will apply the test from the highest level of detail to the lowest one, which corresponds to the model simplified at 90 % and, on the other hand, we must select a number of approximations to be extracted.

In Table 4.2, we show some tests applied to the Bunny model. In this table, each row shows:

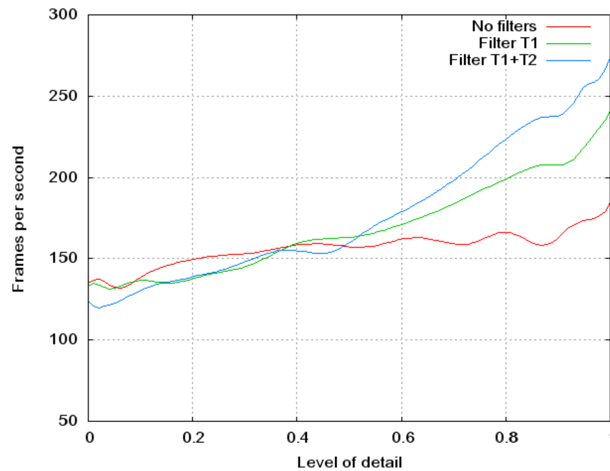


Figure 4.7: Frames per second per level of detail by applying different filters to the Bunny model.

- the number of levels of detail extracted.
- the number of edge collapses processed to transist between one level of detail and the next one.
- the time employed to run the whole test.
- the time employed only to extract the different levels of detail.
- what percentage of the total test time is spent on extracting

At this point, we must underline one of the *LodStrips* characteristics we have already commented earlier: the low time required to retrieve one LOD. We have decided to extract 1,000 levels of detail in the tests carried out in the following sections. In Figure 4.8, we show the percentage of time (when applying a linear test to different models) spent on the extraction of 1,000 levels of detail.

4.5.2. Spatial cost

Spatial costs from the models taken as references are shown in Table 4.3. For each model, we specify the triangle strips that compose them (strips generated by means of the STRIPE algorithm [ESV96b]), the number of approximations or levels of detail available, storage cost in main memory and, finally, the relation between storing the multiresolution model and the original model. We can easily observe that the cost of storing the whole model ranges is between

Extractions	Step	Test (ms.)	Extraction (ms.)	% Extraction
10	3135	28.83	9.48	32.9
100	313	210.44	14.49	6.9
500	62	904.46	19.40	2.1
1,000	31	1,800.52	24.85	1.4
10,000	3	16,776.4	49.46	0.3

Table 4.2: Extraction times of the Bunny model extracting different LODs. The model consists of 31351 levels of detail.

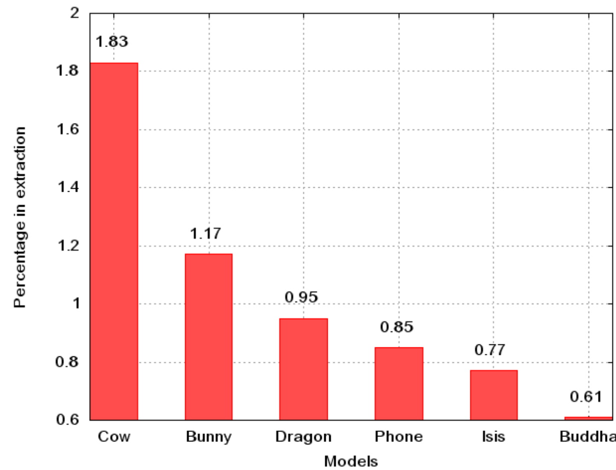


Figure 4.8: Percentage of time spent on extraction for different models by applying a linear test with 1000 LOD extractions.

2.11 and 2.69 times higher than storing the original mesh (mesh at the highest level of detail and represented by triangle strips).

4.5.3. Temporal cost

Results shown in this section were obtained under the conditions mentioned above. We applied a series of tests to the reference models, where levels of detail were in the interval [0, 1], zero being the highest LOD and one the lowest one. We also considered the model simplified at 90 % to be the lowest LOD. Transition between the different LODs used in the tests follows the criteria proposed in section 4.5.1. Results for some models are shown from Figure 4.10 to Figure 4.15. We can see how the extraction time is a small percentage of the model. Graphically, this time consists in the difference between the rendering and the drawing time, we consider rendering as the sum of extracing and

	LodStrips			Ratio
	Strips	LODs	Total (Mb)	
Cow	136	2,614	0.22	2.21
Al Capone	177	3,256	0.26	2.11
Boat	399	5,474	0.55	2.62
Car	1,396	19,179	1.98	2.69
Bunny	1,229	31,351	2.88	2.41
Dragon	1,787	48,865	4.28	2.30
Phone	1,747	74,740	6.56	2.31
Isis	5,141	168,880	15.11	2.35
Buddha	31,596	489,329	46.95	2.52

Table 4.3: List of models with their features and the *LodStrips* storage cost.

drawing a level of detail.

In Figure 4.9, we can observe *LodStrips* performance compared to other well-known multiresolution models. In these figures, it is important to notice that although *LodStrips* sends more vertices than MTS, we obtain better rendering times due to the large amount of time that the latter model spends on extracting.

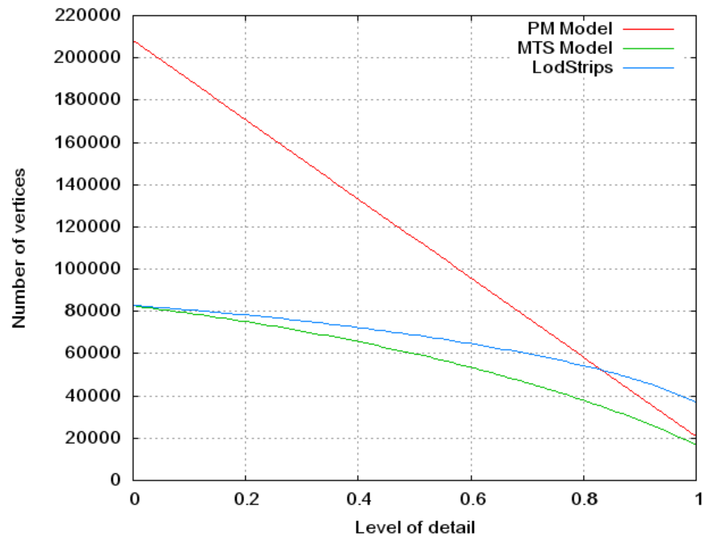
4.6. Conclusions

In this chapter we have introduced *LodStrips* [RC04b], a multiresolution model wholly based on an implicit connectivity primitive: the triangle strip. This model allows us to efficiently transit between different levels of detail in real-time applications [RCC04].

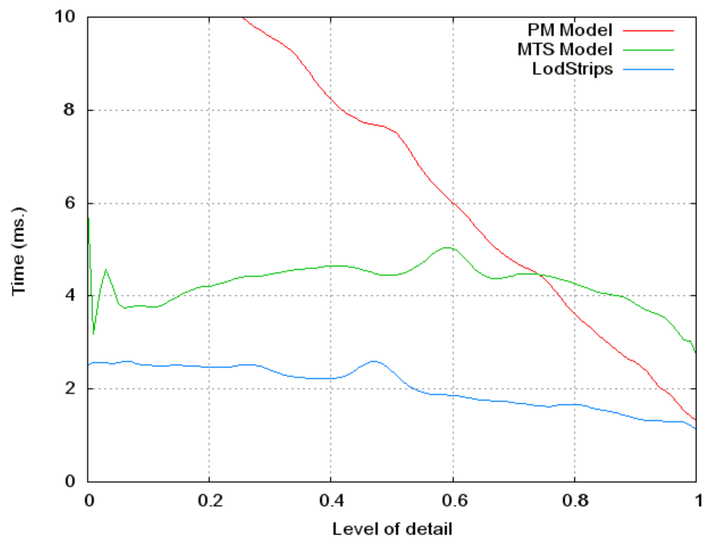
The *LodStrips* model offers many advantages and it should be underlined that it offers a fast level-of-detail extraction which allows us to obtain smooth transitions between levels of detail as well as considerable rendering times because extraction is usually an important part of the total rendering time. This model offers an important reduction in storage and rendering costs when compared to other well-known multiresolution models.

Reduction in spatial costs has been shown in the previous sections, where it can be seen how a *LodStrips* model can store an object in a memory space that is similar in size to twice the size of the original mesh in triangle strips. We thereby can allocate more objects in memory than with other models.

It has been shown how *LodStrips* has better temporal cost than the MTS and PM models. It also allows us to obtain a fast-rendering model, which is important in critical applications and, what is more, it accelerates scene rendering by increasing the frame per second rate.

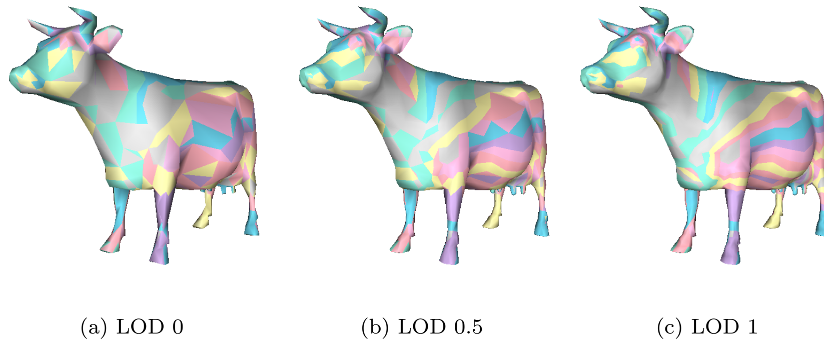


(a) Bunny: Vertices sent per level of detail.

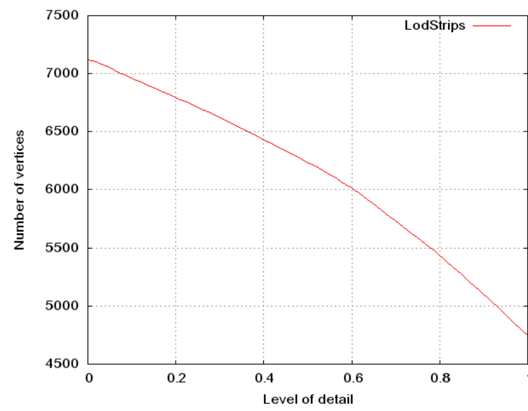


(b) Bunny: Rendering times per level of detail.

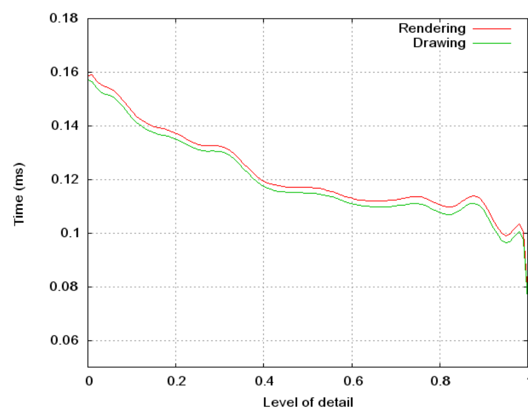
Figure 4.9: Charts obtained for the Progressive Meshes, MTS and LodStrips multiresolution models based on the Bunny object.



(a) LOD 0 (b) LOD 0.5 (c) LOD 1

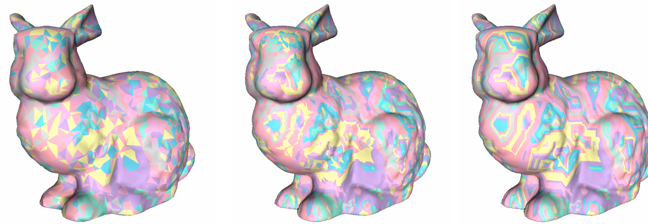


(d) Vertices sent to the GPU per level of detail.

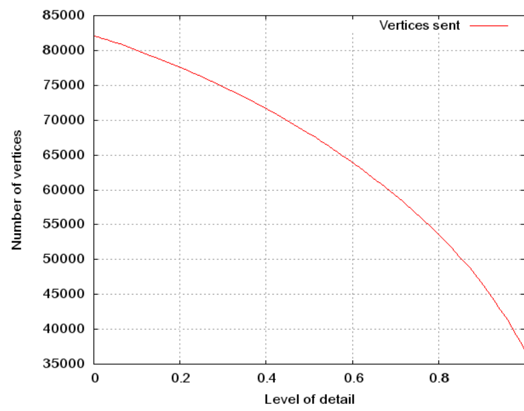


(e) Rendering and drawing times per level of detail. Rendering=Extracting+Drawing. The difference between the curves means extraction times.

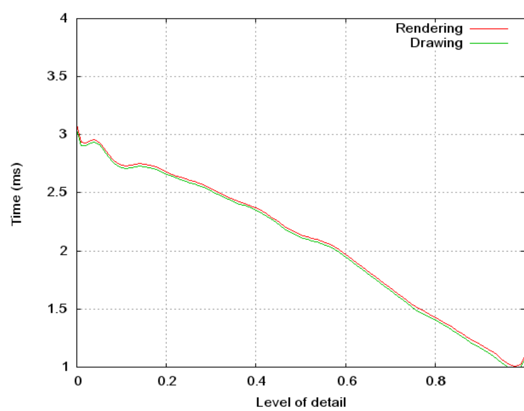
Figure 4.10: Results obtained for the multiresolution model based on the Cow object.



(a) LOD 0 (b) LOD 0.5 (c) LOD 1

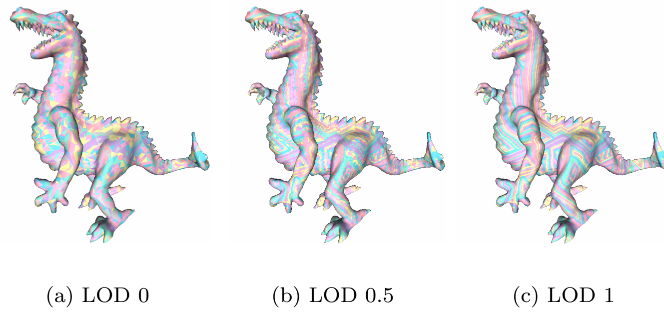


(d) Vertices sent to the GPU per level of detail.

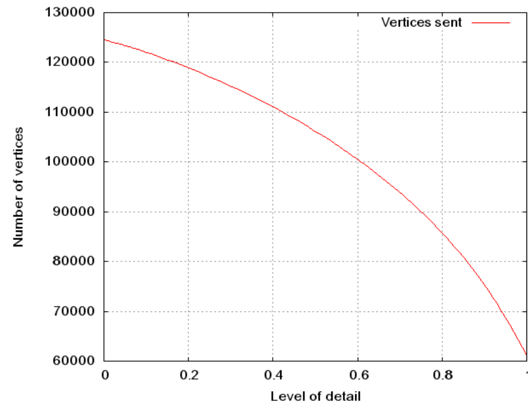


(e) Rendering and drawing times per level of detail. Rendering=Extracting+Drawing. The difference between the curves means extraction times.

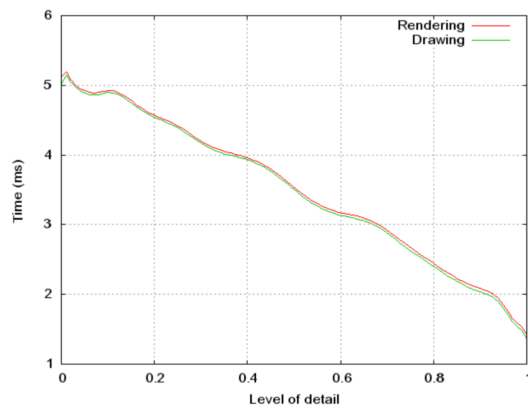
Figure 4.11: Results obtained for the multiresolution model based on the Bunny object.



(a) LOD 0 (b) LOD 0.5 (c) LOD 1

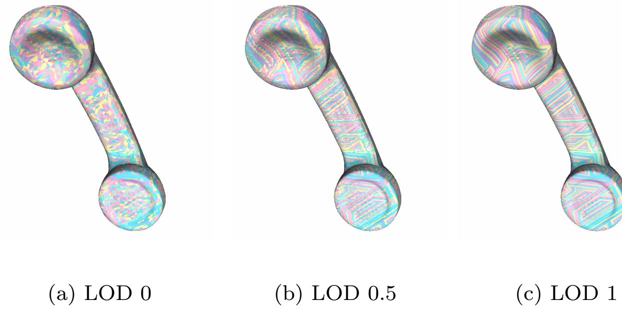


(d) Vertices sent to the GPU per level of detail.

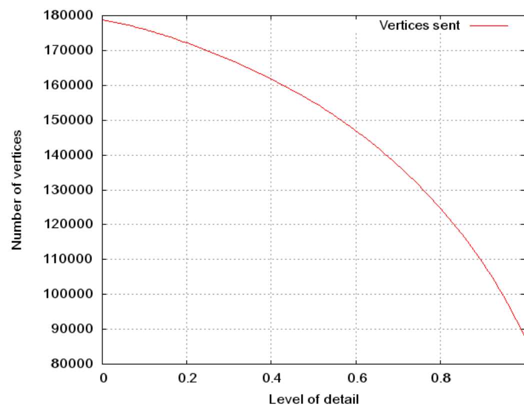


(e) Rendering and drawing times per level of detail. Rendering=Extracting+Drawing. The difference between the curves means extraction times.

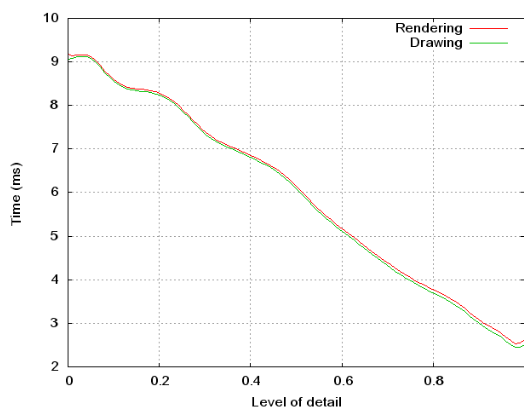
Figure 4.12: Results obtained for the multiresolution model based on the Dragon object.



(a) LOD 0 (b) LOD 0.5 (c) LOD 1

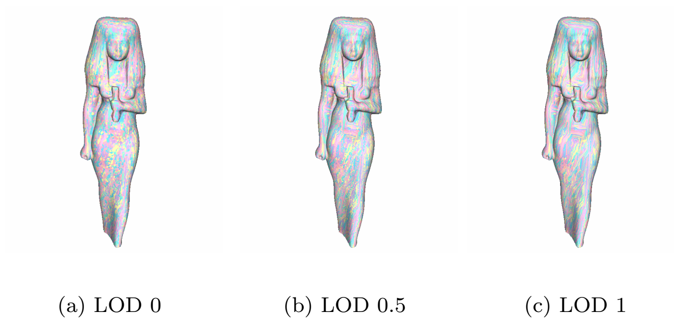


(d) Vertices sent to the GPU per level of detail.

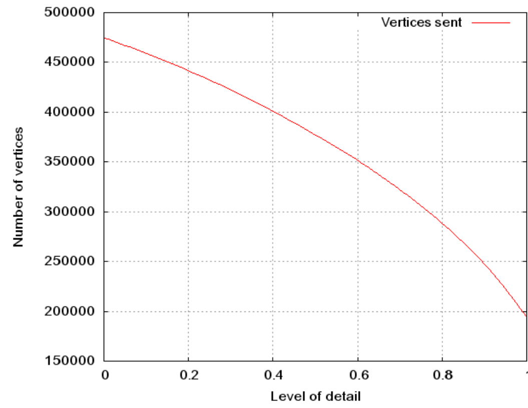


(e) Rendering and drawing times per level of detail. Rendering=Extracting+Drawing. The difference between the curves means extraction times.

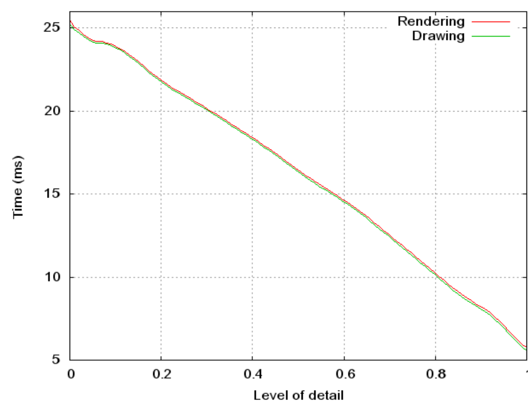
Figure 4.13: Results obtained for the multiresolution model based on the Phone object.



(a) LOD 0 (b) LOD 0.5 (c) LOD 1

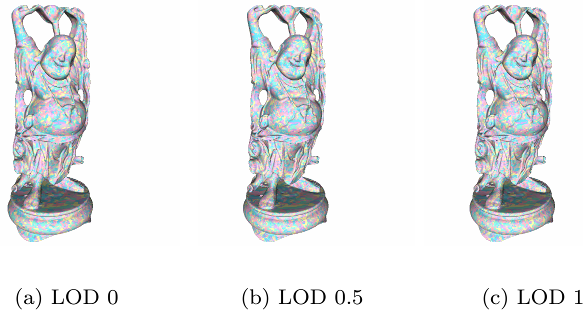


(d) Vertices sent to the GPU per level of detail.

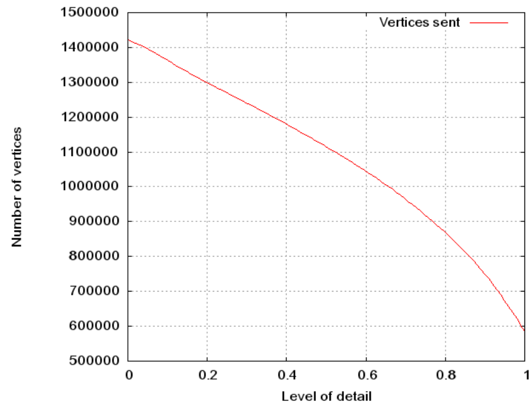


(e) Rendering and drawing times per level of detail. Rendering=Extracting+Drawing. The difference between the curves means extraction times.

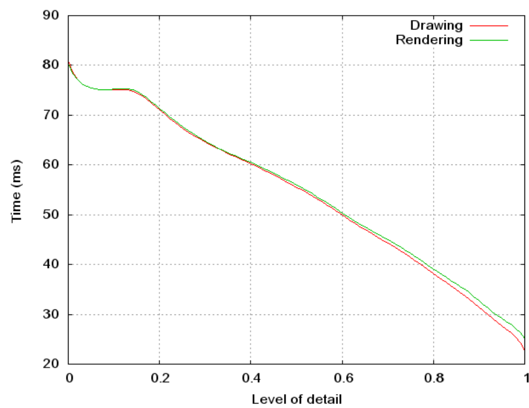
Figure 4.14: Results obtained for the multiresolution model based on the Isis object.



(a) LOD 0 (b) LOD 0.5 (c) LOD 1



(d) Vertices sent to the GPU per level of detail.



(e) Rendering and drawing times per level of detail. Rendering=Extracting+Drawing. The difference between the curves means extraction times.

Figure 4.15: Results obtained for the multiresolution model based on the Buddha object.

CHAPTER 5

LodStrips on the GPU

In the previous chapter, we introduced *LodStrips*, a continuous multiresolution model that manages data structures and algorithms for real-time rendering of meshes with uniform level of detail. Following the same philosophy as that underlying *LodStrips*, we introduce some modifications in the original model which allow us to noticeably improve its performance in recent graphics hardware. Some of the most important changes include a representation of the model on the GPU and the application of new hardware acceleration techniques that take advantage of the new GPU features.

5.1. Introduction

Nowadays GPUs offer new capabilities that, when exploited to the maximum, allow the multiresolution models to accelerate even more. One of these involves storing information directly in the memory located in the GPU. This characteristic allows information to be managed in the GPU while avoiding data transfer between the CPU and the GPU and taking the maximum advantage of the proximity of the memory and the graphics processor.

Another important issue is the kind of bus that joins the CPU and the GPU. AGP buses are far better optimized to upload data than to download it, thus favoring the use of the memory of the graphics card to store static objects that do not change their geometry. But the appearance of the PCI-Express bus makes it possible to use a symmetric bus, which allows data to be uploaded and downloaded to and from the GPU at the same speed, so that it is possible to work with the GPU memory and dynamic geometry in a reliable way. The use of stripification algorithms, which attempt to take maximum advantage

of the GPU cache, and the new extensions of graphics libraries that allow visualization of a whole mesh with only a few instructions are also examples of these new techniques. Thus, the model have been greatly improved both in rendering times and in spatial cost.

We have organized this chapter as follows: we will carry out an analysis concerning stripification (section 5.2) and hardware acceleration techniques (section 5.3). Section 5.4 introduces the *LodStrips* model in the GPU, it details the new representation (section 5.5) and algorithms (section 5.6) for the GPU exploitation. Later, section 5.7 shows the different results obtained. Finally, we finish the chapter with the conclusions, in section 5.8.

5.2. Stripification techniques

The use of optimized rendering primitives elicits a performance gain in graphics systems. Graphics processors allow us to represent 3D objects properly by using triangles, triangle strips, triangle fans, and so on. Given an object representation in triangles, there exist some algorithms that can obtain its equivalent representation in triangle strips. This process is usually named *Stripification*. *LodStrips* is a multiresolution model that admits any kind of mesh composed of triangle strips.

As commented in previous chapters, the stripification process is involved in the construction and management of multiresolution models based on triangle strips. This process can be carried out in a dynamic or a static way. Dynamic stripification involves generating the triangle strips in real time, that is, for each level of detail new strips are generated. On the other hand, static stripification entails first creating triangle strips and then working with versions of the original strips. There are several models that use dynamic stripification [Ste01, SP03], especially variable resolution models. Other models such as [ESAV99, RAO+00, BRR+01], however, use static stripification techniques.

From a performance point of view and given the architecture of present-day GPUs, it is better to employ static stripification techniques since we thereby avoid strip creation and destruction in the GPU, which would imply an additional cost that would make the model much less competitive. Furthermore, there is an additional cost stemming from the calculation of the new triangle strips at each level of detail, which also penalizes the use of dynamic techniques.

In previous chapters, we have analyzed *LodStrips* by using the stripification algorithm STRIPE [ESV96b]. To accelerate the rendering, we will use the NvTriStrip algorithm [BD02]. Both algorithms allow us to obtain a triangle strips representation from a model composed of triangles. However, NvTriStrip has a distinctive and important feature that accelerates geometry visualization: it exploits the vertex cache. Current GPUs have a series of records which store the last 16 or 24 vertices that have been used. Thus, we can send the geometric information to the pipeline in an ordered way, so that page faults are kept to a

"tesis" — 2008/4/28 — 19:02 — page 81 — #97

minimum, that is, by reusing the maximum number of vertices that are in the GPU vertex cache. As commented before, we will use the NvTriStrip algorithm to accelerate rendering.

In Figure 5.1, we show some meshes in triangle strips generated by means of STRIPE and NvTriStrip algorithms. We could accept that triangle strips created by STRIPE offer a better performance than those generated by NvTriStrip because the mesh has less triangle strips and sends fewer vertices. However, performance of the mesh obtained by applying NvTriStrip is better because it reuses the vertex cache far better.

5.3. Hardware acceleration techniques

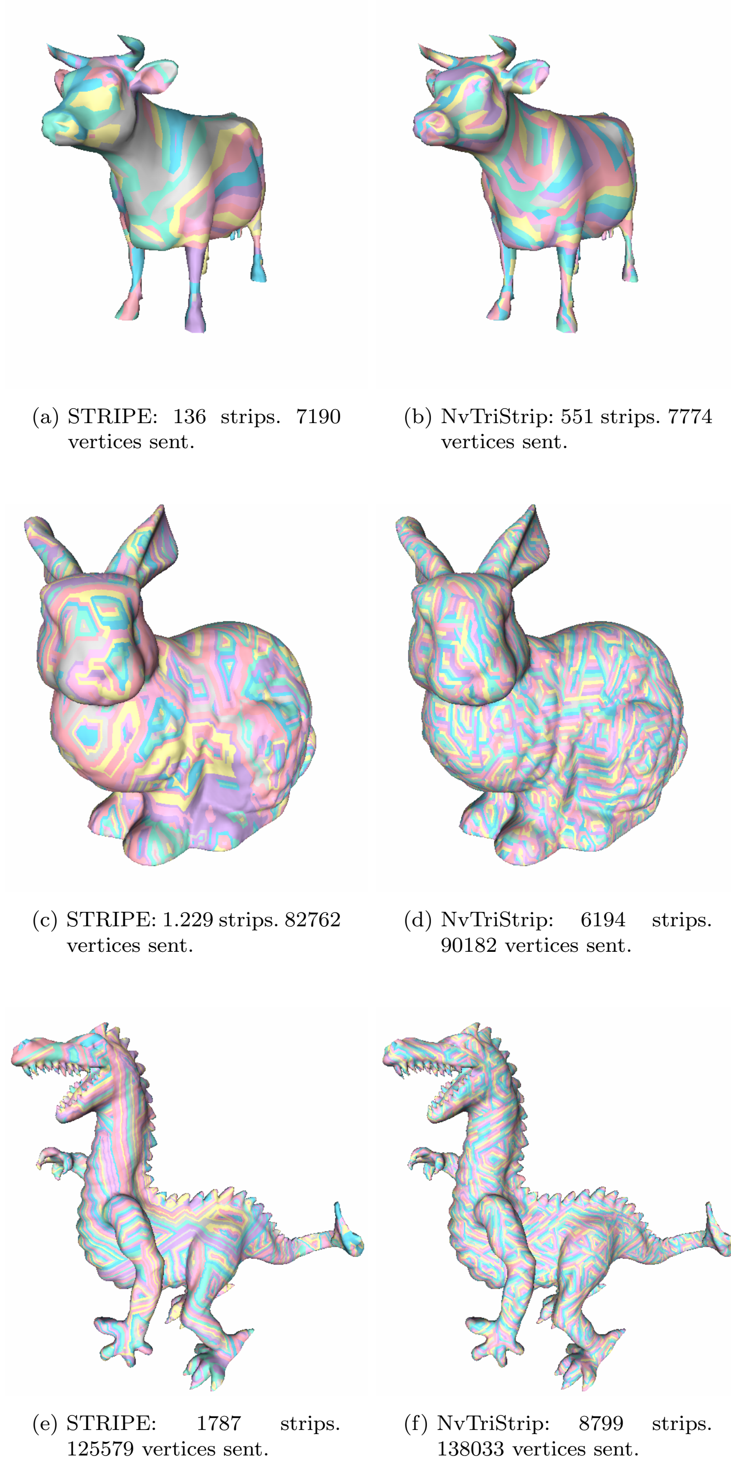
There exist a number of different solutions to exploit graphics hardware and they are directly linked to the graphics library. In this work, we have used OpenGL, although DirectX offers similar features. Traditionally, OpenGL offers two ways to render geometry, namely *immediate mode* and *display lists*.

Use of *immediate mode* implies that, for each frame, that applications must send all the geometric information to the GPU. If data do not change very often, this mode wastes a lot of time on transferring data to the GPU when compared to storing data in the graphics memory. That is, the *immediate mode* transfers data (vertices, normals, ...) in an individual way, it involves a considerable amount of traffic between the CPU main memory and GPU through the bus that interconnects them. Moreover, it noticeably affects to the GPU parallelism.

Implicit connectivity primitives reduce this traffic. However, it continues to be very important. As an alternative to the *immediate mode*, OpenGL provides *display lists*. This allows us to group a series of commands and store them in the graphics memory, thus avoiding the traffic in the graphics bus. However, if mesh geometry changes in a frame, which is very usual in multiresolution solutions, we should create a new *display list* and transfer it to the graphics memory. This would cause a significant bottleneck in applications that work with level of detail.

Another solution that OpenGL offers, which is different to *display lists*, is *vertex arrays*. These allow data associated to the vertices to be grouped in arrays, which leads to similar advantages to those provided by *display lists*. However, whenever we render an object using *vertex arrays*, we must validate them every time that we refer to them, which implies a greater workload on the data transfer.

Vertex buffer objects, or VBOs, increase the capabilities of OpenGL by offering most of the benefits of *immediate mode*, *display lists* and *vertex arrays*, as well as avoiding some of their limitations. They efficiently group and store data, in the form of *vertex arrays*, thus improving their transfer. Moreover, applications are able to modify the data with no extra cost due to data validation.



(a) STRIPE: 136 strips. 7190 vertices sent.

(b) NvTriStrip: 551 strips. 7774 vertices sent.

(c) STRIPE: 1.229 strips. 82762 vertices sent.

(d) NvTriStrip: 6194 strips. 90182 vertices sent.

(e) STRIPE: 1787 strips. 125579 vertices sent.

(f) NvTriStrip: 8799 strips. 138033 vertices sent.

Figure 5.1: Triangle strips generated by STRIPE and NvTriStrip algorithms for the Cow, Bunny and Dragon meshes, respectively.

5.3.1. High-performance memory

A vertex buffer object is a feature that enables us to store data in high-performance memory in the GPU. The basic idea is to provide some buffers, which will be available through identifiers. There are different ways to interact with buffers:

- Bind a buffer: this activates the buffer so it can be used by the application.

- Put and get data: this allows us to copy data between a client's area and a buffer object in the GPU.

- Map a buffer: you can get a pointer to a buffer object in the client's area, but this can lead to the driver's waiting for the GPU to finish its operations.

There are two kinds of vertex buffer objects: array buffers and element array buffers. On the one hand, array buffers contain vertex attributes, such as vertex coordinates, texture coordinates data, per-vertex color data and normals. On the other hand, element array buffers contain only indices to elements in array buffers. The ability to switch between various element buffers while keeping the same vertex array allows us to implement level of detail schemes by changing the elements buffer while working on the same array of vertices.

In order to implement the model on graphics hardware, we used different functions which interact with buffer objects. Among them, we can highlight:

- glBindBufferARB: this function sets up internal parameters so that the next operations work on this current buffer object.

- glBufferDataARB: this function is an abstraction layer between the memory and the application. Basically, this function copies data from the client memory to the buffer object bound.

- glBufferSubDataARB and glGetBufferSubDataARB: its purpose consists in replacing or obtaining, respectively, data from an existing buffer.

Therefore, vertices and triangle strips are directly stored in the GPU. On the one hand, vertices are stored in a *vertex array buffer*. On the other hand, we might allocate each triangle strip in an *element buffer*. However, we have observed that creating as many buffers as there are triangle strips leads to noticeable decreases in performance due to bind operations. A solution to this problem, with optimum results, consists in creating a single element buffer, containing every strip to be rendered. This way, we avoid the need for continuous bind operations to assign an element buffer for each strip.

5.3.2. Specific library extensions

As commented before, we need some OpenGL extensions to exploit *Vertex buffer objects*. In general, what the extensions linked to geometry acceleration attempt to do is to load the data concerning the vertices into the memory of the graphics card while trying to avoid using the main memory because this memory is close to the GPU and is therefore substantially faster. There are many extensions available to render geometry, two of the most important being glDrawRangeElements and glMultiDrawElements.

glDrawRangeElements enables us to render a list of indices to vertices, hence one call per primitive is required. glMultiDrawElements behaves identically to glDrawRangeElements except that it handles multiple lists of indices in one call. Its main purpose is to allow one function call to render more than one primitive, such as a triangle strip, triangle fan, etc. This enables very large models to be rendered with no more than a few small commands to the graphics device.

5.4. The model on the GPU

In this section, we introduce the GPU version of the *LodStrips* model presented in chapter 4. A brief outline of the model is shown in Figure 5.2. We advance the general idea which is based on. At the beginning, information about vertices and triangle strips is uploaded into the GPU. Later, strips are updated in accordance with the current level of detail. More specifically, when a level of detail transition is required, it downloads the strips affected by these changes from the GPU. Later, it modifies and uploads the updated strips to the graphics system. Lastly, strip information in the GPU is then used for display.

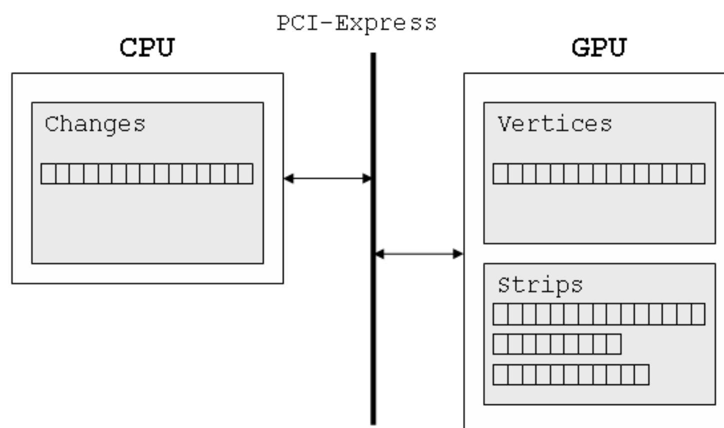


Figure 5.2: Model implementation in GPU.

5.5. Representation

In general, applications such as computer games have high memory requirements. Efficient memory usage is a way to improve this kind of solutions. In our case, we present a way to improve the *LodStrips* data structures in order to minimize its spatial cost while performance remains unaffected.

Two essential data structures of the model are stored in the GPU: *vertices* and *triangle strips*, which constitute the polygonal mesh. The data structure *Changes*, which contains the information for modifying the triangle strips, is stored in the CPU.

Following the nomenclature introduced in previous chapters, we focus on improving the *LodStrips* data structures and taking also into account the performance in real time, we have reformulated the set C as:

$$C = \bigcup_{i=0}^{n-1} C^i \ , n > 0 \tag{5.1}$$

Where n is the number of levels of detail available, and C^i is the set of changes to be applied in the multiresolution triangle strips at the level of detail i. Every element in C^i contains the modifications of a particular triangle strip, that is, it stores what triangle strip changes, where its collapses take place and where the degenerate triangles to be removed are located after applying the collapses. Formally:

$$\forall \ C^i_j \ \in C^i, C^i \ = \{ \ \{t^i_0, P^i_0, R1^i_0, R2^i_0\} \ , \{t^i_1, P^i_1, R1^i_1, R2^i_1\} \ , ...\} \ , \ 0 \leq j < s^i \tag{5.2}$$

s^i being the number of triangle strips to be modified at the level of detail i. Next, we detail the meaning for each tuple that composes C^i_j:

- t^i_j: this is a scalar that stores the index to one of the triangle strips that is modified at the LOD i.
- P^i_j: this is the set of positions, within the triangle strip t^i_j, where vertices to be collapsed at LOD i are located.
- $R1^i_j$: this is the set of positions, within the triangle strip t^i_j, where degenerate triangles of type one that are to be removed are located.
- $R2^i_j$: it is the set of positions, within the triangle strip t^i_j, where degenerate triangles of type two that are to be removed are located.

The construction process has been described in previous chapters. However, we will show a simple example of construction for this new model, from now on the *GPU* model. Thus, if we suppose that sets W and S are:

$$W{=}\{1{,}8{,}3{,}...\}$$

S={{6,5,4,7,0,8,1,9,10}, {6,11,4,3,0,2,1,16,10}, {11,12,3,13,2,14,16,15}}

In Figure 5.3, we can observe an example of construction of the *GPU model*. In previous chapters, we have already explained that, in this model the simplification vertex sequence follows an increasing natural order, that is, the first vertex to be simplified is zero, the second vertex is one, etc. Furthermore, the W set is implicitly ordered. In the example, zero collapses to one, one to eight and so on.

Initially, we start from the highest level of detail, S, and we apply successive collapses to the triangle strips. During their simplification some degenerate triangles could be produced, which will be removed. All this information is stored in C. Thus, when the process finishes, we have the necessary information to reconstruct every LOD in the model.

The process therefore begins by applying the first collapse to set S which implies collapsing vertex zero to one. The process searches for the vertex zero in every triangle strip in order to replace it with vertex one. Once we have found the places where this vertex is, we proceed to update S, and we store that information in C. In the example case, vertex zero is in position four at strip zero, $\{0, \{4\}\}$, and position four at strip one, $\{1, \{4\}\}$, the last zero informs about the type of register, in this case a collapse. Thus, the first subset in C, that is, C^0 is equal to $\{\{0, \{4\}\}, \{1, \{4\}\}\}$. After that, we proceed with the next collapse: vertex one collapses to eight. As well as storing positions of vertex one in order to collapse it (strip zero, positions four and six and strip one, positions four and six: $\{\{0, \{4, 6\}\}, \{1, \{4, 6\}\}\}$, we detect a degenerate triangle in strip zero position four. Finally, subset C^1 contains $\{\{0, \{4, 6\}, \{4, 1\}\}, \{1, \{4, 6\}\}\}$. This process continues its flow until the lowest LOD is reached, storing each and every change needed to transit among the different LODs.

In Figure 5.4, we show the new data structures proposed in our work. As commented previously, the main difference can be found in the data structures that support the level-of-detail transitions.

5.6. Rendering

As described in previous chapters, extraction and drawing algorithms allow us to manage level-of-detail solutions in real time. Obviously, application of hardware acceleration techniques implies certain modifications in those algorithms.

At a high level, the pseudo-algorithm to transit from LOD n to LOD $n+1$ consist in downloading, from the GPU, the chunks of memory corresponding to the strips affected by the change in the level of detail. After that, we replace vertex n by the vertex it collapses to, in every strip where it appears. Later, derived vertex repetitions must be removed. Finally, the strip is uploaded to the GPU for visualization.

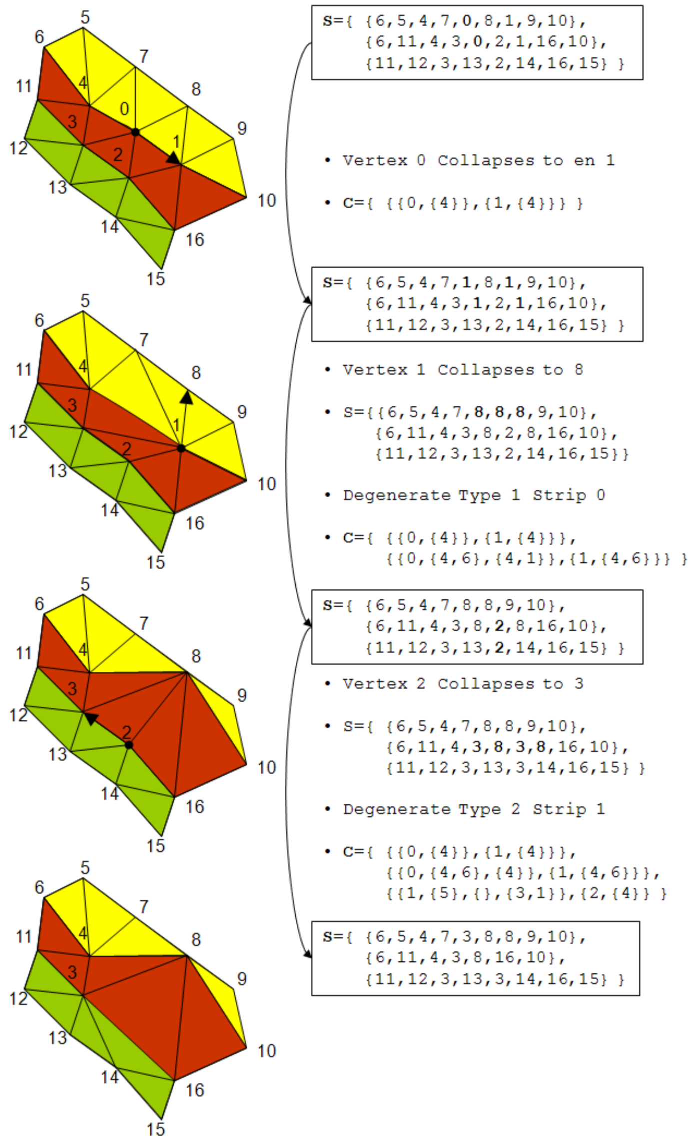


Figure 5.3: Simple construction example of the *GPU* model.

```
struct Vertices {
  real *listOf3DCoordinates[3];
};
struct Collapses {
  struct Vertex *CollapseVertex;
};
struct Strip {
  integer *listOfIndices;
};
struct Strips {
  struct Strip *listOfStrips;
};
struct Change {
  struct Strip *stripToChange;
  integer NumberOfCollapses;
  integer NumberOfDegenerateT1;
  integer NumberOfDegenerateT2;
  integer *Positions;
};
struct ChangesPerLOD {
  integer NumberOfStripsAffected;
  struct Change *listOfChangesPerLOD;
};
struct Changes {
  struct ChangesPerLOD *listOfChanges;
};
```

Figure 5.4: Fundamental data structures for the *GPU* multiresolution model.

In Figure 5.5, we show the pseudo-algorithm associated to the extraction of the level of detail. Essentially, it consists in downloading from the GPU the chunks of memory corresponding to the strips affected by a change in the level of detail. After that, we modify the triangle strips following the *LodStrips* philosophy. Finally, the processed strip is uploaded to the GPU for visualization.

```
for LOD = currentLOD to newLOD
  for Strip = StripsAffected(LOD).Begin() to StripsAffected(LOD).End()
    auxStrip.DownloadFromGPU(Strip);
    auxStrip.Modify(LOD);
    auxStrip.UploadToGPU();
  end for
end for
```

Figure 5.5: Level-of-detail extraction algorithm.

Visualization of the resulting mesh is implemented taking maximum advantage of the new characteristics of current GPUs. To accomplish this aim, we have applied the OpenGL extensions *glDrawRangeElements* and *glMultiDrawElements*. Both extensions allow us to render the geometry directly from the graphics hardware memory. In Figure 5.6, the *glDrawRangeElements* extension is applied for rendering. In the beginning, we activate the GPU buffers which contain the vertices and triangle strips. After that, we call the extension for each triangle strip to be rendered. It is important to underline that information about vertices and strips is wholly located in the GPU, and therefore this kind of information does not transit through the bus.

The drawing algorithm greatly improves rendering times, as shown in the results section. However, one call per primitive is needed to render the whole mesh. With *glMultiDrawElements*, we only need one function call to render more than one primitive such as a triangle strip, that is, the whole mesh. In Figure 5.7, we have an example of the utilization of this extension. Main difference between this extension and the previous one consists in the capability of this extension to manage arrays that contain information about size and location of triangle strips. As shown in the results sections, performance is greatly improved.

5.7. Results

5.7.1. Spatial cost

In Table 5.1, the storage cost for the *LodStrips* and *GPU* model are shown. It can be seen an improvement in this cost that varies from 23 to 30 per cent.

```
glBindBufferARB(GL_ARRAY_BUFFER_ARB, vertex_buffer);
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, strips_buffer);
for IndexStrip = 0 to NumberOfStrips - 1
  glDrawRangeElements (
    GL_TRIANGLE_STRIP, // Primitive
    0, // Beginning of Vertex Buffer
    NumberOfVertices - 1, // Ending of Vertex Buffer
    StripBufferSize(IndexStrip), // Triangle strip size
    GL_UNSIGNED_INT, // Indices data type
    StripBufferOffset(IndexStrip) );// Beginning of strip in Element Buffer
end for
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB,0);
glBindBufferARB(GL_ARRAY_BUFFER_ARB,0);
```

Figure 5.6: Drawing algorithm using the OpenGL *glDrawRangeElements* extension.

```
glBindBufferARB(GL_ARRAY_BUFFER_ARB, vertex_buffer);
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, strips_buffer);
glMultiDrawElements (
  GL_TRIANGLE_STRIP, // Primitive to render
  StripBufferSize, // Strip size array
  GL_UNSIGNED_INT, // Indices data type
  StripBufferOffset, // Strips beginning array
  NumberOfStrips ); // Number of triangle strips
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB,0);
glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);
```

Figure 5.7: Drawing algorithm using the OpenGL *glMultiDrawElemens* extension.

It is also important to underline that this modification improves the temporal cost of the model, as described in the following sections.

5.7.2. Temporal cost

A temporal comparison between the *LodStrips* and *GPU* models is shown in Figure 5.8. We compare extraction and drawing times. These charts show similar behaviors both extracting and drawing. Therefore, we have obtained a implementation of *LodStrips* that offers approximately the same temporal cost but saves around a 30 per cent in the spatial cost. OpenGL immediate mode

	#Strips	#LODs	Spatial Cost in Mb. LodStrips	GPU	Improvement
Cow	136	2614	0.22	0.17	23 %
Capone	177	3256	0.26	0.20	23 %
Boat	399	5474	0.55	0.40	27 %
Car	1396	19179	1.98	1.38	30 %
Bunny	1229	31351	2.88	2.21	23 %
Dragon	1787	48865	4.28	3.32	22 %
Phone	1747	74740	6.56	5.08	23 %
Isis	5141	168880	15.11	11.69	23 %
Buddha	31596	489329	46.95	35.51	24 %

Table 5.1: Improvement produced in the spatial cost when implemented with the new data structure.

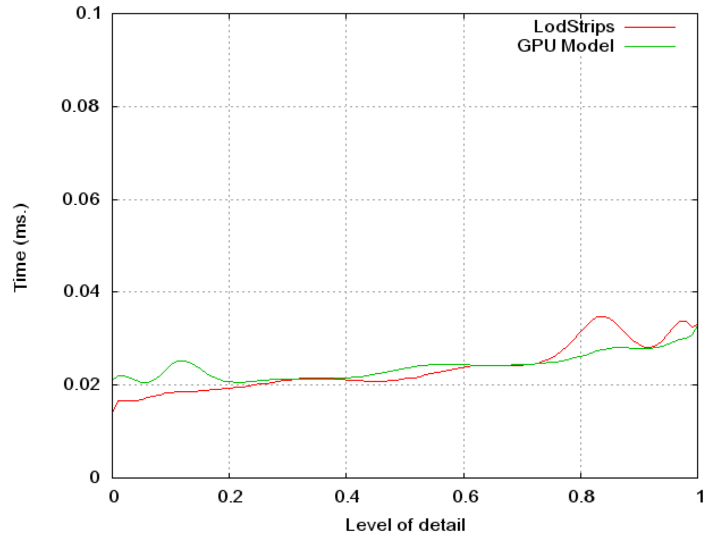
was used to render the models.

5.7.3. Hardware acceleration

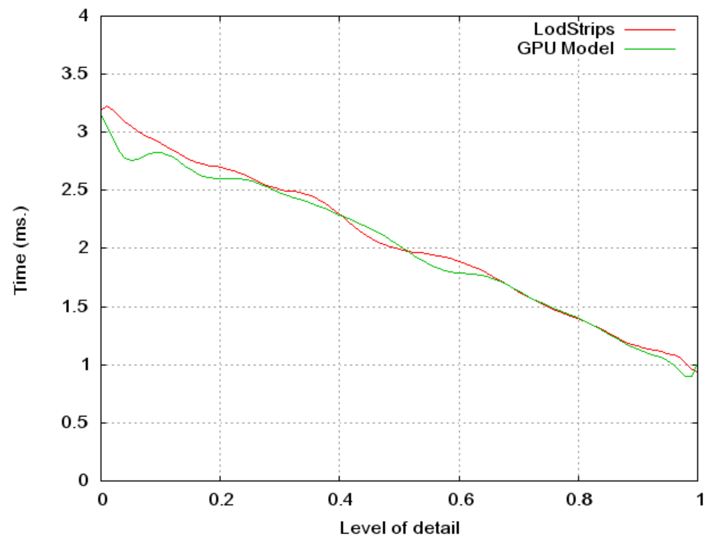
In Table 5.2, we observe an improvement of around ten times in the speed between using extensions and the immediate mode. In the immediate mode, for each frame all the vertex information, as well as triangle strip indices, are sent to the graphics system. However, the other modes only send a minimum amount of information that enables the GPU to correctly interpret data contained in its buffers. On comparing *glDrawRangeElements* and *glMultiDrawElements*, we notice a marked difference in the rendering times. This is due to the number of calls to the driver. While *glMultiDrawElements* calls it once per frame, with *glDrawRangeElements* we need as many calls as there are primitives to be rendered.

We highlight that *LodStrips* rendering times are similar to the *GPU model* when OpenGL immediate mode was used to render (Figure 5.8). Therefore, the *GPU model* is around ten times faster than the original *LodStrips* presented in the previous chapter.

It is important to underline that extensions can be used with any kind of triangle strips, that is, those generated both to minimize the number of vertices sent and to take advantage of the vertices cache. *LodStrips* is able to manage any set of triangle strips. In Figure 5.9, we have created two models from the Bunny object: one using the STRIPE algorithm and the other one with the NvTriStrip utility. However, in spite of taking rendering measures for both models with the *glMultiDrawElements* extension, they have similar extraction costs. In that figure, it is possible to observe how the Bunny object generated from the NVTriStrip Library shows better frame-per-second rates than the Stripe object when the level of detail is higher. However, we obtain



(a) Extraction times.



(b) Drawing times.

Figure 5.8: A comparison of extraction and drawing times comparison for the *LodStrips* and *GPU model* using the Bunny object (OpenGL immediate mode was used to better compare the models).

Immediate Mode		glDrawRange		glMultiDraw	
Render (s)		Render (s)		Render (s)	
%Rec	%Drw	%Rec	%Drw	%Rec	%Drw
16.38		1.64		1.22	
0.11	16.27	0.11	1.54	0.11	1.11

Table 5.2: LodStrips on GPU: Linear test with 1000 extractions applied to the Isis model (187644 vertices and 5141 triangle strips at the highest LOD) by applying different hardware acceleration techniques.

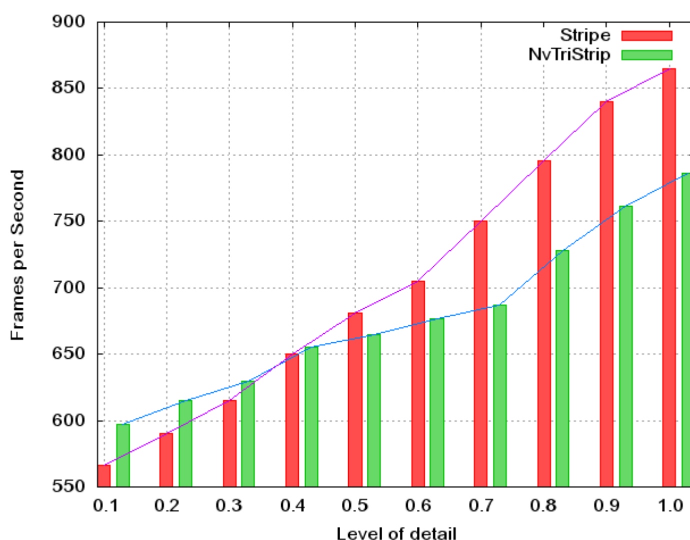


Figure 5.9: Bunny model performance comparison by using triangle strips generated by Stripe and NVidia algorithms. Rendering mode used was *gl-MultiDrawElements*.

better results with STRIPE at coarser levels of detail. This behavior is due to the vertex cache locality being lost whenever the model moves to coarser meshes.

5.8. Conclusions

In this chapter, we introduced a GPU version of the *LodStrips* model by modifying both the data structure and the drawing algorithms [RCRG06a]. As regards the data structure, we noticeable improved its spatial cost [RCRG06b].

It is important to underline how appropriate the *LodStrips* model is for

applying hardware acceleration techniques. This model spends a small percentage of time on extracting the level of detail, which leads to fast transitions between different levels of detail. Moreover, it benefits the application of those techniques. On the other hand, by using cache optimized triangle strips, performance is greatly improved, although it is important to notice that when the model moves to coarser meshes, vertex locality is lost and we lose the advantage over another kind of triangle strips. Hence, improvements in this way would be an interesting line in order to manage multiresolution schemes.

Moreover, the efficiency of the geometric acceleration techniques was tested on a multiresolution model. One of the most important conclusions that must be stressed here is that using hardware acceleration techniques allows us to increase the performance of the models with dynamic geometry. In this regard, the *LodStrips* model increased its performance around ten times.

CHAPTER 6

LodStrips for Deforming Meshes

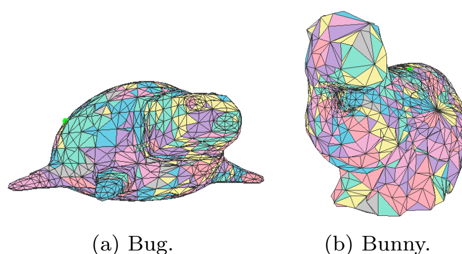
Applications such as video games or movies often contain deforming meshes. The most-commonly used representation of these types of meshes consists in dense polygonal models. Such a large amount of geometry can be efficiently managed by applying level-of-detail techniques and specific solutions have been developed in this field. However, these solutions do not offer a high performance in real-time applications. We thus introduce a multiresolution scheme for deforming meshes. It enables us to obtain different approximations over all the frames of an animation. Moreover, we provide an efficient connectivity coding by means of triangle strips as well as a flexible framework adapted to the GPU pipeline. Our approach enables real-time performance and, at the same time, provides accurate approximations.

6.1. Introduction

Nowadays, deforming surfaces are frequently used in fields such as games, movies and simulation applications. Due to their availability, simplicity and ease of use, these surfaces are usually represented by polygonal meshes.

A typical approach to represent these kind of meshes is to represent a different mesh connectivity for every frame of an animation. However, this would require a high storage cost and the time to process the animation sequence would be significantly higher than in the case of using a single mesh connectivity for all frames. Even so, these meshes often include far more geometry than

is actually necessary for rendering purposes. Many methods for polygonal mesh simplification have been developed (see Section 2.2). However, as multiresolution techniques for static meshes are based on a specific fixed shape, the meshes they produce can yield very poor approximations if the surface highly deforms. A single simplification sequence for all frames can also generate unexpected results (see Figure 6.1). Hence, multiresolution techniques for static meshes are not directly applicable to deforming meshes and so we need to adapt these techniques to this context.



(a) Bug. (b) Bunny.

Figure 6.1: A simplification sequence from the bug mesh applied to the bunny mesh.

Therefore, our goal consists in creating a multiresolution model for deforming meshes. We specifically design a solution for morphing meshes (see Figure 6.2), although it could be extended to any kind of deforming mesh. Our approach includes the following contributions:

- Implicit connectivity primitives: we benefit from using optimized rendering primitives, such as triangle strips. If compared to the triangle primitive, triangle strips lead us to an important reduction in the rendering and storage costs.
- A single mesh connectivity: for all the frames we employ the same connectivity information, that is, the same triangle strips. It generally requires less spatial and temporal cost than using a different mesh for every frame.
- Real-time performance: meshes are stored, processed and rendered entirely by the GPU. In this way, we obtain greater frame-per-second rates.

6.2. Background

6.2.1. Deforming meshes: morphing

A solution to approximate deforming meshes is to employ mesh morphing [ADSS99, Par05]. Morphing techniques aim at transforming a given source

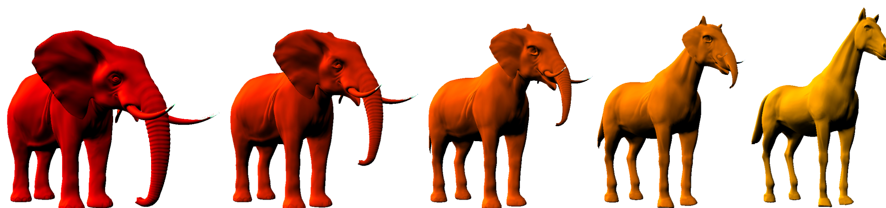


Figure 6.2: A deforming mesh: Elephant to horse morph sequence.

shape into a target shape, and they involve computations on the geometry as well as the connectivity of meshes.

In general, two meshes $M_0 = (T_0, V_0)$ and $M_1 = (T_1, V_1)$ are given, where T_0 and T_1 represent the connectivity (usually in triangles) and V_0 and V_1 the geometric positions of the vertices in R^3. The goal is to generate a family of meshes $M(t) = (T, V(t))$, $t \in [0, 1]$, so that the shape represented by the new connectivity T together with the geometries $V(0)$ and $V(1)$ is identical to the original shapes. The generation of this family of shapes is typically done in three subsequent steps:

- finding a correspondence between the meshes.
- generating a new and consistent mesh connectivity T together with two geometric positions $V(0)$, $V(1)$ for each vertex so that the shapes of the original meshes can be reproduced.
- creating paths $V(t)$, $t \in [0, 1]$, for the vertices.

The traditional approach to generate T is to create a *supermesh* [Par05] of the meshes T_0 and T_1, which is usually more complex in terms of geometry, than the input meshes. After the computation of one mesh connectivity T and two mesh geometries represented by vertex coordinates $V(0)$ and $V(1)$, we must create the paths. The most-used technique to create them is the linear interpolation, see Figure 6.2. Given a transition parameter t the coordinates of an interpolated shape are computed by:

$$V(t) = (1 - t)V(0) + tV(1), t \in [0, 1] \tag{6.1}$$

As commented before, connectivity information generated by morphing techniques usually gives rise to more dense and complex information than necessary for rendering purposes. In this context, we can make use of level-of-detail solutions to approximate such meshes and thus remove unnecessary geometry when required. We can also represent the connectivity of the mesh in triangle strips, which reduces in a factor of three the number of vertices to be processed [ESV96b].

6.2.2. Multiresolution

Some multiresolution models that benefit from using hardware optimized rendering primitives have recently appeared [ESAV99, VFG99, Ste01, SP03, BRR+04, RC04b]. However, as they are built from a fixed and static mesh, they usually produce low quality approximations when applied to a mesh with extreme deformations.

Some methods also provide multiresolution models for deforming meshes [MG03, SP00, DR05], but they are based on the triangle primitive and their adaptation to the GPU pipeline is potentially difficult or does not exploit it maximally. An important work introduced by Kircher et al. [KG05] is a triangle-based solution as well. This approach obtains accurate approximations over all levels of detail. However, temporal cost to update its simplification hierarchy is considerable, and GPU-adaptation is not a straightforward task.

6.2.3. GPU Pipeline

Recent GPUs include vertex and fragment processors, which have evolved from being configurable to being programmable. They execute shader programs in parallel. On the one hand, vertex shaders allow the programmer to alter per-vertex attributes, such as position, color, texture coordinates, and normal vectors. On the other hand, after the rasterizer has converted the transformed primitive to pixels, fragment shaders are used to calculate the color of a fragment, or per-pixel.

Mesh morphing techniques are also favored when they are employed directly in the GPU. With the current architecture of GPUs, it is possible to store the whole geometry in the memory of the GPU and to modify the vertex positions in real time to morph a *supermesh*. This would greatly increase performance. In order to obtain all the intermediate meshes, we can take advantage of the GPU pipeline to interpolate vertex positions by means of a vertex shader.

A combination of multiresolution techniques and GPU processing to deforming meshes can lead us to an approach that offers great improvements in rendering and, at the same time, high quality approximations.

6.3. Technical background

Starting from two arbitrary polygonal meshes, $M_0 = (T_0, V_0)$ and $M_1 = (T_1, V_1)$, where V_0 and V_1 are sets of vertices and T_0 and T_1 are the connectivity to represent these meshes, our approach is built upon two algorithms: We first obtain a *supermesh* (morphing builder) and later we build the multiresolution scheme (lod builder). The general construction process is shown in Figure 6.3.

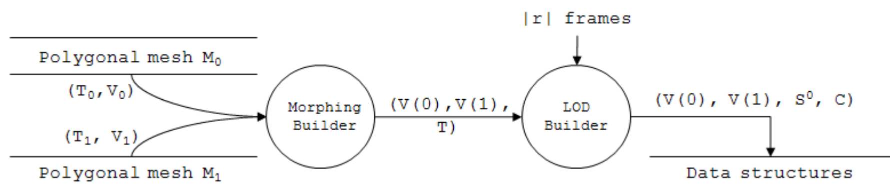


Figure 6.3: General construction process data flow diagram.

6.3.1. Generating morphing sequences

As commented before, linear interpolation is a well-known technique to create paths for vertices in morphing solutions. Vertex paths defined by this kind of technique are suitable to be implemented into recent GPUs offering considerable performance when generating intermediate meshes, $M(t)$. Thus, we apply a method [Par05] to first generate a family of meshes $M(t) = (T, V(t))$, $t \in [0, 1]$. As paths are linearly interpolated, we only need the geometries $V(0)$ and $V(1)$ and the connectivity information T, to reproduce the intermediate meshes $M(t)$ by applying the equation 6.1. The FaceToFace morphing sequence shown in Figure 6.8 was generated by using this method [Par05].

6.3.2. Construction of the multiresolution scheme

Once generated the *supermesh* ($M(t) = (T, V(t))$), we proceed to create the multiresolution model. A strip-based multiresolution scheme for polygonal models is preferred in this context as we obtain improvements both in rendering and in spatial cost. Thus, we perform an adaptation of the *LodStrips* multiresolution model to deforming meshes.

In order to construct the model, we perform two fundamental tasks by means of the LOD builder subprocess (see Figure 6.3). On the one hand, we generate the triangle strips to represent the connectivity by means of these primitives, and, on the other hand, we generate the simplification sequence which allows us to recover the different levels of detail. In Figure 6.4, the LOD Builder subprocess is shown.

Before constructing the multiresolution scheme, we need to convert the connectivity of the *supermesh* from triangles to triangle strips. This stripification process has been already explained in section 2.3.1. After this process, the *supermesh* is topologically equal to the previous one but represented by triangle strips: $M(t) = (S, V(t))$. However, we use the same connectivity information for all frames so that we only need to process one frame to compute the triangle strips.

As previously explained, a single simplification sequence for all frames of the animation can generate incorrect approximations (see Figure 6.1). Therefore, we generate the simplification sequence for each frame that we will consider

```
S=StripifySuperMesh(V(0),T) //Single connectivity
for i=0 to |r|-1
  t=i/(|r|-1) //t ∈ [0,1]
  M=TransformSuperMesh(S,V(t))
  C^i=SimplifyMesh(M)
end for
```

Figure 6.4: LOD Builder subprocess.

in the animation. This task is performed by modifying the t factor in the *supermesh*. The number of frames to be taken into account is called $|r|$. In this way, we linearly transform the *supermesh* by calculating C, which contains the set of changes to be applied in the multiresolution strips so that they represent the required level of detail.

After the general construction process has finished, we obtain the sets $\{V(0)$, $V(1)$, S, $C\}$, where $V(0)$ and $V(1)$ comes from the *Morphing builder*, S is the *supermesh* in triangle strips and C contains the sequences of simplification operations that enable us to change the resolution of the *supermesh* for each frame of the animation.

6.4. Representation in GPU

In the previous section, we described the sets required to represent our approach: $V(0)$, $V(1)$, S and C. According to the multiresolution morphing pipeline that we propose in Figure 6.5, our sets are implemented as follows: $V(0)$, $V(1)$ and S are located in the GPU, whereas C is stored in the CPU. In particular, $V(0)$ and $V(1)$ are stored as *vertex array buffers* and S^0 as an *element array buffer*, which offers better performance than creating as many buffers as there are triangle strips.

It is important to notice that we represent the geometry to be rendered by means of the data structures in the GPU, where the morphing process also takes place. On the other hand, the simplification sequence for every frame is stored in the CPU. These data structures are efficiently managed in runtime in order to obtain different approximations of a model over all the frames of an animation.

6.5. Rendering

Once construction has finished, we must build a level-of-detail representation with morphing during run-time. According to the requirements of the applications, it involves the extraction of a level of detail at a given frame.

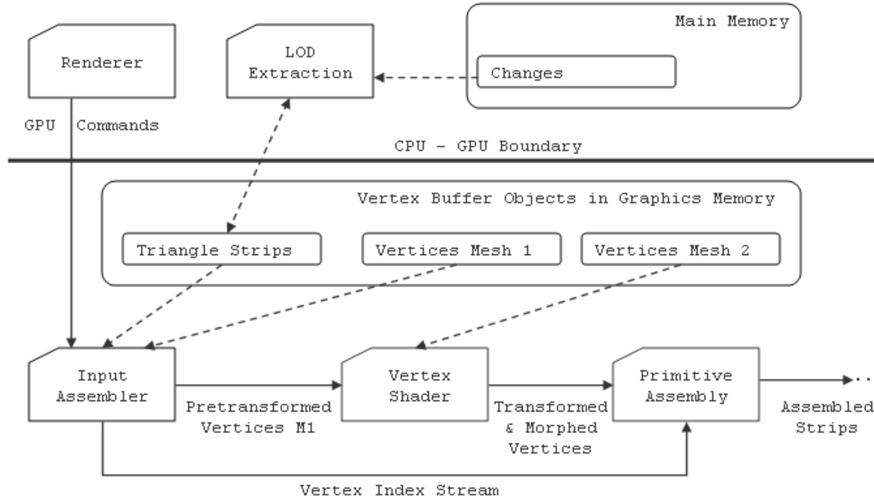


Figure 6.5: Multiresolution morphing pipeline using the current technology.

In Figure 6.5, we show the main functional areas of the pipeline used in our approach.

The underlying method to extract approximations of the models is based on the *LodStrips*. Among other advantages already commented, this model offers a low temporal cost when extracting any level of detail for strip-based meshes. We take advantage of this feature to perform fast updatings when traversing a *supermesh* from frame to frame in any level of detail.

Thereby, according to the frame and level of detail required by applications, the level-of-detail extraction algorithm is responsible for recovering the appropriate approximation in the triangle strips by means of the previously computed simplification sequence. In Figure 6.5, we show the general operation of this algorithm. It reads the simplification sequence of the current frame from the data structure *Changes*, and it modifies the triangle strips located in the GPU so that they always have the geometry corresponding to the level of detail used at the current time. A more detailed algorithm is shown in Figure 6.6.

After extraction, vertices must also be transformed according to the current frame in such a way that the deforming mesh is correctly rendered. When an application uses the GPU to compute the interpolation operations, the CPU can spend time improving its performance rather than continuously blending frames. Thus, by using the processing ability of the GPU, the CPU takes over the task of frame blending. Therefore, after extracting the required approximation, we directly compute the linear interpolations between $V(0)$ and $V(1)$ in the GPU by means of a vertex shader, see Figure 6.10.

Regarding the GPU pipeline, the first stage is the *Input Assembler*. The

purpose of this stage is to read primitive data, in our case triangle strips, from the user-filled buffers and assemble the data into primitives that will be used by the other pipeline stages. As shown in the pipeline-block diagram, once the *Input Assembler* stage reads data from memory and assembles the data into primitives, the data is output to the *Vertex Shader* stage. This stage processes vertices from the *Input Assembler*, performing per-vertex morphing operations. Vertex shaders always operate on a single input vertex and produce a single output vertex. Once every vertex has been transformed and morphed, the *Primitive Assembly* stage provides the assembled triangle strips to the next stage.

```
Function ExtractLODFromFrame (Frame,LOD)
  if Frame!=CurrentFrame then
    CurrentFrame=Frame;
    CurrentChanges=Changes[CurrentFrame];
    ExtractLevelOfDetail(LOD);
  else if LOD!=CurrentLOD then
    ExtractLevelOfDetail(LOD);
  end if
end Function
```

Figure 6.6: Extraction algorithm.

6.6. Results

Tests and experiments were carried out with a Dell Precision PWS760 Intel Xeon 3.6 Ghz with 512 Megabytes of RAM, the graphics card was an NVidia GeForce 7800 GTX 512. Implementation was performed in C++, OpenGL as the supporting graphics library and Cg as the vertex shader programming language.

The morphing models taken as a reference are shown in Figure 6.8 and Figure 6.9. We can observe the high quality of the approximations, some of which are reduced (in terms of number of vertices) by more than a 90 %.

6.6.1. Spatial cost

Spatial costs from the FaceToFace and HorseToMan morphing models are shown in Table 6.1. For each model, we specify the number of vertices and triangle strips that compose them (strips generated by means of the STRIPE algorithm [ESV96b]), the number of approximations or levels of detail available, the number of frames generated and, finally, the spatial cost in Megabytes. It

is divided into cost in the GPU (vertices and triangle strips) and cost in the CPU (simplification sequence). Finally, in the ratio column, we show the cost per frame, calculated as the total storage cost divided by the number of frames. As expected, the cost of storing the simplification sequence of every frame is the most important part of the spatial cost.

Morphing model		FaceToFace	HorseToMan
#Verts		10,520	17,489
#Strips		620	890
#LODs		9,467	15,738
#Frames		25	26
Cost per Frame	GPU	14.2 KB.	22.4 KB.
	CPU	472.1 KB.	848.3 KB.
	Total	486.3 KB.	870.7 KB.

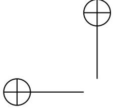
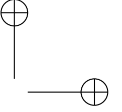
Table 6.1: Spatial cost.

6.6.2. Temporal cost

Results shown in this section were obtained under the conditions mentioned above. Levels of detail were in the interval [0, 1], zero being the highest LOD and one the lowest. Geometry was rendered by using the *glMultiDrawElements* OpenGL extension, which only sends the minimum amount of information that enables the GPU to correctly interpret data contained in its buffers. With *glMultiDrawElements* we only need one call per frame to render the whole geometry.

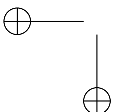
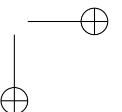
In Figure 6.7a, we show the level-of-detail extracting cost per frame of the FaceToFace morphing sequence. The per-frame time to extract the required level-of-detail ranges between 6 % and 1.4 % of the frame time. If we consider the lowest level of detail as being the input mesh reduced by 90 % (see LOD 0.9 in Figure 6.7a), we obtain times around 6 % of the frame time, which offers us better perfomance than other related works such as [KG05], which employs more time in changing and applying the simplification hierarchy.

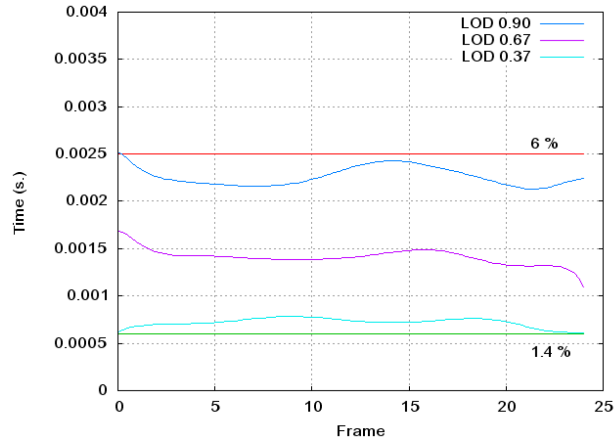
We performed another test by extracting one approximation every 24 frames and, at the same time, we progressively changed the level of detail. This was carried out to simulate an animation which is switching its LOD as it is further from the viewer. In Figure 6.7b, we show the results of this test. As expected, our approach is able to extract and render different approximations over all frames of an animation at considerable frame-per-second rates.

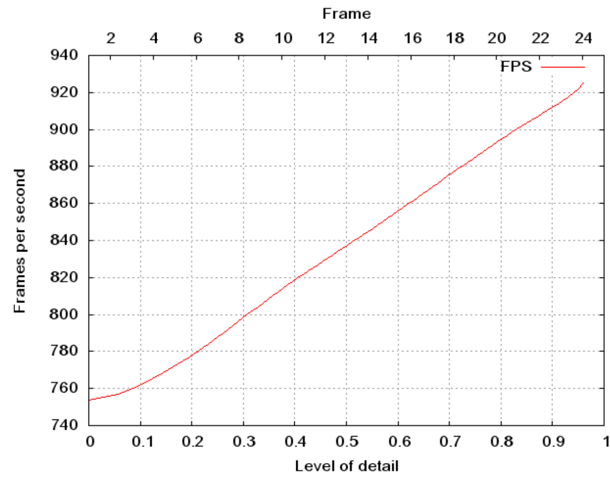
6.7. Conclusions

We have introduced a multiresolution scheme suitable for deforming meshes such as those generated by means of morphing techniques. A solution for morphing sequences was specially designed, although it can be adapted to any kind of deformed mesh by storing the vertex positions of every frame within the animation. We also share the same connectivity information and we store the whole geometry in the GPU, thus saving bandwidth in the typical CPU-GPU bottleneck. Morphing is also computed in the GPU by exploiting its parallelism. We thus obtain real-time performance at great frame-per-second rates. At the same time, we offer high quality approximations in every frame of an animation.



(a) Level-of-detail extraction cost per frame of the FaceTo-Face morphing model at a constant rate of 24 fps.



(b) Frame-per-second rates by performing one extraction every 24 frames. Results obtained by using the FaceTo-Face morphing model.

Figure 6.7: Temporal cost of the model.

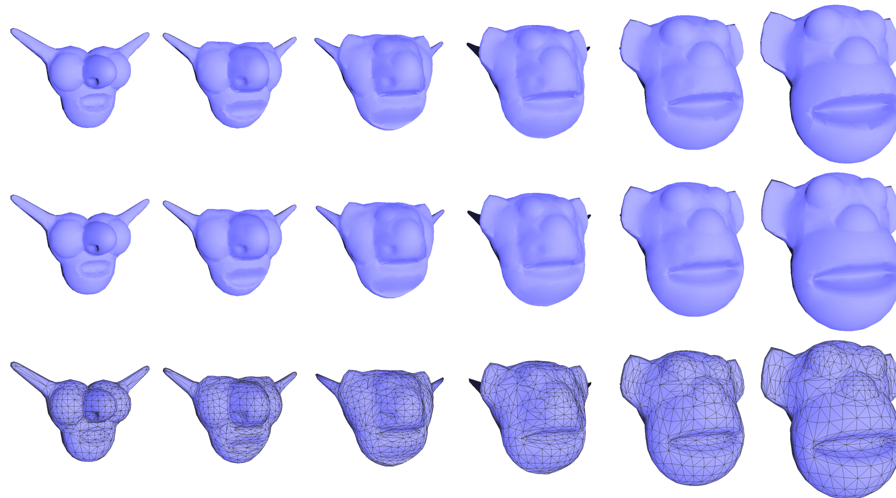


Figure 6.8: Multiresolution morphing sequence for the FaceToFace model. Rows mean level of detail, 10,522 (original mesh), 3,000 and 720 vertices, respectively, and columns morphing adaptation, aproximations were taken with t=0.0, 0.2, 0.4, 0.6, 0.8 and 1.0, respectively.

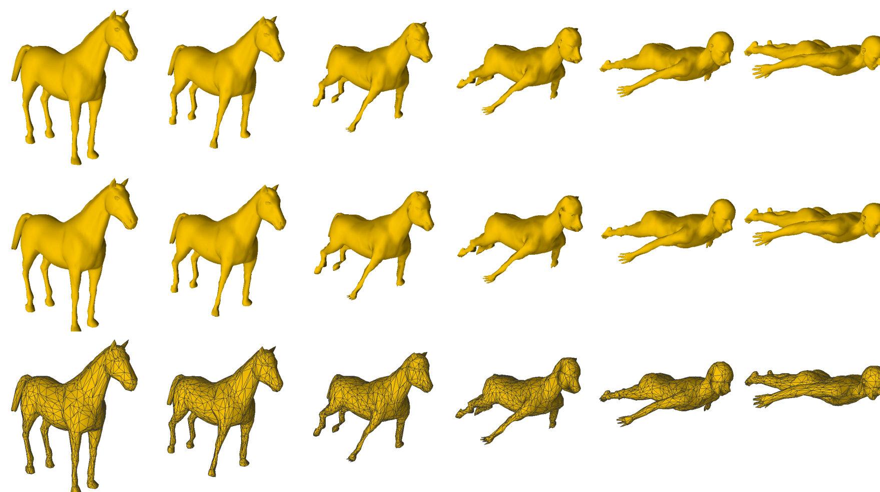
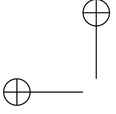
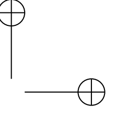
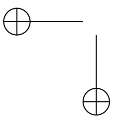
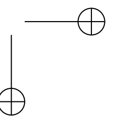


Figure 6.9: Multiresolution morphing sequence for the HorseToMan model. Rows mean level of detail, 17,489 (original mesh), 5,000 and 1,000 vertices, respectively, and columns morphing adaptation, aproximations were taken with t=0.0, 0.2, 0.4, 0.6, 0.8 and 1.0, respectively.

```
struct appdata
{
   float3 positionA : POSITION;
   float3 normalA   : NORMAL;
   float3 positionB : TEXCOORD1;  //Position
   float3 normalB   : TEXCOORD2;  //Normal
};

struct vfconn
{
   float4 HPos : POSITION;
   float4 Col  : COLOR0;
};

vfconn main( appdata IN,
             uniform float4x4 ModelViewProj,
             uniform float keyFrameBlend,
             uniform Light light)
{
   vfconn OUT;     // Variable to handle our output from the vertex
                   // shader (goes to a fragment shader if available).

   float3  positionF   = lerp(IN.positionA,
                              IN.positionB,
                              keyFrameBlend);
   float3  blendNormal = lerp(IN.normalA,
                              IN.normalB,
                                 keyFrameBlend);

   float3 normal = normalize(blendNormal);

   // Transform The Vertex Position Into Homogenous Clip-Space
   OUT.HPos = mul(ModelViewProj, float4( positionF,1));

   OUT.Col  = computeLighting(light, positionF, normal);

   return OUT;
}
```

Figure 6.10: CG implementation of the Vertex shader.

CHAPTER 7

Applications

The integration of *LodStrips* into computer graphics applications implies having an efficient interface between the application and the geometry modules. Thus, we have developed an independent library to satisfy these requirements. It has also been designed to be API independent, thus allowing client applications to use any graphics library to manage the geometry.

7.1. LodStrips Library

The *LodStrips* library provides the ability to construct and render interactive arbitrary 3D meshes in 3D applications. The *LodStrips* class represents a general mesh which is able to change its level of detail.

This module contains functions that handle the levels of detail of the input multiresolution polygonal meshes. For any given resolution of an object, this module returns a set of triangle strips representing the object at that resolution, that is, at the requested level of detail. These modules use *LodStrips* and, therefore, they take advantage of triangle strips to reduce storage usage and to speed up realistic rendering.

A simple specification of the library is shown as follows:

Public Member Functions

- **LodStripsLibrary** (char *fileGeometryMesh, char *fileDecimationMesh,VertexData ∗uservertexdata, IndexData ∗userindexdata)
    - *Class constructor. Constructs a LodStrips multiresolution object from:*

- *The triangle strips geometry information contained in the fileGeometryMesh file.*
- *The decimation information contained in the fileDecimationMesh file.*
- *A user-defined IndexData instance.*
- *A user-defined VertexData instance.*

- ~**LodStripsLibrary** (void)
  - *Class destructor.*
- void **GoToLod** (Real)
  - *Changes the level of detail of the object to a specified factor. The value specified to change the LOD must be in the range [0,1] ([min,max]). After the LOD has been calculated, this function automatically updates the indices using the IndexData interface provided in the constructor.*
- uint32 **MaxFaces** () const
  - *Returns the number of triangles at the highest LOD.*
- uint32 **MinFaces** () const
  - *Returns the number of triangles at the lowest LOD.*
- uint32 **GetValidIndexCount** (int submeshid) const
  - *Returns the index count at the current LOD of a certain mesh.*
- uint32 **GetTotalStripCount** (void) const
  - *Retrieves the total number of strips in a mesh.*
- uint32 **GetSubMeshtripCount** (int submeshid) const
  - *Returns the number of strips in a given submesh.*
- uint32 **GetCurrentTriangleCount** (void) const
  - *Obtains the triangle count at the current LOD.*
- Real **GetCurrentLodFactor** (void) const
  - *Obtains the current LOD factor.*

7.1.1. The VertexData and IndexData interfaces

The *LodStripsLibrary* class is able to change the level of detail of an object. It has been designed to be API independent. However, updating indices or vertices of a mesh is an API-dependent task because it is dependent on how the client application stores them to render the geometry. For example, the client application can use OpenGL or Direct3D, where management of their vertices and indices is different, but our library must support them.

Therefore, the *LodStripsLibrary* class knows where to store the vertices and how to calculate the new set of indices to render the geometry, but it does not know how or where to retrieve vertices or store the resulting indices. To solve this, we developed our library using VertexData and IndexData abstraction interfaces. The *LodStrips* algorithm uses these interfaces to communicate with the client code in order to retrieve the vertices and set the indices at a given LOD.

The user must inherit a custom class from the VertexData and IndexData interfaces and implement their virtual methods to provide the desired functionality. Thus, instances of the user custom VertexData and IndexData classes will be passed to the *LodStripsLibrary* class at creation time. The code below shows both class interfaces.

VertexData class

```
class VertexData
{
public:
  VertexData(void){}
  virtual ~VertexData(void){}

  virtual void Begin(unsigned int numinds)=0;
  virtual void SetVertex(unsigned int i, float x, float y, float z)=0;
  virtual void End(void)=0;
};
```

The methods above will be called when a *LodStrips* instance changes the 3D coordinates of the mesh. The meaning of each method is described below:

- Begin(numverts) indicates the number of vertices to be modified on the mesh. This is a good place to lock a vertex buffer.
- void SetVertex(unsigned int i, float x, float y, float z) specifies the new 3D coordinates for vertex i.
- End() indicates that the changes made to the mesh are finished. This is a good point to unlock a vertex buffer.

IndexData class

```
class IndexData
{
public:
  IndexData(void){}
  virtual ~IndexData(void){}

  virtual void Begin(unsigned int numinds)=0;
  virtual void SetIndex(unsigned int i, unsigned int index)=0;
  virtual void End(void)=0;
};
```

The methods above will be called when a *LodStrips* instance must change the level of detail. The meaning of each method is described below:

- Begin(numinds) it indicates the number of indices to be modified on the mesh. This is a good place to lock an index buffer.
- SetIndex(i,index) it specifies the new value for the index at the position i.
- End() it indicates that the changes made to the mesh are finished. This is a good point to unlock an index buffer.

7.1.2. LOD switching

After the multiresolution has been instantiated, LOD switching involves finding the new set of indices that describe the object at a given LOD. LodStrips class calculates this set of indices and applies them to the mesh with the IndexData user interface implemented by the user. To change the LOD of a multiresolution object the user must call the method `GoToLod()` specifying the desired level of detail in the range [0,1] (0 means the lowest LOD and 1 the highest).

```
myLodStripsObject->GoToLod(lodfactor); // lodfactor in [0,1]
```

After calling this function, the target mesh should have effectively changed its level of detail and it should be ready to be rendered.

7.1.3. Usage

First of all, we need two files containing information about geometry and decimation, that is, the LOD information required to run the *LodStrips* algorithm. To use this file in an external application such a *LodStrips* object the general process would be:

1. LOD Initialization:
    a) Load a mesh file with the geometry.
    b) Load a mesh file with the decimation information.
    c) Create an instance of LOD model feeding it with those parameters.
2. LOD Switching. To change the level of detail:
    a) Call the GoToLOD method.

We highlight that the previous scheme would also be suitable to implement a multiresolution model based on triangles.

7.2. Applications: LodStrips in Ogre

7.2.1. Overview

OGRE (Object-Oriented Graphics Rendering Engine) is a flexible scene-oriented 3D engine. It was written in C++ and designed to make it easier and more intuitive for developers to produce applications utilizing hardware-accelerated 3D graphics. The class library abstracts all the details of using the underlying system libraries like Direct3D and OpenGL and provides an interface based on world objects and other intuitive classes. OGRE can be (and indeed has been) used to create games, but OGRE was deliberately designed to provide just a world-class graphics solution; for other features like sound, networking, AI, collision, physics and so on, it is necessary to integrate it with other libraries.

As regards licensing, the Ogre source is made available under the GNU Lesser General Public License (LGPL), which basically means that the source of any changes to the core engine must be released if a product is distributed. The sources of the application or of new plugins that are created do not have to be released.

In Figure 7.1, we show a diagram of the main core objects of OGRE. At the very top of the diagram is the Root object. This is the *way in* to the OGRE system and the place where the top-level objects, like scene managers, rendering systems and render windows, loading plugins and so on, are created. The rest of OGRE's classes are located in one of these three groups:

- Scene manager: This is concerned with the contents of the scene, how it is structured, how it is viewed from cameras, and so forth. Objects in this area are responsible for providing a natural, declarative interface to the world in construction.
- Resource management: All rendering needs resources, whether it is geometry, textures, fonts, etc. It is important to manage the loading, re-use and unloading of these carefully, and that is what classes in this area do.

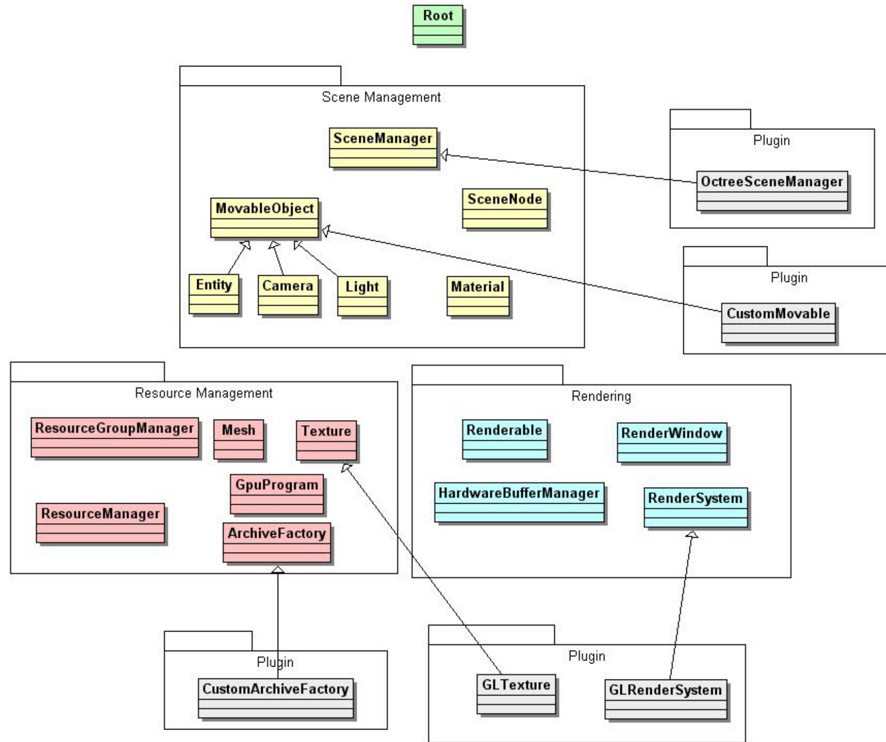


Figure 7.1: Ogre system overview.

- Rendering: This is about the lower-level end of the rendering pipeline, the specific rendering system API objects like buffers, render states and the like and pushing it all down the pipeline. Classes in the Scene management subsystem use this to obtain their higher-level scene information onto the screen.

With reference to plugins, OGRE was designed to be extended and plugins are the usual way to do so. Many of the classes in OGRE can be subclassed and extended, whether it is by changing the scene organization through a custom SceneManager, adding a new render system implementation (e.g. Direct3D or OpenGL), or providing a way to load resources from another source (e.g. from a web location or a database). As a result, OGRE is not just a solution for a single defined problem, it can extended as much as is needed.

Figure 7.2: Ogre and LodStrips interaction.

7.2.2. LodStrips integration

Integration into Ogre-based applications is a very straightforward task. This is due to the fact that the *LodStrips* library was initially designed with this engine in mind. Integration of the *LodStrips* library into existing applications and game engines like OGRE is a very important task. This library was designed to be as flexible as possible, so that it can be successfully integrated into any situation.

As regards meshes, OGRE was designed to use discrete multiresolution models. In particular, mesh objects are the basis for the individual movable objects in the world, which are called entities. An entity is an instance of a movable object in the scene. The only assumption is that it does not necessarily have a fixed position in the world. Entities are based on discrete meshes, which are represented by the Mesh object. Multiple entities can be based on the same mesh, since multiple copies of the same type of object in a scene can be created.

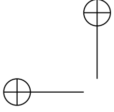
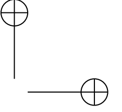
In this context, we have integrated the *LodStrips* library in such a way that it is not necessary to recompile the engine. In the client application, we simply modify the vertex indices used by a mesh object through a *LodStrips* object. In Figure 7.2 we show a simple scheme representing how we interact with mesh objects.

7.2.3. Results

We have developed a number of demos to demonstrate the *LodStrips* library and its integration into a graphics rendering engine: The Ogre Rendering Engine.

Figure 7.3 shows the *LodStrips* multiresolution library in runtime. The application renders a group of models which are able to change their level of detail depending on the distance of the group of objects from the camera. The information panel in the bottom-left corner of the screen shows the current LOD factor, frames per second and the amount of geometry sent to the renderer. The level of detail can be calculated in two ways: automatically, based on the distance of the group from the camera, and manually, where the user changes the level of detail regardless of the distance. This last mode is useful to see the meshes in detail even when their level of detail is set to the minimum by the automatic mode.

In Figure 7.4, we can see several *LodStrips* objects running in OGRE. They

are rendered at different levels of detail according to their distance to the camera. In this demo, each color means a different level of detail.

7.3. Conclusions

The *LodStrips* library which we have developed, provides the ability to construct and render interactive meshes in 3D applications. Using this library, we implemented *LodStrips* in a real-time application. The Ogre3D game engine was extended with this library and it thus enabled Ogre3D to manage continuous multiresolution models.

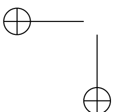
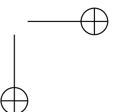



Figure 7.3: Screenshot of a LodStrips demo.

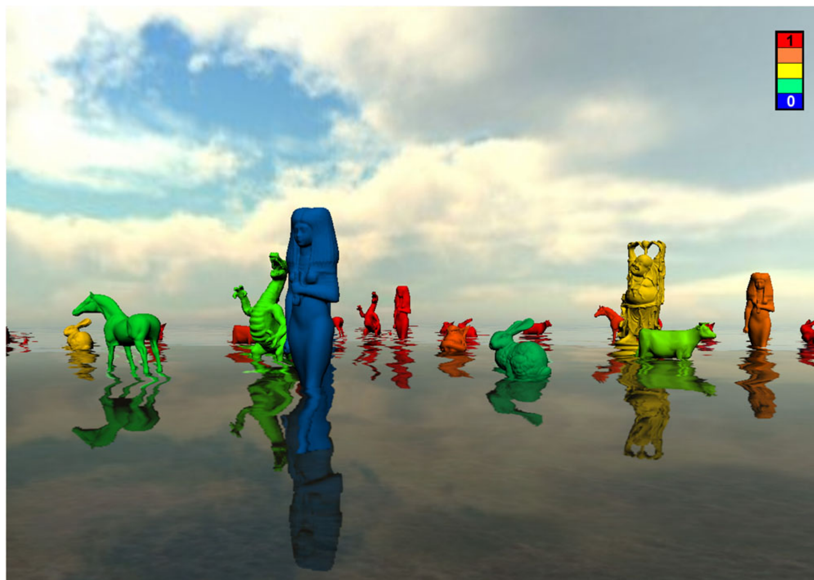
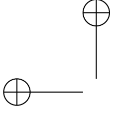
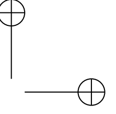
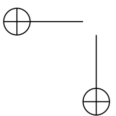
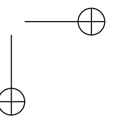


Figure 7.4: LodStrips running in OGRE. The colors denote the level of detail used. According to the distance to the viewer, closer objects (in blue) are more detailed and further ones (in red) less detailed.

Chapter 8
Conclusions and Future Work

In this dissertation, we focus on improving the interactive render of polygonal meshes. To tackle the problem, we have combined geometry simplification and level of detail techniques. Thus, we have defined a multiresolution model that represents any polygonal mesh at any given resolution. This approach is able to manage continuous level-of-detail by smoothly adapting mesh resolution to the application requirements. Moreover, the model can be perfectly integrated into current GPUs and it was successively implemented in some real-time applications.

This chapter is organized as follows: we summarize the conclusions in section 8.1. Later, in section 8.2 we introduce the different publications derived from our work and finally future work lines are presented in 8.3.

8.1. Conclusions

In this section, we review the concepts and results achieved during the development of this dissertation:

- In chapter 2, we introduced previous works in multiresolution modeling, presenting the different existing techniques in the field of real-time visualization of geometric models and analyzing the most outstanding papers to date. From an application point of view, the use of a discrete or continuous multiresolution model in an application depends on its requirements. Discrete multiresolution models are a good alternative in applications which require few levels of detail in their objects. These models also offer a simple implementation and these levels of detail can be polished

and improved considerably. Nevertheless, continuous models offer a high granularity, allowing us to adapt exactly the level of detail desired in each case. Furthermore, their evolution has allowed us to obtain very low level-of-detail extraction times and significant accelerations due to the irruption of the new and powerful GPUs. As previously commented, multiresolution models can be divided into uniform and variable resolution. Although variable resolution schemes are powerful and very flexible, they usually has a high storage cost and they are usually slower than uniform ones. Obviously, requirements will determine its applicability. The design of schemes for specific applications, such as computer games or interactive visualization of vegetation, are open lines.

- In chapter 3, we introduced a simple and effective triangle strips and data structure management for real-time rendering of multiresolution meshes. Conceptually, it enables multiresolution meshes to gain the rendering speed-up from using optimized rendering primitives, such as triangle strips. We introduced an approach that improves the time required to extract different levels of detail in multiresolution schemes based on triangle strips. Moreover, by means of a filter in the visualization, it removes most degenerate triangles produced in the lower levels of detail of the model. This approach offers the following features:

  - Uniform and variable resolution: this scheme permits to extract one or several levels of detail on the surface of a polygonal mesh.
  - Low storage cost: cost diminishes drastically compared to previous multiresolution models.
  - Efficiency: with low level-of-detail recovery time, we obtain better results than MTS [BRR+01], one of the first models wholly based on triangle strips.
  - Speed: using triangle strips, rendering is faster than other models based on triangles.

- In chapter 4, a new uniform multiresolution method was presented: *LodStrips*. This model allows us to efficiently transit between different levels of detail in real-time applications. The *LodStrips* model offers many advantages and it should be underlined that it is a model with only three simple data structures and it is easy to implement. Moreover, it offers a fast level of detail extraction which allows us to obtain smooth transitions between levels of detail, as well as very good rendering times because extraction is usually an important part of the total rendering time. This model is wholly based on the triangle strips primitive, which leads to an important reduction in storage and rendering costs. Reduction in spatial cost has been shown in the previous chapters, where it can be seen how a *LodStrips* model can store an object in a memory space that is similar

in size to two times the original triangle strips mesh size. Hence, we can allocate more objects in memory than with other models. It has also been shown how *LodStrips* has a much better rendering time than other well-known models. It also allows us to obtain a fast rendering model, which is important in critical applications and, what is more, it accelerates scene rendering by increasing the frame per second rate.

- In chapter 5, we introduced a GPU version of the *LodStrips* model by modifying both the data structure and the drawing algorithms. As regards the data structure, we noticeable improved its spatial cost. Moreover, the efficiency of the geometric acceleration techniques was tested on this multiresolution model. One of the most important conclusions that must be stressed here is that using hardware acceleration techniques allows us to increase the performance of the models with dynamic geometry. In this regard, the *LodStrips* model increased its performance around ten times. This rise is mainly due to the design of the model that has been optimized for the hardware, where level-of-detail extraction times are very low and so graphic acceleration is greatly benefited by avoiding long waits to render approximations.

- In chapter 6, we introduced a multiresolution scheme suitable for deforming meshes such as those generated by means of morphing techniques. We share the same connectivity information and we store the whole geometry in the GPU, thus saving bandwidth in the typical CPU-GPU bottleneck. Morphing is computed in the GPU by exploiting its parallelism. We thus obtain real-time performance at great frame-per-second rates. At the same time, we offer high quality approximations in every frame of an animation.

- In chapter 7, we designed and developed an independent library to integrate *LodStrips* in 3D applications. An efficient interface between the application and the geometry modules was designed and implemented. This library provides the ability to construct and render interactive 3D meshes in 3D applications. This module contains functions that handle the levels of detail of the input multiresolution polygonal meshes. For any given resolution of an object, this module returns a set of triangle strips representing the object at that resolution, that is, at the level of detail requested. Obviously, these models use *LodStrips*, they therefore take advantage of triangle strips to reduce storage usage and to speed up realistic rendering. Finally, the Ogre3D game engine was extended with this library and it thus enabled Ogre3D to manage continuous multiresolution models.

8.2. Publications

In this section, publications related to this dissertation are detailed:

- **Tiras de Triángulos con Nivel de Detalle**
  F. Ramos, P. Castelló, M. Chover CEIG 2004
  339-342. 2004

- **Explotación del Hardware Gráfico para Acelerar la Visualización de Geometría**
  P. Castelló, F. Ramos, M. Chover CEIG 2004
  363-366. 2004

- **A Comparison of Multiresolution Modeling in Real-Time Terrain Visualization**
  C. Rebollo, I. Remolar, M. Chover, F. Ramos
  Lecture Notes in Computer Science 3044. ICCSA 2004
  Springer-Verlag. 2. 703-712. 2004

- **An Approach to Improve Strip-based Multiresolution Schemes**
  F. Ramos, M. Chover, O. Belmonte, C. Rebollo
  WSCG 2004
  Journal of WSCG. 12(1). 349-354. 2004

- **A comparison of strip-based multiresolution models**
  F. Ramos, M. Chover
  VIIP 2004
  Acta Press. 239-244. 2004

- **LodStrips: Level of Detail Strips**
  F. Ramos, M. Chover
  Lecture Notes in Computer Science 3039. ICCS 2004
  Springer-Verlag. 4. 107-114. 2004

- **Level of Detail Modeling in a Computer Game Engine**
  F. Ramos, M. Chover
  Lecture Notes in Computer Science 3166. ICEC 2004
  Springer-Verlag. 4. 451-454. 2004

- **Variable Level of Detail Strips**
  F. Ramos, M. Chover
  Lecture Notes in Computer Science 3044. ICCSA 2004
  Springer-Verlag. 2. 622-630. 2004

- **Real-Time Terrain Rendering using LodStrips Multiresolution Model**
  F. Ramos, M. Chover, C. Granell
  DEXA 2004
  819-823. 2004

- **LodStrips in a Game Engine**
  F. Ramos, M. Chover
  ICCCT 2004
  288-292. 2004

- **A Comparative Study of Acceleration Techniques for Geometric Visualization**
  P. Castelló, F. Ramos, M. Chover
  Lecture Notes in Computer Science 3515. ICCS 2005
  Springer-Verlag. 2. 240-247. 2005

- **Quality Strips for Models with Level of Detail**
  O. Ripolles, M. Chover, F. Ramos
  VIIP 2005
  ACTA Press. 268-273. 2005

- **Efficient Implementation of LodStrips**
  F. Ramos, M. Chover, O. Ripollés, C. Granell
  VIIP 2006
  ACTA Press. 365-370. 2006

- **Continuous Level of Detail on Graphics Hardware**
  F. Ramos, M. Chover, O. Ripollés, C. Granell
  Lecture Notes in Computer Science 4245. DGCI 2006
  Springer-Verlag. 460-469. 2006

- **Level-of-Detail Triangle Strips for Deforming Meshes**
  F. Ramos, M. Chover, I. Kolingerova, J. Parus

Accepted in CGGM 2008. Lecture Notes in Computer Science.

- **Adaptive Keyframing Animation on the GPU using Triangle Strips**
  F. Ramos, M. Chover
  Submitted to Visual Computer.

8.3. Future work

There exist several open areas to research in the line of this work. In this section, we present some of them.

Stripification

The process to convert a polygonal mesh, usually composed of triangles, into triangle strips is often called stripification. As explained in previous chapters, there are a lot of heuristics to calculate them. However, schemes which take advantage of vertex cache are not numerous and they only optimize a static mesh. It would be interesting to develop stripification methods that optimize vertex cache reuse for each level of detail.

Out of core

Present representations and extraction algorithms are not scalable for models made up of tens or hundreds of millions of polygons. The extraction cost is proportional to the size of the model, and it can be prohibitive for massive models. Another future line is to develop an out-of-core multiresolution method following the *LodStrips* philosophy.

Comparison tool

Regarding simplification, we have tools to compare methods and thus, to determine different characteristics to improve them, that is, to tune them. In multiresolution, we do not have this kind of tool. Thus, a general tool to compare multiresolution methods can help researchers to efficiently develop new multiresolution models and, what is more, to tune them.

Adaptive morphing with level of detail

Deforming meshes are present in real-time applications. That surfaces are often represented as dense polygonal meshes with static connectivity. However, such high resolution meshes are unnecessary and undesired in many environments. Many works has addressed the simplification of static meshes but they

are not adequate for deforming meshes. We created a multiresolution model for deforming meshes based on triangle strip primitive. However, this model is view-independent. A view-dependent approach to manage multiresolution models based on triangle strips with morphing is another open line.

GPU

Nowadays GPUs offer new capabilities that, when exploited to the maximum, can offer very good results in several aspects. New units have been included in the graphics pipeline: geometry shaders. A geometry shader can generate new primitives from existing primitives like vertices, lines, triangles and triangle strips. It offers us the possibility of simplify a model in real-time in the GPU, avoiding the traffic needed to update the triangle strips located in the GPU buffers. Another open line is to develop a model that stores the information to change the triangle strips into textures within GPU memory. Later, at runtime, the geometry of the mesh would be modified by means of a geometry shader.

Bibliography

[ADSS99] L. Aaron, D. Dobkin, W. Sweldens, and P. Schroder. Multiresolution mesh morphing. In *SIGGRAPH*, pages 343–350, 1999. 6.2.1

[AHB90] K. Akeley, P. Haeberli, and D. Burns. The tomesh.c program. Technical report, 1990. 2.3.1

[AHMS96] E. Arkin, M. Held, J. S. B. Mitchell, and S. Skiena. Hamiltonian triangulations for fast rendering. *The Visual Computer*, 12(9):429–446, 1996. 2.3.1

[Ale02] M. Alexa. Recent advances in mesh morphing. *Computer Graphics Forum*, 21(2):1–23, 2002. 1.2

[AM04] H. Annaka and T. Matsuoka. Memory efficient adjacent triangle connectivity of a vertex using triangle strips. In *CGI*, pages 278–281, 2004. 2.3.1, 2.4.2

[BD02] C. Beeson and J. Demer. Nvstristrip library. 2002. 2.3.1, 5.2

[BG99] A. Bogomjakov and C. Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. In *Graphics Interface*, pages 81–90, 1999. 2.3.1, 2.4.2

[BRR+01] O. Belmonte, I. Remolar, J. Ribelles, M. Chover, C. Rebollo, and M. Fernandez. Multiresolution triangle strips. In *VIIP*, pages 182–187, 2001. 1.3, 2.3.1, 2.4.2, 2.4.2, 2.5, 3.5, 5.2, 8.1

[BRR+04] O. Belmonte, I. Remolar, J. Ribelles, M. Chover, and M. Fernandez. Efficient use connectivity information between triangles in a mesh for real-time rendering. *Computer Graphics and Geometric Modelling*, 20(8):1263–1273, 2004. 6.2.2

[BS05] T. Boubekeur and C. Schlick. Generic mesh refinement on gpu. In *Graphics Hardware*, pages 99–104, 2005. 2.4.1, 2.4.2, 2.5

[CCMS97] A. Ciampalini, P. Cignoni, C. Montani, and R. Scopigno. Multiresolution decimation based on global error. *The Visual Computer*, 13(5):228–246, 1997. 2.2.2, 2.2.3

[Cho97] M. Chow. Optimized geometry compression for real-time rendering. In *Visualization*, pages 347–354, 1997. 2.4.2

[CHVJ05] L. Chang Ha, A. Varshney, and D. Jacobs. Mesh saliency. In *SIGGRAPH*, pages 659–666, 2005. 2.2.3

[Cla76] J. Clark. Hierarchical geometric models for visible surface algorithms. *CACM*, 10(19):547–554, 1976. 2.1, 2.4, 3.1

[CMS98] P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification methods. *Computer and Graphics*, 1(22):37–54, 1998. 2.2

[CSCF07] P. Castello, M. Sbert, M. Chover, and M. Feixas. Viewpoint entropy-driven simplification. In *CGVCV*, pages 249–256, 2007. 2.2.3

[DFMP98] L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulations. In *IEEE Visualization 98*, pages 43–50, 1998. 2.4.2, 2.4.2, 2.5

[DGBGP06] P. Diaz-Gutierrez, A. Bhushan, M. Gopi, and R. Pajarola. Singlestrips for fast interactive rendering. *The Visual Computer*, 22(6):372–386, 2006. 2.3.1

[DP02] C. Decoro and R. Pajarola. Xfastmesh: Fast view-dependent meshing from external memory. In *IEEE Visualization*, pages 363 – 370, 2002. 1.1, 2.4.2, 2.5

[DR05] C. Decoro and S. Rusinkiewicz. Pose-independent simplification of articulated meshes. In *Symposium on Interactive 3D Graphics*, 2005. 6.2.2

[DVS03] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. Sequential point trees. *ACM Transactions on Graphics*, 22(3):657 – 662, 2003. 2.4.2, 2.5

[DZ91] M. DeHaemer and J. Zyda. Simplification of objects rendered by polygonal approximations. *Computer and Graphics*, 2(15):175–184, 1991. 2.2

[EMB01] C. Erikson, D. Manocha, and W. Baxter. Hlods for faster display of large static and dynamic environments. In *Symposium on Interactive 3D Graphics*, pages 111–120, 2001. 2.4.2, 2.4.2, 2.5

[ESAV99] J. El-Sana, E. Azanli, and A. Varshney. Skip strips: Maintaining triangle strips for view-dependent rendering. In *Visualization*, pages 131–137, 1999. 1.1, 2.3.1, 2.4.2, 2.4.2, 2.5, 5.2, 6.2.2

[ESC00] J. El-Sana and Y.J. Chiang. External memory view-dependent simplification. In *EUROGRAPHICS*, volume 3, pages 139–150, 2000. 2.4.2, 2.5

[ESV96a] F. Evans, S. Skiena, and A. Varshney. Completing sequential triangulations is hard. Technical report, Department of Computer Science, State University, 1996. 2.3.1

[ESV96b] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle srips for fast rendering. In *IEEE Visualization*, pages 319–326, 1996. 1.1, 2.3.1, 2.4.2, 4.2, 4.5.2, 5.2, 6.2.1, 6.6.1

[ESV99] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18(3):83–94, 1999. 2.2.2, 3.4.1

[FHWZ04] R. Fernando, M. Harris, M. Wloka, and C. Zeller. Programming graphics hardware. In *EUROGRAPHICS*, 2004. 2.4

[Gar99] M. Garland. Multiresolution modeling: Survey and future opportunities. In *Eurographics*, pages 111–131, 1999. 2.1, 2.4.2

[GE04] M. Gopi and D. Eppstein. Single-strip triangulation of manifolds with arbitrary topology. In *EUROGRAPHICS*, volume 23, pages 61–69, 2004. 2.3.1

[GH97] M. Garland and P. Heckbert. Simplification using quadric error metrics. *Computer and Graphics*, 31:209–216, 1997. 2.2.1, 2.2.2, 2.2.3, 2.2.3, 3.2.1, 4.2

[GPG07] GPGPU. General-purpose computation using graphics hardware (http://www.gpgpu.org), 2007. 2.4

[GW07] M. Giegl and M. Wimmer. Unpopping: Solving the image-space blend problem for smooth discrete lod transitions. *Computer Graphics Forum*, 26(1):46–79, 2007. 2.4.1

[HDD+93] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *SIGGRAPH*, 1993. 2.2.2, 3.2.1

[HON04] M. Hussain, Y. Okada, and N. Niijima. Efficient and feature-preserving triangular mesh decimation. *WSCG*, 12(1):167–174, 2004. 2.2.3, 2.2.3

[Hop96] H. Hoppe. Progressive meshes. In *SIGGRAPH*, pages 99–108, 1996. 1.3, 2.2.1, 2.2.2, 2.2.3, 2.2.3, 2.2.3, 2.4.2, 2.4.2, 2.4.2, 2.5, 3.5, 4.5.1

[Hop97] H. Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH*, pages 189–198, 1997. 1.1, 2.3.1, 2.4.2, 2.4.2, 2.5, 3.4, 3.4.2

[Hop99] H. Hoppe. Optimization of mesh locality for transparent caching. In *SIGGRAPH*, pages 269–276, 1999. 2.3.1, 2.4.2

[JR94] J. Helman J. Rohlf. Iris performer: a high performance multiprocessing toolkit for real-time 3d graphics. In *SIGGRAPH*, pages 381–394, 1994. 2.4.1

[JWLL05] J. Ji, Enhua. Wu, Sheng . Li, and Xuehui. Liu. Dynamic lod on gpu. In *CGI*, 2005. 2.4.2, 2.4.2, 2.5

[KG05] S. Kircher and M. Garland. Progressive multiresolution meshes for deforming surfaces. In *EUROGRAPHICS*, pages 191–200, 2005. 2.5, 6.2.2, 6.6.2

[Kor99] D. Kornmann. Fast and simple triangle strip generation. Technical report, 1999. 2.3.1

[Lak04] A. Lakhia. Efficient interactive rendering of detailed models with hierarchical levels of detail. In *International Symposium on 3D Data Processing, Visualization, and Transmission*, pages 275–282, 2004. 2.4.2, 2.5

[LE97] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *SIGGRAPH*, pages 199–208, 1997. 2.4.2, 2.5

[LPRM02] B. Levy, S. Petitjean, N. Ray, and J. Maillot. Least squares conformal maps for automatic texture atlas generation. In *Siggraph*, pages 362–371, 2002. 2.4.2

[LRC+02] D. Luebke, M. Reddy, J.D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of detail for 3D graphics*, volume 1. Elsevier Science Inc., 2002. 2.2.1, 2.2.2

[LT00] P. Lindstrom and G. Turk. Imagen-driven simplification. *ACM Trans. Graphics*, 19(3):204–241, 2000. 2.2.3

[Lue01] D.P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, 3(24):24–35, 2001. 2.2

[LY06] G. Lin and T. Yu. An improved vertex caching scheme for 3d mesh rendering. *Transactions on Visualization and Computer Graphics*, 12(4):640–648, 2006. 2.3.1

[MG03] A. Mohr and M. Gleicher. Deformation sensitive decimation. In *Technical Report*, 2003. 6.2.2

[MP05] R. Scateni M. Porcu, N. Sanna. Efficiently keeping an optimal stripification over a clod mesh. In *WSCG 2005*, volume 2, pages 73–80, 2005. 2.4.2

[MP06] O. Matias and H. Pedrini. A comparative evaluation of metrics for fast mesh simplification. *Computer Graphics Forum*, 25(2):197–210, 2006. 2.1, 2.2.2, 2.2.3

[NBS06] D. Nehab, J. Barczak, and P. V. Sander. Triangle order optimization for graphics hardware computation culling. In *Symposium on Interactive 3D Graphics and Games*, pages 207–211, 2006. 2.3.1, 2.4.2

[Paj01] R. Pajarola. Fastmesh: Efficient view-dependent meshing. In *PCCGA*, pages 22–30, 2001. 1.1, 2.4.2, 2.5

[Par05] J. Parus. Morphing of meshes. Technical report, 2005. 1.2, 6.2.1, 6.3.1

[PH97] J. Popovic and H. Hoppe. Progressive simplicial complexes. In *SIGGRAPH*, pages 217–224, 1997. 2.4.2

[PR00] R. Pajarola and J. Rossignac. Compressed progressive meshes. *Trans. on Visualization and Computer Graphics*, 6(1):79–93, 2000. 2.4.2

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction.* 1985. 2.3.1

[PS97] E. Puppo and R. Scopigno. Simplification, lod and multiresolution - principles and applications. In *EUROGRAPHICS 1997*, volume 16, 1997. 2.3, 2.4.2

[PS99] E. Puppo and R. Scopigno. Simplification, lod and multiresolution. In *EUROGRAPHICS 99*, volume 16, 1999. 2.4, 2.4.2

[PS03] M. Porcu and R. Scateni. An iterative stripification algorithm based on dual graph operations. In *EUROGRAPHICS 2003*, pages 69–75, 2003. 2.3.1, 2.4.2

[RAO+00] J. Ribelles, Lopez A., Belmonte O., Remolar I., and Chover M. Multiresolution modelling of polygonal meshes using triangle fans. In *DGCI*, pages 431–442, 2000. 2.3.1, 2.4.2, 2.4.2, 2.5, 5.2

[RC04a] F. Ramos and M. Chover. A comparison of strip-based multiresolution models. In *VIIP*, pages 239–244, 2004. 3.6

[RC04b] F. Ramos and M. Chover. Lodstrips: Level of detall strips. In *ICCS*, volume 3039, pages 107–114, 2004. 2.3.1, 4.6, 6.2.2

[RC04c] F. Ramos and M. Chover. Variable level of detall strips. In *ICCSA*, volume 3044, pages 622–630, 2004. 3.6

[RCBR04] F. Ramos, M. Chover, O. Belmonte, and C. Rebollo. An approach to improve strip-based multiresolution schemes. In *WSCG*, volume I, pages 349–354, 2004. 3.1, 3.6

[RCC04] F. Ramos, P. Castello, and M. Chover. Tiras de triangulos con niveles de detalle. In *CEIG*, pages 339–342, 2004. 4.6

[RCG04] F. Ramos, M. Chover, and C. Granell. Real-time terrain rendering using lodstrips multiresolution model. In *DEXA*, pages 819–823, 2004. 3.6

[RCLH99] J. Ribelles, M. Chover, A. Lopez, and J. Huerta. A first step to evaluate and compare multiresolution models. In *EUROGRAPHICS*, pages 230–232, 1999. 3.5, 4.5.1

[RCR05] O. Ripolles, M. Chover, and F. Ramos. Quality strips for models with level of detail. In *VIIP*, pages 268–273, 2005. 2.3.1

[RCRG06a] F. Ramos, M. Chover, O. Ripolles, and C. Granell. Continuous level of detail on graphics hardware. In *DGCI*, volume 4245, pages 460–469, 2006. 5.8

[RCRG06b] F. Ramos, M. Chover, O. Ripolles, and C. Granell. Efficient implementation of lodstrips. In *VIIP*, pages 365–370, 2006. 5.8

[RLB+02] J. Ribelles, A. Lopez, O. Belmonte, I. Remolar, and M. Chover. Multiresolution modeling of arbitrary polygonal surfaces: a characterization. *Computer and Graphics*, 26(3):449–462, 2002. 1.1, 2.1, 2.4, 2.4.2

[Sch97] W. Schroeder. A topology-modifying pregressive decimation algorithm. In IEEE Press, editor, *IEEE Visualization*, pages 205–212. IEEE, 1997. 2.2.2

[SFRV07] M. Sbert, M. Feixas, J. Rigau, and I. Viola. Tutorial: Applications of information theory to computer graphics, 2007. 2.2.3

[SG03] R. Southern and J. Gain. Creation and control of real-time continuous level of detail on programmable graphics hardware. *Computer Graphics Forum*, 22(1):35–48, 2003. 2.4.1, 2.5

[SM05] P. V. Sander and J. L. Mitchell. Progressive buffers: View-dependent geometry and texture for lod rendering. In *Symposium on Geometry Processing*, pages 129–138, 2005. 2.4.2, 2.5

[SP00] A. Shamir and V. Pascucci. Temporal and spatial levels of detail for dynamic meshes. In *Symposium on Virtual Reality Software and Technology*, pages 77–84, 2000. 6.2.2

[SP03] M. Shafae and R. Pajarola. Dstrips: Dynamic triangle strips for real-time mesh simplification and rendering. In *Pacific Graphics Conference*, pages 271–280, 2003. 1.1, 2.3.1, 2.4.2, 2.4.2, 2.5, 5.2, 6.2.2

[SSGH01] P. Sander, J. Snyder, S. Gortler, and H. Hoppe. Texture mapping progressive meshes. In *SIGGRAPH 2001*, pages 409–416, 2001. 2.4.2

[Ste01] J. Stewart. Tunneling for triangle strips in continuous level-of-detail meshes. In *Graphics Interface*, pages 91–100, 2001. 1.1, 2.3.1, 2.4.2, 2.4.2, 2.5, 5.2, 6.2.2

[SZW92] W. Schroeder, J. Zarge, and Lorenson W. Decimation of triangle meshes. *Computer and Graphics*, 26:65–70, 1992. 2.2.2

[TAF92] Seth J. Teller Thomas A. Funkhouser, Carlo H. Sequin. Management of large amounts of data in interactive building walkthroughs. In *Symposion on Interactive 3D Graphics*, pages 11–20, 1992. 2.4.1

[Tur07] P. Turchyn. Memory efficient sliding window progressive meshes. In *WSCG*, 2007. 2.4.2, 2.5

[Ups90] Steve Upstill. *The Renderman Companion.* Addison Wesley, Reading, 1990. 2.4.1

[VFG99] L. Velho, L.H. Figueredo, and J. Gomes. Hierachical generalized triangle strips. *The Visual Computer*, 15(1):21–35, 1999. 2.3.1, 6.2.2

[VK07] P. Vanecek and I. Kolingerova. Comparison of triangle strips algorithms. *Computer and Graphics*, 31(1):100–118, 2007. 2.3.1

[Wer94] Josie Wernecke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics With Open Inventor, Release 2.* Addison-Wesley, 1994. 2.4.1

[WLC+03] N. Williams, D. Luebke, J.D. Cohen, M. Kelley, and B. Schubert. Perceptually guided simplification of lit, textured meshes. In *Symposium on Interactive 3D Graphics*, pages 113–121, 2003. 2.2.3

[XESA97] J. Xia, J. El-Sana, and Varshney A. Adaptative real-time level-of-detail rendering for polygonal models. *Trans. on Visualization and Computer Graphics*, 3(2):171–183, 1997. 2.1, 2.4.2

[XHM99] X. Xiang, M. Held, and J. S. B. Mitchell. Fast and effective stripification of polygonal surface meshes. In *Symposium on Interactive 3D Graphics*, pages 71–78, 1999. 2.3.1

[XV96] J. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal meshes. In *Visualization*, pages 327–334, 1996. 2.2.2

[YSG05] S. Yoon, B. Salomon, and R. Gayle. Quick-vdr: Interactive view-dependent rendering of massive models. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):369–382, 2005. 2.4.2, 2.5

[Zac02] C. Zach. Integration of geomorphing into level of detail management for realtime rendering. In *SCCG 2002*, 2002. 1.1, 2.4.1, 2.5

[ZMK02] C. Zach, S. Mantler, and Karner K. Time-critical rendering of discrete and continuous levels of detail. In *EUROGRAPHICS 2002*, pages 1–8, 2002. 2.4.2, 2.5

[ZS00] D. Zorin and P. Schoeder. Subdivision for modeling and animation. In *SIGGRAPH*, 2000. 2.4.2